Translation Validation for Verified, Efficient and Timely Operating Systems

Thomas Sewell

School of Computer Science and Engineering

University of New South Wales
Sydney, Australia

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

November 2017

‘I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project’s design and conception or in style, presentation and linguistic expression is acknowledged.’

Signed: _____

Date: _____

In light of the above, let me add a note about authorship. In the following chapters I will follow the same convention that I do in the related papers, and always say “we did” instead of “I did”, rather than carefully delineate where my path has joined or diverged with the paths of others.

I will always be indebted to those I have worked with. Nevertheless, the SydTV approach and software described in this thesis is almost exclusively my work. The exception is the contribution of Felix Kam, whose undergraduate thesis I supervised during my time as a PhD candidate. His contribution will be clarified in Chapter 3. The SydTV approach also makes use of an independent tool written by Magnus Myreen, with whom I have collaborated closely in the past. In this text I will describe his work in outline but not claim it as a contribution.

This text incorporates sections of prior publications listed on the next page, which my co-authors Gerwin Klein, Magnus Myreen, Felix Kam and Gernot Heiser have helped to edit.

This thesis incorporates these core publications:

- Thomas Sewell. Formal replay of translation validation for highly optimised C: Work in progress. In *Verification and Program Transformation*, Vienna, Austria, July 2014
- Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *RTAS*, Vienna, Austria, Apr 2016
- Thomas Sewell, Felix Kam, and Gernot Heiser. High-assurance timing analysis for a high-assurance real-time OS. *Real-Time Syst.*, 53:812–853, Sep 2017

This thesis will also make reference to, but not directly build on, other work in which I have participated during the period of my PhD work:

- Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70, Feb 2014
- Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *ASPLOS*, pages 175–188, Atlanta, GA, USA, Apr 2016
- Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *ICFP*, Nara, Japan, Sep 2016
- Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. In *ITP*, Nancy, France, Aug 2016

Finally, this thesis will make reference to, but not claim any contributions of, prior work of mine from before my PhD candidature, especially the first here:

- Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–481, Seattle, Washington, USA, Jun 2013
- Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In *TPHOLs*, pages 500–515, Munich, Germany, Aug 2009
- Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009
- Sascha Böhme, Anthony CJ Fox, Thomas Sewell, and Tjark Weber. Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In *International Conference on Certified Programs and Proofs*, pages 183–198. Springer, 2011
- Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *ITP*, pages 325–340, Nijmegen, The Netherlands, Aug 2011

Abstract

Computer software is typically written in one language and then translated out of that language into the native binary languages of the machines the software will run on. Most operating systems, for instance, are written in the low-level language C and translated by a C compiler. *Translation validation* is the act of checking that this translation is correct. This dissertation presents an approach and framework for validating the translation of C programs, and three experiments which test the approach.

Our validation approach consists of three components, a frontend, a backend and a core, which broadly mirrors the design of the C compiler. The three experiments in this dissertation exercise these three components. Each of these components produces a *formal proof* of *refinement*, and these refinement proofs compose to produce a proof that the binary is a refinement of the source semantics. This notion of refinement can then compose with correctness proofs for a C program, resulting in a *verified binary*. Throughout this work, our case study of interest will be the seL4 verified operating system kernel [KEH+09], compiled for the ARM instruction-set architecture, for which we will produce a verified efficient binary.

The *thesis* of this work is that our translation validation approach offers us great *flexibility*. We can quickly produce verified binaries produced via many complex transformations without specifically addressing each such transformation. We can adapt our frontend to handle low-level source code which does not strictly respect the rules of the C language it is written in. We can also retarget our backend to address important timing concerns as well as correctness ones.

Contents

1 Introduction

1.1 The SydTV Translation Validation Approach

Translation validation is the act of checking that the translation of a computer program is correct. This dissertation presents an approach for validating the translation of C programs, and three experiments which test the approach. This approach is exemplified by a software tool developed in this work. It is time to give the software a name, and so we will dub it the Sydney Translation Validation suite, SydTV for short.

The main function of SydTV is to produce a proof of *refinement*. This refinement is divided into three component proofs, which we sketch in Figure 1.1. These proofs are performed by different components of SydTV in three different logical environments. The three proof steps are connected by two intermediate representations of the program. These shared representations are expressed in a common interchange language, SydTV-GL, which we designed to have a simple known semantics that can be expressed in all three logical environments.

Both the C language and the ARM processors we are interested in have a number of complex features which are not strictly necessary. By contrast, SydTV-GL is designed to be as simple as possible. The main function of the frontend is to reduce the complexities of the C language to a simpler SydTV-GL representation, and prove this reduction was sound. This component is implemented within the Isabelle/HOL theorem prover [NPW02], to connect to the existing Tuch/Norrish C semantics [TKN07] and the C-to-Isabelle parser used in a number of program verification projects. The main function of the backend is to eliminate complexities of the ARM architecture and function calling convention, recovering a structured program. For the backend, we make use of Myreen's method of decompilation into logic [MGS08, MGS12]. This is implemented within the HOL4 theorem prover, and connects to the Cambridge validated ARM semantics [FM10].

The core component of our approach, SydTV-GL-refine, is a custom standalone tool which is backed by a suite of SMT solvers. The key function of SydTV-GL-refine is to discover a proof of refinement between two programs which are semantically related but may be structurally quite different. The discovered proofs are also checked within SydTV-GL-refine.

While all of the components of SydTV perform translation validation, the nature of the problem varies substantially. The outer components address the ideal version of the translation validation problem, where the translation has been deliberately structured to be easy to validate. The inner SydTV-GL-refine component, however, must handle

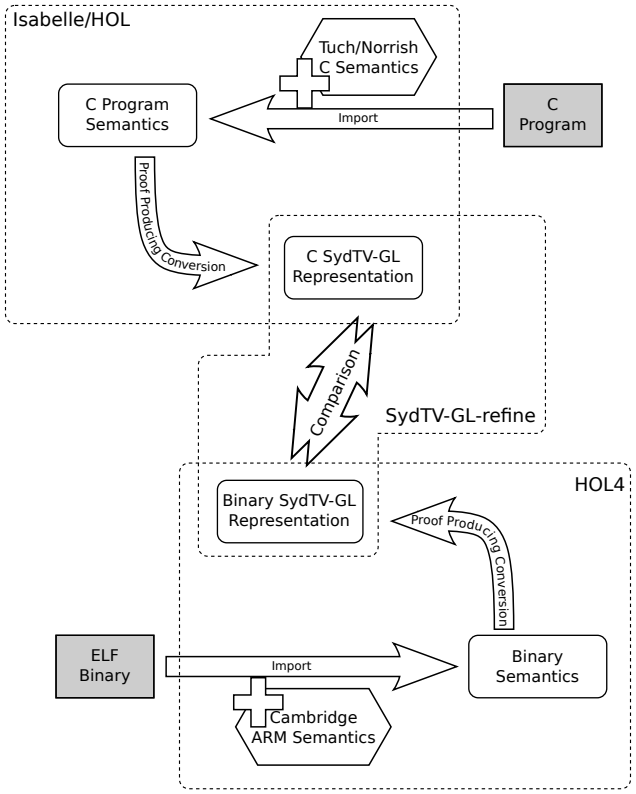


Figure 1.1: Tools and artefacts of SydTV

the challenging variant of translation validation where nothing at all is known about the relationship between the two target programs except that one ought to be a valid translation of the other.

1.2 Experiments

In addition to developing SydTV, we test our approach through three broad experiments, which the three major chapters of this dissertation will explore.

In our first experiment, we test the primary function of SydTV, producing a refinement proof in a challenging setting. We aim to produce a completely verified seL4 binary when it is compiled with complex loop optimisations. This mostly exercises the design and implementation of the SydTV-GL-refine component, which we will explain in detail.

Our second experiment applies SydTV to a task for which it was not originally designed. To reason about the execution time of programs, we link SydTV to the Chronos timing tool [LLMR07]. We extend SydTV-GL-refine to establish timing relevant information, such as bounds on the number of iterations possible in loops. By making use of additional information passed through by the user from the Isabelle-based frontend, we create a rich environment for proving that a binary executes within essential deadlines.

In our final experiment, we revisit the theoretical foundations of the SydTV approach, replicating elements of the SydTV-GL-refine logic within the theorem prover Isabelle/HOL. In this work we confirm that the basic reasoning principles of SydTV-GL-refine are sound, but also discover that our theoretical foundation is more complex that we thought, including connections to variants of the axiom of choice.

1.3 Contributions

The implementation of SydTV builds on a previous nameless prototype, which we built in a previous project. In that work, we showed it was possible to verify a key subset of the compiled seL4 binary [SMK13] when major optimisations were disabled. SydTV inherits its structural design from that prototype, which had the same division across three components, and also included a predecessor of SydTV-GL. To clarify the contribution of this work, we will from time to time make comparisons between elements of SydTV and their corresponding features in the earlier prototype.

Specifically, the contributions of this work are:

- A model-guided SMT-based search process for producing a translation validation proof which handles substantial optimisations and does not assume any knowledge about the way the binary representation was produced from the source program.
- Careful tuning of the proof discovery process to handle exotic cases, improve performance and ensure complete coverage on sizeable binaries.
- A case study in applying the whole validation process to produce an efficient verified OS binary.
- Discovery of minor defects in the well-publicised seL4 verification, adjustments to the C verification environment, and re-verification.
- A novel approach to the WCET (worst-case execution time) analysis problem, in which a translation validation suite and a sofware verification environment are used in high-assurance timing analysis.
- Improvements to the timing behaviour of the seL4 kernel by replacing a crucial operation with a preemptible counterpart, and re-verification of the adjusted kernel.
- Thorough validation of the formal underpinnings of the approach, including modest contributions to the relevant theory.

2 Proving Refinement

The main function of SydTV is to prove refinement between the source and binary semantics of a program. Together with a verified source program, this gives us a verified binary. Various proof approaches can produce such verified source programs, including the the seL4 C refinement approach [WKS+09], the AutoCorres tool [GLAK14], and the proof framework of the Cogent language [OCR+16, RLN+16]. These approaches give us various verified programs of interest, including of course the seL4 microkernel [KEH+09], algorithms from the LEDA framework [MN89, NRM14], and filesystems implemented in Cogent [AHC+16].

Each of these programs has been verified against the Tuch/Norrish C semantics [TKN07], using the C-to-Isabelle parser to produce a formal model of the program in Isabelle/HOL.[1] The various proof approaches connect the C semantics "upward" to higher level properties. The refinement proof produced by SydTV connects to these same semantics "downwards", and links down to the actual semantics of the binary program.

Crucially, the high-level and low-level proofs "meet" at the C semantics. Firstly, this ensures that the correctness conditions which the high-level proofs check are exactly those which the low-level proofs may assume. Secondly, we avoid questions about whether all tools agree on the interpretation of the C standard. Indeed, not all experts agree on the interpretation of the C standard. It is not even important whether the compiler and C-to-Isabelle parser are "correct" with regard to any particular understanding of the C standard, as long as they agree closely enough that the binary of interest can be shown to have the properties that the C program was verified against.

The proof of refinement produced by SydTV is split into three component refinement proofs, as shown previously in Figure 1.1. While the proofs are performed in different tools and there is not a single end-to-end theorem in any one of them, the refinement proofs all logically compose. We could, for instance, hypothesise in Isabelle/HOL the results which we show in HOL4 and SydTV-GL-refine, and derive from those hypotheses a correctness property for the machine code.

While all three proof components of SydTV are important, this chapter will focus on the central proof in SydTV-GL-refine, which is the most challenging aspect. The conversion from the Isabelle/HOL representation produced by the C-to-Isabelle parser is the easiest aspect, although it raises a number of theoretical questions which we will return to in Chapter 4. The decompilation is performed by a special-purpose decompiler

[1] The parser is available as part of the official L4.verified distribution:
`https://github.com/seL4/l4v/blob/master/tools/c-parser/`

component, which forms part of SydTV. This decompiler is one of a family of related tools maintained by Myreen [MGS08, MGS12], customized to support this project. For the purposes of this PhD discussion we will treat the decompiler as a separate work, sketch its function, and leave discussion of its design to another document.

The proof approach of SydTV-GL-refine itself has several major components. As recommended by Pnueli et al, in the original formulation of translation validation [PSS98], we divide SydTV-GL-refine into a search process and a check process, with the search process discovering a proof script which the checker then validates. Both the search process and the check process proceed by reducing questions about programs expressed in SydTV-GL to SMT problems. This reduction is divided into two phases, which we will outline shortly. The search process is also supported by a static analysis component.

This chapter will explore the design of SydTV-GL-refine, working our way from the high-level proof structure to implementation specifics, and then describe the results of our case study in applying the tool to the seL4 binary. The structure of proof scripts, and the way they are converted to SMT proof obligations, is introduced in Section 2.1. The way the search process discovers proof scripts is introduced in Section 2.2. Additional features required to reason about the decompiled binary programs are introduced in Section 2.3, and fine-tuning needed to cover difficult cases is discussed in Section 2.4. Section 2.5 describes our case study in applying the toolset to the seL4 binary, and the adjustments to the seL4 verification that were necessary to complete this work, with the performance results of the case study discussed in Section 2.6. Related work in the field of translation validation is examined in Section 2.7, and Section 2.7.1 specifically compares to the approach of verified compilers.

2.1 Structure of SydTV-GL-refine Correctness Proofs

The three components of SydTV cooperate to produce a proof of refinement between the C source semantics and the binary code. The outer components reduce the semantic complexities of the C language and the ARM architecture to a uniform representation in the language SydTV-GL. This section focuses on the inner component SydTV-GL-refine, and the structure of the refinement proofs that it checks.

However, while the aim of the section is to focus on SydTV-GL-refine, we begin by describing the inputs to SydTV-GL-refine, that is, the expected behaviour of the other two tools.

2.1.1 Conversion from C Semantics to SydTV-GL

Let us begin by examining the pseudo-compilation process which converts functions in the Isabelle C model into SydTV-GL. This also serves as an introduction to SydTV-GL. SydTV-GL stands for SydTV Graph Language, because the primary idea of the conversion is to replace complex language control flow rules with an explicit control flow graph.

This conversion is best illustrated with an example. Consider the function `find`, shown in Figure 2.1. To convert the C statements to the graph language, we number them all, with the steps from statement to statement becoming the edges of a labelled directed graph. Each node of the graph has a statement component as well as an address. The special label `Ret` represents the return point of 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 variables

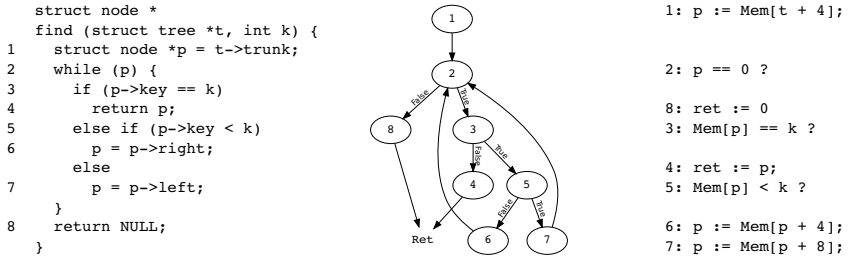


Figure 2.1: Example Conversion of Structure and Statements to Graph Language

with the result of some calculations. Call nodes are used to represent function calls, which are distinguished from other statements. A number of restrictions enforced by the C-to-Isabelle parser assist us in this conversion, for instance, function calls may be embedded in other expressions only in very limited ways, and statements with multiple effects are forbidden.

The conversion also simplifies the expressions of the C language. SydTV-GL has only named variables, no globals, so memory must be treated as a normal variable, accessed and updated with special operators. The only numeric types available are n-bit machine words, for any length n, and there are no container types. Pointers become 32-bit words, and address operations of various kinds are replaced with the relevant arithmetic. Insofar as C can be seen as a portable assembler, the conversion makes this explicit. Some examples are given in Figure 2.1, with implicit memory accesses becoming explicit, and implicit addressing within structures becoming explicit arithmetic on addresses.

Local variables of structure and array type are also 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 are later instantiated by reading the symbol table of the binary. 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.

The C-to-Isabelle parser uses the Tuch memory semantics [Tuc08] to encode memory accesses. It is a theorem of this semantics that a memory write of an aggregate type is equivalent to a write of each of its fields, but only if the aggregate type contains no padding. Padding creates a number of headaches for us. Rather than solving the padding problem in general, we require the input C source to avoid it, which is usually easy for the programmer to ensure. For instance, we can easily remove the use of padding in the seL4 sources, as we discuss in Section 2.5.2.

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 these cannot all be encoded using machine types. 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 C-to-Isabelle parser inserts a number of `Guard` statements into its output. A `Guard` statement wraps another statement with some condition, and the condition is added as a verification obligation in any proof about the inner statement.

The statement which accesses `p->right` in Figure 2.1, for instance, will be wrapped in a guard in Isabelle, which checks that `p` in not `NULL`. The graph conversion then translates the guard statement into a condition node with one of the outbound edges pointing to

the special label `Err`. Both the C and graph variants of these guards were omitted from Figure 2.1 for clarity.

The guards that addresses are non-zero and aligned, and that arithmetic is non-overflowing, can themselves be represented with concrete machine arithmetic. This means that all the expressions and variables used in SydTV-GL can themselves be translated into SMT equivalents. The exception to this relates to the *strict-aliasing* rule of the C standard. This rule says that any given memory address may not be in use with two different incompatible types. This rule is occasionally broken in low-level programming, but usually holds, and is important in enabling many performance optimisations. To address this, we include in SydTV-GL a small number of types and operators from the Tuch memory semantics [Tuc08] in Isabelle/HOL. These operators permit assertions which declare that a typed pointer is valid. We return to the details of this encoding in Section 2.1.12.

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.

We revisit the conversion from Isabelle/HOL to SydTV-GL in Chapter 4, including a detailed presentation of the above semantics in (Section 4.2).

2.1.2 Decompiling Compiler Output into Logic

The lowest-level refinement proof we perform relates the semantics of the compiled binary into a SydTV-GL representation. This is accomplished by a decompiler tool, one of a family of such decompilers maintained by Myreen [MGS08, MGS12]. The decompiler variant included in SydTV targets SydTV-GL directly, producing SydTV-GL representations of each instruction in the binary and a proof in HOL4 that the binary semantics refine those of the SydTV-GL bodies.

This proof connects our work in SydTV-GL-refine down to the Cambridge ARM semantics. It is important to note that the accuracy of these processor semantics has been exhaustively validated by testing the processor model alongside real silicon [FM10]. This gives us great confidence that we are proving results that directly relate to the real-world semantics of our programs.

The way the SydTV decompiler represents stack accesses has substantial implications for SydTV-GL-refine, which we will return to in Section 2.3.

2.1.3 Proof Scripts in SydTV-GL-refine

Each of the three refinement proofs in SydTV is decomposed into one proof per function. The check process of SydTV-GL-refine checks one proof script for each named function in the binary. Any other functions called will be treated as black boxes, specified only by their corresponding refinement theorem.

The proof script for each function consists of a problem space, together with a tree of proof rules. The proof rules are named **Restrict**, **Split**, **CaseSplit** and **Leaf**. These rules give structure to the proof, reducing the refinement problem to a number of proof obligations. The heavy lifting is then done by converting proof obligations on the problem

space into SMT problems. We will describe these proof components as the proof checker sees them.

We also include a more detailed example, and a sketch of the way its correctness proof would be produced, in Section 4.4.2.

The proof rules will, together, reduce the problem of refinement for a pair of programs in SydTV-GL to a collection of specific proof obligations. These obligations can be addressed by conversion to SMT. This forms a reduction of an *undecidable* problem (refinement in a Turing-complete language) to a problem for which SMT solvers are efficient decision procedures. The proof checker aims to be a *complete* decision procedure for checking proof scripts. Given a correct proof script, the checker should always verify refinement. In practice, the checker may sometimes time out because the complexity of the problem defeats the SMT solvers, but it should not decide against a correct proof script.[2] By comparison, the discovery of the proof script by the search process, which we will describe in Section 2.2, is heuristic, and may fail because we have not yet adequately addressed particular kinds of compiler strategies.

2.1.4 Function Pairings

The SydTV-GL-refine refinement proof proceeds one function at a time. For each function, we prove a correctness theorem. Within the proof of each function, other functions are specified only by their correctness theorem.

The first complication in SydTV-GL-refine is that the function call graphs of the C and binary may differ. The compiler has a choice at every C call site whether to generate a binary function call or to include ("inline") the body of the called function. If the called function is marked `static` in C, and it is inlined at all call sites, then it may not appear under its name in the binary at all.

For this reason, many functions that exist in the C SydTV-GL representation have no counterpart in the binary SydTV-GL artefact. This also sometimes happens the other way around, when the compiler may, on rare occasions, generate an anonymous function with no known counterpart in C. More frequently, the compiler may generate "clone" functions which implement a named C function in some sense but do not promise to implement those functions in the usual way. We discuss the issue of clones at length in Section 2.5.7.

Thus, the function-by-function breakdown is really a breakdown by function *pairings*. The first step for SydTV-GL-refine is to generate the list of function pairings. A function pair names the C and binary SydTV-GL functions, and specifies the statement of their correctness theorem.

The correctness theorems are all stated as an implication, with a premise relating the input values and a conclusion relating the output values. Functions in SydTV-GL have an arbitrary length list of named input and output values, which may differ between the paired functions. For instance, all binary SydTV-GL functions produced by the SydTV decompiler have the same list of inputs and outputs, which includes all the CPU registers, the stack and main memory. The input and output relations specify the connection between these values.

The input and output relations are mostly specified by the architecture calling convention. For instance, according to the ARM specification, a C function of two arguments, `x` and `y`, will be paired to a binary function, where the input relation for these arguments

[2] There is an issue with stack equalities which we discuss in Section 2.3, which specifically makes the checker incomplete.

is that $\texttt{x}_{init(C)} = \texttt{r0}_{init(BIN)}$ and $\texttt{y}_{init(C)} = \texttt{r1}_{init(BIN)}$. Input relations in SydTV-GL-refine pairings are always a list of equalities, where each side of each equality is a variable expression that can be evaluated using either the binary or C input variables.

The output relations are also a conjunction of equalities. Output equations may either link output variables to each other, or link output variables to input variables. This allows the output relation to specify return value relations, e.g. $\texttt{retv}_{Ret(C)} = \texttt{r0}_{Ret(BIN)}$, as well as callee saved values, e.g. $\texttt{r6}_{Ret(BIN)} = \texttt{r6}_{init(BIN)}$.

2.1.5 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 with the remaining function calls treated as black boxes.

Inlining is done on the binary side of the problem whenever the called function does not match any C function, that is, is not part of any pairing.

Inlining is done on the C side of the problem at any call site which is reachable and which is not paired with a function that appears on the binary side of the problem. This simple heuristic might fail in the presence of selective inlining, where a C function contains two call sites for the one function and the compiler made different decisions about inlining. The heuristic 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 `-O1` and `-O2` for which we have extensively tested it.

This heuristic makes use of the syntax of the binary function, and considers the semantics of the C function alone to decide reachability. It does not make use of the semantic link between the functions yet. This design decision was made to allow us to perform all inlining before doing more complex analysis, e.g. comparing the semantics of loops.

The problem space produced after inlining is included in the proof script, and the SydTV-GL-refine checker trusts that this problem space can be derived from the functions of interest. To be truly skeptical, the checker could attempt to replay the inlining procedure itself. We have an incomplete project to replay the refinement proofs of SydTV-GL-refine within the Isabelle/HOL theorem prover, which will require an official check of this phase. We discuss our progress on that replay project in Section 4.4.

2.1.6 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 2.2 shows nearly all of the steps of interest. The boxes show the path condition and variable values immediately before execution of each node. The input variables at the entry point, for

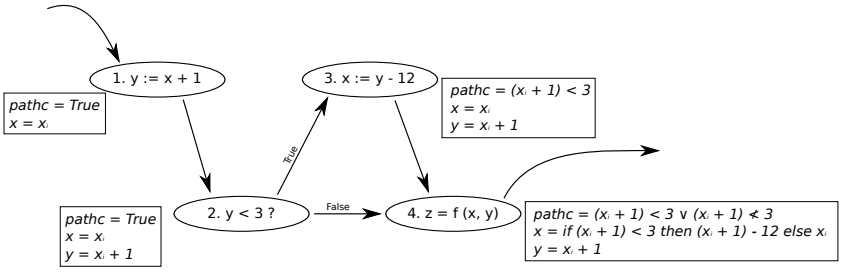


Figure 2.2: Example Conversion to SMT

instance x_i, become SMT unknowns. The path condition at the entry point 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 2.2 was the variable state after calling function f at 4. The return values are named, for instance z_after_4, and become additional SMT unknowns. The SMT conversion process notes that a call to f took these input values and returned the named output values (recall that inputs and outputs include global objects like memory). If f is a part of a function pairing, and if at a later point the paired function f' is also exported to SMT, then the conversion process will instantiate the correctness condition of f/f' for these inputs and outputs.

Figure 2.2 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 addresses the problem we already see at node 4, where the expressions are becoming larger and larger.

This process does not handle loops, and the generalisation will be discussed in Section 2.1.8.

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 bit-vectors, typically 32-bit but any bit width is supported. Memory is represented as an SMTLIB array. We have two choices how to encode memory, either as an array of bytes, mapping 32-bit to 8-bit vectors, or as an array of words, mapping 30-bit to 32-bit vectors. Since the vast majority of loads and stores on our 32-bit architecture are aligned 32-bit accesses, the latter representation usually performs better. This tradeoff is explored further in Section 2.4.5.

Some values cannot be directly encoded, for instance the heap type description and pointer validity assertions described earlier. During this conversion of graph visits to SMT problems, SydTV-GL-refine proceeds as though the SMT logic contained additional types and operators. A second phase then reduces the SMT problems to equisatisfiable problems containing only types and operators of the QF_ABV logic. This second module is effectively a stronger SMT solver, supporting additional theories of pointer validity and stack preservation. The encoding of the theory of pointer validity is discussed in Section 2.1.12, and the theory of stack equalities is discussed in Section 2.3.2.

2.1.7 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 initial assumptions will be the input equalities given by the function pairing of the function we are examining, for example, $x = r_0, y = r_1$ etc.

The main objective is to show that the output equalities of the function pairing hold on the output variables, should the arcs to the `Ret` label be reached. 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.

2.1.8 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 impossible 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 for this function pair must somehow address this.

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 n-th visit to that node is unsatisfiable by SMT. The proof checker then introduces a restriction, which asserts that this node is visited less than n times. This restriction is used in the SMT conversion process, promoting the semantic limit into a syntactic one. The SMT conversion process can now cover more of the nodes of the graph.

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.

2.1.9 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 predicate 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 at least n elements, in which case there exists some $i \geq 0$ and the sequence ends with the elements $i, i+1 \ldots i+n-1$. Finally, 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. We call the case with $\geq n$ iterations the looping case. In the looping proof, a new variable i and 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-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. The **Split** rule may concern a subsequence of visits to the nodes on either side. A C sequence starting 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, etc. 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. In a more dramatic case, a C sequence step width of 2 means that we only consider every second visit to the C split point, for instance relating visits 1, 3, 5…in the C loop to visits 1, 2, 3…in the binary loop. This case is needed for loops unrolled by the compiler, which we will discuss at length in Section 2.4.1. Starting offsets and larger step widths are also allowed on the binary side.

In addition, 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.

2.1.10 Case Division

The **CaseSplit** rule names a particular node, and considers the case where that node is visited and the case where it is not. It has two children, one proof for each case.

This is the simplest of our structuring rules. The checker performs no checks related to the rule, but adds a new hypothesis to each case, the truth and the negation of the path condition of interest.

There are two reasons for the **CaseSplit** rule to be used in proof scripts. The first reason is for simple performance. Many programs themselves have a case-division structure, with a toplevel `if` or `switch` statement, jump-table or similar. An ideal proof can divide the problem into the relevant cases, and then use **Restrict** rules to eliminate the unreachable syntax on both the C and binary side of the problem. This may substantially reduce the complexity of the extracted SMT problems.

The second reason for the **CaseSplit** rule is to handle the complexities of unrolled loops. We will return to this issue in Section 2.4.1.

2.1.11 Prototype Proofs

The proof format of SydTV-GL-refine is largely inherited from the former prototype, which also had proof scripts containing **Restrict**, **Split** and **Leaf** rules. The two new features both relate to the unrolled loop problem we have alluded to, the first being the inclusion of the "step width" parameters in the **Split** rule, and the second new feature being the **CaseSplit** rule itself.

2.1.12 SMT Theory Extensions

The SMT conversion process described in Section 2.1.6 eliminates the graph structure of SydTV-GL programs, resulting in a problem with a specific collection of unknowns and a number of assertions about them, an SMT problem. The operators and constants of SydTV-GL are also replaced with their equivalents in the SMTLIB2 QF_ABV logic (quantifier-free formulae over arrays and bit-vectors). Most of the language elements of SydTV-GL encode directly into QF_ABV.

However, some of the types and operators of SydTV-GL do not have any direct equivalents in SMTLIB2 QF_ABV. The SMT conversion process of Section 2.1.6 does not directly address this problem. Instead, it proceeds as though the QF_ABV logic contained additional types and operators. When the problem is ready to be passed to the SMT solver, a second phase adjusts the problem into a logically equivalent problem with the additional types and operators removed.

Through this second phase, SydTV-GL-refine is really implementing an SMT logic in its own right. SMT stands for "satisfiability modulo theories". The theories are the collections of types and operators on which the satisfiability problems can be stated. The theories are all backed by a decision procedure which can both discover satisfying models and refute unsatisfiable hypotheses. Many theories can be implemented by reducing to other theories, or to the base problem of boolean satisfiability. For instance, the theory of arrays over bit-vectors is typically addressed by reducing to the theory of bit-vectors, a reduction we will describe in detail in Section 2.3.2. SydTV-GL-refine is effectively building a richer SMT logic on top of QF_ABV, with a decision procedure based on reducing to standard QF_ABV.

Some of the additional types and operators encode easily. SydTV-GL has a type of memory, which can be accessed with various word lengths, e.g. 8-bit and 32-bit loads and stores. As we have already mentioned, we have a choice whether to represent memory as principally 8-bit or 32-bit. Once we have made this choice, the loads and stores can be encoded as compound operations of QF_ABV.

Other additional operators include the CPU-provided bitwise operations to count leading zeroes and reverse the order of a bit-vector. These are not SMTLIB2 builtins, but are easy to define for each bit-vector length for which they are used. SydTV-GL also contains a type of "symbols", string constants which support equality comparisons. SydTV-GL-refine maintains a global table of known symbols, giving each encountered symbol a unique number, and encoding the symbols in QF_ABV as a 32-bit encoding of that number.

SydTV-GL-refine implements two more challenging theories, the theory of pointer validity, and the theory of stack equality. The stack equality issue is specific to the binary aspect of the problem, and is discussed in Section 2.3.2.

The theory of pointer validity is needed to handle the *strict-aliasing* rule of the C standard. 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, but most code conforms to it.

To substantiate the strict-aliasing rule, we make use of the heap type validity predicate from the Tuch memory model [Tuc08]. This is the strongest pointer validity operator in the Tuch model. In Isabelle/HOL, the predicate takes the form $htd \models_t p$, where htd is a heap type descriptor object, and p is a pointer to some C type. This notion of pointer validity has the properties we require, requiring any address in memory to be in use with at most one top-level type, but permitting the fields within an aggregate object to be valid pointer targets also. The details of how the heap type descriptor and operator are implemented are discussed in detail by Tuch [Tuc08].

The Isabelle/HOL formulation of pointer validity via $\models_t$ depends on Isabelle's ad-hoc polymorphism, where the type of a pointer p also encodes the type of object it points to as a phantom type parameter. For our purposes, we reformulate the predicate in Isabelle/HOL using a new predicate pvalid which makes the type parameter explicit:

$$htd \models_t (p :: \tau\ ptr) = \text{pvalid}\ htd\ \text{TYPE}(\tau)\ (\text{ptr\_val}\ p)$$

The ptr_val operator on the right extracts the address from the pointer object in the Tuch semantics, discarding the phantom knowledge of the type τ. We make the type parameter explicit in the syntax TYPE(τ). This syntax is itself based on ad-hoc polymorphism and a phantom type, but the details of this are not important for these purposes.

We add the pvalid operator to SydTV-GL, also adding a type HTD of heap type descriptors, and a "type of types" to encode TYPE(τ). We include the global HTD as an explict argument and return value of all the C SydTV-GL functions, the same way memory is treated. We also supply details of the layout of structures in C, a type constructor for named C structs, and a type constructor for arrays of other types, but we provide no operators for any of these types. We use the additional C types only in the "type of types" as arguments to pvalid.

SydTV-GL-refine cannot export the pvalid operator into any equivalent in any SMT theory. Instead, each time it encounters a pvalid expression, it introduces a new boolean unknown, e.g. $pvalid_1$, $pvalid_2$.

The essential task is to make use of the key theorem about pvalid:

$$\frac{\text{pvalid}\ htd\ \tau\ p \qquad \text{pvalid}\ htd\ \tau'\ p'}{\begin{array}{c}\{x \mid p \le x < p + \text{size}(\tau)\} \cap \{x \mid p' \le x < p' + \text{size}(\tau')\} = \{\} \\ \vee\ p' - p \in \text{subtype\_offsets}(\tau', \tau) \\ \vee\ p - p' \in \text{subtype\_offsets}(\tau, \tau')\end{array}}$$

This theorem tells us that two pointers valid in the same heap type descriptor must either be totally disjoint, or one must point into a subtype (a field) of the other.

This general fact reduces to a simpler SMT form, such as $pvalid_1 \wedge pvalid_2 \longrightarrow p + \text{size}(\tau) - 1 < p' \vee p' + \text{size}(\tau') - 1 < p$ for disjoint types τ, τ'. We produce all such theorems, a possibly quadratic expansion. The largest group of pvalid assertions which involve the same heap type description which we have seen so far had size 20.

SydTV-GL-refine does not fully implement the theory of pvalid and HTD objects. In principle the theory should also include a mechanism for proving HTD objects equal. Furthermore, the C program may need to adjust the HTD value, to add new valid pointers or invalidate old ones. SydTV-GL needs to contain operators for adjusting the HTD value,

but SydTV-GL-refine does not provide any support for reasoning about the relationship between pvalid predicates before and after the adjustments.

The reason that SydTV-GL-refine does not implement more of the HTD theory is that we can assume that SydTV-GL-refine will always be assuming pointer validity facts and never trying to prove them. The pvalid predicates will appear as assertions in the C SydTV-GL bodies, but not in normal expressions or anywhere in the binary functions. If we wanted to use SydTV-GL-refine to prove refinement relations between C programs, however, we would have to extend this mechanism.

This support for pointer validity operators is inherited largely unchanged from our former prototype. SydTV-GL-refine includes one major change to the former handling of pointer validity. The change concerns the way arrays are treated, and has significant implications for our experiment with seL4, which we discuss in Section 2.5.5.

2.2 Proof Search

We have seen in Section 2.1 how proof scripts are structured in SydTV-GL-refine, decomposed by **Restrict**, **Split** and **CaseSplit** rules as necessary to handle nontrivial control flow.

The most complex component of SydTV-GL-refine is the search process, which must discover these proof scripts. This section explores the various strategies used by the search process to build components of proof scripts.

2.2.1 Easy Cases

Discovering some **Restrict** and **CaseSplit** rules can be done by straightforward heuristics. For instance, if the problem contains a loop on one side and not the other, clearly it has been fully expanded on the other side of the problem, or is unreachable. We must find a loop bound and use a **Restrict** rule (an unreachable loop has a bound of zero). Loop bounds can be discovered naïvely in this case since we are always searching for modest loop bounds (less than 50).

CaseSplit rules can also sometimes be discovered straightforwardly. If the problem can be decomposed based on early control flow decisions in both the function graphs, and if there is little shared code between the divided subproblems, then a **CaseSplit** rule is appropriate. To decide whether the code shared between subproblems is too substantial, we estimate its difficulty using a simple heuristic that counts nodes, function calls, and particularly loops.

2.2.2 Discovering Split relations

Discovering **Split** rules, however, is more involved. A **Split** rule centers on a split relation, which relates a sequence of visits to some binary node to a sequence of visits to a C counterpart. The path-conditions must be related, and the variables will be related by a series of equalities. We discover split relations through a model-guided validation process. Suppose we have a binary loop for which we need to find a split. The first step is to find candidate split relations which match at their first three subsequence visits.

Suppose we were looking for a split relation which matches the entire sequence of visits to a pair of loop nodes. We could test the first three visits in any such relation with

reference to the variable values in the first three iterations of the relevant loops.[3] To do this, we expand the graph problem into an SMT problem which includes representations of the path conditions and variables at all visits to nodes of the loop up to and including the third iteration.

We can think of this finite SMT problem as looking into the (potentially infinite) loop behaviour through a "window" with a width of 3. Using a minimal SMT window gives the fastest SMT solver performance. For some loops with large complex bodies, using small window sizes is necessary to avoid solver failures and timeouts. If the discovery process fails, we can expand this window.

To match a binary node, we must explain the values of all relevant variables (registers) through split equations. A simple static analysis pass helps us here, by discovering which variables are irrelevant, which variables are constant through the loop, and which variables are changed only by some constant increment resulting in a linear series. The constant and linear variables (registers) can be explained by relating their n-th valuations to their initial valuations. All remaining variables (registers) must be explained by relating them to a matching C value.

We repeatedly pick a possible split relation. The first such relation will be an arbitrary choice of a binary and C loop node, in which all relevant binary variables (registers) are equated with any available C variable of related type. It is highly unlikely this will be a valid split relation. We request an SMT model in which the binary loop is entered, but the split relation fails to hold at the first three instances, either because they are exited during different iterations or because the equations fail to hold.

The solver either responds with the assertion that these constraints are unsatisfiable, and we have found a split relation that holds for three steps, or the solver responds with a satisfying model. These models are crucial, because they narrow the search substantially, eliminating many more path-condition equalities and variable equalities than just the ones we have explicitly tested. We can then continue with a new potentially valid split relation, informed by the data we have seen.

This process continues until either a split relation is found, or we run out of split candidates. If a split relation is found, we test its inductive step. This requires building a new SMT problem, and is far more expensive than the other tests, which is why we only attempt it once we are confident that the relation holds for a few steps.

If we run out of split candidates, we can include more by expanding the SMT representation window. The initial bound we picked was 3. Increasing this bound allows us to test more split relations which match on more complex subsequences. With 4 iterations present, we can test split relations with a starting offset, whose subsequences begin at the second visit. From 5 iterations, we can test split relations involving a step width of two, that is their subsequences include every other visit on one side or both. Increasing the SMT problem sizes can substantially increase the solving time, which is why we start with smaller and simpler queries. We continue expanding until a split is found, up to a maximum window size of 8, after which we declare failure.

Before we expand the window, we try one other strategy. If there are multiple possible paths into the loop, we pick the most common one in the models we have found. We then rerun the search for this window size, under the assumption that this entry path is taken. We can reuse the models where this path is in fact taken. If this restricted search results in a split relation, it is sufficient evidence that we need to perform a **CaseSplit** on the

[3] If the loop contains internal conditional structure, there is a difference between the "third visit" to a node and the "potential visit in the third iteration". We mean the latter. The difference is irrelevant, however, since we will only ever try to split a loop at nodes that are unconditionally visited in each iteration of the loop.

possible entry paths before discovering the relevant **Split** rules.

This search process is novel to SydTV, and had no equivalent in the former prototype.

2.2.3 More Complex Unmatched Loops

We have noted that a fully expanded C loop can be easily detected if there is no loop in the binary. A more complex case involves a C function with two loops in sequence, and a binary function with one loop. It is clear that one of these two loops has been expanded, but which one?

The split discovery process needs to represent a "window" of the first few iterations of a loop to do its job. Suppose in the above example that it was the first of the sequential C loops that was expanded and the second one that matches the binary loop. The split process cannot represent iterations of the second loop until the first loop has been restricted. This would lead the split process to fail.

More generally, in a case where both functions have several loops, it might not even be obvious that any of the C functions had been expanded. Nonetheless, it is important to detect any expansion before attempting the split discovery process described in Section 2.2.2.

To avoid split failures, SydTV-GL-refine applies two tests to the set of loops that can be represented, before attempting split discovery.

The first test is to check that each of these loops can iterate at least eight times. The path condition to the start of the eighth loop iteration is tested, and, if unsatisfiable, it must be possible to handle this loop by expanding with a **Restr** rule rather than by discovering a split. This test may report false positives, loops with small bounds that nonetheless have matching counterparts. However, the matching loop will also have a small bound, which should be discovered, resulting in a correct but possibly inefficient proof.

The number eight in the first test was arrived at by trial and error. We don't want to test higher numbers, because of the risk of highly inefficient proofs. Setting the test any lower has lead to complex search failures.

The second test is whether any of the loops is independent. A loop is considered independent if it is possible to iterate through that loop for eight iterations without visiting any potentially matching loop for more than one iteration. If a loop is independent in this sense, then we can assume that no split relation will hold for that loop. We can then search for a loop bound, possibly much higher than eight. This test is quite effective in distinguishing expanded C loops from those with matching binary bodies.

2.3 Binary-Specific Features of SydTV-GL-refine

SydTV-GL-refine is mostly a generic framework for proving refinement relations between functions expressed in SydTV-GL, with significant support for handling loops. In priciple, SydTV-GL is a uniform language which represents C and binary programs in the same syntax, with the tools that generate SydTV-GL representations responsible for eliminating C-specific and binary-specific features.

In practice, SydTV-GL-refine contains a number of features that are specific to the C and binary sides of the problem. We have already discussed the pointer validity operators that are unique to the C representation. Far more of the implementation of SydTV-GL-refine is dedicated specifically to reasoning about the binary, and nearly all of the binary-specific logic concerns the stack.

The stack is a region of memory used by the compiler to store temporary values. Each function allocates itself a region called a stack frame for this purpose. On the ARM architecture, the stack grows downwards, with new stack frames allocated at lower addresses than the parent stack frames. The stack is under compiler control, with the layout and contents of each stack frame unknown to SydTV-GL-refine.

The stack causes difficulties because it needs to be seen both as a region of memory and as a collection of variables. When there are more C variables in scope that can be kept in machine registers, some variables must be saved to the stack and retreived later. It would be attractive to treat these save slots as variables themselves, much like the C variables and the machine registers appear in SydTV-GL. Such a treatment would allow the standard static analysis of SydTV-GL-refine to track values as they moved between registers and stack slots, for instance.

Unfortunately this clean abstraction of the stack is *fragile*, and can break down entirely, for instance when arrays in the stack frame are accessed at unknown offsets. Instead, we treat the stack principally as a region of memory, accessed with the real stack addresses. This approach is *robust*, but requires dedicated mechanisms for reasoning about the in-memory stack representation at various places in SydTV-GL-refine.

The following subsections address various aspects of this issue.

All the stack handling of SydTV-GL-refine is new, with the former prototype depending on the decompiler to abstract away the stack in the fragile manner described above.

2.3.1 Binary Functions in SydTV-GL

SydTV includes a variant of Myreen's decompilation approach [MGS08, MGS12], which runs in the HOL4 logic environment to produce a SydTV-GL representation of the binary semantics. This decompiler variant was produced by Myreen to support SydTV. This decompiler is actually simpler than previous incarnations of the concept, because it tries to do less abstraction, particularly with regard to the stack. The lack of complex abstractions also makes it the most robust decompiler in the family.

Unlike previous decompiler instances, which produced output functions with similar signatures to the matching C functions, the SydTV decompiler produces functions which all share the same signature. All functions take as arguments all the CPU registers `r0`, `r1` etc, in addition to main memory, the stack, and some extra special-purpose objects. The stack is represented as a complete 32-bit memory, and accessed with the same addresses as actually used in the binary.

The decompiler does detect which memory accesses are stack accesses and which ones are heap accesses by tracking information flow from the stack pointer register carefully. This allows the decompiler to produce output functions which manipulate two different memories, the stack and the heap.

The SydTV decompiler produces output functions natively in SydTV-GL. Supporting the SydTV-GL format natively simplifies our proof chain and our claim of its correctness, and is largely self-explanatory.

The reason that the decompiled functions all have the same signature is to avoid issues with fragile abstractions. The calling convention specifies that each function must return some registers unchanged. It would be nice to avoid explicitly returning these registers in SydTV-GL, and instead leave the registers as they were at each call site. Unfortunately, to make this simplification, the decompiler would have to *prove* that the registers are not changed. Since the callee function typically preserves the registers by saving them to the

stack and then retreiving them, this would require the decompiler to fully analyse all stack accesses, a problem which is as fragile as fully abstracting the stack.

The decompiled functions also faithfully follow the control flow graph of the binary in their SydTV-GL representation. The binary functions actually use the same instruction addresses[4] as the function bodies in the binary.

2.3.2 Handling the Stack in SydTV-GL-refine

The binary SydTV-GL functions processed by SydTV-GL-refine represent the stack the same way as the rest of memory. Like memory, the entire stack is passed to each function as an argument and returned to the parent as a return value. If a function stores values to its stack frame so that it can call another function, and then fetches them again, the values are fetched from the returned stack object, but we need to be able to prove that the values have not changed.

The stack grows downwards in the ARM architecture, with the current end of the stack stored in the register `sp/r13` at all times. The architecture mandates that `r13` must be decremented before new stack slots can be used. We need to prove, in SydTV-GL-refine, that each function preserves the stack frames of its parents. This can be defined as an output relation:

$$\forall x \geq \texttt{r13}_{before}.\ stack_{after}[x] = stack_{before}[x]$$

This simple specification needs to be slightly adjusted by details from the calling convention. If there are too many arguments to fit in registers, the remaining arguments are pushed onto the stack by the calling function, but are considered part of the callee's stack frame and may be modified. If the return value is too large to fit in the (single) return register, instead a pointer to a return address is passed as an extra argument, and the addresses it points to will be on the stack but may be modified. Our C subset does not permit taking the addresses of local variables, so apart from these cases a function cannot manipulate pointers into the stack.

These details can be addressed. The more serious problem is that this constraint cannot be encoded into the quantifier free SMT logic QF_ABV which we are using. There also exist SMT logics extended with quantifiers, but there are good reasons we avoid using them. Logics with quantifiers are supported by far fewer solver implementations, and typically see far worse performance. Instead we wish to encode this stack preservation property directly into the well-supported SMT logic QF_ABV.

Lemmas on Demand

To explain how our solution works, we have to recap a little. The QF_ABV logic is already in some sense a hybrid, a quantifier free logic of bitvectors extended with the implicitly quantified theory of extensional arrays. The extensional property of arrays would be expressed in other logics by quantification:

$$\forall a\ b.\ a = b \iff (\forall x.\ a[x] = b[x])$$

[4] Some binary instructions require multiple SydTV-GL nodes to represent, in which case the first SydTV-GL node will have the binary instruction's address and the others will have small odd-numbered addresses which cannot be addresses of binary instructions.

Instead of introducing this quantified formula, the solver simulates it via two different mechanisms. Firstly, to make use of array equalities that are believed to be true, the solver instantiates this weaker theorem on demand:

$$\forall a\ b\ x\ y.\ a = b \longrightarrow x = y \longrightarrow a[x] = b[y]$$

Secondly, to prove array equalities, or equivalently, to handle the case where array equalities are believed to be false, the solver introduces a *witness value*. The witness value $w_{a,b}$ for the equality $a = b$ is some value where the arrays differ, if such a value exists. This adds the additional theorem:

$$a = b \iff a[w_{a,b}] = b[w_{a,b}]$$

We say above that the solver instantiates a theorem on demand. The solver could simply instantiate this theorem for any four variables a, b, x, y whenever $a[x]$ and $b[y]$ appear in the problem. This would produce the correct results, however, it would result in a quadratic expansion in the size of the problem within the SMT solver.

Instead, these theorems are introduced only when intermediate states of the solver suggest that $a = b$ and $x = y$ are likely. This approach is called the system of lemmas on demand [BB08] and is known to be crucially important to SMT performance on the theory of arrays. A similar strategy can be applied to the related SMT theory of uninterpreted functions.

The Theory of Stack Equalities

As a first step, we can introduce a stack equality operator stackeq(sp_a, st_a, sp_b, st_b) into SydTV-GL. This represents equality between two stack representations a and b, each including a stack pointer and a stack memory. We want to define stack equality to mean that the stack pointers are equal and all addresses above the stack pointers are equal also:

$$\text{stackeq}(sp_a, st_a, sp_b, st_b) \iff sp_a = sp_b \wedge (\forall x \geq sp_a . st_a[x] = st_b[x])$$

We cannot supply this definition in QF_ABV. Instead we implement within SydTV-GL-refine an SMT solver for the theory QF_ABV extended with stack equalities, a logic we will call QF_ABVSt. We implement the theory of stack equalities by reducing problems which contain stackeq operators to problems in pure QF_ABV. This is done in addition to other theory extensions, such as pointer validity, which we discussed in Section 2.1.12. We will call the starting problem, including the stackeq operators, the "QF_ABVSt problem", and the resulting problem the "QF_ABV problem".

For each stackeq operator application in the QF_ABVSt problem, we define a fresh boolean unknown in the QF_ABV problem, named $stackeq_1$, $stackeq_2$, etc. We also introduce a witness value, $stackeqw_1$, $stackeqw_2$, etc, which we use to mimic the witness strategy for array equalities.

The witness value is defined to be some value *above the stack pointers* where the stacks differ, if such a value exists, or otherwise any value above the stack pointers. The stack equality witness has the analagous characterising theorem to the memory equality witness, that the stacks will be equal if they are equal at the witness value:

$$\begin{aligned}\text{stackeq}(sp_a, st_a, sp_b, st_b) \iff\ & sp_a = sp_b \\ & \wedge\ st_a[w_{sp_a,st_a,sp_b,st_b}] = st_b[w_{sp_a,st_a,sp_b,st_b}]\end{aligned}$$

This fact reduces to a simpler representation in QF_ABV:

$$stackeq_1 \iff sp_a = sp_b \wedge st_a[stackeqw_1] = st_b[stackeqw_2]$$

We also know unconditionally that the witness value is above the stack pointers:

$$w_{sp_a,st_a,sp_b,st_b} \geq sp_a \wedge w_{sp_a,st_a,sp_b,st_b} \geq sp_b$$

This witness mechanism gives the solver exactly the information it needs to prove stack equalities.

Half Arrays

We also need to instantiate this theorem for making use of stack equalities:

$$\forall sp_a\ st_a\ sp_b\ st_b\ x.\ \text{stackeq}(sp_a, st_a, sp_b, st_b) \longrightarrow x \geq sp_a \longrightarrow st_a[x] = st_b[x]$$

We have thought of two ways to make use of the above theorem, and implemented one of them. The naïve solution, which we did not pursue, is to instantiate the theorem exhaustively. For every value x which is used to address an array value st_a which also appears in a stack equality, we would assert the fact:

$$\text{stackeq}(sp_a, st_a, sp_b, st_b) \longrightarrow x \geq sp_a \longrightarrow st_a[x] = st_b[x]$$

This would be logically sufficient, however, it results in a quadratic expansion in the size of the QF_ABV problem with respect to the size of the QF_ABVSt problem. We strongly suspect that it will degrade SMT solving time, since the solver will have to adjust models repeatedly to ensure they satisfy many pointless equalities.

We would prefer to approximate the "lemmas on demand" approach. But this requires the SMT solver to decide when to instantiate equalities. The solver will instantiate array equalities on demand, but we want to specify equality only on half memories. The solution we have adopted is to "cut" some stack variables in half, so that we can assert equality only on the relevant half.

We examine the QF_ABVSt problem and look for array-typed unknowns which appear on the right hand side of stackeq predicates. For each of these unknowns in the QF_ABVSt problem we introduce two array-typed unknowns in the QF_ABVSt problem, a "top" memory and a "bottom" memory. To convert expressions to QF_ABV, we initially replace all instances of the original unknown with a four-way tuple, which consists of the top memory, the bottom memory, a division point, and a sequence number. The division point will be the stack pointer value from the right hand side of the stackeq predicate.

This results in a syntactically invalid SMT problem. We now adjust each expression that can enclose such a tuple $(top_a, bot_a, div_a, seq_a)$. Loads from this split memory become SMT if-then-else operations, loading from top_a if the target address x satisfies $x \geq div_a$, and from bot_a otherwise. Stores to split memories are adjusted into new tuples, with the same division point and two if-then-else operations, the top memory being updated only if $x \geq div_a$, etc.

Where tuples appear on a side of an if-then-else operator, because paths converge in the binary graph, the result is itself a tuple. The top, bottom and division components are produced as if-then-else operators in the same way. The sequence number is the maximum

of the two argument sequence numbers. If an if-then-else operator combines a regular memory with a tuple, we consider the top and bottom components of the regular memory to be the memory itself, the sequence number to be zero, and pick the same division point to reduce syntax.

Since the right hand side of the stack equality stackeq(sp_a, st_a, sp_b, st_b) is adjusted with st_b becoming a split memory $(top_b, bot_b, div_b, seq_b)$, we can provide useful information about this split equality in the QF_ABV problem:

$$stackeq_1 \longrightarrow sp_a = sp_b \longrightarrow sp_a = div_a \longrightarrow sp_b = div_b \longrightarrow top_a = top_b$$

The SMT solver can then produce the consequences of the equality $top_a = top_b$ on demand. These equalities should only be relevant when st_b was accessed above st_b in the QF_ABVSt problem, since otherwise the access will interact with bot_b rather than top_b.

For these equations to be sufficient, it must be possible to divide the right hand sides of all stackeq predicates in this way. It happens that stackeq predicates will nearly always apply to otherwise unknown stack values, including the stacks returned from function calls and introduced in induction proofs after an unknown number of loop iterations. There is an exception involving k-induction for $k > 1$, because a stack equality is assumed at multiple iterations in proving the inductive case. It happens to be the case that the extra stack equalities are redundant (but not the other assumptions of the k-induction) and can be discarded.

It is also required that no array appears in both stack equalities and regular equalities. We are fortunate that this is the case. The decompiler divides memory accesses between the heap and the stack ahead of time. We always use regular array equality to reason about the heap and stack equality to reason about the stack.

This approach expands the QF_ABV problem by a constant factor compared to the QF_ABVSt problem. Each operator site in the QF_ABVSt problem can result in at most one amendment, which adds a constant amount of syntax. The additional if-then-else operators are expected to mostly be straightforward for the SMT solver, since most stack addresses are at constant offsets to each other, making the condition trivially true or false.

Correctness of the Half Array Model

This approach of dividing stack memories is intuitively correct, because the construction ensures that the bottom part of each "top" memory is never accessed in normal expressions. Thus it should be OK to specify that the whole top memory is equal to some other value, even though that statement is logically stronger.

Let us sketch a *proof* that indeed the QF_ABV problem is equisatisfiable to the QF_ABVSt problem. We can prove equisatisfiability by showing a mapping that converts a satisfying model of each problem into a satisfying model of the other. We will see in the sketch proof that this conversion doesn't always work, and that we must impose some constraints on the representation. The constraints are:

- All stack equalities in the QF_ABVSt problem have a single SMT unknown rather than a compound expression as their right hand side memory. These unknowns are converted into split representations in the QF_ABV problem.

- Each SMT unknown appears in the right hand side of at most one stack equality.

- If the left hand side of a stack equality is split in the process of converting to the QF_ABV problem, then the resulting sequence number is strictly lower than the sequence number on the right hand side.
- The division point of the split representation of the right-hand side of a stack equality must equal the relevant stack pointer.
- Normal equalities involving split representations are not possible.

To convert a model of the QF_ABV problem back into a model of the QF_ABVSt problem, we must merge each of the top and bottom memory pairs back into a single memory. We do so in such a way as the load operations produce the same result, that is, the merged memory has contents from the top memory at addresses above the division point, and contents from the bottom memory at addresses below the division point. The load, store and if-then-else operations can be confirmed to have the same effect before and after the merging.

The stack equalities that were false in the QF_ABV model were false at their witness addresses. We can use these witness addresses to instantiate the quantified definition of stack equality that exists in the QF_ABVSt problem. Performing this instantiation produces evidence the stack equalities were also false in the QF_ABVSt model.

The stack equalities that were true in the QF_ABV model are also true in the QF_ABVSt model. We assume that the stack equalities relate a tuple representation on the right hand side to either a plain memory on the left hand side or a tuple representation with the same division point. All addresses relevant to the stack equality will be accessed from the top component of the right hand side and the top or whole component of the left hand side. These components are equal, thus the stack equality holds.

To convert QF_ABVSt models to QF_ABV models, we must invent data. Some arrays of the QF_ABVSt model must be converted to a pair of arrays in the QF_ABV model. The bottom such memory is straightforward to build, since it only appears in the QF_ABV problem in branches of if-then-else expressions, where the branch ensures that only the relevant contents matter. We set the relevant contents to be the same as the QF_ABVSt memory, and the irrelevant contents to zero.

The top memories are more interesting. They appear in exactly one equality expression in the QF_ABV problem, the one which corresponds to the stack equality they appear in in the QF_ABVSt problem. We set their relevant contents to be the same as the QF_ABVSt memory, ensuring that all explicit accesses get the matching result. If the QF_ABVSt stack equality was false, then we can set the irrelevant contents of the top memory to zero. The stack equality must be false at some relevant address. We use that address as the witness value. The array equality is false for the same reason, although this isn't even strictly relevant.

For all the top memories whose stack equalities are true, we must set their irrelevant contents to equal the matching contents from the left hand side. The top (or whole) component of the left hand side will be either another such unknown with a lower sequence number, or an expression built on top of one via if-then-else or array stores. Note we can already evaluate all the if-then-else conditions, since they evaluate in the both models without needing the irrelevant data, and the store operations can't get to the irrelevant data either. We pick the irrelevant contents in sequence order, to clarify that we can always pick concrete values without problems of cycles or solving simultaneous equations.[5]

[5]The sequence numbers are probably not necessary, and can be replaced with a more complex argument

The constraints we have given are thus sufficient to ensure that the QF_ABV and QF_ABVSt problems are equivalent. Most of these constraints are syntactic, and are checked during the SMT export process. The exporter stores, for instance, a table of stack equalities indexed by the top memory on the right hand side, so that it can check that each top variable only appears on one right hand side.

The constraint that cannot be easily checked is that the division points are equal to the stack pointers. The right hand side division point is picked accordingly, but the equality on the left hand side is a semantic truth, not a syntactic one. Instead of trying to establish this at export time, we make it a premise of theorem that establishes the array equality (it is already written that way in the most recent theorem above). This extra premise means that the QF_ABV and QF_ABVSt problem are not entirely equisatisfiable. If we have to divide sequential stacks at different split points, we may lose information.

Impact of Constraints

The constraints were picked because they are easy to satisfy in almost every case. Since we do not allow dynamic stack allocation (via the problematic `alloca` function or variable-size local arrays), the compiler knows the maximum stack frame size for each function. To save on instructions and cycles, the compiler always adjusts the stack pointer to reserve the whole stack frame at the start of each function and to restore it at the end. We have never seen an instance of the compiler trying to save on stack consumption by shrinking its stack frame before making a function call, although it is conceivably worthwhile in some circumstances. This means that the current stack pointer (ARM requires `r13` to always contain this value) is unchanged in the body of the function and always the correct division point. The sequence numbers simply count monotonically the number of unknowns produced by the SMT export process.

The way the stack is used is sketched in Figure 2.3. A binary function `f` initially creates a stack frame, and slowly populates it with data. When function `g` is called, the newly returned stack is related to the previous one via the stack equality predicate. Return values may also be known to correspond to those from C. Otherwise nothing is known about this new value. Likewise after an unknown n number of loop iterations, a new stack variable is introduced. The stack equality predicate is used to establish that the majority of the stack is preserved, and this is specifically downgraded at stack slots that may be updated during the loop.

We treat some stack equalities specially to ensure the constraints will be met. For instance, a typical proof obligation is that the function we are analysing returns the stack to its parent with the parent's stack frame unchanged. We need to prove this stack equality with the witness equation from before, but we do not need to characterise the implications of this equality. We do not produce the QF_ABV equation which specifies the consequences of this equality, and we do not count this equality towards the restriction that each unknown appears on the right hand side of only one equality.

In the case of k-induction where $k > 1$, we begin the inductive proof with k instances of the induction hypotheses. This might include k different stack equalities, which might have the same stack on the right hand side, breaking the syntactic constraints. We solve this problem by discarding all but the first such equality entirely. This is done before conversion of the QF_ABVSt problem to QF_ABV. Discarding these assumptions is always sound but not always equisatisfiable, however, this has never been a limitation in practice.

about the graph of dependencies of these unknowns, which have only one outbound edge once if-then-else operators are resolved, leading to either equality cycles or paths to known data.

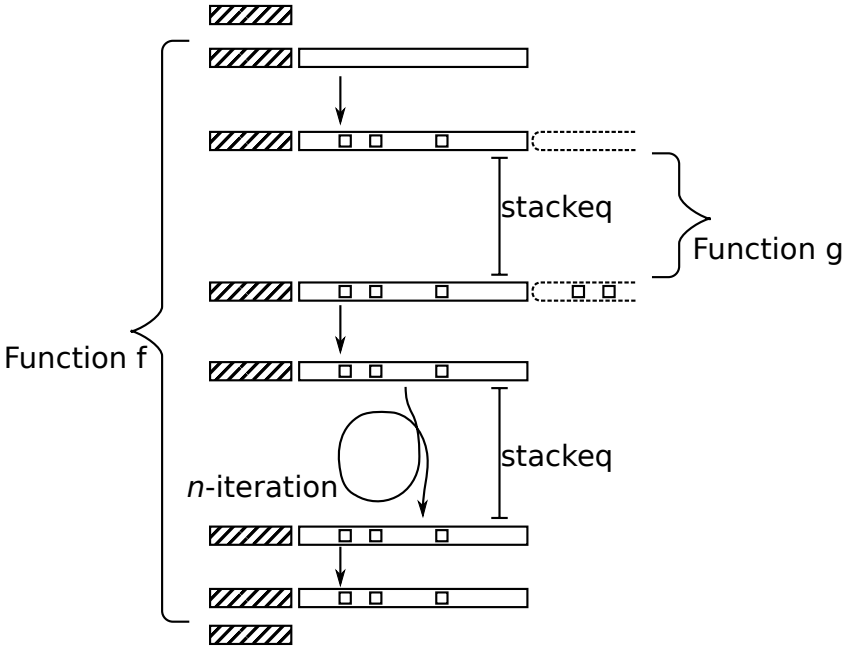


Figure 2.3: Example stack use of function `f`.

The most serious downside of this stack equality approach is that it limits inlining. In our former prototype, when the compiler invented a function with no C counterpart, we could often handle that case by just inlining its semantics into its call sites. In SydTV-GL-refine, however, this breaks our assumption about the stack frame being uniform. In the composed problem with the inlining performed, the binary program moves its stack pointer up and down more than once. If another function call occurs within the body of the function that was inlined, the returned stack will need a different division point to other stacks within the resulting problem. Our problems with functions introduced by the compiler are discussed further in Section 2.5.7.

2.3.3 Stack Use Bounds

Before comparing any functions, SydTV-GL-refine first computes maximum bounds on stack use for all functions. This is not a feature we originally intended to implement, but it is more or less required.

The stack equality predicate above tells us that functions preserve the contents of the stack for addresses $\{x.\ x \geq \mathtt{sp}\}$. A function `f` will typically decrement the stack pointer `sp` to make space for its stack frame before calling other functions `g`, `h`.

We encountered a number of simple SMT counterexamples where the initial stack pointer `sp` is very close to zero, and thus `f`'s stack frame wraps around to the end of the address space. When `g` is called, it preserves addresses above the new value of `sp`, which does not cover all of `f`'s stack frame.

We must assume that the starting stack pointer is not so absurdly small. However, if `g` assumes that $\mathtt{sp} \geq n$ for some n, then `f` must assume that $\mathtt{sp} \geq n + sz_{\mathtt{f}}$, where $sz_{\mathtt{f}}$ is the size of `f`'s stack frame.

To avoid these simple counterexamples, we most solve a substantial problem. The minimum stack bounds, e.g. n for `g`, must be a conservative bound on the possible stack use of each function and its callees.

We could try to avoid the issue by picking a very approximate bound, for example 1 MiB times the potential function call depth. Computing function call depth is trivial unless recursion is present. Avoiding the issue would be reasonable for many programs, for which stack use bounds are not a serious concern. For example, a program running in a 64-bit address space with virtual memory can easily allocate stack space so large that only unbounded recursion would exhaust it.

The program of interest in our case study, seL4, strongly motivates us to take this issue seriously and compute tight bounds on stack consumption. Firstly, seL4 allocates its stack conservatively to avoid wasting physical memory, allocating 64 KiB per kernel stack on most platforms. Secondly, seL4 makes use of some recursion. Limiting recursion is complex in any case, and the maximum bound on stack consumption is an important correctness concern, so we decided that it was important to compute accurate stack bounds.

We assume that `sp` is greater than the stack bound in the input relation for each function pair (see Section 2.1.4). This means that the stack bound calculation must be performed by SydTV-GL-refine before any other proof steps. Our initial implementation used SMT models to calculate the expected stack consumption between the entry pointer of a binary function and the call sites within it. However, since stack bounds must be computed before doing anything else, we switched to a faster implementation in which SydTV-GL-refine performs simple static analysis to determine the constant offsets added to the stack pointer.

Discovering Recursion Bounds

We also implement a simple calculation for discovering recursion bounds, which works for small bounds. Our design was motivated by the two simple instances of recursion that exist in seL4.[6] Both cases involve a cleanup routine with two modes of operation, distinguished by a boolean argument, which can (indirectly) recurse to themselves. The recursion can be broken by considering each cleanup routine to be two different functions depending on the boolean argument. There has been a long-running discussion amongst the seL4 developers as to whether these functions should be split in the source code, which would simplify this kind of automatic analysis of the system but lead to code duplication in both the source and binary.

We detect this style of limited recursion. We can imagine an approach to this problem would be to take a recursive function f and inline its recursion sites. The general instance of f can only recurse into the special case with the given boolean argument. Inside the inlined special case, the call to f should now be semantically unreachable. If we keep inlining at recursion sites whose path conditions are satisfiable, we should eventually reach this limit.

For technical reasons, we don't actually use the inlining mechanism. Instead we create a problem space (a mutable space containing function bodies which we introduced in Section 2.1.5). Instead of inlining a new copy of f at a call site, we add a copy of f as an additional function in the problem space, and assert equalities to link the variables passed into this function from the call site and returned out. Again, we repeatedly add these functions until the joint path condition needed to recurse again becomes unsatisfiable.

Once all recursion sites are unreachable, we know the recursion bound. We need to characterise it, however. We can ask the SMT solver for a satisfying model in which the maximum amount of recursion occurs. In our example, this will require the boolean

[6] Specifically, in the C implementation of seL4. The Haskell prototype and Isabelle/HOL specifications are functional programs, and use recursion extensively.

parameter (e.g. y) to be set one way in the outer copy of f (e.g. $y = 1$) and another way in the inner copy (e.g. $y = 0$). We can check in SMT that $y = 0$ is necessary in the inner copy of f for recursion to occur. This means we can characterise the inner copy by the property $y = 0$.

We can generalise this approach directly to mutual recursion. We can also handle small enumerations, for instance by treating $y = 0$, $y = 1$ and $y = 2$ all as special cases of f.

We will refer to these distinguishing equalities, such as $y = 0$, as *recursion identifiers*. We can use recursion identifiers for f to treat $f_{(y=0)}$ and $f_{(y\neq 0)}$ as different functions. When we recompute the call graph with this distinction, it will become acyclic.

We can compute stack use bounds for all functions in the new acyclic call graph. Finally we compute a stack bound for f using an if-then-else expression which depends on whether $y = 0$.

For the moment, we only detect recursion identifiers which are equality expressions between a function call argument and a constant. We could generalise this approach to a far more powerful one by detecting other kinds of identifying expressions. The recursion identifiers are really Craig interpolants [Cra57, L$^+$59] of the unsatisfiability proof which shows that further recursion was impossible. We could discover more general identifying expressions by installing an interpolant-generating SMT solver such as SMTInterpol [CHN12] or iZ3 [McM11].

The above approach of using Craig interpolants to summarise the meaning of functions has been used extensively in the field of model checking [McM05,SFS11,CMB14]. A similar approach has also been applied directly in the Reve project [KKU16], which produces proofs of equivalence between different implementations of C programs.

Another possible improvement would be to discover numeric recursion identifiers as well as booleans. This would be far more efficient in simple cases where, say, a recursive argument must decrease by 1 in each recursive call. This calculation about recursion is very similar to calculations that are done to establish loop bounds, for instance for WCET (worst-case execution time) analysis. We have previously experimented with computing loop bounds as an extension to SydTV-GL-refine (we will discuss this in Chapter 3), and could possibly reuse that mechanism here to compute larger recursion bounds efficiently.

We have discovered that the stack bound analysis is more reliable if SydTV-GL-refine initially computes these recursion identifiers on the C functions, and then converts them into binary recursion identifiers via the calling convention. This is partly for convenience, because of a case where a function has sufficiently many arguments that the boolean parameter of interest is itself initially stored on the stack. More generally, we use the C representation because some correctness information is present there which might be implicit, concealed by complexity, or missing entirely in the binary. This is one of the advantages of handling stack concerns in SydTV-GL-refine. The competing design where the decompiler abstracts the stack would require the decompiler to perform the same calculation without the C semantics as a guide, unless we created a more complex exchange of analyses between SydTV-GL-refine and the decompiler.

2.3.4 Static Analysis for Stack Slots

Loops in the binary must be handled by the split relation discovery process outlined in Section 2.2.2. This process depends on static analysis to divide variables into 4 groups:

1. variables out of scope

2. variables constant throughout a loop
3. variables modified by a constant offset, forming a linear series through loop iterations
4. remaining loop variables to be matched in the split relation

The usual static analysis can be applied to both the C and binary functions, and will compute which registers are in scope. However the stack is a single variable, which will simply be discovered to be in scope. We require more fine-grained information about the status of various stack slots.

In particular, when a loop includes a function call, it is likely that live variables of the loop will be saved to the stack frame at the function call site. This may include variables which form linear sequences, which we need to detect.

We extend the static analysis component of SydTV-GL-refine to allow some functions to request custom static analysis, and add a stack-aware variant of the loop analysis for handling our binary functions. This stack-aware variant reuses the stack pointer offset calculation needed to compute stack bounds (see Section 2.3.3). Instead of computing linear adjustments to the stack pointer itself, it computes the difference between various stack access addresses and the initial stack pointer. These are used to identify which stack slot is being used in each stack access. For functions with large return values, the function is also supplied with a pointer to a return region in its parent's stack frame, and some stack accesses are at offsets from the return pointer instead, which we also detect.

We can then adjust the function to produce a new representation, still expressed in SydTV-GL, with nodes manipulating stack slots as named variables. The new variables have generated names like "(StackOffset, -42)". The explicit accesses to the stack object disappear. The standard static analysis suite can then analyse this new representation to compute which registers and stack slots are in scope at any point, and how they are adjusted by loop iterations. Finally, we map this analysis back to memory accesses on the stack object.

We mentioned at the start of this section that it would be attractive to adjust the program into an abstraction where stack slots became regular variables. Unfortunately such an abstraction is *fragile*. However, we seem to have contradicted ourselves, having just introduced such an abstraction. The difference here is that we offer no *proof* that this abstracted representation captures the semantics of the concrete stack operations. We only need enough accuracy to extract a useful static analysis.

This stack slot abstraction is only run for functions which contain loops, and only needs to succeed for accesses that occur during loops. We can, and do, handle failure cases where the live loop slots are correctly calculated but later parts of the problem cannot be properly analysed.

The static analysis module is not a trusted component of SydTV-GL-refine. It supports the split discovery process, which is then checked by the proof script checker. We have encountered a number of cases where inaccurate analysis lead to later failures. This analysis is quite complex, and we suspect that further bugs remain which simply have not lead to failures yet.

2.4 Fine Tuning

The process of producing SydTV, starting with a prototype, took over 3 years and was a substantial software development effort. The major features and the design of SydTV have

been introduced already, however, a large fraction of the work was done in fine tuning.

By repeatedly attempting the refinement proof, and carefully examining failures and counterexamples, we tuned the implementation to ensure that our case study would succeed.

In this section we discuss a number of specific investigations that led to slight amendments to SydTV. This includes something of a rogues' gallery of especially problematic functions, and a toolkit of countermeasures.

2.4.1 String Comparison: Accelerating Split Discovery

A key optimisation in SydTV the linear sequence optimisation, which is best explained with an example. The `strncmp` string comparison function from seL4 is a good example. Standalone operating system binaries such as seL4 cannot depend on an external standard library, and so have their own implementation of standard functions such as `strncmp`. This implementation of `strncmp` is a straightforward byte-by-byte procedure. A more complex optimisation which tries to use a word-by-word comparison is possible, but only justified if long strings are expected.

```
int PURE
strncmp(const char* s1, const char* s2, int n)
{
    word_t i;
    int diff;

    for (i = 0; i < n; i++) {
        diff = ((unsigned char*)s1)[i] - ((unsigned char*)s2)[i];
        if (diff != 0 || s1[i] == '\0') {
            return diff;
        }
    }

    return 0;
}
```

While the original authors have chosen a straightforward implementation, knowing that the number of loop iterations will be low anyway, the compiler doesn't know this. The compiler (GCC 4.5.1 with optimisation setting `-O2`) produces quite a complex implementation with the aim of reducing the number of cycles spent testing the bound condition `i < n`. The approximate structure of the source and this implementation is shown in Figure 2.4. The many additional paths which go "around" the loop to the tail of this function exist because the compiler has produced special-case implementations of the first two loop iterations, including their loop exit conditions. The compiler also decides where to enter the loop based on the parity of the argument `n`.

We suspect that some of this complexity is unnecessary. We think that a simpler but equally efficient implementation exists, which initially checks the parity and positivity of `n`, but performs all other tests within the loop body. However, implementing compiler optimisations is a difficult business, and we have not investigated whether there is a good reason for the compiler to behave the way it does.

The proof we are seeking begins with a **CaseSplit** rule which picks between the two possible entry paths to the loop. This is roughly equivalent to a case division on the parity

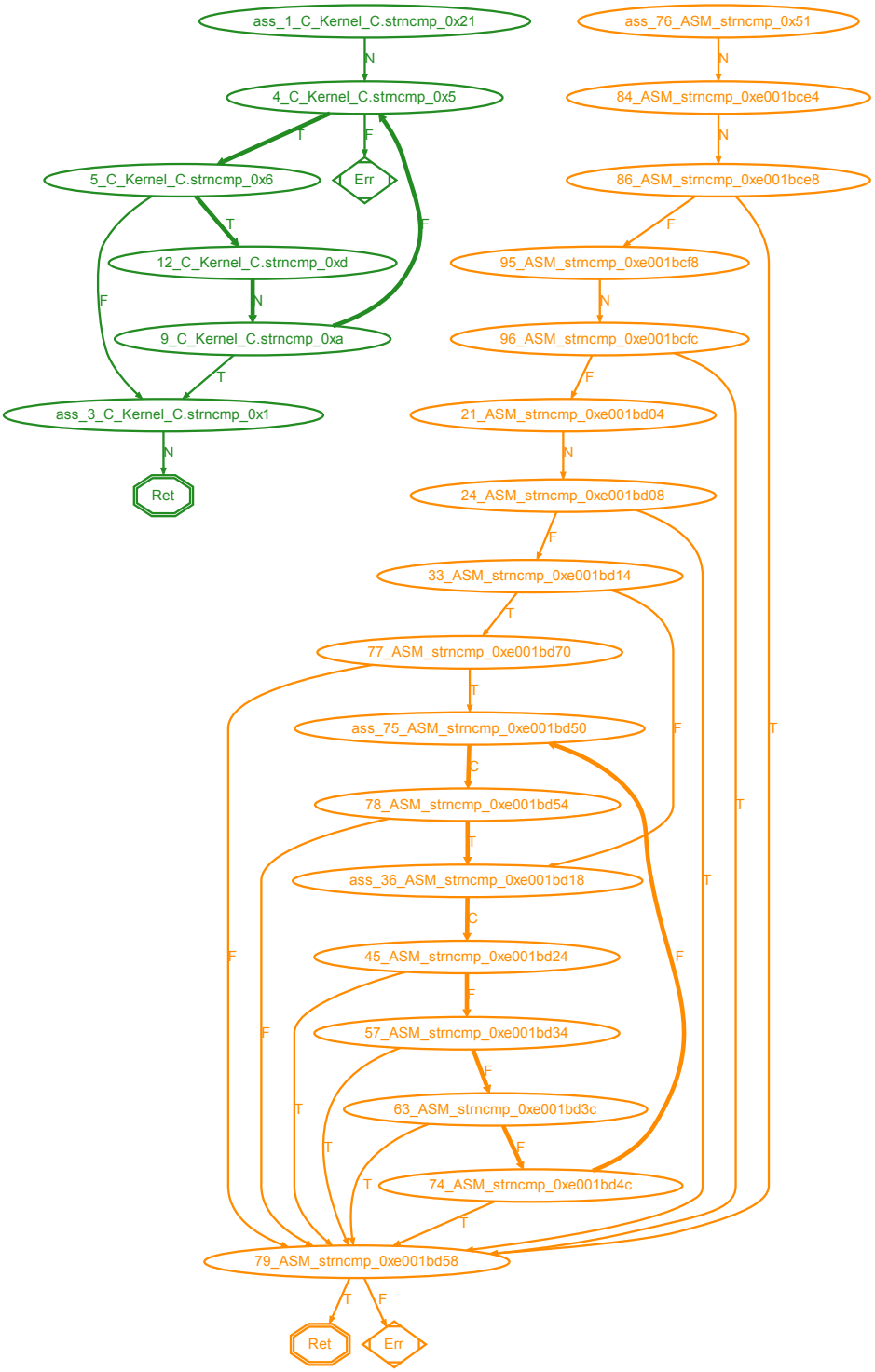


Figure 2.4: Simplified control-flow of source and implementation of `strncmp`.

of `n`. Each case then proceeds with a **Split** rule. In each case, the split relation matches the sequence of visits to the first address in the binary loop to a subsequence visits to the first statement of the body of the C loop. When `n` was even, the subsequence of visits includes the 2nd, 4th, 6th iteration etc. In the odd case, the match is to the 3rd, 5th, 7th iteration etc.

The challenge is to discover these split relations efficiently, using the split relation discovery process that was outlined in Section 2.2.2. We have highlighted the `strncmp` function because it is a pathological case for the discovery process as we first implemented it. The discovery process would run for several hours before failing due to an SMT solver timeout. In investigating this pathological case, we discovered three significant improvements which we introduce below.

2.4.2 Loops Without Variables

The key challenge of the `strncmp` loop is that, from the perspective of the split relation discovery process, it has no variables. The discovery process (outlined in Section 2.2.2) tries to match binary variables (registers) to C variables, but excludes linear series. The `strncmp` loop is a good illustration of why linear series are excluded. The C loop has a single variable `i` being adjusted. The compiled implementation instead allocates three registers to tracking the linear series `i`, `s1 + i` and `s2 + i`. The loop body contains arithmetic to increment each of these registers by 2.

It is not even necessary that the compiled binary contain any register which tracks `i`. Another valid implementation would track `s1 + i`, `s2 + i`, and the number of bytes remaining to be compared `n - i`.

In principle the discovery process could use some kind of linear algebra to discover that `r3 = s1 + i`. This is quite complex to implement. It is much easier to derive a characterising formula for `r3` by static analysis, discovering that $\texttt{r3}_i - 2i$ is constant where i is the iteration count. That is, the current value of `r3` equals its value i iterations previously plus $2i$. This information, together with information from the initial cycles which establishes that initially `r3 = s1`, will allow the inductive proof to succeed.

However, once we exclude these linear series variables, the `strncmp` loop doesn't appear to be computing anything. This is unusual. It is quite common for loops to involve linear counters, but usually there is some kind of data flow from them to other variables, or to the contents of memory, possibly via the arguments to function calls. This loop is computing nothing other than the first value of `i` which will cause the loop to exit.

The absence of any data variables starves the model-guided discovery process of information. Normally the satisfying model returned by the SMT solver will be queried for the values of some binary registers. It is unlikely that two different 32-bit values within an SMT model will be equal by coincidence.[7] A small number of satisfying models should exclude these coincidences and narrow the search to split relations that make some kind of sense. In the `strncmp` case, however, the only relevant information content of the SMT models is 1-bit information about which paths are reached. These variables are not even independent of each other, since the exit conditions of later iterations only matter if the loop was continued in the early iterations.

[7] It is possible that the solver has some default value, e.g. 0, which it tends to use in satisfying models. This does not appear to be the case for the solvers we are using. In addition, the queries we are making tend to require variables to be distinct. For instance, a satisfying model in which a linked-list following loop continues for two iterations and then exits will require distinct values for all the pointers in the list.

2.4.3 Partially Recovering Linear Sequences

The key insight which helped us fix the `strncmp` performance problem was that we should think about *data flow*. We mentioned data flow above in Section 2.4.2. The C version of the `strncmp` loop uses the linear counter `i` as a data source, with intermediate subexpressions `s1 + i`, `s2 + i`, `i < n` etc. Data flows through these to eventually be used in memory lookups and the loop exit conditions, which we can think of as data sinks.

The key observation is that the linear sequence data must flow somewhere (or else the compiler will remove the calculation entirely). The data can flow through intermediate linear calculations for some while. It might then flow on to non-linear calculations in registers, or to data sinks such as memory accesses and function call parameters. In our previously-pathological case with no non-linear calculations, this information *must* flow to data sinks. In general, memory accesses and function calls are more common than non-linear arithmetic, and linear values will often flow towards these data sinks.

The compiler has the option of rearranging the arithmetic involved in the intermediate calculations. However once the data sinks are reached, the values passed are likely to be exactly the same. For instance, in `strncmp`, the addresses of the memory accesses will match in the two loops. Not only will they match, but the nature of the match is revealing.

We can discover by static analysis the points in the loops where memory accesses are performed with addresses that form a linear sequence. There are two such accesses in the C loop `s1[i]` and `s2[i]`, at the same statement. There are four such accesses in the binary loop because of the unrolling. We pick one of these binary accesses, and ask the SMT solver for a satisfying model in which this loop point is reached.

The solver responds with a model in which, for instance, the accesses `s1[i]` and `s2[i]` are performed at addresses `0xbc001244` and `0xe0f34541` in the first C loop iteration. The binary access we have chosen to examine is performed at address `0xe0f34543` in the first instance. We also know that the C addresses increase by 1 in each iteration, and the binary address by 2. It is now clear exactly what kind of split relation we should be looking for. The binary loop iterations loosely correspond to every second C iteration, starting from the third one.

Clearly this kind of split relation causes the accesses to agree within this model. We check whether this match is *necessary*, by seeking a model where this first binary access is not equal to the third access `s2[i]`. This hypothesis is satisfiable, in the case where the alternative loop entry path is taken, and the binary iterations correspond to different C iterations. This suggests that a **CaseSplit** rule should be used.

We use this approach to discover **CaseSplit** rules and quickly narrow the search space for split relations down to ones which agree with the loop unrolling structure. This approach does not give us enough information to manufacture a split relation, as the loop points in which the data sinks appear may not be good choices as loop splits in any case. So we still need to discover a split relation using the usual algorithm, but we can skip over the usual process of slowly expanding the discovery window and excluding simple split relations until the relevant split relations are present.

This process, the linear sequence matching process, is attempted before any split discovery is tried. To clarify the process:

- Linear sequences in all relevant loops are calculated by static analysis. This includes some tricky cases, such as 16-bit linear sequences which are computed within 32-bit registers by repeated addition and truncation.

- Linear sub-expressions, i.e. expressions computed by applying certain kinds of

arithmetic to linear variables, are also computed.

- Linear sequences of interest are identified. These are linear sub-expressions which are passed into "data sinks". Data sinks include memory accesses (the addresses, and the updated values) and function call arguments.
- For each binary linear sequence of interest which has at least one matching C sequence of interest, we attempt to:
  - Request an SMT model where the data sink point is visited.
  - Match the value to an early value within one of the C sequences of interest, and the offset amount to a multiple of the relevant C offset amount.
  - Check whether this subsequence match must always be the case.
  - If this subsequence match is always the case, inform the split discovery process that it should focus on split relations which agree with this match.
  - If this match is not always the case, check whether it is always the case under the assumption that the loop entry path seen in the initial model is taken.
  - If the match holds under path assumptions, derive a **CaseSplit** rule which distinguishes between these cases.

2.4.4 Duplicate Splits

Another simpler optimisation was discovered in reflecting on the `strncmp` performance problem. This is to avoid testing "duplicate" split relations. Recall from Section 2.4.2 that a key challenge in `strncmp` is the small amount of information derived from SMT models which refute split relations. This is partly because the information content of the split relations themselves is low: normally split relations must match a list of variables, but here this list is empty.

For this reason a number of split pairings will generate the same SMT problem. In principle, we can save time by detecting this case and skipping duplicate considerations. However, if a satisfying model refutes one of these duplicated split relations, it will refute all of them. Likewise if a split relation is endorsed by testing, it doesn't matter which of the duplicates was picked. The problem is with split relations that are found to hold in their first three iterations, but which fail to hold when tested in their inductive step. When an inductive step fails to prove, we also record failures for all similar split relations that generate the same initial SMT problem, even if the inductive step problem was slightly different.

This adjustment actually makes no difference for `strncmp`, since no inductive steps fail to prove. It did, however, instantly solve a significant performance problem in a very similar function. Like `strncmp`, the other function does not have any non-linear variables. It scans a small array for the first available entry. The key difference which leads to the performance issue is that the size of the array is known. This means that the inequality exit condition, the equivalent of `i < n` in `strncmp`, cannot be true in the early iterations. The same issue could appear if `strncmp` was inlined into a context in which the value of `n` was known.

Since the exit condition is deactivated in the early iterations, and there are no non-linear variables, the split discovery process has literally zero data to work with, and must test every candidate split until it finds one for which induction works. Repeatedly building SMT problems to test broadly equivalent inductive steps can consume CPU hours.

2.4.5 Variable Width Memory Access

The C language, and the CPU architecture itself, can access memory at 8-bit, 16-bit and 32-bit widths. We map this into the SMTLIB2 array logic by picking one of these widths to be "normal" and emulating the others. When we developed the prototype that SydTV builds on, we found that a 32-bit focused encoding was the most efficient. Memory is represented as an array of words, i.e. an SMTLIB2 array with 30-bit indices and 32-bit values. The usual word-length reads and writes become array accesses and updates. Reads and writes of 16-bit and 8-bit values are encoded by reading or writing the containing 32-bit word and performing bitvector arithmetic as necessary.

Picking a good encoding is important because the performance of SMT solvers on problems extracted from realistic programs is known to be highly sensitive to the way the array theory is used. The extensionality theorem for arrays, which says that if $A = B$ and $x = y$ then $A[x] = B[y]$, can be naïvely instantiated for any pair of memory accesses $A[x]$ and $B[y]$ in the problem. This can lead to a quadratic increase in the underlying SMT/SAT problem size, which modern SMT solvers take great pains to avoid [BB08]. A 32-bit encoding leads to one SMT array access per memory access, whereas the more obvious 8-bit encoding would require 4 SMT array accesses per word-length access, and require the SMT solver to instantiate more array theorems.

We also broadly expect that a program running on a 32-bit machine will perform far more 32-bit memory accesses than at any other width. We also expect that even when 8-bit and 16-bit values appear in structures and stack frames, they do so within structures that are 32-bit aligned. Direct 8-bit and 16-bit accesses within aligned structures are equivalent to accesses of the word they are contained in, and indeed the compiler might adjust one into the other.

The counterexample to this logic is `strncmp`. We initially know nothing about the alignment of the values `s1` and `s2`. If we use a 32-bit encoding, the access to the first byte of `s1` is encoded as a case division between the four possible alignment cases of `s1`. At the inductive step, there are four alignment cases for `s1`, four for `s2`, and another four for the inductive value of `i`,[8] and the simple argument that the C and binary loops are equivalent becomes lost in a 64-way case division to establish what they are actually doing. This leads to long delays and SMT solver timeouts for otherwise simple problems. With an 8-bit memory representation, however, the relationship between different encodings of `strncmp` becomes clear and straightforward.

Fortunately the SMT export mechanism was designed so that the particular SMT memory representation could be replaced easily. This choice was initially made globally in the source code of SydTV-GL-refine, but with a few adjustments the choice can be made once per SMT problem. The remaining question is how to decide between representations. There might exist a reliable way to make this decision, but instead, we chose to build further on an existing experiment, allowing us to avoid making a decision at all.

Instead of trying to pick an optimal SMT solver implementation and memory representation, SydTV-GL-refine passes difficult SMT problems to a number of different solver processes running in parallel. SMT solvers are used by SydTV-GL-refine in both "incremental" and "offline" mode, the difference being that "incremental" solvers can perform multiple queries whereas "offline" solvers run one dedicated process per query. "Incremental" mode offers lower overheads when solving many simple problems, but is only supported by some SMT solvers, whereas "offline" mode is universal and offers better performance for difficult problems. SydTV-GL-refine seeks first seeks a solution

[8] Arguably there are only 2 cases here because of the unrolling.

in incremental mode, and after a short delay (configurable, but usually 2 seconds) runs a dedicated "offline" solver. Our previous prototype switched between solvers in the same way. In addition, however, SydTV-GL-refine can be configured to run multiple "offline" solvers at this point, taking the result from the fastest one.

Having parallel solvers allows us to run solvers with quite different implementations on the same problem, and exploit the advantages of both, at the cost of using up CPU time. We can also simultaneously test the same problem in both 8-bit and 32-bit memory representations. The two representations are syntactically different but logically equivalent. We need to take a little care in the way SydTV-GL-refine handles SMT problems to make sure we keep intermediate representations generic until we know which solver we are sending them to, but this is only a little work in practice. The implementation allows us to use parallel solvers with different representations for both sat/unsat queries and model extraction.

2.4.6 Overflows in Split Induction and k-Induction

The debug facility of SydTV can tabulate the memory activity that occurs in an SMT counterexample. This allowed us to quickly understand many failures. One interesting such failure involved an initialisation function for interrupt states in seL4. We will not go through its code here, the function walks a global array and zeroes values.

The proof search for this function failed in a pathological way. Many split candidates were discovered, and the inductive proof for each one failed. We pick one arbitrarily and investigate its counterexample. We see a clear linear sequence of array updates in the debug trace. The early iterations match, but at the inductive step, the memory write addresses have diverged. This suggests that something might be wrong with the linear sequence static analysis, however it is correct.

The problem is that interrupt identifiers are known to be small values on this particular ARM platform, and so a 16-bit integer `i` is used to represent them, and to iterate through them. The compiler uses a 32-bit register `r3` to store `i`. The compiler would typically increment `r3` and then zero its upper 16 bits to simulate a 16-bit increment of `i`. The usual static analysis for linear sequences includes cases which detect various ways that bitwise operations can be used to encode this calculation.

However, the size of the global array is known to be small enough that `i` never overflows. Thus the compiler can forget about simulating 16-bit arithmetic, and increment `r3` normally. The linear sequence analysis correctly detects the sequences in both variables, but they are a 16-bit and 32-bit sequence respectively. The inductive counterexample involves a case where more than 2^{16} iterations have occured.

We need to strengthen the statement which we prove by induction, to clarify that overflow of `i` is impossible. SydTV-GL-refine did detect a linear sequence match in the memory access addresses in this case, using the optimisation discussed in Section 2.4.3. We considered adding a facility to SydTV-GL-refine that addresses the specific case where a pair of matching linear sequences iterate on values of different widths. This turned out to be fairly involved, however, and targets a very narrow case.

Instead we reused a module which we had previously added to SydTV-GL-refine in work on estimating execution time. We discuss that work in Chapter 3, and also elsewhere [SKH16]. This module examines loop exit conditions and guesses some additional inequalities that might be provable by induction, then quickly tests the induction for each of these inequalities. Those inequalities aided our execution-time work in discovering loop bounds. In this case, one of the candidate inequalities is that `i` is less that the size of

the global array. This inequality indeed can be proven by induction, and it implies that the iteration count is less than 2^{16}, and thus solves the problem in the inductive proof.

We have adjusted the split discovery process to to load this additional module, discover these additional inequalities, and try to make use of them whenever a split induction proof fails.

This adjustment allows SydTV-GL-refine to discover a valid proof for this function.

It is interesting to ask why explicit loop bounds were not needed in other functions, given that the compiler knows how to compute them and use them in optimisations. The looping condition is typically encoded as `i < NUM` in the C code. We prove split relations using k-induction where k defaults to 2. The premises of the inductive case include two related visits to the respective split points. The path between these two previous split points includes a test of `i < NUM` which must have succeeded. This means that the proof process knows that the previous `i` satisfied `i < NUM` "for free".

The compiler frequently encodes `i < NUM` by equality tests `r3 != NUM` instead. Proving that this is safe requires discovering that the final iteration of the C loop must have `i = NUM`. We usually get this information "for free", but in this case the difference in precision loses the information again. We know that the bottom 16 bits of `r3` are equal to `NUM`, but lose information about the full value of the register.

The inequality module guesses from the fact that `r3 != NUM` is tested in a loop exit condition that `r3 < NUM` is a candidate invariant of the loop. With this invariant proven, the upper bits of `r3` are also known to be zero, and thus `i` and `r3` is recaptured.

2.4.7 The Sequential Fixed-Length Loop Problem

The last problematic function which we specifically investigated is part of seL4's boot sequence. It helps initialise a collection of things. Its source contains two loops of its own, with one more inlined from a call to `memzero`. The search process discovers a split for the first loop, and then fails to find a split at the next loop. We left this problem until last, because the failure manifests only after a complete split search failure, a slow process.

When we examine the debug memory trace for one of the models generated during the failed split discovery process, we can clearly see that the second C loop does not match any binary loop. Instead, a linear pattern of 16 memory accesses are performed explicitly at different binary instructions.

The compiler always has this option of fully unpacking loops with small well-known bounds. For example, a `for` loop counting from 0 to 5 might be replaced with 6 copies of its body. SydTV-GL-refine detects bounds of 8 or lower directly, and can also detect "independent" loops, which can be iterated on one side of the problem without any loop iterating more than once on the other side of the problem. We discussed these checks in Section 2.2.3.

The problem is that each of the loops in this particular function executes for a known number of iterations. The three loops also execute unconditionally in the C function. The independence test won't work here, because it's not possible for one loop to execute but not another. Once again, we have found a special case that starves our tools of information.

We could address this issue by explicitly looking for higher loop bounds. We mentioned in Section 2.2.3 that it is undesirable to discover such bounds unconditionally, as it might substantially degrade performance. We could also consider searching for higher bounds incrementally as the loop discovery process fails and expands its search window. This would address this issue eventually, but the downside is that the discovery process can be slow to fail.

When we examined this problem by hand, it was clear from the debug memory trace what the problem was. The sequence of writes we saw in the second C loop, which matched a sequence outside any binary loop, clarified that the second C loop was expanded. The linear sequence optimisation described in Section 2.4.3 already identifies the linear sequences of C write addresses, and tries to match it against binary loops, discovering no matches.

We adjust this linear sequence mechanism to also note the expressions of interest outside of loops. When we generate a model, it contains a concrete value and offset for each interesting linear sequence on the C side of the problem. We can quickly generate the first 100 values, and check which appear on the binary side outside of loops. In this case a sequence of 16 values appear at distinct binary addresses. Whenever such a sequence of length 5 or more is discovered, we adjust SydTV-GL-refine to check for a loop bound for the relevant loop.

With this adjustment in place, the bound of 16 is found, the split discovery process then succeeds, and the problematic function is handled.

2.5 Verifying the seL4 binary

We now turn our attention to our case study, using SydTV to verify the compiled binary of seL4.

We have tried, wherever practical, to solve problems of translation validation by improving SydTV rather than by imposing restrictions on the source code or the compiler. Nonetheless, in several cases we found that the best way to proceed was to make slight amendments to the seL4 source rather than handle pathological cases in SydTV. This section discusses these amendments, and hopefully gives some insight into how much work is required to produce a verified binary using SydTV.

2.5.1 Strengthening Guards

We have said before that the C semantics are produced in Isabelle/HOL by the C-to-Isabelle parser, and exported into a SydTV-GL representation. This is a slight simplification. In fact we first produce a slightly adjusted variant of the original C semantics, where in particular we strengthen some guards and add others. We have adjusted the seL4 verification proofs to target this adjusted representation rather than the original. We also prove a (trivial) refinement proof which establishes that the original C semantics refine the adjusted ones.

Figure 2.5 shows the various Isabelle representations of the seL4 kernel together. The C semantics is converted first to its Isabelle encoding, then to an adjusted form with stronger guards, and finally to SydTV-GL. The adjusted form is the target of the refinement proof which connects to the higher-level verification of seL4. The refinement chain completes in two directions. We prove that the SydTV-GL semantics refines the higher-level models, and also that the original semantics of the C-to-Isabelle parser does also. The former refinement connects to the binary semantics, the latter refinement means we preserve the original seL4 verification result [KEH+09] also.

The main adjustment made in the conversion involves the pointer validity guard that is asserted at every memory access. The parser supports various possible configurations for this guard. Unfortunately, we need to use both the strongest and the weakest settings. To permit the compiler to make use of the *strict-aliasing* rule, we must make the strong pointer validity assertion which strictly excludes the pointer being aliased by other types. These “strong” validity checks are the pvalid checks discussed in Section 2.1.12. However,

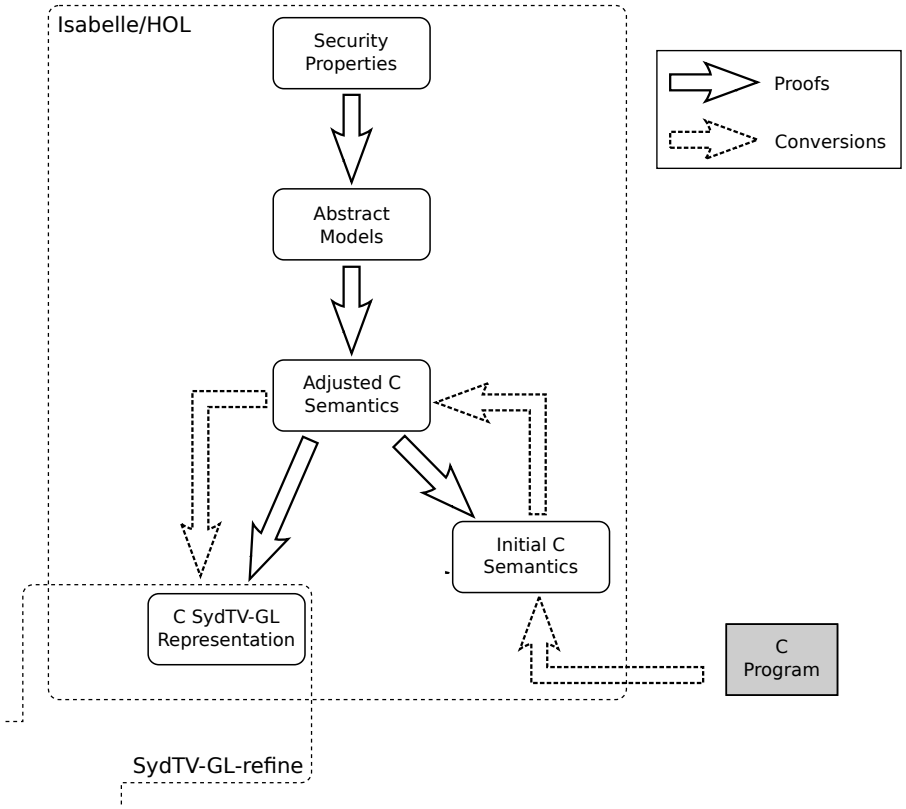


Figure 2.5: Multiple C representations in the refinement chain.

the memory copying and zeroing operations done in seL4 are performed word by word rather than byte by byte, which is technically invalid against the C standard. We need to use weak pointer validity checks within these functions to be able to prove them correct. We use the weak mode of the parser, then strengthen the guards at every other point. This workaround dates back to our previous work with our former prototype.

We plan to adjust the C-to-Isabelle parser with more fine-grained configuration mechanisms. We will then retire this mechanism, and export the original C semantics directly.

2.5.2 Padding Etcetera

Cases where the compiler has a genuine choice create difficulties for SydTV. One such case involves writes to structures which contain padding. In principle the C-to-Isabelle parser could include a nondeterministic choice operator, where the outcome for the padding bytes is unknown. This would require adding nondeterminism to SydTV-GL, resulting in a quantified choice within SydTV-GL-refine. It would also require substantial adjustment to the Tuch memory semantics [Tuc08]. In short, these padding decisions are deeply undesirable.

Instead we eliminated all padding in the seL4 source code. It is not uncommon for embedded systems to avoid padding anyway.

The most common cause of padding was C enumeration constants. The C standard defers to the architecture standard on the default type of such constants, and the ARM reference specifies that the compiler should pick the smallest type available. This led to padding in a number of structures.

The common idiom in C headers is to create an enumeration, and then immediately give the enumeration type a shorter name via `typedef`. We adjusted all such typedefs, keeping the enumeration but having the short name be a synonym for `uint32_t` rather than the enumeration type. The enumeration constants still have their own types, but are always fetched and stored within full length words. This eliminates nearly all padding. There is a slight memory cost in a small number of structure types, but most structure types remain the same size.

The remaining padding can be eliminated by adding dummy fields, or using larger field widths than required, in a handful of structure definitions.

We also adjusted a function to make it syntactically obvious that a structure it manipulates and then returns is fully initialised. In the C standard this is a semantic restriction, not a syntactic one. A standard conforming function may appear to contain a path along which a field of a structure is used uninitialised, as long as that path is semantically impossible. We could try to supply this information to SydTV-GL-refine by adding layers of additional guards, but it is far easier to supply a pointless initialiser in the source function.

Both of these changes, the padding changes and the initialiser changes, date back to our previous work and former prototype.

2.5.3 Nested Loops

A significant feature that is missing in SydTV is support for nested loops. This is largely because we haven't had a strong enough reason to support them. The verified source code of seL4, for instance, contains no nested loops.

The unverified initialisation code, however, contained a small number of nested loops, as did an unverified hardware interface function. Moreover, when compiling with optimisation flag `-O2`, the compiler introduces a number of nested loops via inlining.

The first step we took in addressing this problem is to prevent the compiler introducing nested loops this way. GCC allows us to intervene in its selective inlining process. Functions marked with `__attribute__((noinline))` should not be inlined under normal circumstances. We use this attribute in three places to prevent the compiler inlining a function into a loop body and creating a nested loop.

We also use the same attribute to prevent the compiler inlining a member of a trio of mutually recursive functions. This is because the mechanism for handling recursion bounds which we discussed in Section 2.3.3 tries to map recursion limits from C to the binary, and inlining of recursive calls confuses the mapping.

We make larger refactors to remove the two remaining nested loops. In one case, we move the inner loop into a function call, and again order the compiler not to inline it. In another, we adjust a loop which calls a custom function followed by `memzero`. We don't want to forbid inlining of `memzero`, because it is used in many places and specialising its loop to a given length can improve the code. Instead, we move the `memzero` call within the custom function, a modest refactor which the seL4 developers were happy to accept.

2.5.4 The End of the Line

We also mark the `halt` function as not to be inlined, and specify to GCC that it never returns with `__attribute__((__noreturn__))`. The `halt` function exists in the seL4 source and binary, but the formal verification establishes that it is never called. The actual implementation enters an infinite loop.

The reason for adding the attribute is that, with optimisation flag `-O2`, the compiler inlines `halt` at most of its call sites. This is despite the fact that `halt` is clearly an infinite loop, which suggests that optimising for performance might be futile. Requiring that `halt` not be inlined actually saves instructions, an improvement to the implementation. Saving instructions, especially branch instructions, will (very slightly) improve instruction cache and branch predictor utilisation, and thus performance.

The `halt` function is also a unique function for the analysis. The C-to-Isabelle parser has been told to ignore the body of the halt function. This is also done for some machine interface functions as well. Instead, the parser is given an axiomatised specification to use for the function. In the case of `halt`, however, the axiomatised specification has the empty set for its precondition, that is, it requires that `halt` is never called. Since `halt` is not given a normal body by the C-to-Isabelle parser, it is left underspecified in the SydTV-GL C representation also, and cannot be inlined. SydTV-GL-refine gives up whenever the inlining of the binary is impossible because of underspecification in C.

We could probably adjust the inlining heuristic to cope with this case, and then try to prove that all the inlined loops were unreachable, however the adjustment to the C code was desirable for performance anyway.

The fact that `halt` is marked non returning needs to be handled carefully. The compiler knows that paths that follow halt are unreachable, and may omit code we expect to be present. For instance, stack pointer may never be returned to its starting value. We know from the verification that `halt` is unreachable. To make this information more directly available to SydTV-GL-refine, we add a guard in the guard strengthening phase (see Section 2.5.1). We guard all call sites to `halt`, making it even more explicit in the verification that they are unreachable.

This is not the only performance improvement that has resulted from the translation validation work. When reading a debug trace, we noted that the compiler was issuing instructions to discover the addresses of objects in the globals section, and realised that a number of `const` markers were missing. These markers have now been added, save a handful of instructions, and presumably save a number of cycles from some operations.

2.5.5 The Array Offset Validity Constraint

The largest change we make concerns the validity guards produced by the C-to-Isabelle parser when a pointer value is used as an array.

Consider this trivial function:

```
int
f (int *p) {
  int x = p[42];
  return x;
}
```

The existing C-to-Isabelle parser would encode this memory access by first constructing the address of this array element (`&(p[42])` or `(p + 42)`) and then accessing it. It would also produce a pointer validity guard to ensure that `&(p[42])` was a valid pointer to an integer. We could easily amend the parser to also check that `p` was a valid pointer to an integer also.

The C standard, however, asserts more. The compiler may assume not only that `p` and `&(p[42])` are valid, but that there exists a contiguous array which includes both of them and the space in between.

This issue lead to complex counterexamples involving some message functions of the kernel. These copy values between various message buffers, each of which is an array of `unsigned int`. One such buffer `x->buf` is enclosed within a larger structure, and another `y` is handled simply as a pointer to `unsigned int`. In principle `y` might point to some part of `x->buf`, but then a large constant offset is used `y[OFFSET]` which is larger than the size of `x->buf`. The compiler then concludes that `x->buf` and `y` are totally disjoint, and any accesses can be reordered.

We need to assert, in our trivial function `f`, that there exists an array that covers both `p` and `&(p[42])`. Up until we encountered this issue, there was no support within the Tuch memory semantics for such assertions.

An Assertion

It is easy enough to define a predicate which can be asserted in this case. We define a validity principle pvalidarith for pointer arithmetic, by appealing to the existence of the containing array:

$$\begin{aligned}\text{pvalidarith } htd\ \tau\ p\ n \quad = \quad (\exists p'\ N\ i\ j.\quad & \text{pvalid } htd\ (\tau[N])\ p' \\ & \wedge\ p = \&(p'[i]) \\ & \wedge\ \&(p[n]) = \&(p'[j]) \\ & \wedge\ i < N \wedge j < N \wedge 0 \leq i \wedge 0 \leq j)\end{aligned}$$

The predicate pvalidarith $htd\ \tau\ p\ n$ declares that it is correct to perform arithmetic on pointer p, moving it by offset n, when p is a pointer to type τ, given the heap type description htd. The heap type descriptors and the pvalid operator were introduced in Section 2.1.12. The definition of pvalidarith $htd\ \tau\ p\ n$ says that there exists a specific array pointer of size N which is valid in the normal sense, and from which both p and $p + n$ are offsets.

The above definition is for presentation only. The actual Isabelle/HOL definition is more complicated, because there is no mechanism in higher-order logic which allows us to quantify on an number N and produce from it a type $\tau[N]$. To do the equivalent of the above, we have to dig into several layers of the definition of the Tuch memory semantics, and construct the type information that characterises $\tau[N]$ without actually constructing the type.

Provenance

A confusing technicality in the C standard concerns the case where the address `&(p[42])` is computed but not dereferenced. According to the C standard, a pointer derived from an array pointer is valid for dereferencing if it points within the array, and valid for arithmetic if it points within the array or to the first address outside the array. This permits a well-known idiom where a function iterates a pointer over an array and detects the stopping condition by comparing the pointer to this boundary address.

This boundary confusion becomes important in a case where two arrays `int[] x` and `int[] y` are known to be side by side. The pointer to the start of `y` is also the boundary address for `x`. This single value is valid for arithmetic and access when considered as the pointer to the start of `y`, but as the boundary pointer of `x` it is only valid for arithmetic. The same value would appear to have different semantics depending on where it came from. This is called the issue of pointer *provenance*.

The provenance issue has been discussed in detail by others [HER15, Kre15, MML+16]. At the time of writing, this issue has recently been considered by the ISO/IEC working group (JTC1/SC22/WG14) which officiates over the C standard. In their October 2016 meeting, the group agreed in principle with a proposed amendment by Memarian et al [MS16] which would clarify the standard further, but further rounds of discussion are expected before any judgement is made.

Our solution to this problem is to largely ignore it. We define a "strong" form and a "weak" form of the pvalidarith operator from before. The version shown in the formula above is the "strong" form. When pointer offsets are constructed to be immediately dereferenced in the same statement, we use the strong form. When offsets are computed but not dereferenced, we use the weak form, which is the same but permitting equality $j \leq N$. We only consider whether the value is immediately dereferenced, and do not bother tracking the provenance of the resulting pointer further.

Authors discussing the provenance issue claim that there are example situations in which real compilers such as GCC use this provenenance information to make valuable optimisations [MS16]. However, the examples which we have personally understood up to this point are all somewhat artificial, and seem unlikely to appear in practice. We will continue to avoid the issue as long as we can.

Exporting to SydTV-GL

We have not yet amended the C-to-Isabelle parser to produce the pvalidarith predicate. Instead, for the time being, we add it as an additional assertion via the guard strengthening phase (see Section 2.5.1).

The definition of pvalidarith above includes an existential quantifier, which we do not wish to appear in SydTV-GL-refine. Instead of unfolding the definition, we add an additional operator to SydTV-GL. To reduce complexity, we add a slightly different operator parrayvalid, which is essentially the same as pvalidarith in the case where the offset is positive:

$$\begin{aligned} \text{parrayvalid } htd\ \tau\ p\ sz \quad = \quad (\exists p'\ N\ i.\quad & \text{pvalid } htd\ (\tau[N])\ p' \\ & \wedge\ p = \&(p'[i]) \\ & \wedge\ i + sz < N) \end{aligned}$$

$$\begin{aligned} \text{pvalidarith } htd\ \tau\ p\ n \quad = \quad & ((n = 0 \wedge \text{parrayvalid } htd\ \tau\ p\ 1) \\ & \vee (n \wedge \text{parrayvalid } htd\ \tau\ p\ n) \\ & \vee (n < 0 \wedge \text{parrayvalid } htd\ \tau\ (\&(p[-n]))\ (-n)) \end{aligned}$$

This predicate parrayvalid $htd\ \tau\ p\ n$ declares that there is an array of elements of type τ, including at least the pointer p and the next n elements, within the heap type description htd.

We now have two predicates that specify an array of constant size, parrayvalid $htd\ \tau\ p\ n$ and pvalid $htd\ (\tau[n])\ p$. Their semantics is subtly different. The pvalid assertion specifies a *closed* array starting at p. This array may be a field of a bigger structure, but it may not be a subarray of a larger array with the same type. The parrayvalid assertion specifies an *open chunk* of array, which might be part of a bigger array.

Proving Distinctness

We mentioned the *open*/*closed* distinction above, because our next task after including the parrayvalid operator in SydTV-GL is to support it in SydTV-GL-refine. We need to use this operator to prove addresses are distinct.

We mentioned in Section 2.1.12 that each pair of pvalid assertions on the same htd generate an extra assertion that we feed to the SMT solver. If pointers p and p' are both valid, we assert that either:

1. p equals p' (same address and types).
2. p and p' are totally disjoint, with the structures they point at having no common byte.
3. p points at a field within the aggregate structure p'.
4. vice-versa, p' points within p.

We can generalise each of these notions to cover either kind of point validity operator, array or otherwise. When we say that p points at a field within an aggregate structure p', an *open* array chunk may point at any subarray of any field of the aggregate structure p', or may be directly a subarray of p' itself.

We also add one more case:

- p and p' are both *open* array chunks, and they share at least one element.

Verifying the Assertions

The next task was to prove the new assertions in the seL4 verification. This turned into a major proof effort, because as originally stated, the new assertions were not true. The heap type descriptor object is "ghost data" in the C semantics. It is always available as a global object. Specially formatted comments in the C source are interpreted by the C-to-Isabelle parser as update operations on the available types. The seL4 Retype operation, which is called to create new objects, includes updates to the descriptor object. When Retype created some new array objects, including new capability table objects, it updated the descriptor to contain each new capability storage slot, but not to include the array structure which contained them.

Perhaps it was not such a surprise that the array was not created, since the Tuch semantics initially lacked any mechanism to create it. There is an operator for retyping to make a given pointer valid, including pointers to arrays of any given constant length. This does not help us with dynamic length arrays. Once again, we need to dig into some levels of the Tuch semantics to define a new retype operator which can create arrays of variable length.

The existing proof of correctness of seL4 involves three levels, connected by refinement proofs. Validity of pointers being used in C is usually proven by appeal to the validity of pointers being used at the other levels, as the refinement proof steps through the programs together. This reasoning does not directly apply to array pointers, which are represented in a somewhat complex way in the other two models of seL4. To permit a similar approach, we adjust the intermediate seL4 model to check some pointer offset properties that will be relevant to prove validity of the C pointers. We must also adjust the state relation between the C heap and the specification's memory model to ensure that arrays exist in C when they are expected from the memory model.

Finally, the real work is to return to the proofs about the Retype operation, and prove that the new kinds of pointer updates preserve all the invariants, and also that the new arrays in the heap type descriptor are preserved when necessary by all the existing updates. This was the hardest part of the array assertion change, with the final change adding over 1600 lines and removing over 1400 lines from the Retype theory alone. This region of the seL4 verification has previously been described as somewhat aesthetically unpleasant.

In total, this proof effort requires the addition of 758 and removal of 785 lines from the correctness proof of the intermediate specification, and the addition of 3985 and removal of 2614 lines from the C/intermediate refinement proof. This project took 7 weeks of work, despite us already having substantial relevant experience.

2.5.6 Additional Adjustments

We also made some further minor adjustments to the seL4 source to address minor issues with the translation validation.

We suppressed inlining in two cases, to avoid the creation of two monster functions. The first of these functions would accumulate nearly all of the seL4 boot sequence into a single function body. The other would accumulate nearly all of seL4's user-callable architecture-specific operations in a single function body. The monolithic architecture-specific function would then contain 1531 instructions, about 10% of the instructions in the kernel, 14 different loops, and code from 106 additional C functions, copied into a total of 326 inline sites.

Each of these monster functions can be divided into a few large functions with requests to suppress inlining (we discussed using such requests to prevent nested loops in Section 2.5.3). We needed 3 source annotations to break up each of these monster functions, 6 annotations in all.

The largest subfunction of the architecture-specific function, for instance, contains 879 instructions, 8 loops, and code from 61 additional functions inlined at 218 sites. This might sound like a minor adjustment, but significant amounts of proof time can be saved by reducing the number of loops present in any single large problem. Performing this division, however, added 169 instructions to the binary to support the three call sites. This overhead is unfortunate, but we think acceptable.

Another minor adjustment involved an unverified boot-time function of seL4 which checked a hardware-provided value against the constant `0xf « 28`. This left shift results in a sign change, which is forbidden by the C standard, and so this function would be impossible to verify.

This function unconditionally results in undefined behaviour according to the C standard. The guards produced by the C-to-Isabelle parser are unsatisfiable, and in the SydTV-GL representation, the `Err` address must be reached. Since SydTV-GL-refine assumes that `Err` is not reached for most checks, any binary function could be proven to refine this C function. This is exactly what the C standard means when it says this behaviour is undefined, the compiler can provide whatever implementation it wants. It also clarifies why these guards are so important to check.

We detected this case because we have a safeguard in SydTV-GL-refine which checks that the various guards are together satisfiable. This was first introduced to help find a bug in the handling of the array pointer validity constant we introduced in Section 2.5.5, and seems to be a worthwhile case to detect generally.

Having detected the problem, it is very easy to amend `0xf « 28` to the correct syntax `0xful « 28`.

2.5.7 The Clone Problem

The proof of refinement performed by SydTV-GL-refine is decomposed by function *pairing*, as we discussed in Section 2.1.4. In the cases we analyse, the vast majority of binary functions are members of pairings, whereas many C functions are marked `static inline` and do not have binary counterparts.

The exception to this case involves "clone" functions. A clone function is an additional private symbol in the binary, with a symbol name such as `isRunnable.clone.132`. Thi function is a special-purpose implementation of the C function `isRunnable`. The compiler does not give it the canonical symbol name `isRunnable`, because the compiler does not wish to claim that this is an implementation of `isRunnable` in the usual sense. The clone might make some assumptions about its arguments, or about its calling environment, or might violate the usual calling convention.

The advantage of cloning is that it provides an intermediate alternative to function inlining. Inlining the body of a function allows the compiler to eliminate the overhead of the function call, and also to specialise the body of the function to its particular arguments. The downside of inlining is that it may lead to code duplication. Cloning may allow specialised implementations to be shared between call sites. The overhead of the function call cannot be entirely avoided, but it can be reduced by specialising the calling convention to fit the call sites.

The problem for SydTV-GL-refine is that clones cannot be paired against the functions they were generated from. There probably exists some input/output relation that makes `isRunnable.clone.132` a refinement of `isRunnable`, but we do not know what it is. The obvious way to proceed is to inline `isRunnable.clone.132` in all proofs about functions in which it is called. The problem, as discussed in Section 2.3.2, is that the way we handle stack equality does not permit binary functions to be inlined into refinement problems.

We work around this problem by using the GCC attribute `__attribute__((noclone))` to prevent the creation of problematic clone functions. This attribute prevents the compiler from emitting a clone copy of the given function.

Clone functions do not appear in the binary when seL4 is built with GCC 4.5.1 and optimisation flag `-O1`. With `-O2`, we see a single clone function, and so eliminating it with this override seems a reasonable alternative to revisiting the problem of SMT stack representation. With optimisation flag `-O3`, we see an additional five clone functions, three of which are problematic. At the higher optimisation setting, the compiler is more willing to expand the binary by creating clones of existing functions. Clone functions that make no function calls themselves are not a problem. Once again, we address these three problematic functions with the no-cloning attribute.

When we build the kernel with flag `-Os` ("optimise for binary size"), however, there are 96 clone functions. This is for the opposite reason to `-O3`. The compiler is now extremely reluctant to expand the binary by inlining multiple copies of functions. This includes a great many functions marked `static inline` in the seL4 headers. We have not yet come up with a workaround for this case.

We also add the no-cloning attribute to functions we have previously marked no-inline (see Section 2.5.3, Section 2.5.4, Section 2.5.6). This adds a further 11 no-clone requests, for a total of 15. It is no help to us if the compiler avoids inlining a function, but produces a clone body for it that we must inline at the problem level anyway.

This level of workarounds is sufficient for the time being, although it leaves us no satisfactory way to address the `-Os` binary. It is clear that we need to find a more durable

solution to the clone problem in the near future.

2.6 Evaluation and Discussion

2.6.1 Results

We have run the SydTV-GL-refine analysis on the seL4 binary, as produced by GCC with optimisations levels `-O1` and `-O2`.

For GCC `-O1` SydTV-GL-refine reports the results:

- 268 function pairings are checked
- a further 16 are skipped[9]
- no proof failures
- the slowest problem took 69052.05s (19:10:52)
- the total time taken is 160312.37s (1 day, 20:31:52)

For GCC `-O2` SydTV-GL-refine reports:

- 258 function pairings are checked
- a further 16 are skipped
- there are 3 failures
- the slowest problem took 67734.66s (18:48:54)
- the total time taken is 382531.45s (4 days, 10:15:31)

There were three failures in our final `-O2` proof run. Two of them we know how to address, but the remaining one involves an SMT solver timeout and may require further source-level workarounds. We are confident we will produce a completely verified binary in the very near future.

These times are clocked on a virtual machine controlling 16 Intel Xeon E5-2640 cores running at 2.40 GHz. The high core count allows us to run five SMT solvers in parallel for difficult problems. These include CVC4 [BCD+11], SONOLAR [PVL11] and Yices2 [Dut14]. The SMTLIB2 [BST10] shared input language allows us to interact with these tools uniformly.

We have previously used Z3 [dMB08] in our experiments, but an incompatibility with the standard libraries available in our VM causes it to run unreliably. It is extremely useful to us that the SMTLIB2 [BST10] standard input format allows us to switch out SMT solvers to address various issues, including trivial ones.

These long execution times are an issue. To produce accurate times, we ran the whole experiment sequentially on an otherwise unloaded machine. We can split the work between parallel workers to save some time. In addition, the vast majority of problems are solved fairly quickly. The fastest 90% of the `-O1` problems are finished in 9425.68s (2:37:6), 5.9% of the total running time. For `-O2`, the fastest 90% take 17356.87s (4:49:16). The remaining difficult problems account for nearly all the running time. While working

[9] These are machine interface functions and other assembly stubs which do not have C bodies to compare to.

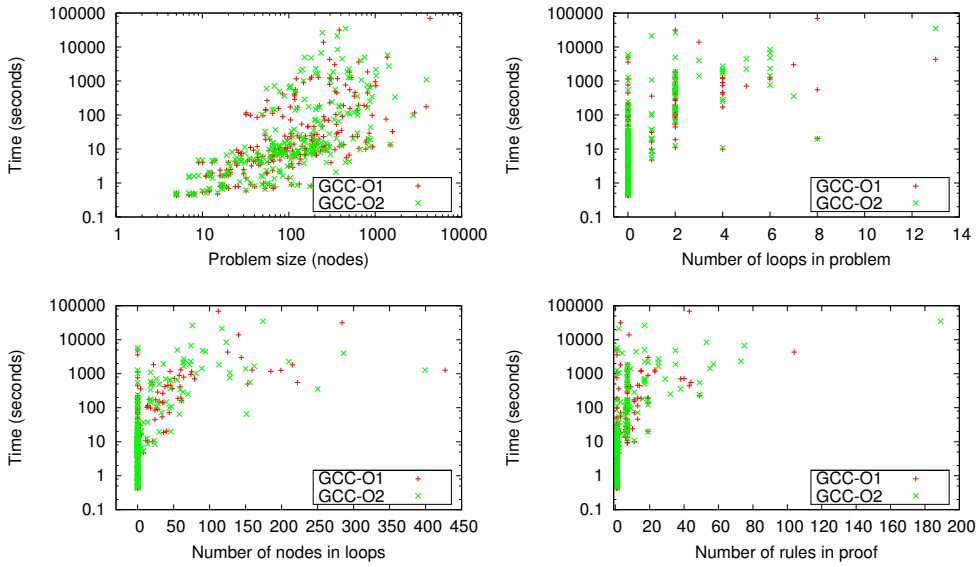


Figure 2.6: Correlation of problem size, loop count, loop node count, and proof size to analysis time.

on the tool we typically focus on these faster examples for testing. It is possible that further improvements to SydTV or SMT solvers may lead to substantially faster analysis times for the very difficult problems.

2.6.2 Correlations for Proof Time

Clearly the time taken for SydTV-GL-refine to verify a function depends enormously on the function involved. Many of the slowest functions have numerous cases and loops. Figure 2.6 shows the correlation between analysis time and various obvious factors. These include the number of nodes in the problem representation, the number of loops present (counted across both the source and binary), the total number of nodes in those loops, and the number of rules in the resulting proof script.

The y-axes of the graphs in Figure 2.6 are logarithmically scaled because the variation of analysis time is so substantial. Clearly the larger problems with more loops dominate the analysis.

2.6.3 Benefit of Model Guidance

We have claimed that the model-guided split discovery approach of SydTV-GL-refine saves analysis time, because candidate pairings that do not have to match are unlikely to match by accident in satisfying models. Figure 2.7 shows the decay in the number candidate loop pairings after successive SMT queries. The decay is shown for 16 randomly selected example problems. In each case, the results are shown with the search window set correctly, so that the process ends with a candidate pairing being discovered.

The y-axes are once again scaled logarithmically, since the number of candidate pairings in some problems is so high. The roughly linear downward trend we see at the start of most of these arcs represents exponential decay, which we regard as success. Without this exponential decay, many more SMT queries would be necessary to eliminate

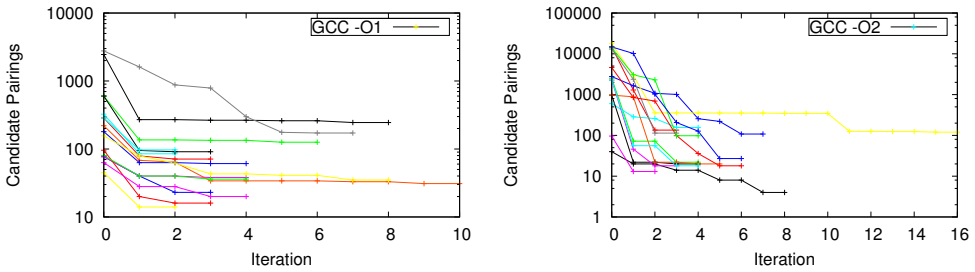


Figure 2.7: Decay of the number of candidate loop pairings with successive SMT queries.

	GCC -O1	GCC -O2
Window size 3:	32 / 62 (51.6%)	4 / 15 (26.7%)
Window size 4:	3 / 4 (75.0%)	1 / 2 (50.0%)
Window size 5:	n/a	17 / 18 (94.4%)
Window size 6:	n/a	8 / 12 (66.7%)
Window size 7:	n/a	45 / 57 (78.9%)

Table 2.1: Hit rate of the linear sequence optimisation.

all the invalid pairings.

Many of these arcs stabilise with a substantial number of candidate pairings remaining, suggesting that most of these candidates will survive further iterations. This is quite likely the case. There may be many potential candidate pairings which are all valid for any problem.

2.6.4 Benefit of Linear Sequences

The linear sequence mechanism accelerates the process of split discovery by narrowing the search to the correct pairing type and search window size.

The linear sequence optimisation applies frequently, and applies especially frequently in problems which require wider search windows (e.g. unrolled loops).

With GCC `-O1`, the vast majority of loops can be matched without expanding the search window. A small number require a sequence offset. The linear sequence optimisation succeeds fairly frequently. Table 2.1 clarifies the number of loop problems that the linear sequence optimisation produces data for. With GCC `-O2`, the number of loop problems that require window expansion is substantial. Crucially, the linear sequence optimisation applies to the majority of problems that require deep expansion.

Figure 2.8 gives some suggestion as to the value of the linear sequence optimisation. It once again graphs the decay in the number of candidate splits after a number of SMT queries. Instead of graphing a number of independent problems at the correct search width, however, this graph shows the repeated analysis of the `strncmp` problem at different search widths if the linear sequence optimisation is deactivated.

This graph shows the substantial value of the linear sequence analysis. By picking only the correct search window, the discovery process can reduce the number of SMT queries required from 26 to 6, a 77% reduction.

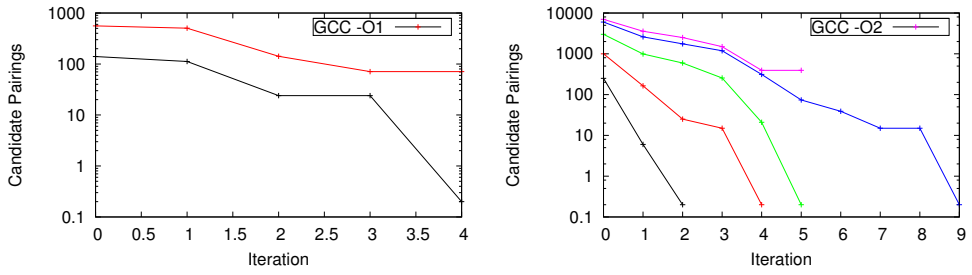


Figure 2.8: Decay of the number of candidate loop pairings in `strncmp` with successive SMT queries.

2.7 Related Work to Translation Validation

The field of translation validation began with two pioneering works. In nearly twenty years since, dozens of works have tried to fulfil the ambitions of one of these founding projects.

Translation validation was first proposed by Pnueli et al [PSS98], as a pragmatic alternative to maintaining a verified compiler. Their SIGNAL compiler translates programs in the SIGNAL synchronous language into implementations in C, and also produces evidence of correctness, which a separate validator process reads. The validator upgrades that evidence to a proof of correctness within the TLV [PS96] proof environment.

The second founding work of translation validation is by Necula [Nec00], who applies the approach of Pnueli et al in a far less formal setting. Necula studied the optimisation phases of GCC 2.7. The aim of that project was not to produce any formal evidence of correctness, but rather to quickly and easily increase the general trustworthiness of GCC. The validators for the various optimisation phases were simple efficient programs, with an entire validation pass typically much faster than a complete run of the compiler. Failures of validation were to be used as bug reports.

Over a decade of translation validation experiments [ZPF+02, ZPG03, GZB05, BSPH07, SG05, RS09, TSTL09, KLG10, TGM11] have followed. We do not have space to detail the contributions of all these systems here. Most of these projects aspire to the formal precision of Pnueli et al, but few achieve it. This is because Pnueli et al had the advantage of a disciplined compiler for the well structured SIGNAL language. By comparison, performing translation validation on languages that appear in the wild, such as C [ZPG03, GZB05, TGM11, STL11], Java [TSTL09], or Simulink [RS09] introduces complications that make it difficult to be truly precise.

Instead, translation validation approaches are mostly following Necula's lead, becoming a best-effort strategy rather than a way of ensuring correctness. Many concepts in computer science have followed this path from formal precision to pragmatic tradeoffs, with type systems being perhaps the best known example. Some projects aim to do the reverse, however. For instance, the Vellvm [ZNMZ13] and Alive [LMNR15] projects build verified optimisation passes, and insert them into the LLVM compiler framework to make them pragmatically useful in the short term. Perhaps these efforts will eventually result in a highly optimising verified compiler.

Real-world translation validation is also a problem that is steadily getting more difficult. Necula's original validation for GCC 2.7 focused on algebraic transformations. When Goldberg et al [GZB05] attempted similar work five years later, they discovered

that C compilers had begun to perform aggressive structural optimisations such as loop rearrangements which Necula's approach could never have handled.

Our approach is yet another variation within the space of possible translation validations. Like Tristan et al [TGM11] in their work on LLVM phases, we do not make use of any hints from the compiler. Like Ryabtsev [RS09] et al in their work on Simulink, we target the end-to-end transformation rather than any internal compiler step. Furthermore, like Pnueli et al's original work, we do not accept any failures or false positives in the process.

However, unlike other authors, we handle a well understood subset of the language C rather than all C code and optimisations that might exist in the wild. Our work is also strongly grounded in well understood semantics at both ends, rather than focusing on 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.

2.7.1 Producing a Verified Binary

Pnueli et al [PSS98] originally introduced the approach of translation validation as an alternative to maintaining a verified compiler in their SIGNAL project. In both the SIGNAL work and ours, the advantage of separating the translator from the certifying step was *flexibility*. For the SIGNAL compiler, this mostly meant the flexibility to maintain and engineer the translator without having to revisit its correctness proof for each minor change.

In both the SIGNAL work and ours, the objective is to eventually produce a verified output program. As we have seen above, the field of translation validation has expanded since then, with most works not aiming to actually construct a proof of correctness.

Let us return to the original question. How best should we construct a verified binary? What can we learn by surveying the various projects in the field?

A few projects have produced small verified binaries by reasoning directly over the semantics of the binary program itself. The first verified operating system, Bevier's KIT [Bev89], was completely verified in this manner at the assembly level. However, it measured only a few hundred lines of assembly in total. More recently, Ni et al [NYS07] verified modern context switching code using the XCAP x86 model. Chlipala [Chl11] has demonstrated that some of the complexity of this approach can be addressed with sophisticated automatic reasoning tools.

An alternative to reasoning about the machine semantics by hand is to use a simple hand-tuned compiler within the proof environment to produce binaries from low-level functions. Myreen has produced a hand-extensible miniature compiler [MSG09] for this purpose, and Chlipala's Bedrock framework [Chl10, Chl13] has taken this concept far further. These tools are useful for verifying smaller specialised functions, especially those that perform niche tasks, but it has not yet been demonstrated that they can scale well enough to manage substantial systems.

The remaining strategies include the use of verified compilers, translation validation, and certifying compilation or proof carrying code. To the best of our knowledge, the only recent work on translation validation that aims to produce a verified binary is our own.

Proof carrying code was another approach suggested by Necula [Nec97]. In this approach, the compiler is not verified but instead produces a certificate of correctness. The Verve project [YH10] uses a hybrid of this approach with hand verification to produce a

somewhat unusual verified OS. The system consists of a directly verified minimal runtime system, a certifying type-preserving compiler, and a certificate checker built into the runtime, to ensure type safety. The whole OS is then built above this runtime using a single language environment.

The Verisoft project [APST10] was the first to present a verified low-level component (a hypervisor) produced by a verified compiler. The project developed a simple minimal 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.

There are a number of recent practical verified compilers, such as CompCert [Ler09] and CakeML [KMNO14], as well as verified compiler components [BDP12, ZNMZ13, LMNR15]. However, given these, there are fewer resulting verified binaries in the world than we might expect. Merely presenting a verified compiler only addresses half the problem. To produce verified outputs, the compiler must also provide a useful methodology for verifying source programs against its semantics.

This is not a trivial task. In the case of the CompCert compiler [Ler09], roughly half a decade passed between the first clear specification of an approach to verifying CompCert-compiled programs [App11] and the point at which this approach became mature enough to yield substantial rewards [BPYA15, GSC+16]. This required addressing both technical issues about the way that programs are represented as well as deeper issues about the memory model [LABS14] and notion of partial compilation [SBCA15].

Now that this work is mature, sizeable verified programs written against CompCert, such as an implementation of OpenSSL HMAC [BPYA15] and the CertiKOS [GVF+11, GSC+16] OS kernel, are beginning to appear. It is really only at this point that we can begin to say that there is enough literature to meaningfully compare the productivity of this approach to others.

The CakeML verified compiler [KMNO14] is following the same path. The first verified CakeML program was the compiler itself, but substantial progress has recently been made on a more productive verification methodology [GMKN17], with the expectation that more verified programs will now follow quickly. The Standard ML dialect that CakeML targets is a much higher level language than C, and more amenable to verification, with known challenges for low-level programming and performance.

2.8 Concluding: A Verified Binary

We have presented the design of SydTV, a translation validation approach capable of verifying the correctness of compiled binaries even when a diverse collection of complex loop optimisations are applied.

The specific contributions of this work are:

- A model-guided SMT-based split discovery process.
- Extensions to the SMT theory to support pointer validity, pointer arithmetic validity, and partial array equalities.
- Discovery of important characterising linear sequence information that is not directly necessary for the search process but greatly improves its efficiency.
- Additional fine tuning of the proof discovery process to handle exotic cases.

- A case study in applying the whole validation to the substantial seL4 source base, and discovery of necessary adjustments.
- Discovery of missing proof obligations concerning pointer arithmetic in the Tuch/Norrish C semantics and seL4 verification, and significant re-verification work to prove those obligations.
- Performance analysis of the proof process, highlighting the room that remains for improvement.

The key result of this work is that we can transfer the essential functional correctness property of seL4 from the C semantics to the semantics of the binary. This means that we have robust evidence that the binary conforms to the high level correctness properties stated in Isabelle/HOL. The C source, its semantic model, and the compiler need no longer be trusted, enabling the use of off-the-shelf tool chains in a high-assurance environment.

Running SydTV on our case study requires a modest number of human interventions and a substantial amount of CPU time. We have detailed these efforts as a guide to others interested in using this tool or following a similar approach. We think that we still have room for improvement in the performance aspects of SydTV, but we are satisfied with its coverage. We would also like to reduce the amount of human intervention, which is a downside of our approach when compared to a verified compiler. However, as we have outlined in Section 2.7, producing a verified binary is a major human effort, even given a verified compiler. In fact, the flexibility of our approach in allowing us to tweak our source semantics to fit our target program may well compensate for the overheads of running the validation process.

The strength of our work is the *flexibility* by which we can combine a commodity compiler, a diversity of SMT solvers, a designed-for-verification C semantics, and a homebrew proof search procedure to quickly obtain a verified binary.

In conclusion, we are confident that the verified seL4 binary we produce with GCC `-O2` in this project is the most efficient and trustworthy binary yet produced amongst comparable works.

3 Real-Time Applications

This chapter closely follows the presentation of the article "High-assurance Timing Analysis for a High-assurance Real-Time OS" (see [SKH17]), which was written in collaboration with Felix Kam and Gernot Heiser.

Felix contributed substantially to the implementation of this work during his undergraduate thesis year, developing the SydTV/Chronos connection discussed in Section 3.4.1 and the explicit loop bound search strategy discussed in Section 3.4.2, and supervising countless experimental runs leading up to the successful experiments we describe here.

3.1 Introduction

Timeliness is a crucial correctness requirent for some software systems. These systems, like others, must be assured to produce the correct outputs, but must also be assured to respond within a known period of time. Failing to react in a timely manner may be as serious a defect as failure to act correctly. Systems with timeliness constraints are called *real-time* systems.

Timeliness requires, among other things, sound estimation of worst-case execution time (WCET). This is nearly always calculated by some kind of static analysis of the *binary code*. This introduces a key tension in the problem of WCET estimation. Analysis of the binary-level program must struggle to recover information that was obvious at the source level, such as typing information, bounds on some variables, distinctness of some pointers, etc. Source level analysis, however, has easy access to rich sources of correctness information, the most valuable being the insights of the software developers themselves.

All WCET analysis must address this question. Should it consider source information? If so, how can this be converted into precise information about binary-level timing?

This chapter proposes a new approach, based on our translation validation tool SydTV. SydTV implicitly relates control flow at the binary and source level, which allows our WCET analysis to make use of information missing in the binary. We can also manually intervene in the binary-level analysis by adding information at the source level. We do this by annotating the source code with extra assertions formatted as special comments. These comments are ignored by the compiler, but are part of the formal model of the C program, are proven correct, and may be used by SydTV as additional assumptions.

Our WCET analysis follows the pattern of nearly all others in the literature. Firstly, we extract a control-flow graph (CFG) from the binary, which is used to generate candidate execution paths. The execution time of a path is estimated (conservatively) with the use

of a micro-architectural model of the hardware. Where paths include loops, we must determine safe upper bounds on the number of iterations possible in that loop. Many candidate paths may also be infeasible (the conditions required to follow the path at every branch are impossible) and these paths should be eliminated from the analysis to avoid an excessively pessimistic WCET.

We evaluate this WCET approach on our target of interest, the seL4 microkernel [KEH+09]. By adding a handful of new annotations to seL4, we can discover all loop bounds necessary to compute seL4's WCET. We identify a number of operations in the kernel which make large contributions to WCET. Fortunately there exist system configurations which prevent application code from exercising these operations, leading to much improved time bounds. We also explore an alternative implementation of one of these operations, verify that the new implementation is functionally correct, and demonstrate that incorporating this change can allow more of the kernel API to be used with acceptable WCET.

Finally, our WCET approach is not limited to functionally-verified code such as seL4. Any C code that is understood by the C-to-Isabelle parser can be analysed. We do however need to assume that the C program behaves as the parser expects, including in particular that it does not contain any unspecified or undefined behaviours. The absence of such behaviours could, for instance, be proven by static analysis.

We make the following key contributions:

- high-assurance construction of the binary control-flow graph, with a proof of correctness of all but the final simplification (Section 3.4.1).
- WCET analysis supported by a translation-validation framework, allowing C-level information to be used in computing provable loop bounds and infeasible paths (Sections 3.4.2–3.4.4);
- computation of all loop bounds needed for WCET of the seL4 kernel, with the support of source-level assertions, but no manual inspection of the binary program (Section 3.6.1), and similarly elimination of infeasible paths (Section 3.6.6);
- improvement of the WCET of the seL4 kernel by reimplementing one of its key operations (Sections 3.5.2 and 3.5.3);
- demonstration that the approach is applicable to code that is not formally verified, by analysing a subset of the Mälardalen benchmarks (Section 3.6.5);
- further evidence of the *flexibility* of the SydTV approach, showing that its three main components can all be easily repurposed into a binary analysis suite.

3.2 Background

This section summarises material on which we build directly. Section 3.3 summarises other related work from the literature.

3.2.1 Chronos

For WCET analysis we use the Chronos tool [LLMR07], which is based on the *implicit path enumeration technique* (IPET), to perform micro-architectural analysis and path analysis. The attraction of Chronos is its support for instruction and data caches, a flexible approach

to modeling processor pipelines, and an open-source license. It transforms a simplified CFG, with loop-bound annotations, into an integer linear program (ILP). We solve this using an off-the-shelf ILP solver – IBM's ILOG CPLEX Optimizer – to produce the estimated WCET. Infeasible path annotations can generally be expressed as ILP constraints.

We build on previous work by our colleagues, Blackham et al [BSH12], who adapted Chronos to support certain ARM microarchitectures for the WCET analysis of seL4. While seL4 can run on a variety of ARM- and x86-based CPUs, presently only the ARM variant is formally verified (but verification of the x86 version is in progress). Blackham et al picked the Freescale i.MX 31 for analysis, thanks to its convenient cache pinning feature, which is unavailable in later ARM processors. The i.MX31 features an ARM1136 CPU core clocked at 532 MHz, has split L1 instruction and data caches, each 16 KiB in size and 4-way set-associative. The processor uses pseudo random cache-line replacement. The cache is modeled as a direct-mapped cache with the size of one of the available ways (4 KiB), based on the pessimistic prediction that the other three ways contain useless data which is at random never replaced.

Blackham et al [BSH12] carefully validated this model against cycle timing on the real processor. They concluded that modeling the 4-way cache as 1-way was pessimistic but sound, never overestimating cycle times. They also discovered that the L2 cache degrades worst-case times substantially. When it is enabled, the total cycle time to miss all caches and access main memory increases significantly. In pessimistic calculations such as these, expected L2 hits are rare, and the time lost outweighs the time saved. We configure the system and the Chronos model to have the L2 cache disabled.

In this work we keep the microarchitecture model unchanged from the work by Blackham et al. The Freescale i.MX 31 is now an old architecture, however, validating the timing model on a new architecture requires a lot of experimental work, and is not the focus of the current project.

3.2.2 The seL4 Operating System Kernel

We have already introduced the seL4 microkernel, and its formal verification. It provides a minimal set of mechanisms but nonetheless is designed to support a broad spectrum of use cases, including use as a pure separation kernel, a minimal real-time OS, a hypervisor supporting multiple Linux instances, a full-blown multi-server OS, or combinations of these.

Mixed-criticality workloads are a target of particular interest. Such systems consolidate mission-critical with less critical functionality on a single processor, to save space, weight and power (SWaP), and improve software and certification re-use [BBB+09]. Examples include the integrated modular avionics architecture [ARI12], and the integration of automotive control and convenience functionality with Infotainment [HH08]. These systems require strong spatial and temporal isolation between partitions, for which seL4 is designed. The various proofs of the seL4 verification directly address issues of reliability [KEH+09], spatial security [SWG+11], and information security [MMB+13].

We build on prior work by Blackham et al [BSC+11, BSH12, BH13, BLH14] on performing WCET analysis for seL4. Discovering a known WCET bound to the kernel is a crucial step towards supporting real-time and mixed-criticality systems. More work also remains to be done on seL4's scheduling model [LH14, LH16].

The kernel executes with interrupts disabled, for (average-case) performance reasons as well as to simplify its formal verification by limiting concurrency. To achieve reasonable WCET, preemption points are introduced at strategic points. These need to be used

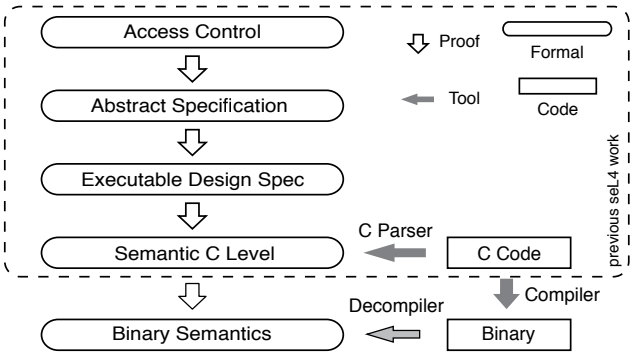


Figure 3.1: The seL4 functional correctness stack.

sparingly, as they may substantially increase the code complexity and the proof burden. A configurable preemption limit (presently set to 5) controls how many preemption points a kernel execution must pass to trigger preemption. Adjusting this limit adjusts the tradeoff between the worst-case time to switch to a higher-priority task on interrupt and the worst-case time to complete a complex task in the presence of interruptions. The preemption model is discussed in detail in Section 3.2.4, and also by Blackham [BSH12].

This preemption mechanism fits reasonably well with the ILP approach. If we assume that an interrupt is ready to fire shortly after kernel entry, it follows that the preemption point function will never be called more than 5 times in a single kernel entry. This is trivial to encode as a global ILP constraint. This constraint also implicitly bounds those loops which include a preemption point, meaning we don't need to calculate bounds for them. Chronos still requires such a bound, so we use a nonsense bound (10^9).

To cover the case where an interrupt arrives at an arbitrary point, we must adjust the ILP problem slightly more. We are interested in the maximum latency between the interrupt arriving and kernel execution completing. This latency would always be increased by the interrupt arriving one step earlier, except if the presence of the interrupt would change the behaviour of that step. The only steps at which the interrupt affects (our model of) CPU execution are preemption points. Thus we can handle this case by specifying that the graph of configurations within the ILP may be entered either at the kernel entry address or immediately after any preemption point.

In addition to producing the first WCET analysis for seL4, Blackham et al aggressively optimised the kernel for latency [BSC+11, BSH12]. Among other measures, this involved placing additional preemption points in long running operations. These changes and analysis proved that a variant of the seL4 kernel can achieve interrupt latency competitive with a single-purpose real-time OS. In contrast, our intention here is to develop a *high-assurance analysis process*. Thus we apply our approach to the most recent *verified* version of seL4, which lacks these unverified modifications. We note that the number of loops we analyse is significantly larger than in Blackham et al's previous work, which set the preemption limit to one.

3.2.3 The seL4 Verification & TV Framework

The WCET analysis we are doing does not stand alone. By adding annotations to the source code and verifying them in the theorem prover, we build the WCET analysis *into* the existing functional correctness stack.

To recap, the verification of seL4 comprises over 200,000 lines of proof script, manually

written and automatically checked by the theorem prover Isabelle/HOL [NPW02]. The headline proof is built from multiple proofs of refinement, seen in Figure 3.1, of which only the last concerns the C source code. The formal model of the C code in Isabelle/HOL is produced by the C-to-Isabelle parser [TKN07]. The verification stack is then extended by SydTV down to binary level.

3.2.4 The seL4 Timing and Preemption Model

The focus of this chapter is determining the WCET of the seL4 kernel under various assumptions. These time bounds can then be used to answer questions about the execution time of real-time systems built on top of the kernel. There are various established approaches to timing analysis for such systems, some of which call for slightly different worst-case timing measures, including *worst-case response time* and *worst-case interrupt latency*. The WCET of the kernel (under various assumed scenarios) is in fact a sufficient measurement, thanks to the specifics of seL4's timing model.

Firstly let us clarify that the WCET of the seL4 kernel is known to be finite. seL4 is an event-reactive kernel with a single kernel stack. The kernel has no thread of execution of its own (except during initialisation) and executes in response to specific external events. These events include system calls, hardware interrupts, and user-level faults. Each kernel entry uses the same kernel stack to call a kernel top-level function, e.g. `handleSyscall` for system calls. This C function executes atomically to normal completion, rather than stopping abruptly (e.g. via `longjmp` or `exit`) or being suspended (e.g. via `yield`). Interrupts are also disabled while these top-level functions are executing. Thus the WCET of seL4 exists; it is just the maximum WCET of the various entry points.

We can compute the WCET of each of the kernel entry points, of which the system-call handler will always be by far the greatest contributor. This is because seL4 follows the microkernel philosophy, and does not fully handle faults or interrupts itself (apart from some timer interrupts). Instead it despatches messages to user-level handlers, and the messaging facility of the microkernel is designed to be fast. Some system calls take much longer to complete, partly because seL4 avoids managing its internal memory allocation itself, and instead allows user level managers to request major configuration changes. To prevent substantial delays to other tasks, these long-running operations include *preemption points*.

When a preemption point is reached, seL4 can check for pending interrupts, and if there are any the current operation is discontinued. A configurable preemption limit adjusts how often the actual interrupt check is performed compared to the number of preemption-point function calls. Note that the exit process still results in a normal completion of the top-level `handleSyscall` function, even though the logical operation is still incomplete. This was done for verification reasons: the model of C semantics used to verify seL4 does not allow abrupt stops (e.g. `exit`) or any form of continuation yielding. The interrupt is handled as the last step in the execution of `handleSyscall`, usually resulting in a context switch to its user level handler. The preempted operation resumes as a fresh system call the next time the preempted task is scheduled.

In this model of kernel entry and preemption, the execution time of seL4 contributes to the completion time of some real-time task in three ways:

1. Time spent in the kernel during the task's timeslices, performing system calls on behalf of this task. This includes as many attempts as are necessary to complete any preemptible system calls.

2. Time spent in the kernel during the task's timeslices , when the task is being interrupted. This includes the time overhead of switching to and from any higher priority tasks which resume as the result of an interrupt. This also includes the time taken to handle a hardware interrupt and queue a lower priority task to be scheduled, but not to switch to it.

3. Delays to the start of the thread's timeslice or to delivery of its interrupts caused by the kernel executing atomically on behalf of another task (of any priority).

While all of these execution times are important for real-time performance, the first two contributions can be managed by system design. For the first point, a critical real-time thread should obviously avoid expensive system calls that have a major impact on its WCET. As all the expensive calls involve system reconfigurations, these should not be needed during steady-state operation of a real-time task. In fact, if needed at all, such operations should be delegated to a less-critical task that runs in slack time. The kernel provides mechanisms that support such delegation.

Point two requires that that rates of high-priority interrupts are limited, a standard assumption in real-time schedulability analysis.

The final kind of contribution is the most concerning. The kernel is designed for a mixed-criticality environment, in which non-real-time and untrusted tasks can make system calls. If the kernel takes too long to complete or preempt some of these system calls, it may substantially degrade real-time performance. The countermeasure is to limit which kernel operations can be performed by untrusted tasks; we will discuss the limits this imposes on system design in Section 3.2.5. The WCET figures we report for unconstrained systems assume that an interrupt which will release a high-priority task happens just as the kernel began the longest-running operation.

3.2.5 Using seL4 Security Features to Limit WCET

Long-running operations in the seL4 kernel may substantially degrade the real-time performance of the system. The ideal solution is to eliminate all such long-running operations, and analyse the system afterwards to prove they no longer exist. As an alternative, if we identify a small number of problematic operations, we can try to restrict their use.

Use of the seL4 API is restricted through its capability-based security model. Tasks require capabilities to individual kernel objects to perform operations on those objects. This system can be used to constrain the set of objects a task may ever use [SWG+11], for instance to create spatial separation between tasks. However, the only way to prevent a task performing a particular *operation* is to ensure it never has the appropriate capabilities.

This has implications for system design. The simplest way for a trusted supervisor to initialise the system is to distribute capabilities to *untyped* memory regions, which the client tasks may then use to create kernel objects of any type. This will typically permit client tasks to perform any kind of operation. The opposite approach is to keep all untyped capabilities in the control of the trusted supervisor or other trusted tasks, requiring untrusted clients to receive resources only via trusted channels. This ensures that access to complex operations can be carefully controlled. However, the downside of this approach is that it requires more complexity within the trusted components, especially if they must coordinate with clients to dynamically reconfigure the system. The trusted components may themselves need to be verified, so reducing their complexity is highly desirable.

Hardware support for binary virtualisation is now commonplace, and provides a useful compromise. A guest OS running within a virtual machine (VM) environment may dynamically reconfigure its virtual environment while the external configuration of the VM remains static. The static environment can be created by the trusted supervisor, which can then provide minimal support to the guest OS, while the guest OS may run arbitrarily complex legacy software environments. An seL4 variant supporting such virtualisation extensions exists, and its verification has commenced. More work remains to be done to consider the impact of such a platform on our timing analysis.

Another compromise we will consider is an *object size limit*. A task with a capability to an untyped memory range may create any object, as long as it fits within the untyped memory range. The initial supervisor can enforce a limit on the size of untyped memory ranges by first dividing the initial untyped memory objects before distributing them. Once divided, the untyped ranges cannot be combined again. This simple restriction permits tasks access to most of the kernel's API but prevents some problematic cases involving very large objects.

This gives us a number of hypothetical system configurations:

- A *static* system, where all configuration decisions must be made at startup, before entering real-time mode. User tasks may only use system calls to exchange messages. Various simple embedded systems running on seL4 use a static configuration. Various separation kernels [Rus81], including Quest-V in separating mode [LWM13] and MASK [MWT02], would enforce similar static restrictions. Virtualisation improves the usefulness of this configuration.

- A *closed* system, where user tasks are not given access to untyped capabilities, and may not create or delete kernel objects. Unlike in the static case, tasks may have capabilities to manipulate their address spaces. This use case was evaluated by Blackham et al [BSC+11] in their previous WCET analysis of seL4.

- A *general* system, where all operations are permitted.

- A system with an *object size limit*, as discussed above. All objects and capabilities in the system are known to fit within the maximum object size. We prove some assertions to support this configuration, which we will discuss in Section 3.5.1.

- A *managed* system design has been considered, where untrusted tasks have few capabilities themselves, but request additional operations via trusted proxies. This design is the most general, allowing the proxy to add any additional constraints to the kernel API. This approach may be useful in the future in implementing mixed-criticality systems on seL4. We will revisit the implications of such a system for timing analysis once a working example exists.

3.2.6 Verifying Preemptible seL4 Operations

The abort style of preemption used in seL4 (see Section 3.2.4) was chosen to simplify verification. No matter what style of preemption is chosen, the verification of a preemptible operation must consider three concerns:

1. *Correctness*: the usual requirement that the preemptible operation is functionally correct.

2. *Non-interference*: other operations that are running must not interfere with the safety and correctness of the operation.

3. *Progress*: the preemptible operation must eventually run to completion.

In most approaches to concurrency verification, it is the non-interference concern that is most complex. The key advantage of the abort style is that it avoids all concerns about interference. There is no need to calculate the atomic components of preemptible operations, instead, all kernel entries are fully atomic. There is no need to calculate what variables and references an operation has in scope while preempted, or consider the impact on these references when objects are updated or deleted elsewhere. Instead, a preempted operation will forget all references, and will rediscover its target objects and recheck its preconditions when it resumes.

These advantages make the verification of an abort-style preemptible operation straightforward. Compared to the verification of a non-preemptible operation, the only additional requirement is that the system is consistent (all system invariants hold) at each possible preemption point.

The downside of the abort style is that it complicates the design of preemptible operations. These operations must completely reestablish their working state when resumed after preemption, which might have substantial performance costs for long-running operations that are frequently preempted. The operations must also be designed to make it possible at all to discover how much work has already been completed. For instance, in this work, we add a preemption point to an operation which zeroes a range of memory. There is no efficient way to examine a partly-zeroed range of memory and decide where to resume the operation; information about progress must somehow be tracked in another object. Our solution to this problem is discussed in Section 3.5.2.

3.3 Related Work

3.3.1 WCET Analysis

WCET analysis is a broad field of research with a vast wealth of literature. The field has been broadly surveyed by Wilhelm et al. [WEE+08], and we refer the reader to their summary for a more comprehensive overview.

Standard strategies for WCET analysis include hierarchical timing decomposition [PK89, PS91], explicit path enumeration [LS98, HAWH99], and implicit path enumeration [LM95, BR06]. We reuse the Chronos tool [LLMR07] in this work, which follows the implicit approach.

Whichever core WCET approach is chosen, the analysis requires additional loop bound and path information, usually discovered by static analysis, frequently supported by user annotations. There is a vast diversity of possible static analysis approaches to this problem, and again we refer the reader to Wilhelm et al's survey [WEE+08]. In recent years, Rieder at al. have shown that it is straight-forward to determine some loop counts at the C level though model checking [RPW08]. Other authors use abstract interpretation, polytope modeling and symbolic summation to compute loop bounds on high-level source code [LCFM09, BHHK10]. These source level loop bounds must then be mapped to the compiled binary, for instance via a trusted compiler with predictable loop optimisation behaviour. We would like to avoid trusting the compiler as far as possible.

The aiT WCET analyser uses dataflow analysis to identify loop variables and loop bounds for simple affine loops in binary programs [CM07]. The SWEET toolchain [GESL06]

uses abstract execution to compute loop bounds on binaries, and is aided by tight integration with the compiler toolchain, which improves the knowledge of memory aliasing, but this again implies relying on the compiler. The r-TuBound tool [KKZ11] uses pattern-based recurrence solving and program flow refinement to compute loop bounds, and also requires tight compiler integration.

Some of the same techniques are used for eliminating infeasible paths, e.g. abstract execution [GESL06, FHL+01], with the same limitations as for loop-count determination. The earlier WCET analysis of seL4 used binary-level model checking [BH13] to automatically compute loop bounds and validate manually specified infeasible paths. The CAMUS algorithm was also used for automating infeasible path detection [BLH14]. However, this work was inherently limited to information that could be inferred from an analysis of the binary, and failed to determine or prove loop bounds that required pointer aliasing analysis.

3.3.2 Using Formal Approaches for Timing

In this work we reuse our formal verification apparatus to support our WCET analysis. While most WCET approaches are based on static analysis tools such as abstract interpretation [EaJGBL07, KZV09], we are aware of few other projects which address the questions of timing and functional correctness using the same apparatus.

The ambition of combining verification and WCET analysis was suggested by Prantl et al [PKK+09], who propose interpreting source-level timing annotations as hypotheses to be proven rather than knowledge to be assumed. The associated static analysis must verify the user's beliefs about the system's timing behaviour. This replaces the most error-prone aspect of the WCET analysis with a formally verified foundation. The challenge which remains is to discover some sound static analysis which is sufficient for verifying whatever annotations the user supplies.

Our analysis also interprets annotations/assertions as hypotheses to be proven (see Section 3.5.1). In our case the assertions are simple logical expressions, as used in Floyd or Hoare style program verification [Flo67, Hoa69]. It is the task of our WCET analysis to derive temporal properties from these simple stateful assertions. A more feature-rich version of this approach was suggested by Lisper [Lis05]. In their survey of WCET annotation styles, Kirner et al. [KKP+11] place this style in their "other approaches" category. It is interesting that this approach is considered unusual, while for us, approaching WCET analysis coming from formal verification makes the approach seem entirely natural. Perhaps this is because user-supplied assertions may require user-supported interactive verification. We are accustomed to doing such verification, but others may consider it prohibitively expensive.

The CerCo "Certified Complexity" project [AAB+13, AARG12] set out to produce a compiler that would produce provably correct binaries together with provably correct specifications of their execution time. The project followed in the footsteps of the CompCert certifying compiler [Ler09], building a compiler directly within a formal apparatus complete with proofs of correct translation and timing equivalence. The resulting execution time contracts can be extremely precise, especially since the project mostly targets simple microprocessors with predictable timing behaviour. Unfortunately this design makes compiler optimisations particularly complex to implement.

3.3.3 Verification

This chapter discusses the design and verification of a simple preemptible algorithm (see Section 3.5.2), with substantial design effort put into simplifying verification. Preemptible and concurrent algorithms have been of great interest to the field of formal verification for some while. This is partly because the verification of preemptible algorithms is challenging [Sch10,ALM15] and the verification of fully concurrent algorithms is extremely challenging [dRdBH$^+$01, FFS07, CS10, TVD14, GHE15].

Operating system kernels are also an attractive target for formal verification, given their small size and critical importance. Starting with UCLA Secure UNIX [WKP80] and the KIT OS [Bev89], a number of OS verification projects have been attempted, of which Klein [Kle09] has written a detailed overview. In addition to seL4 [KEH$^+$09], recent projects include the Verisoft project [APST10], the Verve kernel [YH10] and the CertiKOS project [GSC$^+$16].

Of these projects, the Verisoft and CertiKOS project are implemented in a similar manner to seL4, using restricted C-like languages, whereas Verve experiments with much higher-level language features. Using a high-level language simplifies producing a verified OS, but probably means that the timing behaviour of the system will be too unpredictable for real-time applications. The initial Verisoft project made very different simplifying design decisions. The project sought to develop and verify a complete system stack, including silicon architecture, operating system, compiler and end-user applications. To accomplish this, Verisoft implemented simplified versions of all of these layers, which introduces performance issues that would be an impediment in real-time use. The recent CertiKOS project has similar goals and design to seL4, and also tackles multicore design issues. Timing analysis for CertiKOS, including analysis of inter-core timing issues, would be an interesting and challenging project.

3.4 WCET Analysis

The design of the WCET analysis process is shown in Figure 3.2. We extend SydTV-GL-refine with new modules to extract the control-flow graph (CFG) of the binary, and to provably discover loop bounds. Chronos then reduces the WCET problem to an integer linear program. We solve the ILP and pass the worst-case path of execution to the infeasible-path module to be refuted. Given any refutations, we find a new worst-case path, continuing until the candidate path cannot be refuted.

This repeated process of examining a candidate worst-case path and refuting infeasible ones was performed by hand in some of our former work [BSC$^+$11], and generally reflects the "counterexample guided" approach to static analysis [CGJ$^+$03, HJM03]. The approach has also previously been used by Knoop et al [KKZ13].

The rest of this section explains the various components in detail.

3.4.1 CFG Conversion

In general, reconstructing a safe and precise binary CFG is difficult and error prone due to indirect branches [BHV11, KZV09]. In the previous analysis of seL4, the CFG was reconstructed from seL4's binary using symbolic execution [BSC$^+$11]. The soundness of the CFG so obtained, and thus the resulting WCET estimation, depended on the correctness of the symbolic execution analysis.

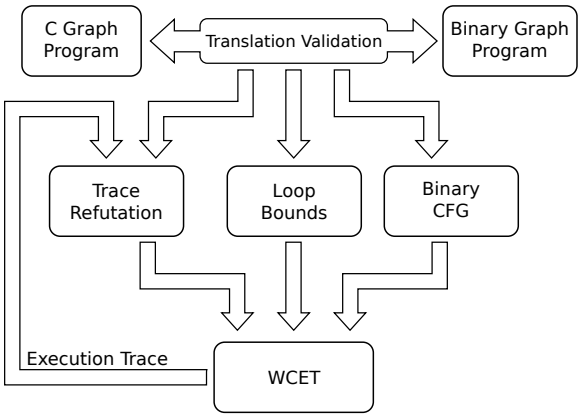


Figure 3.2: Overview of dataflow in the analysis.

We now present a high-assurance approach to construction of the CFG. The decompiler generates the graph-language representation of the binary program, *together with a proof* (in HOL4) that the representation is accurate. The representation consists of a collection of graphs, one per function, with both the semantics and the binary control flow embedded in the graph, and with function calls treated specially.

It is important for this project that each function in the decompiled program is structurally identical to the control-flow-graph of the relevant function in the binary, including sharing the same instruction addresses. To aid us in this work, Myreen adjusted the decompiler slightly to ensure that this was the case.

Chronos, in contrast, expects a single CFG in which function-call and -return edges are treated specially. The two representations are logically equivalent, and we perform the conversion automatically. The conversion also gathers instructions into basic blocks and removes some formal features, such as assertions, that are not relevant to the binary control flow.

In principle, the conversion could be done inside the decompiler, and we could formalise the meaning of the CFG and prove it captured the control paths of the binary. However, this makes the relationship between the decompiler and SydTV-GL-refine framework more complicated, and we leave this to future work. Instead we perform the simplification inside a module of SydTV-GL-refine for now. While this means that the CFG is not proved correct, it is still highly trustworthy, since the most difficult phases have been performed with proof.

3.4.2 Discovering and Proving Loop Bounds

We employ two primary strategies for discovering loop bounds on the binary. Both reuse existing features of SydTV-GL-refine. The first constructs an *explicit model of all possible iterations* of the loop, while the second *abstracts over the effect of loop iteration.*

Consider this simple looping program:

```
for (i = 0; i < BOUND; i ++) {
    x += val[i];
    /* ... */
}
```

The *explicit strategy* for discovering a loop bound is to have the SydTV-GL-refine

export an SMT model[1] of the program including values of `i`, `x`, etc, for each iteration of the loop up to some bound. The model includes state variables for each step in the program, and also a path condition. Loop bounds can be tested by testing the satisfiability of various path conditions, e.g. a bound of 5 will hold if the path condition of the first step of the 6th iteration of the body of the loop is unsatisfiable.

This approach is simple and fairly general. We can analyse complex loops by considering, in SMT, all possible paths through them. However the size of the SMT model expands linearly with the size of the hypothetical bound. As SMT solving is, in general, exponential in the size of the problem, this approach is limited to loops with small bounds. In practice we have been able to find bounds up to 128 this way.

If we suppose `BOUND` in the loop above is 1024, the explicit approach would be impractical. However, it is intuitively clear that this simple loop stops after 1024 steps, because variable `i` equals the number of iterations (minus 1) and must be less than 1024. The *abstract strategy* replicates this intuition.

For this strategy we have the SydTV-GL-refine generate an SMT model for proof of loop equalitites by induction. This includes all the program up to and including the first iteration of the loop, and then fast-forwards to some symbolic n-th iteration, and includes the next iteration or two after that. The variable state at the n-th iteration is unknown. In the above example we can prove that `i` is one less than the iteration count at every iteration. We prove that it is true in the initial iteration, and then, assuming that it is true at the symbolic iteration n, we prove it is true at iteration $n + 1$. This is a valid form of proof by induction, and is closely related to the split induction done by SydTV-GL-refine for matching related loops in the source and binary (see Section 2.1.9).

This strategy applies equally well at the binary level. Consider this disassembled binary code fragment:

```
e1a00004    mov    r0, r4
ebffffe     bl     0 <f>
e2844004    add    r4, r4, #4
e3540c01    cmp    r4, #256    ; 0x100
1affffa     bne    4568 <test+0x8>
```

This code is a loop which increments register `r4` by 4 at every iteration. We can prove by induction in the above manner that the expression `r4`$-4n$ is a constant, where n is the iteration count as above.[2] We reuse the SydTV-GL-refine static analysis which discovers these linear series (see Section 2.3.4).

The above example is complicated by the looping condition, which is `r4 != 256` rather than `r4 < 256`. We show the additional invariant `r4 < 256` by induction. The abstract strategy contains a feature for guessing inequalities of this form that may be invariants. It assembles these inequalities by inspecting the linear series and the loop exit conditions, and then discovers which of its guesses can be proved by induction. In this example, the proof requires the knowledge that the initial value of `r4` was less than 256 and divisible by 4.

Once we have the inequality `r4 < 256`, the loop bound of 64 can be proved easily. Any larger bound will also succeed, which is convenient, because it allows us to refine any

[1] In this chapter we use "SMT model" to mean a set of definitions in the SMT language, used to phrase a satisfiability query, rather than a satisfying model of such a query.

[2] This expression is constant at each address in the loop. If the initial value of `r4` were 4, the expression would be evaluate to the constant 0 whenever execution was at the first two instructions, but 4 after the `add` instruction.

bound we guess down to the best possible bound by means of a binary search. The SMT model does not change from query to query during this search, only the hypothesis that fixes n to some constant. SMT solvers supporting incremental mode can answer these questions very rapidly.

These two strategies do all the work of finding loop bounds, but as presented are not powerful enough for all loops. We extend them in three ways to cover the remaining cases: (i) using C information, (ii) using call-stack information, and (iii) moving the problem to the C side.

The first extension, *using C information*, exploits correctness conditions in the C program while reasoning about the binary. This works because the refinement proof created by SydTV-GL-refine establishes that each call to a binary symbol in the trace of execution of a binary program has a matching C function call in a matching trace of C execution.

Consider, for instance, these C and binary snippets:

```
int
f (int x, int y) {
    x += 12;
    /* ... */
    return 2;
}

0000f428 <f>:
f428: e92d4038 push  {r3, r4, r5, lr}
f42c: e1a05001 mov   r5, r1
f430: e280400c add   r4, r0, #12
      ...
f464: e3a00002 mov   r0, #2
f468: e8bd4038 pop   {r3, r4, r5, lr}
f46c: e12fff1e bx    lr
```

The calling convention relates visits to the two functions `f`. A binary trace in which address `0xf428` is visited three times will be matched by a C trace in which `f` is called at least three times, with the register values `r0`, `r1` matching the C values `x`, `y` at the respective calls.[3] This is established by SydTV-GL-refine, so the WCET analysis can consider this C execution trace simultaneously with the binary execution trace. Concretely this means that SMT problems will contain models of both binary `f` and the matching C `f`. The correctness conditions of the C `f` will be taken as assumptions. The `x += 12` line in `f` above, for instance, tells us that adding 12 to either `x` or `r0` must not cause a signed overflow.

The second extension, use of *call-stack information*, is useful in the case where the bound on a loop in a function is conditional on that function's arguments. Common examples include `memset` and `memcpy`, which take a size parameter, `n`, which determines how many bytes to loop over. To bound the loop in `memset`, we must look at the values given to `n` at each of its call sites. We might in fact have to consider all possible call stacks that can lead to `memset`. Concretely this means that the SMT model will also include a

[3]The story is a little more complex. Some calls to `f` in the source code may not be present in the binary thanks to inlining, and functions which are known not to inspect memory may be moved across memory updates, or each other. A more accurate explanation is that SydTV-GL-refine establishes a collection of pairings, and proves that the binary trace has a matching C trace which matches calls across these pairings.

model of the calling function up to the call site, and the input values to `memset` will be asserted equal to the argument values at the call site. This additional information then feeds into the two core strategies above.

The final extension, *moving the problem to the C side*, maximises the use of SydTV-GL-refine, by asking it to relate the binary loop to some loop in the C program. If SydTV-GL-refine can prove a synchronizing loop relation, that implies a relation between the C bound and the binary bound. The explicit and abstract strategy can then be applied to the C loop to discover its bound. It is convenient that both programs are expressed in SydTV-GL, so we can use exactly the same apparatus. Finding the C bound will sometimes be easier because dataflow is more obvious in C. It also ensures that assertions placed in the body of the C loop will be directly available in computing the loop bound.

By default the apparatus will set up an SMT model which includes the target function and the matching C function. If the function is called at a unique site, we also include its parent and its parent's matching C function. If no bound is found directly, we try to infer a bound from C. If this also fails, we add further call stack information as necessary, by considering all possible call stacks that can lead to our loop of interest.

3.4.3 Refuting Infeasible Paths

Refuting an impossible execution path amounts to expressing the conditions that must be satisfied for the execution to follow that path, and testing whether all those conditions are simultaneously satisfiable. SydTV-GL-refine about path conditions by converting them into boolean propositions in the underlying SMT logic. It is then straightforward to have the SMT solver test whether a collection of path conditions is possible.

To narrow the search space, we only attempt to refute path combinations that appear in a candidate execution trace. The final ILP solution produced by running Chronos and CPLEX specifies the number of visits to each basic block, and the number of transitions from each basic block to its possible successors. Since some basic blocks will be visited many times, with multiple visits to their various successors, we may not be able to reconstruct a unique ordering of all blocks in the execution. Instead, we collect a number of smaller arcs of basic blocks that must have been visited together in a single call to a function. We can also link some of these arcs with arcs that must have occurred in their calling context.

The refutation process then considers each of these arc sections, and checks whether they are simultaneously satisfiable as described above. If the combination is unsatisfiable, we reduce it to a single minimal unsatisfiable combination, and export an ILP constraint equivalent to this refutation.

This approach is simpler than the previous analysis of seL4, which considered much larger sets of path conditions and used the CAMUS algorithm to find all minimal conflicts [BLH14]. The trade-off is that, after eliminating refuted paths, we have to re-iterate the process on the next candidate ILP solution. We believe this approach will usually be more efficient, since the candidate solutions will probably converge on the actual critical path quickly and we will consider only a small fraction of the path combinations of the binary. There is however the possibility, which we have not yet encountered, that the cost of repeated ILP solving will outweigh the benefits of this approach.

3.4.4 Manual intervention: Using the C model

The techniques described in the two preceding subsections discover loop bounds and refute infeasible paths automatically. In cases where these fail, we can manually add (and prove) relevant properties at the C level. Besides the assurance gained by the formal, machine-checked proofs, our ability to leverage properties that can be established at the C level is a powerful tool that most distinguishes our approach from previous work, including former work on seL4 [BLH14].

In Section 3.4.2 we discussed how C correctness conditions, such as integer non-overflow, can be assumed in the WCET process, by constructing simultaneous SMT models of the C and binary programs. Manual assertions added to the C program appear in exactly the same manner as these standard assertions arising from the C standard. However, the manual assertions we supply can be directly related to the WCET problem.

For ordinary (application) programs, such as the Mälardalen benchmarks, we assume that the source conforms to the C standard, specifically that it is free of unspecified or undefined behaviour. This allows SydTV-GL-refine to assume some pointer-validity and non-aliasing conditions which derive from the C standard, but would be hard to discover from the binary alone. While this implies a potentially incorrect WCET for non-standard conformant programs, standard conformance is essential for safety-critical code, and can (and should!) be verified with model-checking tools. In fact, industry safety standards, such as MISRA-C [MIS12], which is mandatory in the transport industry, impose much stronger restrictions.

Additionally, the C-to-Isabelle parser provides syntax for annotations in the form of specially-formatted comments, which add assertions to the C model. This feature is used occasionally in seL4 for technical reasons to do with the existing verification. We can reuse this mechanism to explicitly assert facts which we know will be of use to the loop-bound and infeasible-path modules. The assertions create proof obligations in the existing proofs, which must be discharged, typically by extending the hand-written Isabelle proofs about the kernel. We will describe our changes to the kernel, and its verification, in the following section.

This same mechanism can be used for application code, if an assertion can be known with certainty (eg. by proving it through model checking).

3.5 Improving seL4 WCET

The seL4 kernel is designed for a number of use cases, including a minimal real-time OS. While the kernel's design broadly supports this use case, a number of non-preemptible operations are known to have long running times, which is a problem for timeliness. It was previously shown that by adding further preemption points to the kernel we can reduce its WCET to a level competitive with a comparable real-time OS [BSH12]. Unfortunately these modifications increase the complexity of some operations dramatically, impacting average-case performance and complicating verification.

This section describes two modifications we have made to verified seL4 to improve its WCET bound. Firstly, we add a number of assertions to the source code, supporting our WCET analysis as described above. These changes have all been incorporated into the official verified seL4 as of its release at version 2.1. Secondly, we pick one of the preemption points added in the previous work [BSH12], adapt it to the current kernel design, and adjust the formal verification accordingly. This is a significant step toward competitive WCET for the verified seL4 kernel.

3.5.1 Assertions

We add 23 source assertions to the kernel source to support the WCET analysis. With these manual interventions, we can calculate and prove *all* loop bounds[4] in the seL4 kernel binary, and eliminate the WCET-limiting infeasible paths. We add assertions of five kinds.

1. We add an assertion that the "length" field of a temporary object is at maximum 16. This information actually exists in the binary, but to find it the WCET process would have to track this information across several function calls. Instead, we propagate this information through the preconditions of several proofs about the C program. While manual, this process is not particularly difficult.

    - There are 4 annotations of this kind.

2. We assert that each iteration of a lookup process resolves at least one bit of the requested lookup key. The kernel uses a *guarded page table* [Lie94] for storing user capabilities, in which each level of the table resolves a user-configured number of bits. It is an existing proved kernel invariant that all tables are configured to resolve a positive number of bits, thus, the loop terminates. The assertion is trivial to prove from this invariant. Thus, the assertion transports the invariant into the language of the WCET apparatus.

    - There is 1 annotation of this kind.

3. We assert that a capability cleanup operation performed during the exchange of so-called reply capabilities cannot trigger an expensive recursive object cleanup. Capability removal is the trigger for all object cleanup in seL4, however, this cleanup operation targets a dedicated reply slot which can only contain reply capabilities. This is the same information that we have in previous work provided to the compiler to improve optimisation [SBH13].

    - This requires 7 annotations, six at the call sites of the capability cleanup operation, and one within the operation.

4. We assert that the number of bytes to be zeroed in a call to `memzero` is divisible by 4 (the word length on our 32-bit platform). This implementation of `memzero` writes words at a time and decrements the work remaining by the word length. The stopping condition is that the work remaining is zero, which requires divisibility to be reached.

    - This is the only annotation of this kind.

5. We assert that various objects are smaller than a configurable maximum size parameter. We do not specify in the seL4 source code what this parameter is. In particular we establish that a number of zeroing and cache-cleaning operations cover fewer bytes than this maximum size.

    - There are 10 annotations of this kind.

The final assertion above is needed to address a WCET issue with the present verified kernel version. The seL4 kernel allows a user level memory manager to use the largest

[4] Some loops in the binary are preemptible and do not have bounds.

available super-page objects (16 MiB) if it has access to sufficiently large blocks of contiguous memory. Zeroing or cache-cleaning these pages are very long running operations. The (trusted) initial user-level resource manager can avoid this issue, by intentionally fragmenting all large memory regions down to chunks of some given size.

This fragmentation may add modest overheads. Subsequent resource managers will have to perform more operations, and cannot employ super-pages. However this will not create any further complications for application code.

We argue that the initial manager can ensure a size limit. To formalise this argument, we prove as an invariant across all kernel operations that all objects are smaller than the configurable size limit, which establishes the assertions. This invariant holds for any given size limit, onwards from the first point in time that it is true. Thus, once the initial resource manager configures the system appropriately, the invariant remains true for the system lifetime. The resource manager may choose what size limit to set. For the WCET analysis, we will assume a particular value for the limit, in this case 64 KiB.

This configurable value, and our assertion that it equals 64 KiB, are "ghost data" added to the C program. The actual C program and binary do not manipulate this variable anywhere, but the Isabelle model contains all the assertions about it.

Should the initial configuration violate the constraint, the system's operation will still be functionally correct, but the WCET bounds are no longer guaranteed.

Note that since all four types of manual assertions are specified at the source level, they will still be available if the kernel is re-compiled. We do not expect to have to add further annotations until major code changes require them. The compiler might, however, move information out of scope by changing the inlined structure of the binary, which might require further manual intervention. Clearly, in any case, the WCET analysis must be rerun on each actually-deployed binary.

3.5.2 Design of Preemptible Zeroing

We want to achieve the best possible WCET for a fully verified kernel. Ideally we would accomplish this by incorporating all the prototype changes made in the previous WCET analysis of seL4 [BSH12] into the verified version. As a first step towards this, we incorporate and verify one major change: making object creation preemptible. This allows the kernel to create large objects (e.g. 16 MiB super pages) without compromising its WCET.

Objects are created as part of the seL4 `invokeUntyped_Retype` operation. This is an operation on a so-called *untyped* memory region, a range of kernel memory available to user-level resource allocators to create various kinds of kernel objects. The Retype operation may both remove old objects from an untyped region and create new ones. Creating new objects mainly involves zeroing the relevant memory. The removal of old objects only impacts the verification picture of the kernel memory, as the objects must be unreachable to the implementation already.

To make the Retype operation preemptible, we split the creation phase into two phases, the first zeroing all the relevant memory, the second doing the necessary object setup. The preemption point is inserted in the zeroing phase. Object setup given zeroed memory is fast enough even for large objects. Zeroing a large range of memory in blocks and adding a preemption point is straightforward except for the problem of ensuring progress.

Ensuring progress is the challenging aspect of seL4's abort-style preemption model (see Section 3.2.6). Some long-running operations, such as emptying a linked list, can be preempted and resumed easily. The resumed operation continues unlinking elements from the list in exactly the same manner as the initially aborted operation. In fact, there is

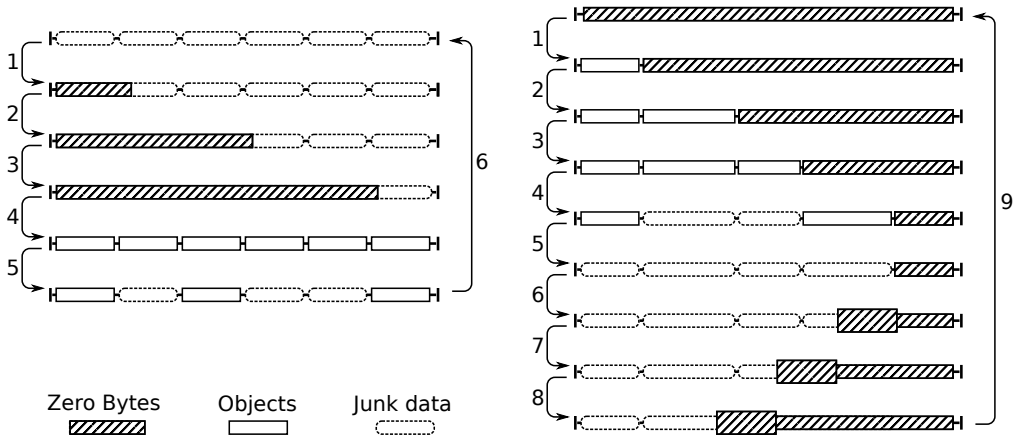


Figure 3.3: Preemptible Retype designs from previous and current work. Previous steps (left): Starting state is junk data. 1-3: Preemptibly zero the region. 4: Complete zeroing and create new objects. 5-6: Objects become unreachable over time. New steps (right): Starting state is zeroed region. 1-4: Objects are created in separate system calls, and may also expire. 5: All remaining objects expire. 6-9: Preemptible zeroing of the region, one chunk at a time.

no need to detect that the operation was previously begun and aborted. Zeroing a large region, however, cannot be efficiently resumed without some knowledge of how much memory has already been zeroed. Adding preemption points to operations of this kind requires storing more information about the progress of the operations within the objects being manipulated. This additional information, and its consistency requirements, then complicates the rest of the implementation and verification.

The Retype operation can scan a capability-related structure to determine whether all the objects in the untyped region have become unreachable. The first-ever Retype implementation would check the region was reusable, then fill the region with newly initialised objects in a single step. In the previous work [BSH12] on this operation, it was adjusted to be preemptible by using a spare word to store a count of how much of the untyped region had been zeroed out. The Retype operation would preemptibly expand this zero region and then fill it with new objects. This implementation is sketched on the left side of Figure 3.3.

Unfortunately this spare word is no longer spare. The seL4 API has been updated to allow untyped regions to be used incrementally, and the additional word now measures the amount of space still available. The incremental Retype implementation allocates new objects from the start of this available space, except in the case where it can detect that all objects previously in the untyped range have expired, in which case it resets the untyped range and begins from the start.

We want to support both incremental allocation and incremental initialisation, but we have only one spare word available. The key insight to solving this problem is to make the available space and the zeroed space the same. Untyped objects continue to track the amount of space available for new objects, but now the space available (for use) is also the part that is known to be zeroed – a new system invariant. The special case of the Retype operation where the untyped range can be reset must now zero the contents of the untyped range as well as marking it available. The zero bytes are also flushed from the cache to main memory.

A peculiarity of this design is that the zeroing happens backwards. The existing API specifies that objects are created forward from the beginning of the untyped range, so the available range is always at the end. Thus the zeroing process, which expands the available range, must begin at the end of the range and proceed towards the start. In fact we subdivide the region to be zeroed into chunks (with a default size of 256 bytes) and zero the chunks in reverse order but each individual chunk in forwards order, for better expected cache performance.

3.5.3 Verification of Preemptible Zeroing

The implementation of the preemptible zero operation is straightforward, requiring the addition of 62 lines to seL4's C code and the removal of 56. Roughly half (32 lines of C) of the addition is the new preemptible zero function, and roughly half (24 lines) of the lines removed were memory zero and cache clean function calls within the creation routines for various specific object types. We make similar modifications to the two higher-level specifications of seL4.

The verification of these changes, however, is far more involved. The final changeset committed to the proofs requires roughly 20,000 lines of changes (`diff` reports 147 files changed, 11,805 insertions, and 9,390 deletions.) This required 9 weeks of work, and we found it a challenging project, despite extensive former experience with the seL4 proofs.

The main reason the verification is so complex is that the Retype process has some of the most involved proofs in the kernel. Most operations manipulate one or two objects at a time, preserving the types of all objects, whereas Retype not only changes types, but it requires several component operations to accomplish this (clearing the region of old objects, updating the untyped range, creating new objects, issuing caps to them, etc). The new proof of invariant preservation for Retype, for instance, is assembled from 31 different sub-lemmas about the component operations. One of these sub-lemmas concerns the new preemptible zero operation. In addition to adding this lemma, the proof structure had to be substantially modified.

We must also verify a new invariant, that the available section of each untyped range of memory is zeroed. Similar invariants in seL4 are proven at the specification levels, and apply to the implementation thanks to the functional correctness proof. Unfortunately this is impossible for this invariant, since the specifications do not accurately track the contents of the relevant memory. Different regions of memory are treated differently in the kernel's specifications. Memory shared with user tasks is represented as-is. Memory used by kernel objects is represented by abstractions of those objects, so the specifications do not need to specify the in-memory layout of these objects.

However, the memory in the available untyped ranges is neither covered by kernel objects nor shared with users. Thus we cannot prove anything about it using the existing specifications. To address this, we add a field to the specification state which tracks the locations of the untyped ranges expected to be zeroed, and require memory there be zeroed as an additional component of the state relation between the specification state and C memory model. This complex approach then requires numerous changes to the proofs.

After the verification of this change was completed, it was included in the official seL4 development version (see `https://github.com/seL4/seL4/commits/03c71b6`). This change also appears in official seL4 relases from 4.0.0 onwards.

3.6 Evaluation and Discussion

3.6.1 Loop Bounds in seL4

We successfully compute the bounds of all 69 bounded loops in seL4 version 3.1.0, which is in contrast to earlier work on seL4, which only succeeded on 18 of 32 loops[5] (56%) [BH13]. A further 5 loops in the binary contain preemption points and have no relevant bound, these are bounded by the preemption limit, as discussed in Section 3.2.2.

Computing all the bounds in the kernel demonstrates that our approach is sufficient for a real-world real-time OS.

To more thoroughly investigate our WCET apparatus and our kernel modifications, we go on to analyse three different versions of seL4, and six different WCET problems:

- *3.1.0-64K:* The standard verified kernel, as of version 3.1.0, with all system calls enabled and a 64 KiB object size limit (see Section 3.5.1).
- *3.1.0-static:* The standard verified kernel, version 3.1.0, in a "static" system configuration with most complex system calls forbidden.
- *preempt-64K:* Our branch of the kernel, with preemptible zeroing for object creation, as discussed in Section 3.5.2, with a 64 KiB object size limit.
- *preempt-nodelete:* Our branch of the kernel, with no object size limit. This is not exactly a "static" variant, since creation of new objects of arbitrary size is allowed. However deletion of objects, and various cache management operations, are forbidden.
- *rt-branch-64K:* The "RT" branch of seL4 as of version 1.0.0. This is an officially maintained but experimental version of seL4 which introduces a more powerful and principled scheduling and timing model [LH16], designed to provide better support for mixed-criticality systems. We assume a maximum of 10 scheduling contexts and also impose a 64 KiB object size limit. As scheduling contexts represent independent (asynchronous) threads of execution, 10 seems a reasonably limit for most critical real-time systems, although it is likely too restrictive for mixed-criticality systems. We will revisit these bounds when analysing the advanced real-time features more thoroughly.
- *rt-branch-static:* The same RT branch, version 1.0.0, in a "static" configuration without complex system calls, and with a maximum of 10 scheduling contexts.

We want to demonstrate a number of points through these studies. Firstly, we want to show that the WCET apparatus we have built works for a number of cases and a realistic system. We also want to show that the current kernel can achieve modest WCET performance goals if some limits are placed on the way its API is used, and that planned adjustments to the scheduling API will not invalidate this. Finally, we show that the verification we have done of new preemption points can be used to allow more of the kernel's API to be exercised without compromising WCET. In future work we hope to complete and combine all of these endeavours, resulting in a verified OS with a general API, predictable real-time scheduling behaviour and a competitive WCET.

[5] Note that the total number of loops here is higher than in earlier work. This results from this work targeting the verified kernel, and thus using preemption points less aggressively, see Section 3.2.2.

Note that the "static" and "nodelete" variants have identical source and binary to the more permissive environments. For the latter, we exclude a large fraction of the binary from analysis by assuming that certain functions in the binary will never be reached. Thanks to seL4's capability-based access control, it is possible for the initial supervisor to enforce these restrictions (this was discussed in Section 3.2.5). The loop analysis and infeasible path analysis use these limitations to quickly exclude loops and refute paths.

In all these configurations, we also make one change to the kernel's standard build-time configuration, to adjust a configurable limit called the "fan-out limit" to the minimum. We make this change everywhere, but it is irrelevant for the "static" configurations. This avoids an issue involving a nested loop with a complex bounding condition.[6] We compile the kernel with `gcc-4.5.1` with optimisation setting `-O2`, which is the default for building the kernel.[7] Finally, we remove the `static` keyword from a small number of sites. This prevents GCC from inlining so other functions into a single symbol that the resulting block runs for several thousand instructions and dominates the analysis time.

seL4 version	3.1.0				Preemptible				RT			
configuration	general		static		general		nodelete		general		static	
Explicit model	20	28%	8	11%	18	29%	11	17%	19	24%	7	8%
Abstraction	42	60%	5	7%	35	56%	8	12%	42	53%	5	6%
From C	1	1%	1	1%	1	1%	1	1%	1	1%	1	1%
Call cases	1	1%	0	0%	0	0%	0	0%	3	3%	2	2%
Skipped (preemption etc)	6	8%	56	80%	8	12%	42	67%	8	10%	58	74%
Not found	0	0%	0	0%	0	0%	0	0%	5	6%	5	6%
Total	70		70		62		62		78		78	

Table 3.1: Loop bounds found by different strategies.

The success rates of the strategies discussed in Section 3.4.2 are listed in Table 3.1. The explicit strategy typically discovers smaller bounds, and the abstraction strategy finds all the higher bounds, which vary from 16 up to 8192. The exception is a bound of 32 discovered by the explicit strategy on the C program. This is the capability lookup loop, manually annotated, which we discussed in Section 3.4.4. This bound is transferred across the discovered split relation to bound the binary loop implicitly. A small number of loops cannot be bounded without considering each case of their calling contexts individually. This is reported in our framework as a fourth strategy, and the subproblems are always solved by the two main strategies. We have not investigated which solvers solve the (small collection of) subproblems.

The "static" and "nodelete" variants exclude far more loops as unreachable. Loops containing preemption points are also detected and excluded by the same mechanism.

Some of the loops in the "RT" branch of the kernel are limited by the number of scheduling contexts, or other limits related to system configuration. These bounds could be discovered if appropriate annotations were added, using similar configurable limits to the object size limit. However, the "RT" branch is still in a rapid development phase, with further major code changes expected. Once the branch is more mature and verification is underway, we will carefully address the loop bound issue. Until then, we let the bound discovery process fail, and manually add appropriate loop bounds based on the symbol

[6] The minimum setting, 1, eliminates the outer loop entirely.

[7] Higher optimisation settings usually result in larger binaries, and instruction cache pressure is known to be an important factor in microkernel performance.

names of the binary functions in which the loops appear.

3.6.2 Loop Analysis Timing

The analysis time for each loop differs greatly, with the explicit strategy discovering small bounds in under a second in some cases and some analysis attempts taking several minutes. To investigate this further, we have timed the loop analysis for the six configurations above.

	Strategies		Setup		Unaccounted		Total	
3.1.0-64K	4,055 s	84.0%	201 s	4.2%	573 s	11.9%	4,828 s	(1:20:28)
3.1.0-static	757 s	53.4%	102 s	7.2%	560 s	39.4%	1,419 s	(0:23:39)
preempt-64K	2,835 s	63.8%	185 s	4.2%	1,424 s	32.0%	4,444 s	(1:14:04)
preempt-nodelete	1,078 s	62.3%	106 s	6.1%	546 s	31.6%	1,730 s	(0:28:50)
rt-branch-64K	12,014 s	58.5%	349 s	1.7%	8,164 s	39.8%	20,527 s	(5:42:07)
rt-branch-static	9,849 s	54.1%	248 s	1.4%	8,101 s	44.5%	18,197 s	(5:03:17)

Table 3.2: Loop analysis time breakdown.

The overall running time for the six variants varies enormously, between 20 and 90 minutes for the versions similar to seL4 3.1.0, and far longer for the experimental RT branch. The majority of the running time is spent in the various loop analysis strategies, as listed in Table 3.2, with a small minority of the time measured spent preparing analysis problems in SydTV-GL-refine and otherwise unaccounted for. All timing is done on a desktop machine with an Intel i7-4770 CPU running at 3.40GHz and 32 GiB RAM.

The analysis time is dominated by the execution time of the explicit and abstract strategies, which is itself dominated by time spent running the SMT solvers. SMT solving time is known to be exponential in the worst case and otherwise difficult to estimate. The analysis runs on each loop separately, with broadly linear complexity in the number of loops to be analysed. However, the analysis time varies enormously between loops. We hypothesise that larger and more complex loops, and larger and more complex SMT problems, are contributors to analysis time. More complex SMT problems are in turn created by complex loop *contexts*: the total size of the function the loop is in and any other functions from the calling context. Small bounds found by the explicit strategy are also discovered more quickly than larger bounds found by the abstract strategy, and falling back to the more complex strategies takes longer again.

The scatter plots in Figure 3.4 test these hypotheses. They compare loop analysis running time to the number of instructions in each loop, an its function and in its whole calling context. These correlations go some way toward explaining expected running times.

The clearest indicators of the analysis time variation are the eventually discovered bound and the eventually successful strategy. The plots in Figure 3.5 clearly indicate this. Small bounds can be discovered by the explicit strategy with only a couple of SMT solver invocations. The abstract strategy must first discover and prove a number of inductive invariants before making further progress. The bound transfer strategy is more complex again, as is considering various possible calling contexts. Not only are these final strategies more expensive, they are run only once the previous strategies have run and failed. Considering calling context cases does not appear in Figure 3.5 as the relevant statistics contain timing for each of the subproblems instead. The most consistently expensive

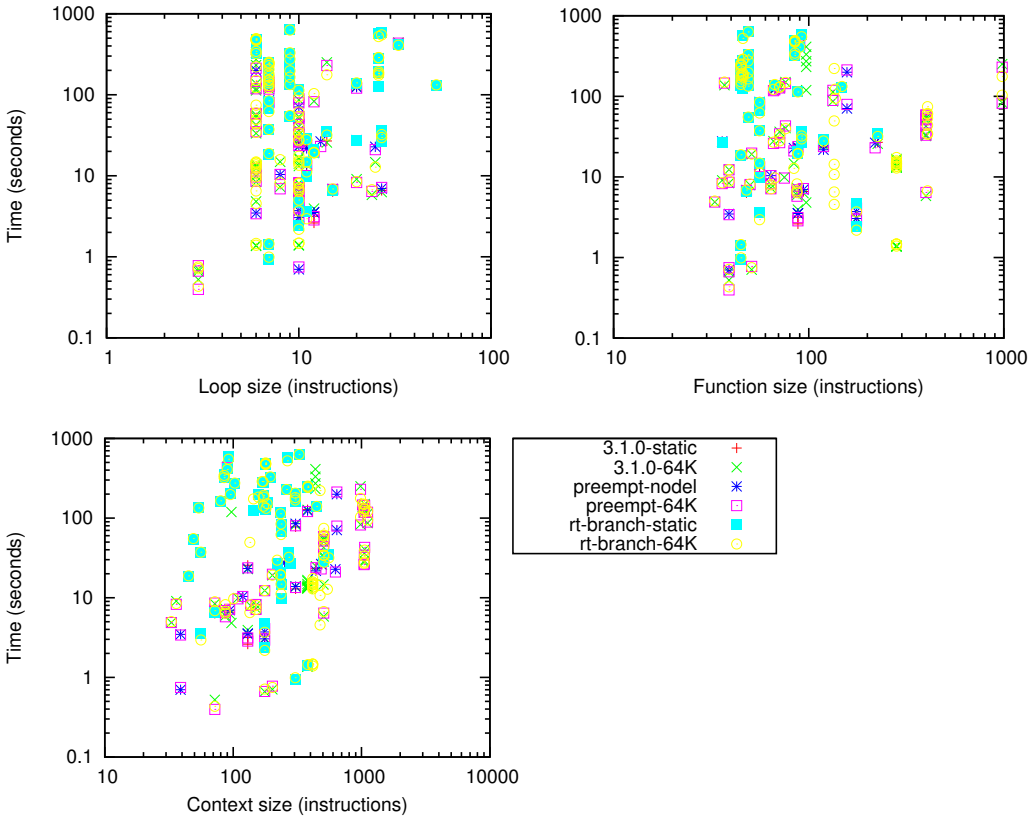


Figure 3.4: Correlation of loop, function and context size to analysis time.

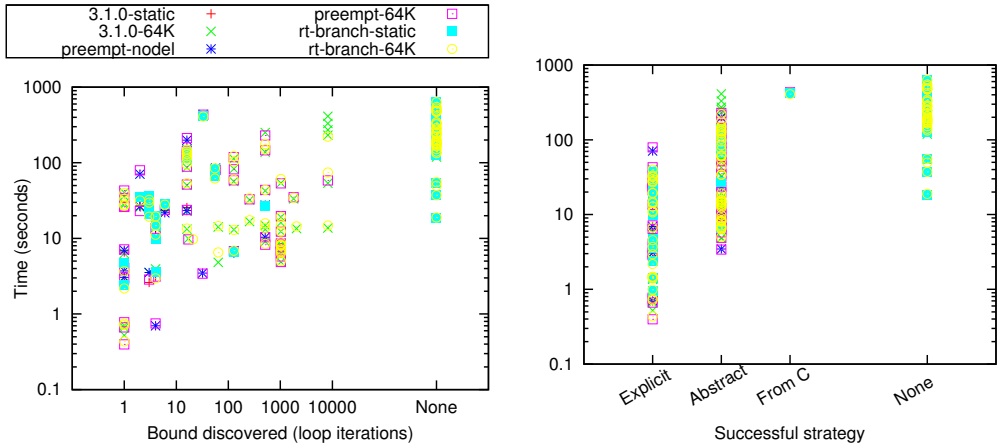


Figure 3.5: Correlation of loop bound and successful strategy to analysis time.

strategy is failure: running all strategies and failing to discover a bound explains many of the most expensive outliers.

We have not yet provided source annotations to the "RT" branch to bound some loops whose iteration limit depends on system configuration. Once the "RT" branch matures further and verification begins, we will add the relevant annotations. Until then we allow the process to fail. The time cost of failure of all the available strategies largely explains the slow analysis time for the "RT" branch.

It is a downside of our implementation that analysis time is reasonable once the program is sufficiently annotated, but the initial process of discovering which annotations to add can be far more expensive.

3.6.3 Binary-Only Analysis and Comparison to Previous Work

	This work				Prior work	
	Full analysis		Binary-only		[BH13]	
Explicit model	25	35%	16	22%	N/A	
Abstraction	42	60%	27	38%	N/A	
From C	1	1%	0	0%	N/A	
Call cases	1	1%	5	7%	N/A	
Excluded	1	1%	0	0%	N/A	
Total found	70	100%	48	69%	18	56%
Not found	0	0%	22	31%	14	44%

Table 3.3: Loop bounds found without C-level information.

For comparison to previous work, we reran the analysis of the "static" case of seL4 3.1.0 with all C-level information discarded, only using information available in the binary. The results are in Table 3.3. This mode makes more use of the last-resort strategy of finding loop bounds by considering multiple calling contexts. We speculate that this approach was needed less often in the previous analysis because assertions we provide through the C code usually make this step redundant. In total we find 48 of 69 bounds (70%) using only information from the binary. This is a slight improvement on the level of coverage we achieved in our earlier work (56%), possibly because the abstract strategy

can discover some large bounds more easily than our previous approach, or because of differences in the placement of preemption points. The reason the binary-only strategies fail to find the remaining bounds are the same as in our earlier work: inability to perform a *memory aliasing analysis* on the binary and the lack of an *invariant maintained by a loop's environment.*

3.6.4 Annotation Reuse

The advantages of source-level annotation became obvious when re-running the analysis repeatedly. We began our work on a development version of seL4 prior to version 2.1.0, and have now repeated our analysis for a variety of successive versions, with many intermediate changes. This includes a major maintenance patch which adjusts over 500 source lines. The source level annotations were preserved across this adjustment, so, when we switched versions, the automatic analysis immediately rediscovered all but one of the expected loop bounds. The failure was because we had rebuilt the kernel binary having forgotten to adjust the kernel build parameters as mentioned above. This failure is also somewhat reassuring: the process is robust, in that the analysis checks the assumptions we are making and will report failures if changes to the code base invalidates them.

3.6.5 Loop Bounds in the Mälardalen Suite

We use the Mälardalen WCET benchmark suite [GBEL10] to further characterise the effectiveness of our approach. A similar evaluation for the earlier tools we build on was performed before by Blackham et al [BH13]. As in that evaluation, we compile the C sources for the ARMv6 architecture, with gcc (4.5.1) and the `-O2` optimisation setting, and omit benchmarks using floating point arithmetic. Floating point arithmetic is not presently supported by our C semantics nor the Cambridge processor model (see Section 3.6.9).

The results are listed in Table 3.4. We must also omit a number of benchmarks which were attempted in the previous work. The current design depends on the C parser and SydTV-GL-refine to handle both the C and binary resulting from each test problem. We skipped some tests which employed the `goto` statement, took references to local variables, or made extensive use of side-effecting operators such as `«=`, `*p++`, none of which are in our verification C subset. We also skipped some tests which involve nested loops, which SydTV-GL-refine does not yet handle, or involve complex recursion.[8] We also reject some use of padding in memory, but this was not an issue for the remaining benchmarks. Finally, we skip the `ndes` test, which exposes an issue in the decompiler's stack analysis causing it not to terminate.

This highlights the tradeoff inherent in our approach. The translation validation apparatus is clearly worth making use of, if we assume that it has already been successfully applied to our target program. Likewise if there is a proof document, we should be making use of the facts in it. The more tools we depend on, however, the more constraints we put on the target program for all the tools to succeed. The seL4 kernel was designed with the source verification in mind, and only needs slight adaptations for translation validation.

We discovered an interesting anomaly with the "bs" and "bsort100" benchmarks. By default the tool discovers loops with a bound of zero, which defies common sense. Restricting the use of the calling context or information from the C level results in the correct bound, for "bs", and a search failure for "bsort100". Further investigation reveals that the `main` function in the two benchmarks does not have a `return` statement, despite

[8] We handle some simple cases of recursion with small bounds in SydTV-GL-refine, see Section 2.3.4.

Benchmark	Loops	Bounds	Failures
BS	1	1	0
BSORT100	2	1	1
COVER	3	3	0
FDCT	2	2	0
FIBCALL	1	1	0
JFDCTINT	2	2	0
STATEMATE	1	0	1

Table 3.4: Mälardalen loop bounds

having return type `int`. Reaching the end of a non-void function is invalid C and the C parser forbids it. The WCET analysis makes use of exactly the restrictions that the C parser checks, and so, since this failure occurs unconditionally whenever `main` is entered, the system decides that `main` must be unreachable.

We could take additional care to avoid making use of C parser restrictions which the programmer knowingly ignored. Since our tool is designed for a case where the checks in the C model are proven true, we are confident that we can use them without further analysis. Compilers must be more cautious, as even confident programmers misunderstand the C standard, as Dietz et al. [DLRA12] have convincingly shown. We think this is a strong argument for the merits of pairing WCET and translation-validation analysis with a source-level proof of safety (e.g. through static analysis, as required by MISRA-C [MIS12]), as no safety-critical code should depend on invalid language constructs.

3.6.6 Eliminating Infeasible Paths

We evaluate infeasible path elimination on the six seL4 configurations from above. Note that the "static" and standard configurations of seL4 3.1.0 broadly match the *open and closed* use cases that were evaluated in previous work [BSC+11]. In the open systems all kernel operations are allowed. In the static/closed system, user tasks are not given capabilities that would allow creation, deletion or recycling of kernel objects (such as address spaces or thread-control blocks) once the system is initialised. Our current "static" system restricts more operations than our previous "closed" system because the previous analysis considered an seL4 variant with more preemption points and fewer long-running operations.

The "static" and "nodelete" systems also forbid three particular operations for cancelling message sends which have no satisfactory WCET in the currently verified version of seL4. These problematic operations are also long-running for small target objects, so the object size limitation does not help. We plan to eventually make these operations preemptible. Preemptible implementations were prototype by Blackham et al [BSC+11], but this time we plan to verify the preemptible implementations. Unfortunately we have not yet had time for such a major verification effort. For the time being we perform our WCET analysis as though these operations already contained preemption points.

The automated process iteratively identifies the worst-case execution trace and eliminates paths within it, until no refutable paths are found. In all scenarios, a large number of infeasible paths are found, with varying impact. The "static" variants see a greater improvement, as shown in Table 3.5. The more general variants are typically dominated by instances of a cleanup operation on a 64 KiB sized object, which contributes over 80% of the cycles spent. Refinement of paths outside the hot loop makes little difference to

seL4 version	3.1.0		Preemptible		RT†	
configuration	general	static	general	nodelete	general	static
Initial est. (k cycles)	6,894	1,193	6,888	1,191	8,256	781
Final est. (k cycles)	6,349	532	6,347	525	7,397	682
Improvement	7.9%	55.4%	7.8%	55.9%	10.4%	12.6%
Analysis Iterations	7	11	6	10	10	9
Total Refutations	1854	2873	1887	3333	3814	2371
ILP Solving Time	708s	488s	642s	443s	2476s	942s
	0:11:48	0:08:08	0:10:42	0:07:23	0:41:16	0:15:42
Unique contexts	1418	1456	1232	1331	4623	1937
Refutation Time	5410s	8002s	5813s	10775s	31566s	5588s
	1:30:10	2:13:22	1:36:53	2:59:35	8:46:06	1:33:08

Table 3.5: Infeasible path analysis statistics. Note the "RT" statistics were impacted by hand workarounds.

the headline WCET. In the restricted variants, more of the kernel code contributes to the WCET, creating more productive work for the infeasible path analysis to do. The improvement is typically steady for a small number of iterations, as shown in Figure 3.6, before continuing for a number of further iterations without a significant change in WCET estimate.

The expensive cleanup operation which dominates the WCET is the same in each of the general variants, with identical C code. Chronos produces a slightly higher estimate of its cycle cost in the "RT" case. It seems likely that the variation is due to differences in placement of the binary code across cache lines, although we have not confirmed this.

We have inspected the worst-case paths by examining which binary function symbols are called. The restricted cases all seem feasible in this regard, and we conclude that the bounds are fairly tight. In the general cases, the function call graph is feasible, however, the estimate is still not tight. The number of calls to the expensive cleanup operation is too high. It is called from the capability cleanup process, a complex nested loop bounded by preemption points. The discovered worst case path moves between the outer and inner loops in a manner that calls twice as many object cleanup operations as preemption points. This path is not feasible, but our trace refutation process cannot currently refute a path entangled in a loop in this manner.

We could improve the general estimates by manually specifying a maximum number of calls to the object cleanup mechanism, with the usual concerns about soundness. We could also in principle extend the trace refutation process to handle these loops. Encoding infeasible paths that interact with loops as ILP constraints can be complex, but effective approaches have been found by others [KBC10, Ray14]. Discovering these refutations would also be challenging for us, for various reasons involving the loop structure itself, alias-analysis for key variables stored on the stack, and differences between the C and binary loop structures. We have not attempted to solve these challenges. We plan in the future to add more preemption points to the deletion processes, which will solve the problem indirectly.

The "RT" branch introduces a performance problem for our analysis. Not only does it contain a few more loops, its function call graph is more connected, and contains significantly more arcs through which loops can be reached. Chronos creates unique

ILP variables for each visit to each function through each possible context, which means the "RT" branch is significantly more difficult for Chronos, the ILP solver, and also the trace refutation process. This is seen in Table 3.5, which lists the number of unique function calling contexts which are encountered in candidate traces during the refutation process. These differences initially resulted in effective timeouts of both Chronos and trace refutation, i.e. no results after 24 hours. We worked around these problems by running Chronos on a different machine with more than 32 GiB RAM, by manually directing the refutation process to skip certain calculations, and by manually excluding some paths in the initial problem. For this reason the times of the "RT" column are not directly comparable to the others.

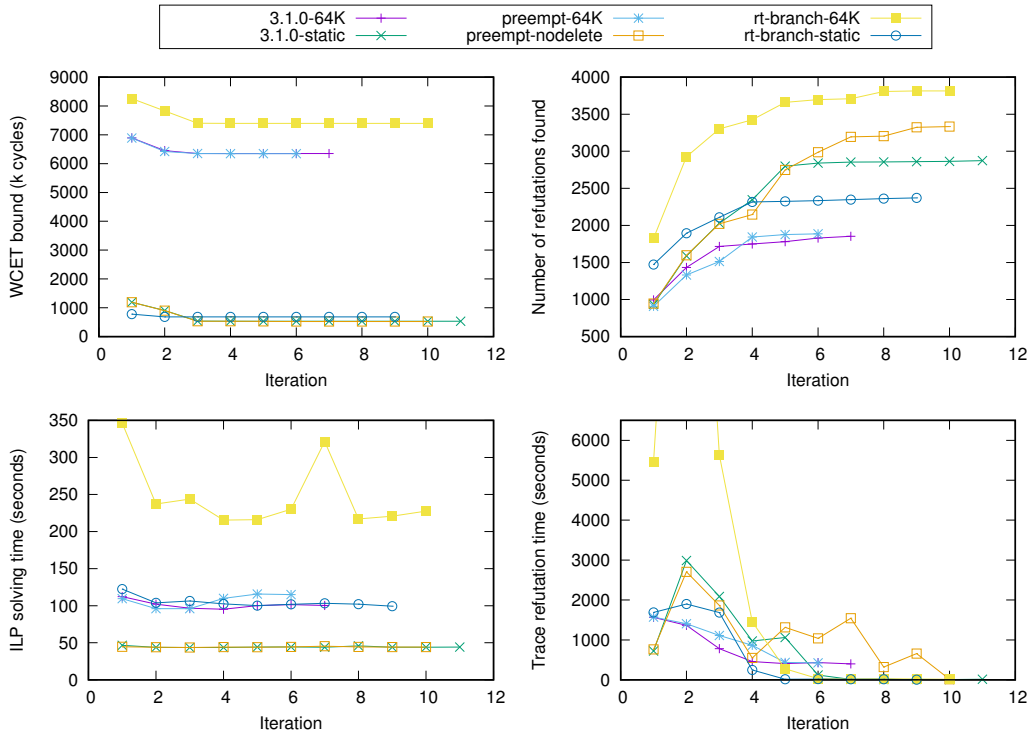


Figure 3.6: WCET bound at each iteration, number of refutations discovered, ILP solving time, and trace analysis time. Note again that manual intervention in the "RT" branch prevents direct comparison.

3.6.7 Performance

We studied the performance of the loop bound analysis in detail in Section 3.6.2. The trace refutation process has some broadly similar characteristics. For instance, it focuses individually on each function context with some calling context included, so it should scale linearly to cover a larger codebase with similar functions. However the total running time is highly sensitive to the number of necessary iterations, which is determined by the number of paths through the binary that have WCET similar to that of the critical path.

This variation is displayed in Figure 3.6, which graphs the time taken during ILP solving and path refutation for each iteration of each process. Broadly speaking, the ILP solving phase is usually cheap, and the refutation process usually becomes faster as the candidate path stabilises on the critical path. Substantial variation exists, including an

outlying refutation iteration for the "RT" branch which took nearly six hours to complete.

There is substantial room for improvement in these performance characteristics. The refutation phase employs sophisticated mechanisms to refute trace fragments, but has no particular strategy in how to employ them. Figure 3.6 demonstrates that the system discovers thousands of refutations which make little or no difference to the estimated WCET. In the future we plan to extract data from the ILP problem as well as solution, and use this to prioritise trace fragments which are likely to have an impact on the solution.

We also anticipate that the performance problem with the "RT" branch will be resolved. This code will be verified as it matures, and the verification process itself is likely to result in changes which make the codebase more amenable to analysis in general. We may also make changes with the explicit intent of improving WCET, which may include minor changes designed to improve analysis time.

The seL4 3.1.0 kernel consists of about 9,000 source lines of code (SLOC) and compiles to about 14,000 instructions in about 2,100 basic blocks. After virtual inlining by Chronos, this increases to an ILP problem for about 650,000 basic blocks. Hypothetically the ILP solving phase, which is currently the cheapest phase, would dominate the analysis for very large code bases. The analysis is helped by the small average size of functions in seL4. If instead we analysed a codebase with a few very large functions, we would produce much larger SMT problems. Our experience with the Mälardalen benchmarks is that the size (number of statements) of the largest loops has a heavy impact on the performance of SydTV-GL-refine.

3.6.8 API Availability and Future Work

The final WCET estimates for the preemption modified kernel and seL4 3.1.0 were listed in Figure 3.6. The key accomplishment of our verification is that the final WCET estimate for the *preempt-nodelete* and *3.1.0-static* variants are nearly identical. This is despite the fact that the *3.1.0-static* kernel is restricted to an entirely static system configuration, whereas with the preemption change, new objects can be created while the system is in real-time mode. This includes the creation of objects larger than the 64 KiB maximum permitted in the *3.1.0-64K* variant, even though the time taken to *complete* the creation of these objects may be substantially longer.

This change already simplifies the construction of modular real-time systems on seL4. In the *3.1.0-static* use case, the initial supervisor task must coordinate the setup of all address spaces and kernel objects itself, and it must complete this task before the system becomes static and the real-time guarantees hold. In the *preempt-nodelete* case, the supervisor can set up tasks in priority order, or delegate task setup to trusted initialisation routines within each component. Since the object creations performed during setup do not impair responsiveness, the high priority tasks can operate in a real-time setting while lower priority tasks are still doing setup.

Unfortunately the supervisor cannot yet delegate setup to untrusted modules. As we discussed in Section 3.2.5, seL4's security API does not provides only coarse-grained control over which operations a task may perform. An untrusted task with the authority to create kernel objects can always create for itself a means to trigger deletion events.

This is only a first step. The clear next step is to split up the long-running components of the object deletion and cache management operations, which would allow a fully dynamic task to run at low priority alongside a high-priority real-time application. In the longer term, incorporating and verifying features of the seL4 real-time branch will allow more complex real-time and mixed-criticality system designs. We hope that future work

will eventually result in a verified OS with a general API, flexible real-time scheduling behaviour and a competitive WCET.

3.6.9 Limitations

We build on a number of existing tools and inherit their limitations. For instance, the C-to-Isabelle parser does not support floating point arithmetic, string constants, or taking the address of a local variable. It also requires the program to be single-threaded and to be written in clean C which strictly conforms to some aspects of the standard. The HOL4 ARM model does not specify floating point or division operations (which are optional on the relevant ARM cores). SydTV-GL-refine does not provide support for reasoning about nested loops, though it handles loops with multiple entry paths or exit conditions.

None of these affect the analysis of seL4, which is unsurprising, as the parser has been co-developed with seL4, and the HOL4 ARM model was enhanced to satisfy the needs of the seL4 translation validation. Hence, the kernel code satisfies all those limitations. Furthermore, nested loops can be accommodated if the inner loop is encapsulated into a function.

While we use the proof apparatus from SydTV-GL-refine extensively, we make relatively little use of the refinement proofs themselves; we only use the loop relations for a few challenging loop bound problems. In principle, we could use the loop and path relations to map every candidate binary execution trace back into a trace through the C program, and therefore convert any path constraint we could discover in the C program into a binary equivalent. Such an approach would be both theoretically and practically attractive. It would allow us to always derive a binary control flow analysis as strong as the best available source analysis.

There are two reasons we did not pursue this. Firstly it would be computationally very expensive to map every binary branch back to its C counterpart (or lack thereof) rather than just the looping conditions. Secondly, seL4 (like any OS kernel) contains a small number of hardware-control routines that use in-line assembly. At the time we performed this analysis, the C-to-Isabelle parser could not understand them. This created a number of "blind spots" for the refinement framework – functions which must simply be assumed to match the semantics of the relevant binary routine. When the compiler is permitted to inline aggressively (we use `gcc -O2`), it occasionally moves these simple routines upwards into the loops they are called from. This means we depend on binary-only loop bound analysis to operate within these blind spots.

3.7 Conclusions

We propose a WCET analysis approach supported by a translation validation suite and the functional correctness apparatus used to verify the target program. In particular we build on the Tuch/Norrish C semantics used for manual verification and the SydTV translation validation framework. Together these give us a convenient environment for reasoning about binary execution and adding source level annotations if necessary, without trusting either the compiler or the annotation author.

We apply this approach to the seL4 microkernel, and determine (tight) bounds on all of the loops in its binary. The majority of bounds are found without providing any additional information, while a few required adding extra assertions (which needed to be proved) at the C level. After this one-off manual interference, all remaining loop bounds are found and proved. All the discovered loop bounds seem to be tight.

Similarly, the tool chain (provably) refutes infeasible paths. While in this case there is no guarantee that all such paths have been refuted, the result is comparable to earlier work (which identified infeasible paths by manual inspection). The identified worst-case execution trace that remains after refutation concludes seems possible, though this is laborious to confirm by inspection.

We have also shown via the Mälardalen benchmarks that the approach works, in principle, for other real-time code that has not been formally verified, although restrictions in the SydTV toolchain limit the class of programs that can be analysed. Obviously, without being able to leverage formal verification artefacts, the analysis is less complete than in the case of seL4. However, the support for manual code annotations to specify assertions can compensate for this, especially where such assertions can ibe proved by other means, e.g. model checking.

Finally, we have used our framework to clearly enumerate the remaining real-time deficiencies in the verified seL4 kernel as of version 3.1.0. We have implemented and verified an improvement to the single largest deficiency, the object creation operation. While plenty of work remains to be done before seL4 becomes a full-featured real-time operating system with complete verification and competitive WCET, this is a substantial step in that direction.

This analysis has made use of the strengths of all three components of SydTV. The complementary decompiler gives us a model for reasoning about binary path information, and great assurance that our CFG reasoning is valid. The connection to Isabelle/HOL allows us to add and verify timing-related assertions within a rich proof environment. Finally, the analysis suite of the SydTV-GL-refine component can be easily expanded from a specific tool for discovering refinement proofs to a more general binary analysis framework.

In summary, we believe that the WCET analysis framework based on our translation validation approach is an extremely promising approach for establishing WCET bounds on high-assurance software. In the specific case of the seL4 microkernel, it constitutes a major step towards reaching a similar level of confidence in its timeliness as already exists in its functional correctness.

4 Formal Aspects

The previous chapters detail the refinement and timing computations about SydTV-GL programs which we can perform within SydTV-GL-refine. This chapter focuses on the proofs we produce within Isabelle/HOL. This includes the frontend proofs which link the SydTV-GL semantics to the formal C semantics. This also includes a project in which we replicate the proof rules of the SydTV-GL-refine proof checker with proof rules and apparatus in Isabelle/HOL.

SydTV-GL-refine interfaces with SMT logics for its low-level judgements, but higher-level concerns about SydTV-GL semantics cannot be expressed in an SMT logic, and are instead encoded in the implementation of SydTV-GL-refine. The SydTV-GL semantics can, however, be precisely specified in Isabelle/HOL. We have already explained to the reader the logic behind the proof rules of SydTV-GL-refine (in Section 2.1). We now repeat this explanation to Isabelle/HOL.

We do this for three reasons. Firstly, the proof principles of SydTV-GL-refine might simply be wrong, and Isabelle/HOL would force us to confront that. Secondly, and more likely, the *formal* presentation of the SydTV-GL-refine logic will lead to the discovery of subtleties that we had previously missed. Thirdly, this formalisation is a significant step toward a project which we hope to return to in future, in which we gather all of the reasoning of SydTV into a single theorem in a single proof environment.

4.1 The Isabelle/HOL Proof Assistant

This chapter discusses a number of results which we prove, or reprove, within the Isabelle/HOL theorem prover. We use theorem provers such as Isabelle and HOL4 because they are designed to give us the highest possible assurance of the results they prove, a level of assurance which we consider the "gold standard" for this kind of work.

While we have considerable confidence in the correctness of the results produced by our custom tool SydTV-GL-refine, we acknowledge that the level of assurance that can be attained by proving a theorem in Isabelle/HOL or HOL4 would be higher again.

These theorem provers are designed to be *skeptical* and *foundational*. Skeptical means they only accept results proven according to their own proof rules. Foundational means that the only proof rules they implement correspond directly to the foundational mechanisms of a well-studied mathematical logic. The tools we use generally build on the well-understood theory of higher-order logic (HOL), a descendant of Church's theory of types [Chu40]. The HOL4 prover [SN08] directly implements a modern version of higher-

order logic. The Isabelle approach is slightly different, with a core Isabelle prover [Pau90] which implements a generic logic onto which the Isabelle/HOL module adds higher-order logic as a collection of axioms.

There are minor differences between these presentations of HOL, however, we can say with confidence that all of the theorems we prove within either logic are established within a well-understood mathematical formalism.

4.2 Formalising SydTV-GL

Before we can prove anything about SydTV-GL programs in Isabelle/HOL, we must introduce a formal semantics for the language. We have said on a number of occasions that SydTV-GL is designed to have an elementary semantics, so that programs in SydTV-GL can be manipulated in a number of of proof tools without risk of confusion. We do not have any precise methodology with which to evaluate that claim, but we hope that the simplicity of the definitions below go some way towards convincing the reader.

Recall that SydTV-GL is a low-level language designed to permit proofs about compiler optimisations. Its name derives from "graph language", because it was designed to make the control-flow graph of the program explicit, especially so that control-flow adjusting optimisations could be treated straightforwardly. SydTV-GL is goto-structured, where all nodes are numbered and specify their successor nodes by number. There are also two exceptional addresses, `Ret` and `Err`, which represent successful return from a function and unrecoverable failure respectively.

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, or normal instructions in the binary, and update the value of variables with the result of some calculation (memory is represented as a variable, as are registers). Call nodes are used to represent function calls, which may not be embedded in other statements.

The graph language was designed so its semantics would be straightforward to formalise in Isabelle/HOL or HOL4. The node types are introduced as a datatype:

```
datatype next_node = NextNode ℕ | Ret | Err
(* variable names and function names are strings *)
type_synonym vname = string
type_synonym fname = string

(* variable state is a function *)
type_synonym state = (vname ⇒ value)
(* expressions/state-accessors are shallowly embedded
   as functions from state to variable *)
type_synonym acc = (state ⇒ value)

datatype node =
   Basic next_node ((vname × acc) list)
    | Cond next_node next_node acc
    | Call next_node fname (acc list) (vname list)
```

The syntax we will use here is slightly amended for presentation. Isabelle/HOL is a logic with a functional style, so for instance "$f\ n$" is the application of function f to

parameter n. We use font to distinguish variables (e.g. f) and type variables (e.g. α) from constants (e.g. `Basic`) and type constructors (e.g. `list`). Isabelle/HOL follows the ML style where type constructors are applied as suffixes. We use mathematical syntax (e.g. $\times$ for cartesian products, $\mathbb{N}$ for the set of natural numbers) where possible. Function types are written $\alpha \Rightarrow \beta$.

These definitions are taken from the **GraphLang** theory of the official L4.verified distribution.[1]

In this representation variables are indexed by name (a string) only. A `Basic` node contains a next program address and list of variables to update by evaluating expressions. A `Cond` node has a pair of successors and a decision expression. A `Call` node contains a successor, a function name to call, a list of argument expressions and a list of return variables to overwrite.

A graph function is essentially a tuple of an input variable list, output variable list, graph body and entry point:

datatype `graph_function`
$=$ `GraphFunction` (`vname list`) (`vname list`) ($\mathbb{N} \Rightarrow$ `node option`) $\mathbb{N}$

The semantics of the graph language are defined by a small-step relation. The configuration of the system at any step is represented by a stack of executing functions. Making the stack explicit introduces some complexity into our proofs, but it allows us to model recursion faithfully, without first having to show a recursion bound or termination. Each stack frame contains an executing address, the state mapping of variables to values, and the name of the currently executing function. The step relation also takes as parameter an environment object that maps function names to bodies.

type_synonym `stack` $=$ (`next_node` $\times$ `state` $\times$ `fname`) `list`

type_synonym `graph_env` $=$ (`fname` $\Rightarrow$ `graph_function option`)

constant `exec_graph_step` :: `graph_env` $\Rightarrow$ (`stack` $\times$ `stack`) `set`

The step relation specifies all possible steps. It builds on a pair of functions `exec_node` and `exec_node_return`, which specify the normal steps and return steps respectively. Normal steps begin at a node address in some function. Return steps begin at the `Ret` or `Err` address in the top stack frame when there is another stack frame to return to. Execution ends when `Ret` or `Err` is reached in the bottom stack frame. The `exec_node` function captures the behaviour of the three node types:

[1] The file can be found at:
`https://github.com/seL4/l4v/blob/master/tools/asmrefine/GraphLang.thy`

primitive_recursive
 exec_node :: graph_env ⇒ state
 ⇒ node ⇒ stack ⇒ stack set
where
 exec_node Γ *st* (Basic *cont upds*) *stack*
 = {upd_stack *cont* (K (upd_vars *upds st*)) *stack*}
 | exec_node Γ *st* (Cond *left right cond*) *stack*
 = {upd_stack (if *cond st* then *left* else *right*) I *stack*}
 | exec_node Γ *st* (call *cont fname inputs outputs*) *stack*
 = (case Γ *fname of* None ⇒ {upd_stack Err I *stack*}
 | Some (GraphFunction *inps outps graph ep*)
 ⇒ {(NextNode *ep*, init_vars *inps inputs st*, *fname*)
 <Cons> *stack*})

The partner function exec_node_return, which we will omit here, is similar to exec_node. It also investigates Call nodes to handle the way variables are saved on return from function calls, however, if it encounters a Basic or Cond node it simply reports an error. The exec_graph_step definition combines these partial specifications together.

The aim of these simple definitions is to be *shared* between a number of proof tools without risk of confusion. It is perhaps instructive to compare the definition of exec_node above with its equivalent from the HOL4 formalisation below:

```
val exec_node_def = zDefine ‘
  (exec_node Gamma st (Basic cont upds) stack =
    {upd_stack cont (K (upd_vars upds st)) stack}) /\
  (exec_node Gamma st (Cond left right cond) stack =
    {upd_stack (if cond st then left else right) I stack}) /\
  (exec_node Gamma st (Call cont fname inputs outputs) stack =
    case Gamma fname of NONE => {upd_stack Err I stack}
      | SOME (GraphFunction inps outps graph1 ep) =>
          {(NextNode ep, init_vars inps inputs st, fname) :: stack})‘;
```

The aim of this formalisation is to produce proofs of refinement. We can define refinement by appeal to the set of possibly-infinite traces generated by the small-step relation. We can define this set of traces for any small-step relation and continuing condition:

definition
 trace_set :: (σ ⇒ bool) ⇒ (σ × σ) set
 ⇒ (ℕ ⇒ σ) set
where
 trace_set *C r* = {*tr*. (∀ *i s*. *tr* (Suc *i*) = Some *s*
 ⟶ (∃ *s'*. (*s'*, *s*) ∈ *r* ∧ *tr i* = Some *s'*))
 ∧ (∀ *i s*. *tr i* = Some *s* ∧ *C s* ⟶ *tr* (Suc *i*) ≠ None)}

The traces are defined as partial maps from natural numbers to states, where the trace evaluated at any number i tells us the state of execution after i steps. Partial maps are constructed using the option type of Isabelle/HOL with constructors Some and None. Any

point in the trace where the state is defined must be reached from a preceding state by a step in the step relation. If the state is judged to be continuing at any step, there must be a next state. For the graph language, continuing means that the final `Ret` or `Err` label has not yet been reached.

These traces are possibly infinite, and thus describe the semantics we are interested in slightly more accurately than the transitive closure of the small-step relation. We can prove that a state s' can appear in a trace starting at s if and only if (s, s') is in the reflexive transitive closure of the small-step relation.

We can define and prove some useful properties about the trace set already. We start by defining the end of the trace, which may or may not exist. Because the small-step relation executes on a stack of states, we prove a number of helper results about the stack. Firstly, we show that the name of the initially executing function in the bottom stack frame will never change, making it clear that the trace is an execution of this function.

Secondly, we show that we can extract executions of called functions. We can take an inner chunk of the trace where the stack depth is always at least n, and discard the bottom $n - 1$ stack elements at every point, resulting in a trace with valid steps. If this subtrace begins at an appropriate entry point and ends at `Ret` or `Err` or never ends, then it is a valid trace in its own right.

Finally, we show that any point in a trace where a `Call` node is reached will permit such an extraction starting from the next step. These results are needed to reason about function calls in later results.

4.3 Connecting C/Simpl to SydTV-GL

We gave an overview of the export process from Isabelle/HOL to SydTV-GL in Section 2.1.1. This section describes this process in further detail, and introduces the proof procedure that establishes that the exported SydTV-GL definitions refine the Isabelle/HOL C representation.

4.3.1 The C/Simpl Language

The C program is given a semantics by the C-to-Isabelle parser. The parser uses a number of facilities within Isabelle/HOL to represent the semantics of C. These mechanisms, taken together, form a dialect of Isabelle/HOL which is the source language for the SydTV-GL export process.

The most substantial components of this dialect of Isabelle/HOL are the Tuch memory semantics [Tuc08], which we have already discussed, and Schirmer's Simpl imperative language framework [Sch06].

The Simpl language framework is a generic facility for deeply embedding imperative languages in Isabelle/HOL. The central component of the Simpl formalisation is a syntax datatype that allows the structure of imperative programs to be represented. The Simpl development also includes as much additional Isabelle support for this language as possible, including a small-step semantics, big-step semantics, sound & complete VCG,[2] custom syntax, etc.

The C-to-Isabelle parser uses Simpl constructors to assemble its representation of the C program. All statement-level C constructs map to Simpl constructors. For instance

[2]Verification Condition Generator, an essential feature for proving safety properties about programs in the Simpl language.

sequential composition in C (`;`) is mapped to its Simpl counterpart (called `Seq` in Isabelle/HOL, and also given the syntax `;;` as an operator). Loops in C are encoded using Simpl's `While` constructor, `if/else` statements use Simpl's `Cond` conditional operator, and so on. Simpl was designed with the C language in mind, specifically for use in the C0 programs [LPP05] of the Verisoft project [AHL+08]. All statement-level C constructs map into Simpl equivalents with minor adjustments.

We mentioned the Simpl `Guard` constructor previously, for instance in Section 2.1.1. The `Guard` constructor wraps a potentially unsafe operation with a check, which is a verification condition for the program. Since nearly all C operations are potentially unsafe in various ways, the output of the C-to-Isabelle parser usually contains more `Guard` statements than anything else.

The Simpl infrastructure provides deep embedding for statements, allowing a formal function in Isabelle/HOL to recurse across a function's statements. The Simpl VCG is exactly such a function. The expressions in Simpl, however, are shallowly embedded. The Simpl datatype takes a type parameter σ, the type of states, and the expression components of statements manipulate it directly. For instance, the condition parameters to the `While` and `Cond` constructors have type σ `set` within Isabelle/HOL. State update actions are encoded using the `Basic` constructor and a function of type $\sigma \Rightarrow \sigma$.

The C-to-Isabelle parser converts the expressions of C into Isabelle/HOL equivalents. Various forms of arithmetic have equivalents using Isabelle/HOL's native implementation of bitvector arithmetic. Aggregate types in C become datatypes in Isabelle/HOL. The C-to-Isabelle parser distribution includes a custom type family for arrays. The Tuch memory semantics is used to manipulate memory, and also supplies guard conditions for pointer validity etc. Variables, both local and global, are collected into the state type using the record package of Isabelle/HOL.

We discussed before that we adjust the C semantics for seL4 (see Section 2.5.1). This adjustment consumes the parser's definitions and produces modified definitions in the same C/Simpl dialect. The new definitions are encoded to "mimic" the C-to-Isabelle parser, so that most tools designed to work with the parser's output also work with the adjusted functions. The export process is such a tool: either the originally parsed C bodies or a mimic set produced by adjustment can be exported.

4.3.2 The Export Process

The export process is conceptually straightforward. It consists of a collection of Standard ML functions that run in the Isabelle/HOL environment and convert the various language elements of the C/Simpl dialect to their SydTV-GL equivalents. Most Simpl statements will map to a single SydTV-GL node, but there are some special cases. The conversion of the expressions of the C/Simpl language is also conceptually straightforward, but intricate in practice.

Since the C/Simpl function bodies are decomposed at the statement level by Simpl constructors, the export process decomposes this way also. The central worker of the export process is an ML function called `emit_body`. It recurses across the *body* of a C/Simpl function. It emits a collection of SydTV-GL node definitions which correspond to a block of C/Simpl syntax. We can see a small excerpt from the definition of `emit_body` below:

```
fun emit_body ctxt outfile params
        (Const (@{const_name Seq}, _) $ a $ b) n c e = let
```

```
    val (n, nm) = emit_body ctxt outfile params b n c e
    val (n, nm) = emit_body ctxt outfile params a n nm e
in (n, nm) end
| emit_body _ _ _ (Const (@{const_name com.Skip}, _)) n c _
  = (n, c)
| emit_body _ _ _ (Const (@{const_name cbreak}, _) $ _) n _ (_, b)
  = (n, b)
| emit_body ...
```

The full definition of `emit_body` has 16 such cases, and we do not wish to go into such detail here.[3] We also edited out some error-formatting code for simplicity.

The `emit_body` function recurses across the underlying Isabelle term representation of the C/Simpl function body. It matches on the `Const` and `$` constructors of the underlying term representation, which encode constants and function application in Isabelle. The source code of `emit_body` uses some Isabelle-specific antiquotation trickery to use elements defined in the relevant Isabelle/HOL theories in the ML source, for instance the way that the fully qualified name of the `Seq` constant is matched on here.

The first, trivial, design choice was that `emit_body` does not return the SydTV-GL definitions it produces. It writes directly to the output file specified. To prove properties about the new definitions in Isabelle/HOL, we parse the textual representation, the same way that SydTV-GL-refine does.

The second design choice concerns the numeric identifiers given to the nodes of the exported program graph. The obvious way to do this would be to have one SydTV-GL node for every statement, to number the statements in textual order, and use those numbers as node identifiers in the graph. When we have displayed such a graph (for instance in Figure 2.1 within Section 2.1.1) we have numbered the nodes in this way.

The problem with this approach is that it requires us to number all the C/Simpl statements ahead of time, store the numbers in some datatype, and look it up. Instead, we number SydTV-GL nodes as they are emitted. However this means that we need to work through the statements in reverse order, so that we know the numbers of the successor statements as we export each statement.

This is what `emit_body` does. We see above that it handles a sequential composition by emitting the second part of the composition first. Each call to `emit_body` returns a two-element tuple. The first part is the next available numeric identifier `n`. This value is threaded through all the recursion of `emit_body`. The second part is the identifier of the emitted graph node which embodies the C/Simpl argument.

This approach allows us to omit some statements from the SydTV-GL graph entirely. When we process an empty statement (`Skip`), instead of emitting a SydTV-GL node that does nothing and points at some next statement, `emit_body` reports that the implementing SydTV-GL node for the `Skip` statement is the same as for its successor. We see this above, where `emit_body` returns as the node address the input parameter `c`. The parameter `c` is the continuing address, the identifier of the SydTV-GL node to be visited next.

The "next" node to be visited may not be unique. There are additional parameters which specify the statement that control flow would move to after a `break`, `continue` or `return` statement. We do not support the `goto` statement. The SydTV-GL addresses that would be reached after these statements are are also passed to `emit_body`. We see above

[3] The full implementation of the export process can be found in the **SimplExport** theory of the official L4.verified distribution, for instance at:
`https://github.com/seL4/l4v/blob/master/tools/asmrefine/SimplExport.thy`

that the `break` statement can be treated like the empty statement, although proceeding directly to the break address.

This hopefully explains the entire process for emitting a sequential composition which we see above. Firstly the second component is emitted, with the same continuing and exceptional addresses as the whole composition. The first component is then emitted, with the same exceptional addresses, and the start of the second component as the continuing address. The start of the whole composition is the start of the first component.

This approach can be extended straightforwardly to other structuring elements of the C language. The only complex case concerns the `While` constructor, which introduces loops. It is the exception to the rule that the program is emitted entirely in backward order. To handle `While` constructs, `emit_body` first reserves some node numbers to represent the start of the loop. The body of the loop is then emitted, with the first reserved address as the continuing address. Finally the reserved nodes are emitted, out of order with the rest of the sequence. These nodes handle the looping condition, and also check some error conditions.

The above process handles the statement level of C/Simpl reasonably well. The complex part of the export process is handling the C/Simpl expressions. There are two reasons for the complexity. The first is the sheer quantity of different kinds of expression syntax, which the C-to-Isabelle parser cobbles together using a large library of different Isabelle/HOL theories. Much of the export process involves querying various Isabelle/HOL packages, e.g. the datatype package of Isabelle/HOL, to discover what the relevant constants are intended to do.

The second source of complexity is that the export process performs a conversion in addition to changing syntax. Many standard operations in C, e.g. arithmetic operations in the `long` type, have uniform representations in C/Simpl and langname. Operations on aggregate types, however, are converted into field-by-field equivalents. This can be quite intricate for some expression types. Some global objects, which are treated by the C/Simpl semantics as global variables, must also be relocated into memory.

4.3.3 Verifying Export Correctness

Having exported the C/Simpl semantics to a SydTV-GL program, we now prove that the exported program is a refinement of the original. The proof process will follow the statement-by-statement structure of the export process, which we outlined in Section 4.3.2.

The SydTV-GL functions have a semantics of possibly-infinite traces which we introduced in Section 4.2. The Simpl formalism includes both a small-step semantics and a big-step semantics. The seL4 proofs, for instance, connect to the Simpl big-step semantics. We can converge semantics somewhat by expressing the Simpl semantics as a set of traces, using the small-step relation and the same `trace_set` operator as used for the SydTV-GL semantics (Section 4.2). We prove that the big-step semantics, including its notion of termination, can be reformulated via this trace semantics.

Refinement then means the usual concept of trace refinement. Each trace of the SydTV-GL function must have a matching trace of the C/Simpl function. Matching means either that both functions terminate with matching return values, that both functions have infinite traces, or that the C/Simpl trace faults at a `Guard` statement.

The refinement problem concerns a complete C/Simpl function body and a complete SydTV-GL function graph. We will prove that excerpts of the C/Simpl body match the semantics of the function graph with different starting points, eventually proving that the entire C/Simpl body matches the semantics of the SydTV-GL function starting at its true

entry point.

We will prove this refinement property by presenting a rule calculus for the predicate `simpl_to_graph`. We will use this predicate to establish that a (part of a) SydTV-GL function graph and a (part of a) C/Simpl function body have matching semantics. The `simpl_to_graph` predicate looks like this:

$$\texttt{simpl\_to\_graph}\ \Gamma_S\ \Gamma_G\ f\ nn\ simpl\ n\ trS\ P\ I\ eqs_I\ eqs_O$$

This predicate addresses two different concerns. For the moment, we're going to leave aside its actual definition, and the intricacies of what the parameters n and trS are for, and focus on the "common" cases.

The predicate says that the part of the SydTV-GL graph function f which begins at the node address nn refines the semantics of the C/Simpl statement $simpl$. The environment parameters Γ_S and Γ_G specify the function calling environment at the C/Simpl and SydTV-GL levels respectively. The function of interest f is also specified by name rather than implementation, with the body fetched from Γ_G.

We want to prove that all execution traces of f have matching traces of execution in $simpl$. We assume a pair of starting states, related by the input relation eqs_I. We will always phrase this input relation as a collection of equalities. The final states must satisfy the output relation eqs_O, or the traces must both never end. Any case in which $simpl$ can transition to a faulting state by failing a `Guard` check is also considered a matching trace to any possible SydTV-GL trace.

In addition to eqs_I, the starting state on the C/Simpl side is assumed to be an element of the sets P and I. We will use the parameter P to accumulate known information from `Guard` checks. The parameter I is an invariant of the C/Simpl state. It is used in practice to assert that the global variables are disjoint from all regular heap values according to the heap type description.

Core Proof Calculus

The core approach of the `simpl_to_graph` proof calculus is to work through the syntax of the SydTV-GL and C/Simpl programs one step at a time.

For example, the following rule processes a `Guard` statement at the C/Simpl level. We proceed to checking that the inner statement of the `Guard` has the correct semantics. In the inner proof, the state is known to be an element of the guard set `G`.

lemma `simpl_to_graph_Guard` :
$$\begin{aligned}&\texttt{simpl\_to\_graph}\ \Gamma_S\ \Gamma_G\ f\ nn\ (\texttt{add\_cont}\ c\ con)\\&\quad n\ trS\ (G \cap P)\ I\ eqs_I\ eqs_O\\&\longrightarrow \texttt{simpl\_to\_graph}\ \Gamma_S\ \Gamma_G\ f\ nn\ (\texttt{add\_cont}\ (\texttt{Guard}\ F\ G\ c)\ con)\\&\quad n\ trS\ P\ I\ eqs_I\ eqs_O\end{aligned}$$

The `add_cont` wrapper ("add continuations") is used in our proof calculus to make handling of composition more uniform. When we process a sequential composition `Seq`, we need to focus on the first component and put the second component somewhere to be resumed. The con argument to `add_cont` is a list of statements to be resumed. Since the list may be empty, we can use the rule to handle a `Guard` statement whether it is the first step in a sequence or the only statement present. The `add_cont` wrapper also

handles exceptional composition (for handling `break` etcetera), but we will not go into these details.

The `Guard` rule is simpler than others, because we do not require a one-to-one mapping between `Guard` statements and nodes in SydTV-GL. We can move the whole effect of `Guard` here into the precondition and continue.

A more conventional rule is the rule for conditional execution. We expect the `Cond` operator of Simpl to map into a SydTV-GL `Cond` node. We apologise for the confusion between Simpl's `simpl.Cond` and SydTV-GL's `graph.Cond` here, and similarly for `Basic` and `Call`.

lemma `simpl_to_graph_Cond` :
$nn = \texttt{NextNode}\ m \longrightarrow \Gamma_G\ f = \texttt{Some}\ g$
$\longrightarrow \texttt{function\_graph}\ g\ m = \texttt{Some}\ (\texttt{graph.Cond}\ l\ r\ cond)$
$\longrightarrow \texttt{eq\_impl}\ nn\ eqs\ (\lambda\ s_g\ s_s.\ cond\ s_g = (s_s \in C))\ (P \cap I)$
$\longrightarrow \texttt{eq\_impl}\ nn\ eqs\ eqs_2\ (P \cap I \cap C)$
$\longrightarrow \texttt{simpl\_to\_graph}\ \Gamma_S\ \Gamma_G\ f\ l\ (\texttt{add\_cont}\ c\ con)$
$(\texttt{Suc}\ n)\ trS\ (P \cap C)\ I\ eqs_2\ eqs_O$
$\longrightarrow \texttt{eq\_impl}\ nn\ eqs\ eqs_3\ (P \cap I \cap (-C))$
$\longrightarrow \texttt{simpl\_to\_graph}\ \Gamma_S\ \Gamma_G\ f\ r\ (\texttt{add\_cont}\ d\ con)$
$(\texttt{Suc}\ n)\ trS\ (P \cap (-C))\ I\ eqs_3\ eqs_O$
$\longrightarrow \texttt{simpl\_to\_graph}\ \Gamma_S\ \Gamma_G\ f\ nn\ (\texttt{add\_cont}\ (\texttt{simpl.Cond}\ C\ c\ d)\ con)$
$n\ trS\ P\ I\ eqs\ eqs_O$

The first thing the reader will observe here is that these rules quickly become complex in practice. This rule will always be applied by an automatic tool, and is verbose partly because it is designed to be uniform. The `eq_impl` predicate here provides one kind of uniformity. All additional proof obligations which are not to be solved in the `simpl_to_graph` phase are wrapped in `eq_impl` statements and left for later work. These `eq_impl` goals have a redundant nn parameter which makes it easier to understand where they come from when debugging the later phase. The `eq_impl` goals relate to the expression-level conversion to SydTV-GL. For example, the first such goal above requires that the condition expressions be equivalent in the two languages.

The rule above requires that the label to be executed is a node label (rather than `Ret` or `Err`), that the node looked up at that label is a `Cond` node, and that the next C/Simpl statement to execute is a Simpl `Cond` statement. The two `simpl_to_graph` subgoals require that the two branches of the respective `Cond` statements refine.

The equality parameters such as eqs are collections of equalities which link the C/Simpl states and SydTV-GL states. The C/Simpl state is an Isabelle/HOL record of variables, whereas the SydTV-GL state is an explicit mapping from variable names to values. We assert equality for all relevant local variables. Because some variables are initially uninitialised, we construct this equality parameter for each graph label, specifying only the variables relevant at that label. Since this varies between addresses, the rule may be instantiated with multiple equality parameters eqs, eqs_2, eqs_3 for the various graph labels. Some addition proof obligations above require that the new equalities can be derived from the initial ones.

There are several further structural rules which cover all of the elements of C/Simpl which are converted into SydTV-GL nodes. There are also rules for skipping through some constructs, for instance if `Skip` appears with a continuation we proceed to the continuation without examining any SydTV-GL nodes (this treatment was implied by

the `emit_body` code we saw in Section 4.3.2). Finally, there are straightforward rules for arriving at the `Ret` label when only `Skip` remains to process.

Caching

Care needs to be taken in proving `simpl_to_graph` to avoid repeating work. Consider the case of a C/Simpl body `Cond` $C\ a\ b\ ;;\ c$ which begins with a conditional expression and then executes another statement. The predicate calculus described above will move the additional component into the continuation list of `add_cont`, and then split on the condition C. The continuation c will appear in both subsequent problems.

The proof calculus will follow the graph forward rather than backward, which means the two proofs will converge on a proof about the c component. The two proofs should then share this subproof.

The proof strategy for `simpl_to_graph` uses a simple caching mechanism to accomplish this. It stores each intermediate result in a database indexed by key parameters to the `simpl_to_graph` predicate in the conclusion. The strategy will apply a previously finished subproof in preference to applying a standard rule.

Induction

The remaining complexity in the `simpl_to_graph` statement is to permit a particular style of induction.

Loops in C/Simpl are encoded using the `While` constructor. The loops of the SydTV-GL function body are simply `Cond` nodes where one of the conditional branches loops back in the graph structure.

As a first step, we can prove the same sort of rule for `While` $C\ c$ as we showed above for `simpl.Cond` $C\ c\ d$, one which pushes the proof forward by one step of the respective small-step relations. Like in the `Cond` rule, we expect to find a `graph.Cond` node, and the first obligation is to show that the C/Simpl and SydTV-GL conditions are equivalent. matching node of the C/Simpl `While` statement is a conditional node. The remaining two subgoals will cover the two possible next steps, depending on whether C holds.

We could apply this rule and proceed forward from these configurations, but one branch will loop back to the same configuration. Deriving a logical fact in a cycle from itself is pointless, but suggests that we could rephrase the problem into a proof by induction. All our rules have pushed the proof "forward", moving at least one step deeper into the trace. If we knew that the trace was of finite length, we could handle this proof by induction.

The n parameter to `simpl_to_graph` counts how many steps have already necessarily taken place. When we introduced the predicate we said that the graph label nn, assumed state information eqs_I, I, etc, all applied at the starting states. This is true, but by "starting states" we do not necessarily mean the first states of the traces. We assume there exists an index i into the SydTV-GL trace at which point nn is the next graph label to be visited. We also assume there exists a trace prefix of the execution of some C/Simpl body which after j steps has $simpl$ the statement to be executed. We assume that $i \geq n$ and $j \geq n$, that is, at least n steps have occurred in both traces.

One aspect of the `Cond` rule that we did not draw attention to was the replacement of n by `Suc` n in each branch. That is, the inner proof can assume that the the number of steps that have taken place is one higher than before. The same is true for the structural `While` rule above. If we knew our programs terminated, we could introduce a variable

N which bounds the length of the traces, and induct on the possible number of steps remaining $N - n$.

We do not, however, know that our programs terminate. Compiling a nonterminating C program into a nonterminating binary is the expected behaviour for a compiler, so we aim to be able to validate it.

We do this by proving the key induction rule for `simpl_to_graph`:

$$
\begin{array}{l}
\textbf{lemma}\ \texttt{simpl\_to\_graph\_induct}: \\
\quad (\forall S'\ n'. \\
\qquad \texttt{simpl\_to\_graph}\ \Gamma_S\ \Gamma_G\ f\ r\ c \\
\qquad\quad (\texttt{Suc}\ n')\ S'\ P\ I\ eqs\ eqs_O \\
\qquad \longrightarrow \texttt{simpl\_to\_graph}\ \Gamma_S\ \Gamma_G\ f\ r\ c \\
\qquad\quad n'\ S'\ P\ I\ eqs\ eqs_O \\
\quad) \longrightarrow \texttt{simpl\_to\_graph}\ \Gamma_S\ \Gamma_G\ f\ nn\ c \\
\qquad n\ (\{tr\} \times UNIV)\ P\ I\ eqs\ eqs_O
\end{array}
$$

This rule says that, roughly speaking, if we can prove our predicate under the assumption that it will hold in the future, then it holds always. This is exactly what we need to complete our proof. When we first wrote this proposition down, we thought that it encoded a well-known induction principle of temporal logic. This was a misconception, the temporal logic principle would move toward the past, not the future. Nonetheless, this rule is provable in Isabelle/HOL.

The trS parameter can specify restrictions on the SydTV-GL trace or on prefixes of the C/Simpl trace. It is used here to initially assume that we know the name of the SydTV-GL trace, tr.

To prove this rule, we consider the space of all possible matching trace prefixes. We consider all traces tr' of the C/Simpl semantics, and all starting points i in tr and j in tr' which can make the starting conditions P, I, eqs_I etc valid.

There must be at least one such configuration in the space, since our outer `simpl_to_graph` predicate assumes it. We call this reference configuration X.

This space of all matching trace prefixes has an ordering, where a pair of trace prefixes $A \leq B$ if $i_A \leq i_B$, $j_A \leq j_B$, and tr'_A and tr'_B agree for j_A steps.

The proof is essentially a case division on the question of whether there is a terminal element in this order which is above X. That is, is there some matching configuration Y where $X \leq Y$ but there is no following element $Y < Z$.

If such a Y exists, we can carefully instantiate the n' and S in our induction premise, in particular with n' being j_Y. We ensure that the configuration from Y can serve as a starting configuration in the n' case, but no matching configuration for `Suc` n can exist as it would create a following element of Y. This then implies the needed trace which matches tr.

If there is no such terminal element, then every configuration reachable from X can be extended to another such configuration. This immediately implies that tr is infinitely long. The trace prefixes on the C/Simpl side can also always be extended, implying there must be a matching infinite trace there, as required.

There is a technicality in the infinite case which the perceptive reader may have spotted. We will return to this in Section 4.5.

The induction rule we have proven here and the structural `While` rule we suggested above can be combined to provide a sufficient mechanism for coping with `While` constructs in the binary. The induction rule above needs to be slightly tweaked for nested while

loops, to clarify that the unknown n and S' inside the induction will be tighter bounds than the existing ones.

Additional care needs to be taken in the caching process, which should never cache a goal with a `While` constant in it. The proofs about the body elements will still be cached. An outer mechanism will combine cached step applications with `While` induction to solve the required problems.

Mistakes that We Have Made

Let us for a moment cast our mind back to the original question of translation correctness, and the two competing approaches, validation of each translation and verification of the translator once and for all. These ideas are not true opposites, but rather ends of a spectrum. Some tools consist mostly of algorithms that are proven correct, but might for flexibility include one or two phases that are checked instead. One such example is CompCert [Ler06], which was originally a fully verified implementation but eventually adopted a validation phase for its optimising register allocator [TL08].

Reflecting on the complexities of our C/Simpl export proof, it occurs to us that perhaps some translation validation approaches should consider including verified transformations as well.

The complexities of the `simpl_to_graph` proof calculus contrast with the simplicity of the `emit_body` export process. The proof must take care not to cover the same ground twice, and must carefully instantiate inductive hypotheses, whereas the `emit` process simply works its way through the C/Simpl syntax one element at a time.

Perhaps in this work we stayed too much in the translate/validate mindset.

The C/Simpl syntax is deeply embedded in Isabelle/HOL. We could define a function `emit` in Isabelle/HOL which recursed through the syntax in the same way as `emit_body`. It could operate on statements but not expressions, instead having to return a list of requirements about expressions, such as a requirement that graph node 12 must be a `Cond` node whose condition C matches some C/Simpl expression. The export process could then use these requirements as a starting point in generating the actual SydTV-GL syntax.

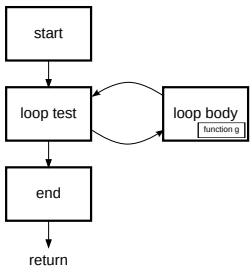
We could then *prove* about `emit` that any SydTV-GL function which satisfies all its output requirements must refine the input C/Simpl program. The complexity of the trace induction discussed above would still have to be considered, but it would be contained within the proof of correctness of `emit`, which would be performed once and for all.

One reason this option did not occur to us initially is that the C/Simpl programs are not fully deeply embedded. This `emit` process could recurse at the statement level, but would have more difficulty examining the inner expressions. This creates headaches for C statements like `break` and `continue` which are converted into the same C/Simpl statement structure with an inner expression that sets a global variable based on exception type. Handling this inner expression would require tricky queries on the function semantics, or some oracle to clarify what should be done.

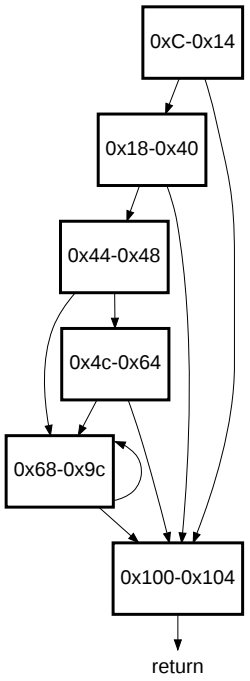
Another alternative would be to stay closer to Pnueli's original treatment [PSS98] in which a simulation relation is produced. Such a relation would map states of the SydTV-GL program to states of the C/Simpl program. Once again, care needs to be taken, because SydTV-GL programs are not fully deep embeddings either, but the relation probably only needs to consider the node address. The logical complexity of handling potentially infinite loops would move to the proof that a valid simulation relation implies a trace refinement.

```
int
g (int i) {
  return i * 8 + (i & 15);
}

void
f (int *p, int x) {
  int i;

  for (i = x; i < 100; i ++) {
    p[i] = g (i);
  }
}
```

(a) Example source code



(b) Example C CFG



(c) Example binary CFG

Figure 4.1: An example program.

4.4 Replaying the Translation Validation Proof

The refinement proof completed by SydTV-GL-refine establishes a simple notion of refinement between two programs, both expressed in SydTV-GL. The search component of SydTV-GL-refine produces a proof script, which the check process then checks. This section discusses our work in reflecting the steps of the check process into rules in the Isabelle/HOL [NPW02] proof environment.

Proving the proof procedures formally against a foundational implementation of HOL increases our confidence in the *design* of the SydTV-GL-refine check process. It also consitutes a significant first step toward repeating the entire proof within the Isabelle/HOL environment, which would eliminate all concerns about the *implementation* of SydTV-GL-refine.

We will introduce the steps of the check procedure as a collection of proof rules in Isabelle/HOL. Firstly, however, we will recap what the check procedure does via an example program we will introduce in Section 4.4.1 and a sketch of its proof of compilation correctness in Section 4.4.2. We will then introduce the key rules needed to formalise this proof in Section 4.4.4.

4.4.1 An Example Program

Consider the simple example program of two functions and one loop presented in Figure 4.1. While this is a simple program, when we compile it with GCC 4.5.1 and optimisation flag `-O2`, it becomes far more complex. The control flow graph (CFG) of the program, together with the simplified CFG of the compiled binary, are also shown in Figure 4.1.

The call to the function g is inlined by the compiler. The compiler has also introduced

far more conditional branching structure. The intent here is to unroll the central loop, with a single iteration of the loop starting at address `0x68` performing the actions of two iterations of the loop in the source. This combined iteration only checks `i < 100` once, which saves numerous cycles. This single test is sufficient, because the compiler ensures that `i` is always even at the start of the combined loop. If `i` is even and `i < 100`, it follows that `i + 1 < 100`, so the second test can be skipped. The additional code complexity before the loop entry in the binary CFG is to ensure that `i` is even by the start of the loop.

The first two checks of the binary CFG handle specifically the cases where the source loop is executed 0 or 1 times. The first execution of the source loop is also performed. The next key conditional checks whether the loop is to be executed an even number of times in total. If so, an additional copy of the loop body is executed, followed by a test for the case in which the loop was to be executed exactly 2 times and execution is now finished. The binary loop body then begins, executing the source loop actions twice and then checking whether to continue.

4.4.2 Informal Proof

Given this function `f`, and an intuition for the way it was compiled, we now work through the proof generated by SydTV-GL-refine.

The intuition we have given takes the compiler's view, describing a series of transformations on the source to unroll the loop and arrive at the binary structure. It would be possible to prove the correctness of the compilation in the same style, guessing which transformations had been applied and showing that each transformation was valid. The correctness proofs we will produce do not take this approach. Instead, we approach the problem analytically, considering an arbitrary collection of inputs to the two representations of `f`, and deriving a proof that their outputs will be equal.

The key component of the overall proof will be a proof by induction on the sequence of visits to a particular point in the binary loop. We will prove a relationship to the sequence of visits to a matching point in the source loop. To complicate this, the relationship between visits to the binary loop and visits to the source loop of `f` changes depending on which of the entry paths was taken in the binary loop. These two paths correspond to the cases where the loop in `f` would be executed an even or odd number of times. To avoid this complication, the first step of our proof is a case division on whether or not the instruction at binary address `0x4c` (see Figure 4.1) is ever executed. We will sketch the proof of the case where it is visited, the even case, in this section. The odd case is essentially the same with different parameters.

Given that instruction `0x4c` is visited, the next step of our proof is to show by induction that a sequence of visits to the address `0x68` within the binary (the start of the loop basic block) correspond to a sequence of visits to the start of the loop body of the source CFG. Because two executions of the source loop are handled specially in the binary, the first such visit to the binary point actually corresponds to the third visit to the source point. Because the binary loop is unrolled, the subsequent visits will correspond to every second source visit. So the binary visit sequence matches the subsequence of the 3rd, 5th, 7th etc visits to the source loop.

We prove the binary sequence and the source subsequence correspond by induction. By correspondence, we mean that the given visits occur in the same cases, so, if the binary point is visited 3 times but not a 4th time, the source point must be visited 7 times, but not 9 times. The corresponding visits also have related variable values. So the contents of memory will be the same at corresponding visits and the values of the relevant registers

can be expressed as functions of `i`, `x`, etc.

We are omitting the details here: SydTV-GL-refine discovers this relationship, and we are not interested in the specifics. The model-driven split discovery process was outlined in Section 2.2.2, and the linear sequence optimisation outlined in Section 2.4.3 is of particular assistance in computing this loop relation.

The induction proof must establish that, given related n-th visits of these visit sequences, the two loops either proceed to related $(n + 1)$-th visits or they both exit. It must also show that the 1st visits to the visit sequences (which are the 3rd and 1st visits to the actual points of the CFGs) either both occur or both do not occur. These checks all concern only a bounded number of steps of the source and binary programs, and the relevant proof obligations can be converted to a finite-size, decidable expression in the SMT language, and can be checked by an SMT solver.

Having proven that the visit sequences are related, we now consider three cases for the length of the visit sequences. The first case is that they are both infinite, which is possible for loops in C. It happens we can exclude this case since our source loop is bounded, but we don't need to use this reasoning. If both programs execute forever, then the compilation was valid.[4]

The second case is where both subsequences are empty. This means that the binary point is not visited and the source point is visited at most 2 but not 3 times. In this case the number of possible paths through the two CFGs is bounded again, and we can once again derive expressions for the total effect of the two functions. We then need to prove that the results of these functions are the same. Function `f` having return type `void`, the only obligation is to prove that the output memory values are equal across all of the paths possible given the CFG constraints we have mentioned. This equality can once again be exported to SMT and checked.

The third case is where the subsequences contain some finite number of visits. If we call this number of visits n, then we know that the two executions reach related n-th visits, but that the $(n+1)$-th visits are not reached. Specifically, this means that the binary address `0x68` is visited n times, but not $(n + 1)$ times, and that the start of the C loop is visited $(2n + 1)$ times, with matching values at visit $(2n + 1)$, but it is not visited $(2n + 3)$ times. These constraints once again limit us to finitely many paths through the CFGs. We can once again derive expressions for the final memory values at the return points of the functions, this time as expressions over the unknown but related values of the variables at the matching visits. Once again we can prove the needed equality via SMT.

This is a sketch of a proof of correct compilation for the even case. Together with the proof of the odd case, which is similar, we have a proof that the compilation of this program was correct.

SydTV-GL-refine discovers this proof automatically. The aim here is to formalise the proof fully in Isabelle/HOL.

4.4.3 Refinement

The objective is to prove refinement, using the small-step and trace semantics for the graph language that was defined in Section 4.2.

The conventional notion of refinement would specify that for every trace of the binary

[4] In principle our program might have periodic external effects that can be observed before it terminates. We have not yet added such observable effects to our semantics. If we did, our induction treatment would still be valid, because the infinite traces would synchronize with each other infinitely often. We would additionally need to specify that the observable states were related at each point of synchrony.

SydTV-GL program, there must be a matching trace of the C SydTV-GL program. We will also require a starting state for the C program which matches the starting binary state according to the input relation of the function pairing (we discussed function pairings in Section 2.1.4). We must then discover a matching C execution. Normally this would be a terminating C execution which reaches `Ret` with return values related to the binary return values by the output relation of the function pairing. We also allow two other cases. If neither trace terminates, we will consider the traces to match. If the C function reaches `Err`, we consider the traces to match, whatever the binary trace is.

We also have one more choice about refinement semantics. Our choice is what to do if the C SydTV-GL program would diverge, either by looping forever or recursing without bound. C programmers would usually expect the compiler to respect their intentions even if an infinite loop clearly serves no other purpose. We call this the precise semantics. However GCC provides a language extension for `const` and `pure` functions. The compiler may remove calls to these functions if their results are discarded. In principle this means that a non-terminating program might be optimised into a terminating one. We also allow an imprecise semantics, which permits this optimisation. It is certainly debatable whether these attributes should ever be used for a possibly-nonterminating function, but we support the option.

4.4.4 Proof Rules

The proof scripts we introduced in Section 2.1 have four structuring proof rules, **Split**, **Restrict**, **CaseSplit** and **Leaf**, with which they present a refinement argument. The proof script captures the part of the proof which must be discovered heuristically. Once the proof script is known, the undecideable proof of refinement reduces to a collection of (decidable) proof obligations. In this work we convert these rules into proof rules in Isabelle/HOL.

We have already discussed these rules, in some sense, in the informally sketched proof we gave already in Section 4.4.2. Recall the logic we used in multiple cases of the proof, where we observed that there were now only finitely many possible paths through the CFGs, allowing some expression at some point in the graph to be expressed by exhaustively considering all the paths by which it might be defined. We formalise the idea of "finitely many possible paths" with the concept of a restriction on a trace. A restriction is placed on the number of visits to a given node in a graph, for instance, we might restrict the number of visits to the start of the loop body in `f` to be 0, 1 or 2.

We can restrict the total number of visits within a trace. We also use restrictions to identify different visits to the same node within a trace. In our sketched proof we considered a sequence of visits to a point p within the loop. We can identify the 3rd visit to p as the visit to p where the number of previous visits to p is restricted to the set $\{2\}$.

type_synonym `trace` = (ℕ ⇒ `stack`) `option`
type_synonym `restrs` = (ℕ ⇒ ℕ `set`)

constant `restrs_condition` :: `trace` ⇒ `restrs` ⇒ ℕ ⇒ `bool`

constant `visit` :: `trace` ⇒ `next_node` ⇒ `restrs` ⇒ `state option`
constant `restrs_eventually_condition` :: `trace` ⇒ `restrs` ⇒ `bool`

definition `pc` :: `trace` ⇒ `next_node` ⇒ `restrs` ⇒ `bool`
where
`pc` *tr nn restrs* = (`visit` *tr nn restrs* ≠ `None`)

constant `restrs_list` :: (ℕ × ℕ) `list` ⇒ `restrs`

Formally the constant `restrs_condition` defines if, at a given step in a trace, the restricted nodes have already been visited a matching number of times. From this we derive `visit`, which gives us the variable state at the first satisfying visit, if there is one, and `restrs_eventually_condition` which tells us if the restrictions hold eventually (at some point and then forever afterwards) on the trace. We define `pc`, the path condition of a visit, as the condition that a satisfying visit occurs.

The `restrs_list` constant defines a set of restrictions by explicitly listing them.

constant `restr_trace_refine` :: `bool`
⇒ `graph_env` ⇒ `fname`
⇒ `graph_env` ⇒ `fname`
⇒ `restrs` ⇒ `restrs` ⇒ `output_relation`
⇒ `trace` ⇒ `trace` ⇒ `bool`

The `restr_trace_refine` predicate captures our proof obligation: trace refinement given restrictions. The first argument specifies the precise or imprecise semantics, which we will explain in a moment. The next four arguments specify the graph functions (by name) and the function environments they exist in.

The two restriction sets apply to the two traces. We may assume that each restriction set holds, in the sense of `restrs_eventually_condition`, on the trace of interest. The output relation defines the equalities that must hold on the values returned by the two programs. In this formalisation we also specify the two traces, that is, we assume we can name the traces ahead of time. This is because we formalise SydTV-GL programs as deterministic and always enabled, thus, from the starting SydTV-GL C state, we can already name the trace that the program generates. Our objective is to prove that the traces match, not discover a trace. We will revisit this assumption in Section 4.5.

We will formalise our key proof rules as a rule calculus for this predicate.

The restriction sets above are key to the proof structure. By imposing syntactic limits on the possible paths the trace may take, the restrictions allow us to talk about specific visits to nodes, whether those visits occur within loops or after loops. Notionally nodes within loops can be reached via an infinite sequence of possible paths until a restriction is added.

The **Restrict** rule introduces a new restriction on the visits to node n. The rule introduces the restriction that n is visited at least i times and less than j times overall. When we said in our informal proof that "the source point is visited at most 2 but not 3 times" we were appealing to the **Restrict** rule, with n the address of the source point, $i = 0$ and $j = 3$.

This rule adds to a restriction set listed by the `restr_list` operator. We add to a list with the Isabelle/HOL list constructor which we write `<Cons>` here. The restriction is added in the sense that we must now finish the proof, but we have the new restriction available in the future.

The restriction is on visits to node n. We show n must be reached (`pc` is true) at a point where n has been encountered $i - 1$ times. Likewise we show n cannot be reached (`pc` is false) when it has already been encountered $j - 1$ times. Thus n is reached at least i times and less than j times overall.

theorem `restr_trace_refine_Restr1` :
$j \neq 0$
$\longrightarrow$ `distinct` (`map fst` rs)
$\longrightarrow$ `wf_graph_function` f $ilen$ $olen$
$\longrightarrow$ Γ nm = `Some` f
$\longrightarrow$ $rs_i = (n, [i - 1])$ `<Cons>` `restrs_visit` rs (`NextNode` n) f
$\longrightarrow$ $rs_j = (n, [j - 1])$ `<Cons>` `restrs_visit` rs (`NextNode` n) f
$\longrightarrow$ ($i \neq 0 \longrightarrow$ `pc` (`NextNode` n) (`restrs_list` rs_i))
$\longrightarrow$ $\neg$ `pc` (`NextNode` n) (`restrs_list` rs_j)
$\longrightarrow$ `restr_trace_refine` $prec$ Γ nm Γ' nm'
(`restrs_list` $((n, [i.. < j])$ `<Cons>` $rs)$) rs' $orel$ tr tr'
$\longrightarrow$ `restr_trace_refine` $prec$ Γ nm Γ' nm' (`restrs_list` rs) rs' $orel$ tr tr'

Formalising this rule led to the discovery of a number of side conditions that were only checked implicitly in SydTV-GL-refine. Arithmetic on naturals underflows in Isabelle/HOL, so we must check for the case of zero. We also require some simple wellformedness properties on our graph functions, such as that all graph arcs go to nodes defined within the graph. Within SydTV-GL-refine, the problem representation combines both program graphs within a single numerical namespace, allowing a single set of restrictions to apply to both cases. This was a challenge for our formalisation, where instead we distinguish between this rule **Restrict1** and a symmetric counterpart **Restrict2** which affects the right hand side trace. These are expected variations between the formal and informal developments.

One variation was not expected. The `restrs_visit` constant here *discards* some of the existing restriction information. It drops any restrictions on nodes still reachable from n while we are testing the preconditions. We know that these restrictions hold for the total trace, in the sense of `restrs_eventually_condition`. However it is possible that these restrictions talk about nodes we have yet to reach at the time we reach node n. The `distinct` constraint above is needed for `restrs_visit` to have the correct effect.

It happens that we never explicitly checked for this case in SydTV-GL-refine, allowing its check process to potentially admit an unsound proof. This does not lead to a serious issue, since SydTV-GL-refine generates proofs in which restrictions are created in a sensible order. It would never return to a node prior to where a restriction has already been applied. Nonetheless exposing these technicalities is exactly the purpose of this formalisation.

The **Leaf** rule is the terminal rule of proof scripts. In SydTV-GL-refine, the proof ends once the number of possible paths through the CFGs is finite, and all output equalities can be expressed directly. The **Leaf** rule encodes this logic, checking the output relation of the traces directly under the assumption that enough restrictions are in place that the visit to the return points can be described concretely. It requires the return point of the binary trace to be reached, which essentially means that possibly-nonterminating loops have been handled. In the precise semantics, we need to know the source trace reaches its return point also.

theorem `restr_trace_refine_Leaf` :
`wf_graph_function` $f\ ilen\ olen$
$\longrightarrow \Gamma\ nm =$ `Some` f
$\longrightarrow$ `wf_graph_function` $f'\ ilen'\ olen'$
$\longrightarrow \Gamma\ nm' =$ `Some` f'
$\longrightarrow$ `pc Ret` $rs\ tr$
$\longrightarrow$ `output_rel` $orel\ (f, f')\ (rs, rs')\ (tr, tr')$
$\longrightarrow (prec \longrightarrow$ `pc Ret` $rs'\ tr')$
$\longrightarrow$ `restr_trace_refine` $prec\ \Gamma\ nm\ \Gamma'\ nm'\ rs\ rs'\ orel\ tr\ tr'$

The most involved of these rules is the **Split** induction rule, which provides the mechanism for reasoning about loops. In the proof sketch in Section 4.4.2 we proved by induction a relation on subsequences of visits to a source and binary graph node. We then considered three cases, the case of infinitely many visits, which automatically implies refinement, the case where the subsequence does not begin, and the case where the subsequence contains exactly n visits for some positive n. The **Split** rule performs both these logical divisions within a single step. The complication, compared to our informal description in Section 4.4.2, is that the general **Split** rule is designed for k-induction, where k previous visits $n \ldots n + k - 1$ are used to show the inductive step to visit $n + k$. The case we described previously is the special case where $k = 1$.

theorem `restr_trace_refine_Split` :
$vres = (\lambda\, i.\ \texttt{restrs\_list}\ ((sp, [start + i * step])\ \texttt{<Cons>}\ rs))$
$\longrightarrow vres' = (\lambda\, i.\ \texttt{restrs\_list}\ ((sp', [start' + i * step'])\ \texttt{<Cons>}\ rs'))$
$\longrightarrow vpc = (\lambda\, i.\ \texttt{pc}\ (\texttt{NextNode}\ sp)\ (vres\ i)\ tr)$
$\longrightarrow vpc' = (\lambda\, i.\ \texttt{pc}\ (\texttt{NextNode}\ sp')\ (vres'\ i)\ tr')$
$\longrightarrow visits = (\lambda\, i.\ (\texttt{visit}\ tr\ (\texttt{NextNode}\ sp)\ (vres\ i),$
$\texttt{visit}\ tr'\ (\texttt{NextNode}\ sp')\ (vres'\ i)))$
$\longrightarrow rel = (\lambda\, i.\ vpc\ i \wedge vpc'\ i \wedge vrel\ i\ (visits\ 0)\ (visits\ i))$
$\longrightarrow (\forall\, i.\ vpc\ (\texttt{Suc}\ i) \longrightarrow vpc\ i)$
$\longrightarrow (\forall\, i.\ i < k \longrightarrow vpc\ i \longrightarrow rel\ i)$
$\longrightarrow (\forall\, i.\ vpc\ (i + k) \longrightarrow (\forall\, j.\ j < k \longrightarrow rel\ (i + j)) \longrightarrow rel\ (i + k))$
$\longrightarrow k > 0 \longrightarrow step > 0 \longrightarrow step' > 0$
$\longrightarrow (\neg\ vpc\ k \longrightarrow \texttt{restr\_trace\_refine}\ prec\ \Gamma\ nm\ \Gamma'\ nm'$
$(\texttt{restrs\_list}\ rs)\ (\texttt{restrs\_list}\ rs')\ orel\ tr\ tr')$
$\longrightarrow (\forall\, i.\ \neg\ vpc\ (i + k)$
$\longrightarrow (\forall\, j.\ j < k \longrightarrow rel\ (i + j))$
$\longrightarrow \texttt{restr\_trace\_refine}\ prec\ \Gamma\ nm\ \Gamma'\ nm'$
$(\texttt{restrs\_list}\ rs)\ (\texttt{restrs\_list}\ rs')\ orel\ tr\ tr')$
$\longrightarrow \texttt{restr\_trace\_refine}\ prec\ \Gamma\ nm\ \Gamma'\ nm'$
$(\texttt{restrs\_list}\ rs)\ (\texttt{restrs\_list}\ rs')\ orel\ tr\ tr'$

The additional variables here abbreviate $vpc\ i$ as the path condition of the i-th subsequence visit in the binary program, $rel\ i$ as the condition that the i-th subsequence visits both occur and are related, etc.

The rule requires that the subsequence visits happen in order, in particular that a visit to $i + 1$ implies a visit to i.

The initial condition of the induction is that the first k binary visits have matching visits if they occur ($vpc\ i \longrightarrow rel\ i$ for $i < k$). The inductive condition is that if binary visit $i + k$ occurs and the previous k visits are related (by rel) then the source visit $i + k$ must also be related.

The rule also requires some extra elementary checks, such as $k > 0$.

The last two premises of the **Split** rule are the subproofs to be addressed. The first case is where the binary subsequence does not include k visits. The second case is where there are exactly $i + k$ binary visits, and the last k visits are known to be related by rel. No restrictions are added, but the rules are designed so that in each case sufficient information exists to immediately restrict the visits to the two split points using the **Restrict** rules.

Unlike the proof of **Restrict**, the proof of the induction rule in Isabelle/HOL was complex, but did not introduce any interesting checks that were not performed in SydTV-GL-refine. The formal constraints that k and the subsequence step sizes must be positive, and that the binary path conditions must be monotonic, were unsurprising formal additions.

The **CaseSplit** rule is the path-condition case-split which we appealed to at the beginning of the informal proof in Section 4.4.2. We divide into subproofs depending on whether a given node is visited. This is logically trivial but necessary for our tool to produce **Split** rules handling unrolled loops like the one we have seen, with different entry paths for different cases.

theorem `restr_trace_refine_PCCases1` :
$(\texttt{pc}\ nn\ rs_1\ tr \longrightarrow$
$\quad \texttt{restr\_trace\_refine}\ prec\ \Gamma\ nm\ \Gamma'\ nm'\ rs\ rs'\ orel\ tr\ tr')$
$\longrightarrow (\neg\ \texttt{pc}\ nn\ rs_1\ tr \longrightarrow$
$\quad \texttt{restr\_trace\_refine}\ prec\ \Gamma\ nm\ \Gamma'\ nm'\ rs\ rs'\ orel\ tr\ tr')$
$\longrightarrow \texttt{restr\_trace\_refine}\ prec\ \Gamma\ nm\ \Gamma'\ nm'\ rs\ rs'\ orel\ tr\ tr'$

The **Err** rule generates the assumption that `Err` is not visited, with any restrictions that are appropriate. This is done implicitly within SydTV-GL-refine, with various restriction choices applied. This rule allows this process to be made explicit.

theorem `restr_trace_refine_Err` :
$(\neg\ \texttt{pc}\ \texttt{Err}\ rs_1\ tr \longrightarrow$
$\quad \texttt{restr\_trace\_refine}\ prec\ \Gamma\ nm\ \Gamma'\ nm'\ rs\ rs'\ orel\ tr\ tr')$
$\longrightarrow \texttt{restr\_trace\_refine}\ prec\ \Gamma\ nm\ \Gamma'\ nm'\ rs\ rs'\ orel\ tr\ tr'$

The final structural rule we add is the **Call** rule. We decompose our refinement proof at function boundaries, and this is the rule that allows us to appeal to the proof that links the called functions. The **Call** rule is technically intricate both to state and prove, and the version we have proven is still insufficiently general. We will leave the technicalities out of this discussion, and say only that we are confident that the call matching part of the SydTV-GL-refine SMT export process can indeed be shown correct in Isabelle/HOL.

These proof rules form the proof calculus we need for `restr_trace_refine`. We can express refinement as an instance of `restr_trace_refine`, and then apply these rules to decompose a refinement problem until only decidable proof obligations remain.

For our example program `f`, the proof structure begins with a **PCCases** on the path into the loop, each possibility being addressed by a **Split** rule. Each split rule generates an initial ($< k$) and inductive ($\geq k$) case, a total of 4 cases, each of which is addressed by a pair of **Restrict** rules to fix the number of visits to the two loops followed by a **Leaf** rule. We have implemented an automatic mechanism for composing and applying this compound proof rule. When applied to our example problem, it discharges all `restr_trace_refine` goals, leaving 70 remaining proof obligations, 30 of them non-trivial.

4.4.5 Further Work for Isabelle/HOL Replay

We have formalised sufficient rules to reduce our refinement problem for `f` down to 30 non-trivial proof obligations. These obligations are logical expressions which concern various specific `visit` instances.

These proof obligations match, at a high level, the proof obligations that are generated within SydTV-GL-refine. To continue to realise the logic of SydTV-GL-refine within Isabelle/HOL, we would next need to replicate the SMT export process which we introduced in Section 2.1.6. This would take the form of a rewrite mechanism that would expand applications of the `visit` function into concrete SMT-compatible values, depending on whether the previously visited nodes were `Basic`, `Cond` or `Call` nodes, or if the node has multiple predecessors, etcetera.

The final step would be to export the problem to SMT, and replay the SMT "unsat" judgement as a theorem in Isabelle/HOL. SMT export and replay has been studied in detail by Böhme [BW10], and we have also made minor contributions to this problem [BFSW11]. We suspect, however, that more work remains to be done here. Replay of SMT proofs for

the bitvector problem can be quite expensive. In addition, not all SMT solvers produce traces to be replayed, and the most effective solvers are usually the ones that do not produce proof traces.

We hope to return to this issue in future work.

For the time being, we have formalised the essential logical principles of SydTV-GL-refine, and are satisfied that they are correct. Formalising the rules has uncovered a modest number of side conditions that we had not checked in SydTV-GL-refine, but none that lead to concern about our existing results.

4.5 The Axiom of Dependent Choice

This section is a note about the theoretical underpinnings of this work that we encountered more or less by accident. It relates to the style of induction which we used to prove trace refinement in both our replay proof (Section 4.4) and export proof (Section 4.3.3). Both of these proofs establish that for each trace of the implementing programs there exists a matching trace of the specifying program. We allow nontermination, so we must sometimes show the existence of infinite traces.

When we formalised these proofs in Isabelle/HOL, we were surprised that we were often unable to complete the proofs by induction arguments alone, and instead had to make explicit use of the axiom of choice. We were surprised again to realise that this was not specific to our approach. An infinite trace is a single object which potentially encodes infinitely many decisions. The existence of such single decision objects is (roughly speaking) what the axiom of choice asserts.

Let us be more precise. The induction schema we used in the export proof (Section 4.3.3) is *equivalent* to the *axiom of dependent choice*, a related axiom to the general axiom of choice. The split induction theorem of SydTV would also be equivalent if we extended it to include nondeterminism. Moreover, some quite different trace refinement results from the literature are also equivalent.

In fact, we can make a stronger hypothesis:

> **Hypothesis 1.** All approaches for proving trace refinement which permit both nondeterminism and nontermination are by default logically equivalent to the axiom of dependent choice.

We say “by default” above because there exist a number of simple countermeasures to reduce the logical strength of the problem. For instance, we can insist that all types in our programs are countable, which gives us a simple choice procedure. We will discuss other strategies in Section 4.5.4.

This discovery might not have profound implications, since it can be ignored by those comfortable with the axiom of choice and worked around by those who are not. Moreover, once we demonstrate instances of our hypothesis, it will become instantly clear why it is true. What is surprising about this fact is that it does not seem to be well-known in cases where it is relevant.

4.5.1 The Axiom

The axiom of choice is an optional member of the axioms of Zermelo-Fraenkel (ZF) set theory [Zer08, FBHL73]. Conventional mathematics texts, including the venerable

"Principia Mathematica" [RW10] use ZF set theory including the axiom of choice (ZF+C) as the logical foundation from which other mathematics is derived.

The axiom of choice says that, given an indexed family of nonempty sets, here represented by a function S from index elements in the domain D to sets, there exists a function which picks one element of each set:

$$(\forall i \in D.\ \exists x.\ x \in S\ i) \longrightarrow (\exists X.\ \forall i \in D.\ X\ i \in S\ i)$$

The axiom of choice is important in coping with complexities introduced by infinity. It is also controversial, giving rise to some counterintuitive results. Some mathematicians have gone so far as to develop parallel mathematics texts which derive all possible results from ZF with the axiom of choice excluded (ZF-C).

A compromise is to consider weaker variants of the axiom of choice. The axiom of *countable* choice is identical to the axiom of choice, but insists that the domain of the indexed family of sets is countable. This is equivalent to the previous presentation where the domain D is the set of natural numbers:

$$(\forall i \in \mathbb{N}.\ \exists x.\ x \in S\ i) \longrightarrow (\exists X.\ \forall i \in \mathbb{N}.\ X\ i \in S\ i)$$

The axiom of countable choice is less controversial. It does not give rise to such surprising results as the axiom of choice itself.

Unfortunately, the axiom of countable choice is not strong enough for the results we need to prove. The axiom of *dependent* choice generalises the axiom of countable choice by saying that later decisions can depend on earlier ones. For simplicity, we say that later decisions depend on the previous one, so our function S now takes an additional parameter from the domain D:

$$(\forall i \in \mathbb{N}.\ \forall x \in D.\ \exists y.\ y \in S\ i\ x \wedge y \in D) \longrightarrow (\exists Y.\ \forall i \in \mathbb{N}.\ Y\ (i+1) \in S\ i\ (Yi))$$

These variations of the axiom of choice are well-known, but their consequences have not yet been as well studied as the choice axiom itself. It is clear that general choice implies dependent choice implies countable choice.

These axioms form a sequence of strictly stronger logics, i.e. ZF-C, ZF with countable choice, ZF with dependent choice, and ZF+C are four different logics with different universes of provable facts.

4.5.2 Foundations

This may seem to some readers like a pointless exercise. After all, we are using the Isabelle/HOL logic, which fundamentally assumes the Hilbert choice operator, one of the strongest presentations of the axiom of choice. Why does it matter whether or not we use a weaker variant as well?

The reason is that, while we are using Isabelle/HOL, we are not ideologically committed to its logic, and we are committed to ongoing dialogue with the rest of the field of formal methods.

The field of formal methods and formal verification has its foundation in the principles of mathematical logic. The problem, which we encounter here, is that the foundations of mathematical logic have never been adequately settled. In addition to variants of Zermelo-Fraenkel set theory, which we have discussed, there are various constructive logics [CH88]

and intuitionistic logics [Kri65, Dum75] all making reasonable cases that they should be the foundation stone of formal reasoning.

We claim that the working formal methodist hopes to avoid such deep discussions. Instead, he or she picks a given implementation of a given logic, but hopes that his or her contributions (results and approaches) have universal relevance even if the specific details of the proofs do not.

This is why this particular observation is interesting. We thought we were studying the simplest possible refinement ordering on programs which do not necessarily terminate, one which had been studied by many others. Instead, we discover that we cannot satisfactorily define this problem until we answer the most difficult questions about the theoretical universe we inhabit.

4.5.3 Logical Equivalence

We claim that any approach for proving trace refinement which permits both nondeterminism and nontermination will be equivalent to the axiom of dependent choice by default. We develop this proof in detail (including an Isabelle/HOL presentation) for a handful of such proof approaches, but firstly let us sketch the concept of the proof.

The interesting aspect of the proof for our purposes is the part that shows that the axiom of dependent choice is implied by trace refinement. The axiom can be simplified down to the problem of discovering an infinite trace of steps within some relation X on elements of domain D:

$$X \subseteq D \times D \longrightarrow (\forall x \in D.\, \exists y.\, (x, y) \in X) \longrightarrow (\exists Y.\, \forall i \in \mathbb{N}.\, (Y\ i, Y\ (i + 1)) \in X)$$

Now that the axiom is phrased in this way, it is almost transparent what we need to do. We will prove trace refinement between two programs. The specifying program will make decisions from X. The implementating program will execute an infinite loop doing nothing. Refinement will imply the existence of an infinite trace of the specifying program. This trace will encode an infinite trace of steps within relation X.

One foundational approach to trace refinement is to present a simulation relation [LV95]. Given a pair of programs specified by a small-step semantics, a simulation relation maps some configurations of the implementing program to related configurations in the specifying program. A simulation relation requires that there exists some bound n. The implementing program must reach a state with a related counterpart at least every n steps. For each path in the implementing program between configurations that are related, there must exist a path of at most n steps in the specifying program that connects the related states.

It is trivial to recover the axiom of dependent choice from the general principle of simulation relations. We pick a specifying program whose small-step semantics is exactly the relation X, an implementing program whose small-step relation is $(0, 0)$, and our simulation relation maps the state 0 to the set of states D. Given that X has forward steps within D, the next step of the implementation (to 0) will be related to some next step of the specification (within X). This proves the simulation relation. The implementing program has an infinite trace all of whose elements is 0. If the simulation relation proves trace refinement, then it follows there is an infinite trace of the specification, which is exactly the infinite trace of X steps which the axiom specifies.

In short, this general formulation of the principle of simulation relations fits more or less *exactly* to the shape of the axiom of dependent choice.

Picking the small-step relation of a program to fit our problem is a bit unorthodox. In a more conventional setting the small-step relation would be fixed by the semantics of some language. Instead we should pick a program based on X.

As a simple illustration, we borrow a simple `While` language presented by Nipkow [Nip06]. The language syntax has four constructors, `Semi` (semicolon, sequential composition), `Cond` (conditional, if-then-else), `While` (looping) and `Do`. The `Do` operator abbreviates a few different kinds of actions. It takes a parameter of type $\alpha \Rightarrow \alpha$ `set`, where α is the state. This allows the action "done" to introduce nondeterminism.[5]

We borrow this syntax unchanged, but present the semantics differently. Nipkow uses a big-step semantics and addresses the notion of termination separately. We produce the counterpart small-step semantics. We have defined this small-step relation in Isabelle/HOL, and will elide the details here. We produce a trace semantics for the language using the `trace_set` operator presented in Section 4.2.

We can now specialise the argument about simulation relations to this language. Abusing syntax slightly, we try to prove:

$$\texttt{While True (Do } (\lambda s.\ \{s\})) \texttt{ <refines> While True (Do } (\lambda s.\ \{s'.\ (s, s') \in X\}))$$

We have, on the left, yet another instance of the program that does nothing forever. The program on the right picks steps from X. The small step semantics relates configurations that consist of a program to execute and a starting state. We can employ a simulation relation here. We pick a simulation relation which relates the left hand side program and any state to the right hand side program and any state from D. We set n to 2.

Indeed, after 2 steps, the left hand side program will return to where it was. The requirement that a 2-step path exist on the right hand side is once again equivalent to the condition that X has forward steps within D.

Refinement will again establish a trace. Here we must discard every second step, and then the state elements will form the desired trace with forward steps in X.

Another classic approach to proving refinement is to introduce a syntax-directed refinement calculus. The composition rule for the constructor `While` might look something like this:

$$\begin{aligned}&(\forall (s, s') \in S.\ (s \in C) = (s' \in C')) \\ &\quad \longrightarrow c \texttt{ <refines>}_S\ c' \\ &\quad \longrightarrow \texttt{While } C\ c \texttt{ <refines>}_S \texttt{ While } C'\ c'\end{aligned}$$

This notion of refinement includes a state relation S which allows some data refinement. It would probably require some more parameters to be really useful, but these are not necessary for our purposes.

It should be clear how we will derive an infinite X trace from the `While` refinement rule. We use the same programs as before. The state relation again relates any left state to right states that are in D. The looping conditions trivially match. The condition that the loop bodies refine should reduce to the expected requirement on X.

We could go on, but the reader is hopefully convinced at this point.

We proved some additional more complex cases in Isabelle/HOL. The future style of trace induction we needed in our export proof (Section 4.3.3) also implies the axiom. This

[5] This is true in Nipkow's 2006 version of this work [Nip06], but not in an earlier version [Nip02] which handled nondeterminism differently.

follows easily from the fact that the future refinement can be used to prove the `While` refinement rule above.

Our split induction principle also implies the axiom, if we extend SydTV-GL with nondeterminism by saying some functions might have weakly specified nondeterministic bodies. Since this proof will require function calls to involve choices from the relation X, it becomes quite involved.

To complete the proof of equivalence, we must also prove all of these results using the axiom of dependent choice and not any stronger variants. We first prove the future style of trace induction (see Section 4.3.3) using the axiom of dependent choice. Our other results follow from future induction. The essential step of that proof involves a set of "configurations" which have a partial order and in which every element has a strictly greater element. We take the order relation as X, the set of configurations as D, and use the axiom to pick a single infinite sequence of configurations which are strictly increasing according to the order. From this sequence we can extract a single infinite trace as needed in the relevant proof.

4.5.4 Countermeasures

While this observation applies by default to any notion of trace refinement we can think of, there are several possible adjustments that remove the issue.

The simplest would be to require that all variables take their values from some countable or ordered set. This gives us a choice mechanism whenever it is necessary.

Another possibility is to permit uncountable variables, but enforce countable choice. Whenever the specifying system makes a nondeterministic decision, it must do so from among a countable collection of options.

Yet another possibility is to require the simulation relation to be functional, to map states of the implementing system to specific states of the specifying system. This moves the decision process from the proof into this function. A related strategy is to require that key aspects of the proof are *constructive*, which will imply the existence of a function that converts implementation traces to specification traces. The proof rules for loops will have to stitch together these functions, but won't actually have to make any decisions.

The other possibility is to adjust the semantics of trace refinement, abandoning the construction of infinite traces. For instance, one option would be to have the refinement imply for each natural number N the existence of a specification trace that matches for at least N steps. This avoids ever constructing an infinite trace, but has the same implications within any specific window of time.

4.5.5 Implications

This discovery will not change the world. The implications of the axiom of dependent choice are in any case less troubling than those of the full axiom of choice, and many authors may simply accept them. Others may employ one of the workarounds described above.

What is interesting about this discovery is that it concerns some theorems that are so well known. The statements "forward simulation implies refinement" or "simulation relations imply refinement" are so well understood in the literature that venerable authorities appeal to them as theorems without detailing their proofs [BvW94, LV95, BO01], including in infinite trace cases. In some cases, by careful reading of the proofs, we can actually see where the authors have appealed to the axiom of choice without explicitly

mentioning it, for instance in proposition 8.2 of van Glabbeek's discussion of simulation semantics [vG99].

While we have no objections to any of the existing literature, we think that we add slightly to it by noting the logical strength of the statements being made.

5 Conclusion

We have explored an approach for translation validation of C programs and its implementation in SydTV. We have also, through a series of experiments, comprehensively studied both the applicability of the approach and the effectiveness of the implementation.

Translation validation is an approach to the problem of binary correctness which aspires to be both *flexible* and *lightweight*. We have seen throughout this work evidence for the flexibility of our SydTV approach.

We have shown in Chapter 2 that the core SMT-powered model-guided proof search of SydTV can verify binaries produced via intricate optimisations. By combining with the verified seL4 source, we produce a final efficient verified binary. We are confident that this is the most comprehensively verified OS binary produced to date.

We have seen in Chapter 3 that SydTV can adapt to the problem of timing analysis, a task that typically requires a specialised real-time compiler toolchain. By threading additional timing information through the seL4 verification proofs and the SydTV validation process, we can produce high-assurance evidence of the kernel's real-time performance.

While SydTV is flexible, it is grounded in precise formal logic. We have studied these theoretical foundations in Chapter 4, and discovered that doing so introduces unexpected insights into the theory of trace refinement.

In conclusion, we present SydTV, a comprehensively studied translation validation engine for producing verified and real-time binaries.

Availability

All aspects of this work are open-source. The three components of SydTV are available as part of the L4.verified distribution, the HOL4 distribution, and the SydTV-GL-refine repository (also called graph-refine) respectively. The seL4 proof manifest gathers all the necessary tools to run the binary verification. It can be found at:

`https://github.com/seL4/verification-manifest`

6 Bibliography

[AAB+13] Roberto M Amadio, Nicolas Ayache, Francois Bobot, Jaap P Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P Mulligan, Mauro Piccolo, et al. Certified complexity (CerCo). In *International Workshop on Foundational and Practical Aspects of Resource Analysis*, pages 1–18. Springer, 2013.

[AARG12] Nicolas Ayache, Roberto Amadio, and Yann Régis-Gianas. Certifying and reasoning on cost annotations in C programs. In *FMICS 2012 - 17th International Workshop on Formal Methods for Industrial Critical Systems*, Paris, France, Aug 2012.

[AHC+16] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *ASPLOS*, pages 175–188, Atlanta, GA, USA, Apr 2016.

[AHL+08] Eyad Alkassar, Mark Hillebrand, Dirk Leinenbach, Norbert Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In *VSTTE 2008*, volume 5295 of *LNCS*, pages 209–224, 2008.

[ALM15] June Andronick, Corey Lewis, and Carroll Morgan. Controlled Owicki-Gries concurrency: Reasoning about the preemptible eChronos embedded operating system. In *Workshop on Models for Formal Analysis of Real Systems (MARS 2015)*, pages 10–24, Suva, Fiji, Nov 2015.

[App11] Andrew Appel. Verified software toolchain. In *20th ESOP*, volume 6602 of *LNCS*, pages 1–17, 2011.

[APST10] Eyad Alkassar, Wolfgang Paul, Artem Starostin, and Alexandra Tsyban. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In *VSTTE 2010*, volume 6217 of *LNCS*, pages 71–85, Edinburgh, UK, Aug 2010.

[ARI12] *Avionics Application Software Standard Interface*, Nov 2012. ARINC Standard 653.

[BB08] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 6–11. ACM, 2008.

[BBB+09] James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russell Urzi. A research agenda for mixed-criticality systems. Available at `http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/`, Apr 2009.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[BDP12] Gilles Barthe, Delphine Demange, and David Pichardie. A formally verified SSA-based middle-end. In *European Symposium on Programming*, pages 47–66. Springer, 2012.

[Bev89] William R. Bevier. Kit: A study in operating system verification. *Trans. Softw. Engin.*, 15(11):1382–1396, 1989.

[BFSW11] Sascha Böhme, Anthony CJ Fox, Thomas Sewell, and Tjark Weber. Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In *International Conference on Certified Programs and Proofs*, pages 183–198. Springer, 2011.

[BH13] Bernard Blackham and Gernot Heiser. Sequoll: a framework for model checking binaries. In *RTAS*, pages 97–106, Philadelphia, USA, Apr 2013.

[BHHK10] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: algebraic bound computation for loops. In *16th Int. Conf. Logic for Progr., Artificial Intelligence & Reasoning*, pages 103–118, 2010.

[BHV11] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based CFG reconstruction from unstructured programs. In *Int. Conf. Verification, Model Checking & Abstract Interpretation*, pages 54–69, 2011.

[BLH14] Bernard Blackham, Mark Liffiton, and Gernot Heiser. Trickle: automated infeasible path detection using all minimal unsatisfiable subsets. In *RTAS*, pages 169–178, Berlin, Germany, Apr 2014.

[BO01] Manfred Broy and Ernst-Rüdiger Olderog. Trace-oriented models of concurrency. *Handbook of process algebra*, pages 101–195, 2001.

[BPYA15] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security*, pages 207–221, Washington, DC, US, Aug 2015. URL: `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer`.

[BR06] Claire Burguière and Christine Rochange. History-based schemes and implicit path enumeration. In *6th WS Worst-Case Execution-Time Analysis*, 2006.

[BSC+11] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *RTSS*, pages 339–348, Vienna, Austria, Nov 2011.

[BSH12] Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *EuroSys*, pages 323–336, Bern, Switzerland, Apr 2012.

[BSPH07] Jan Olaf Blech, Ina Schaefer, and Arnd Poetzsch-Heffter. Translation validation of system abstractions. In *Proc. 7th Int. Conf. on Runtime verification*, RV'07, pages 139–150, Vancover, Canada, 2007.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *Proc. 8th Int. Workshop on Satisfiability Modulo Theories*, Edinburgh, UK, 2010.

[BvW94] R. J. R. Back and Joakim von Wright. Trace refinement of action systems. In *CONCUR'94: Concurrency Theory*, pages 367–384. 1994.

[BW10] Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In *International Conference on Interactive Theorem Proving*, pages 179–194. Springer, 2010.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, Sep 2003.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and computation*, 76(2-3):95–120, 1988.

[Chl10] Adam Chlipala. A verified compiler for an impure functional language. In *ACM Sigplan Notices*, volume 45, pages 93–106. ACM, 2010.

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proc. 32nd PLDI*, pages 234–245, San Jose, California, USA, 2011.

[Chl13] Adam Chlipala. The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In *18th ICFP*, pages 391–402, 2013.

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *International SPIN Workshop on Model Checking of Software*, pages 248–254. Springer, 2012.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(02):56–68, 1940.

[CM07] Christoph Cullmann and Florian Martin. Data-flow based detection of loop bounds. In *7th WS Worst-Case Execution-Time Analysis*, 2007.

[CMB14] Franck Cassez, Christian Mueller, and Karla Burnett. Summary-based interprocedural analysis via modular trace refinement. In *FSTTCS*, pages 545–556, India, Dec 2014.

[Cra57] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(03):269–285, 1957.

[CS10] Ernie Cohen and Norbert Schirmer. From total store order to sequential consistency: A practical reduction theorem. In *1st ITP*, volume 6172 of *LNCS*, pages 403–418, Edinburgh, UK, Jul 2010.

[DLRA12] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 760–770, Piscataway, NJ, USA, 2012.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, Budapest, Hungary, Mar 2008.

[dRdBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. Concurrency verification: Introduction to compositional and non-compositional methods. *Cambridge Tracts in Theoretical Computer Science*, 2001.

[Dum75] Michael Dummett. The philosophical basis of intuitionistic logic. *Studies in Logic and the Foundations of Mathematics*, 80:5–40, 1975.

[Dut14] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[EaJGBL07] Andreas Ermedahl, Christer Sandberg anf Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *WS Worst-Case Execution-Time Analysis*, 2007.

[FBHL73] Abraham Adolf Fraenkel, Yehoshua Bar-Hillel, and Azriel Levy. *Foundations of set theory*, volume 67. 1973.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, pages 173–188, 2007.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, pages 469–485, London, UK, 2001.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19:19–32, 1967.

[FM10] Anthony Fox and Magnus Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *1st ITP*, volume 6172 of *LNCS*, pages 243–258, Edinburgh, UK, Jul 2010.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In *10th WS Worst-Case Execution-Time Analysis*, pages 137–147, Brussels, BE, Jul 2010.

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, pages 57–66, Washington, DC, US, 2006.

[GHE15] Peter Gammie, Tony (Antony) Hosking, and Kai Engelhardt. Relaxing safely: Verified on-the-fly garbage collection for x86-TSO. In *PLDI 2015: the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation.*, page 11, Portland, Oregon, United States, Jun 2015.

[GLAK14] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *PLDI*, pages 429–439, Edinburgh, UK, Jun 2014.

[GMKN17] Armaël Guéneau, Magnus O Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *ESOP*, pages 584–610, Apr 2017.

[GSC+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *OSDI*, Nov 2016.

[GVF+11] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. CertiKOS: A certified kernel for secure cloud computing. In *2nd APSys*, 2011.

[GZB05] Benjamin Goldberg, Lenore D. Zuck, and Clark 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.

[HAWH99] Christopher A. Healy, Robert D. Arnold, Frank Müller David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *Trans. Computers*, 48:63–70, Jan 1999.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. In *ACM SIGPLAN Notices*, volume 50, pages 336–345. ACM, 2015.

[HH08] AndrÃľ Hergenhan and Gernot Heiser. Operating systems technology for converged ECUs. In *6th Emb. Security in Cars Conf. (escar)*, page 3 pages, Hamburg, Germany, Nov 2008.

[HJM03] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Counterexample-guided control. In *30th ICALP*, pages 886–902, Eindhoven, The Netherlands, Jul 2003.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12:576–580, 1969.

[KAE+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70, Feb 2014.

[KBC10] Tai Hyo Kim, Ho Jung Bang, and Sung Deok Cha. A systematic representation of path constraints for implicit path enumeration technique. *Software Testing, Verification and Reliability*, 20(1):39–61, 2010.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009.

[KKP+11] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Software & Systems Modeling*, 10(3):411–437, 2011.

[KKU16] Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. Relational program reasoning using compiler IR. In *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers 8*, pages 149–165. Springer, 2016.

[KKZ11] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for WCET analysis. In *International Andrei Ershov Memorial Conference*, 2011.

[KKZ13] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In *RTNS*, RTNS '13, pages 161–170, New York, NY, USA, 2013.

[Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, Feb 2009.

[KLG10] Sudipta Kundu, Sorin Lerner, and Rajesh K. Gupta. Translation validation of high-level synthesis. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29(4):566–579, Apr 2010.

[KMNO14] Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *POPL*, pages 179–191, San Diego, Jan 2014.

[Kre15] Robbert Krebbers. *The C Standard formalised in Coq.* PhD thesis, Radboud University Nijmegen, 2015.

[Kri65] Saul A Kripke. Semantical analysis of intuitionistic logic I. *Studies in Logic and the Foundations of Mathematics*, 40:92–130, 1965.

[KZV09] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *10th Int. Conf. Verification, Model Checking & Abstract Interpretation*, pages 214–228, 2009.

[L+59] Roger C Lyndon et al. An interpolation theorem in the predicate calculus. *Pacific Journal of Mathematics*, 9(1):129–142, 1959.

[LABS14] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model. In *Program Logics for Certified Compilers*. Apr 2014.

[LCFM09] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *7th Symp. Code Generation & Optimization*, pages 136–146, Washington, DC, US, 2009.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *33rd POPL*, pages 42–54, Charleston, SC, USA, 2006.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[LH14] Anna Lyons and Gernot Heiser. Mixed-criticality support in a high-assurance, general-purpose microkernel. In *WS Mixed Criticality Syst.*, pages 9–14, Rome, Italy, Dec 2014.

[LH16] Anna Lyons and Gernot Heiser. It's time: OS mechanisms for enforcing asymmetric temporal integrity. *arXiv preprint*, 2016.

[Lie94] Jochen Liedtke. Page table structures for fine-grain virtual memory. *IEEE Technical Committee on Computer Architecture Newsletter*, Oct 1994.

[Lis05] Björn Lisper. Ideas for annotation language (s). Technical report, Technical Report Oct. 25, Department of Computer Science and Engineering, University of Mälardalen, 2005.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming, Special issue on Experimental Software and Toolkit*, 69(1–3):56–67, Dec 2007.

[LM95] Yau-Tsun Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, pages 456–461. ACM, Jun 1995.

[LMNR15] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 22–32, New York, NY, USA, 2015.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *3rd Int. Conf. Softw. Engin. & Formal Methods*, pages 2–12, Washington, DC, USA, 2005.

[LS98] Thomas Lundqvist and Per Stenström. Integrating path and timing analysis using instruction level simulation techniques. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, LNCS, Montreal CA, Jun 1998.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations. *Information and Computation*, 121(2):214–233, 1995.

[LWM13] Ye Li, Richard West, and Eric S. Missimer. The Quest-V separation kernel for mixed criticality systems. In *WS Mixed Criticality Syst.*, pages 31–36, Dec 2013.

[McM05] Kenneth L McMillan. Applications of Craig interpolants in model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12. Springer, 2005.

[McM11] Kenneth L McMillan. Interpolants from Z3 proofs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 19–27. FMCAD Inc, 2011.

[MGS08] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Machine-code verification for multiple architectures: An application of decompilation into logic. In *2008 FMCAD*, Piscataway, NJ, USA, 2008.

[MGS12] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation into logic – improved. In *2012 FMCAD*, pages 78–81, Cambridge, UK, Oct 2012.

[MIS12] MISRA. *Guidelines for the Use of the C Language in Critical Systems*, 2012.

[MMB+13] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *S&P*, pages 415–429, San Francisco, CA, May 2013.

[MML+16] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–15. ACM, 2016.

[MN89] Kurt Mehlhorn and Stefan Näher. LEDA a library of efficient data types and algorithms. In *International Symposium on Mathematical Foundations of Computer Science*, pages 88–106. Springer, 1989.

[MS16] Kayvan Memarian and Peter Sewell. N2090: Clarifying pointer provenance (draft defect report or proposal for C2x). Defect Report JTC1/SC22/WG14 N2090, 2016.

[MSG09] Magnus O Myreen, Konrad Slind, and Michael JC Gordon. Extensible proof-producing compilation. In *International Conference on Compiler Construction*, pages 2–16. Springer, 2009.

[MWT02] W. B. Martin, P. D. White, and F. S. Taylor. Creating high confidence in a separation kernel. *Automated Softw. Engin.*, 9(3):263–284, 2002.

[Nec97] George C. Necula. Proof-carrying code. In *24th POPL*, pages 106–119, Paris, France, Jan 1997.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94, Vancouver, British Columbia, Canada, 2000.

[Nip02] Tobias Nipkow. Hoare logics in Isabelle/HOL. In *Proof and System-Reliability*, pages 341–367, 2002.

[Nip06] Tobias Nipkow. Abstract hoare logics. *Archive of Formal Proofs*, August 2006. `http://isa-afp.org/entries/Abstract-Hoare-Logics.shtml`, Formal proof development.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. 2002.

[NRM14] Lars Noschinski, Christine Rizkallah, and Kurt Mehlhorn. Verification of certifying computations through AutoCorres and Simpl. In *NASA Formal Methods*, volume 8430 of *LNCS*, pages 46–61. 2014.

[NYS07] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic system code: Machine context management. In *20th TPHOLs*, volume 4732 of *LNCS*, pages 189–206, Kaiserslautern, Germany, Sep 2007.

[OCR+16] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *ICFP*, Nara, Japan, Sep 2016.

[Pau90] Lawrence C Paulson. Isabelle: The next 700 theorem provers. In *Logic and computer science*, volume 31, pages 361–386, 1990.

[PK89] Peter Puschner and Ch Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[PKK+09] Adrian Prantl, Jens Knoop, Raimund Kirner, Albrecht Kadlec, and Markus Schordan. From trusted annotations to verified knowledge. In *WS Worst-Case Execution-Time Analysis*, pages 35–45, Dublin, IE, Jun 2009.

[PS91] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source–level timing schema. *Trans. Computers*, 24(5):48–57, May 1991.

[PS96] Amir Pnueli and Elad Shahar. A platform for combining deductive with algorithmic verification. In *International Conference on Computer Aided Verification*, pages 184–195. Springer, 1996.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *4th TACAS*, pages 151–166, Lisbon, Portugal, Mar 1998.

[PVL11] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In *NFM*, pages 298–312, Pasadena, CA, USA, 2011.

[Ray14] Pascal Raymond. A general approach for expressing infeasibility in implicit path enumeration technique. In *Proceedings of the 14th International Conference on Embedded Software*, page 8. ACM, 2014.

[RLN+16] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. In *ITP*, Nancy, France, Aug 2016.

[RPW08] Bernhard Rieder, Peter Puschner, and Ingomar Wenzel. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement based WCET analysis. In *Intelligent Solutions in Embedded Systems, 2008 International Workshop on*, pages 1–7, Jul 2008.

[RS09] Michael Ryabtsev and Ofer Strichman. Translation validation: From Simulink to C. In *Proc. 21st Int. Conf. on Computer Aided Verification*, CAV '09, pages 696–701, Grenoble, France, 2009.

[Rus81] John Rushby. Design and verification of secure systems. In *SOSP*, pages 12–21, Pacific Grove, CA, USA, Dec 1981.

[RW10] Bertrand Russell and Alfred North Whitehead. Principia mathematica vol. I. 1910.

[SBCA15] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W Appel. Compositional CompCert. *ACM SIGPLAN Notices*, 50(1):275–287, 2015.

[SBH13] Yao Shi, Bernard Blackham, and Gernot Heiser. Code optimizations using formally verified properties. In *OOPSLA*, pages 427–442, Indianapolis, USA, Oct 2013.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[Sch10] Bastian Schlich. Model checking of software for microcontrollers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(4):36, 2010.

[Sew14] Thomas Sewell. Formal replay of translation validation for highly optimised C: Work in progress. In *Verification and Program Transformation*, Vienna, Austria, July 2014.

[SFS11] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-based function summaries in bounded model checking. In *Haifa verification conference*, pages 160–175. Springer, 2011.

[SG05] Ofer Strichman and Benny Godlin. Regression verification-a practical way to verify programs. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 496–501. Springer, 2005.

[SKH16] Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *RTAS*, Vienna, Austria, Apr 2016.

[SKH17] Thomas Sewell, Felix Kam, and Gernot Heiser. High-assurance timing analysis for a high-assurance real-time OS. *Real-Time Syst.*, 53:812–853, Sep 2017.

[SMK13] Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–481, Seattle, Washington, USA, Jun 2013.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[STL11] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for LLVM. In *Computer Aided Verification*, pages 737–742. Springer, 2011.

[SWG+11] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *ITP*, pages 325–340, Nijmegen, The Netherlands, Aug 2011.

[TGM11] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *Proc. 32nd PLDI*, pages 295–305, San Jose, CA, USA, 2011.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *POPL*, pages 97–108, Nice, France, Jan 2007.

[TL08] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th POPL*, pages 17–27, 2008.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. *ACM SIGPLAN Notices*, 44(1):264–276, 2009.

[Tuc08] Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, UNSW, Sydney, Australia, Aug 2008.

[TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *ACM SIGPLAN Notices*, volume 49, pages 691–707. ACM, 2014.

[vG99] Rob J van Glabbeek. The linear time–branching time spectrum I. *Handbook of process algebra*, pages 3–99, 1999.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. Emb. Comput. Syst.*, 7(3):1–53, 2008.

[WKP80] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.

[WKS+09] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In *TPHOLs*, pages 500–515, Munich, Germany, Aug 2009.

[YH10] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *2010 PLDI*, pages 99–110, Toronto, Ont, CA, Jun 2010.

[Zer08] Ernst Zermelo. Investigations in the foundations of set theory I. *Translated by S. Bauer-Mengelberg. In van Heijenoort (1967)*, pages 199–215, 1908.

[ZNMZ13] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. *ACM SIGPLAN Notices*, 48(6):175–186, 2013.

[ZPF+02] Lenore D. Zuck, Amir Pnueli, Yi Fang, Benjamin Goldberg, and Ying Hu. Translation and run-time validation of optimized code. Runtime Verification 2002 (RV'02). *Electr. Notes Theor. Comput. Sci.*, 70(4):179–200, 2002.

[ZPG03] Lenore D. Zuck, Amir Pnueli, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *J. UCS*, 9(3):223–247, 2003.