Real-Time Syst (2017) 53:812–853
DOI 10.1007/s11241-017-9286-3

High-assurance timing analysis for a high-assurance real-time operating system

Thomas Sewell[1] · Felix Kam[1] · Gernot Heiser[1]

Published online: 27 July 2017

Abstract Worst-case execution time (WCET) analysis of real-time code needs to be performed on the executable binary code for soundness. Obtaining tight WCET bounds requires determination of loop bounds and elimination of infeasible paths. The binary code, however, lacks information necessary to determine these bounds. This information is usually provided through manual intervention, or preserved in the binary by a specially modified compiler. We propose an alternative approach, using an existing translation-validation framework, to enable high-assurance, automatic determination of loop bounds and infeasible paths. We show that this approach automatically determines all loop bounds and many (possibly all) infeasible paths in the seL4 microkernel, as well as in standard WCET benchmarks which are in the language subset of our C parser. We also design and validate an improvement to the seL4 implementation, which permits a key part of the kernel's API to be available to users in a mixed-criticality setting.

Keywords Real time · Static analysis · Worst case · Timing · OS · seL4 · High assurance · Verified · WCET

✉ Thomas Sewell
Thomas.Sewell@data61.csiro.au

Felix Kam
Felix.Kam@data61.csiro.au

Gernot Heiser
Gernot.Heiser@data61.csiro.au

[1] Data61 and UNSW, Sydney, Australia

1 Introduction

Real-time systems are required to meet timing constraints in addition to their functional requirements. Critically important real-time systems must be assured to meet these timing and functional correctness requirements. Functional correctness is usually assured by traditional means such as testing, code inspection and controlled development processes (US National Institute of Standards 1999; RTCA 1992; ISO 2011), or more recently by formal methods RTCA (2011). The highest assurance is obtained by formal correctness proofs based on theorem proving, as was done with the seL4 microkernel (Klein et al. 2009) and several other systems (Bevier 1989; Leroy 2009; Alkassar et al. 2010; Yang and Hawblitzel 2010). Functional verification is generally performed on the *source-code* level (i.e. the C or other implementation language), which is then translated into a binary using a trustworthy compilation tool chain.

Timeliness requires, among other things, sound estimation of worst-case execution time (WCET). This is generally performed by static analysis of the *binary code*, in order to account for code changes by the compiler. The process typically first extracts 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.

However, this requires first determining safe upper bounds for all loop iterations. Furthermore, many candidate execution paths turn out infeasible (depending on branch conditions which are mutually exclusive) and must be eliminated to avoid an excessively pessimistic WCET. Frequently, loop bound determination and infeasible path elimination is done by manual inspection, but this is tedious, error-prone and difficult to validate, and thus unsuitable for safety-critical code.

For high assurance, we require an entirely automatic and trustworthy means of discovering loop bounds and path information in the binary. While there is a wealth of literature on using static analysis to derive loop bounds on binaries, getting complete coverage of all loops is impossible in theory (equivalent to the halting problem) and difficult to approximate in practice. An alternative approach is to instrument the compiler, and pass information across from the source side.

We propose a different approach, based on our existing work on *translation validation*. Translation validation (TV) is an approach to ensuring the compilation tool chain is trustworthy. Other approaches include extensive testing and even formal verification (Leroy 2009) of the compiler itself. In the TV approach, an unmodified optimising compiler is used, and a separate validation tool discovers evidence that the compiler has translated the source correctly (Sewell et al. 2013). The TV tool relates control flow at the binary and source level, which allows our WCET analysis to make use of source-level information missing in the binary. This source-level information includes pointer aliasing information by default. We can also manually intervene in the process by annotating the source code with certain special comments. These comments are ignored by the compiler, but are part of the formal model of the C program and may be used by the TV and WCET tools as additional assumptions.

Our target of interest is the seL4 microkernel, which has undergone comprehensive formal verification, including proof of security enforcement and functional correctness of the implementation (Klein et al. 2014). In the case of seL4, many useful properties have already been proved and are immediately available to the WCET analysis; any additional annotations create new proof obligations which must be discharged in the existing framework (and with the help of previously proved invariants). The result has the same high assurance as the formal correctness proof.

The proposed approach is not limited to functionally-verified code such as seL4. Any code that is in the subset understood by our C parser can be analysed. The parser's syntactical restrictions are that all *struct* declarations occur at the top level of source files, and the prohibition of side effects in almost all expressions. Assignments thus become statement-forms, and functions that return values may be called only as the right hand side of an assignment. Semantically, the C program cannot contain *unspecified* nor *undefined behaviours*, and the parser is parameterised by the architecturally dictated details of *implementation-defined behaviours*. This means that the source code needs to be re-verified for each architecture. These semantic assumptions, especially the absence of unspecified or undefined behaviour, can be verified using model checking. Obviously, manual annotations are of lesser assurance if not formally checked.

We apply our WCET analysis tool to the seL4 microkernel. Using our annotation mechanism, 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. For one of these operations we provide an alternative implementation, verify it functionally correct, and demonstrate that incorporating this change can allow more of the kernel API to be used with acceptable WCET.

We make the following contributions:

– high-assurance construction of the binary control-flow graph, with a proof of correctness of all but the final simplification (Sect. 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 (Sects. 4.2–4.4);
– 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 (Sect. 6.1), and similarly elimination of infeasible paths (Sect. 6.6);
– improvement of the WCET of the seL4 kernel by reimplementing one of its key operations (Sects. 5.2 and 5.3);
– demonstration that the approach is applicable to code that is not formally verified, by analysing a subset of the Mälardalen benchmarks (Sect. 6.5).

2 Background

This section summarises material on which we build directly. Section 3 summarises other related work from the literature.

2.1 Chronos

For WCET analysis we use the Chronos tool (Li et al. 2007), 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.

In earlier work (Blackham et al. 2012) we 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). We picked the Freescale i.MX 31 for our 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. We model the cache 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.

In our previous work (Blackham et al. 2012) we carefully validated this model against cycle timing on the real processor. We concluded that modeling the 4-way cache as 1-way was pessimistic but sound, never overestimating cycle times. We 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 our pessimistic calculations, 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 our previous one. 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.

2.2 The seL4 operating system kernel

seL4 is a general-purpose operating system (OS) microkernel, implemented mostly in C with a minimum of assembly code. In line with the tradition of high-performance L4 microkernels (Heiser and Elphinstone 2016), seL4 provides only a minimal set of mechanisms, including threads, a simple scheduler, interrupts, virtual memory, and inter-process communication, while almost all policy is implemented by user-mode processes. seL4 uses capability-based protection (Dennis and Van Horn 1966; Bromberger et al. 1992) and a resource-management model which gives (sufficiently privileged) user-mode managers control over the kernel's memory allocation—this is key to its strong spatial isolation.

The general-purpose design of seL4 means that the verified kernel can be adapted to support a broad class 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 (Barhorst et al. 2009). Examples include the integrated modular avionics architecture (Avionics Application Software Standard Interface 2012), and the integration of automotive control and convenience functionality with Infotainment (Hergenhan and Heiser 2008). These systems require strong spatial and temporal isolation between partitions, for which seL4 is designed.

The main attraction of seL4 is that it has been extensively formally verified, with formal, machine-checked proofs that the kernel application binary interface (ABI) enforces integrity (Sewell et al. 2011) and confidentiality (Murray et al. 2013), that the ABI is correctly implemented at the C level (Klein et al. 2009), and that the executable binary produced by the compiler and linker are a correct translation of the C code (Sewell et al. 2013). This make it arguably the world's highest-assured OS. Its WCET analysis (Blackham et al. 2011) is a step towards supporting mixed criticality systems, although more work remains to be done on its scheduling model (Lyons and Heiser 2014, 2016).

The kernel executes with interrupts disabled, for (average-case) performance reasons (Blackham et al. 2012), as well as to simplify its formal verification by limiting concurrency. To achieve reasonable WCET, preemption points are introduced at strategic points (Blackham et al. 2012). These need to be used 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 Sect. 2.6.

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 do not 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 outcome 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.

Our previous work focussed on aggressively optimising the kernel for latency (Blackham et al. 2011, 2012). Among other measures, we placed additional preemption points in long running operations. 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 to analyse is significantly larger than in our previous work (which used a non-verified kernel fork), where we had set the preemption limit to one.

2.3 The seL4 verification framework

The verification of the functional correctness of seL4 comprises over 200,000 lines of proof script, manually written and automatically checked by the theorem prover Isabelle/HOL (Nipkow et al. 2002). The proof contains four models of the behaviour of the kernel, as sketched in Fig. 1. The most abstract one (access control) is manually written in Isabelle, and the most detailed one (semantic C) is derived from the C source code of the implementation. There are three main proof components: a proof that a number of crucial invariants are maintained, and two proofs of refinement which establish that behaviours observed of the lower models must be subsets of those permitted by the higher models.

The C-level model is created by a C-to-Isabelle parser (Tuch et al. 2007). This produces a structured program in the Isabelle logic which roughly mirrors the syntax of the input C program. The parser adds a number of assertions which make explicit the correctness requirements of the C program, for instance involving pointer alignment and the absence of signed overflow. These constraints are roughly those that are prescribed by the C standard, with some additions for formal reasons, and some requirements of the standard relaxed to allow the kernel to implement its own memory allocator. Note that all these assumptions are proved correct for the seL4 source.

The translation validation process extends this verification stack, but uses automatic proofs in an SMT-based logic rather than manual proofs inside Isabelle/HOL.

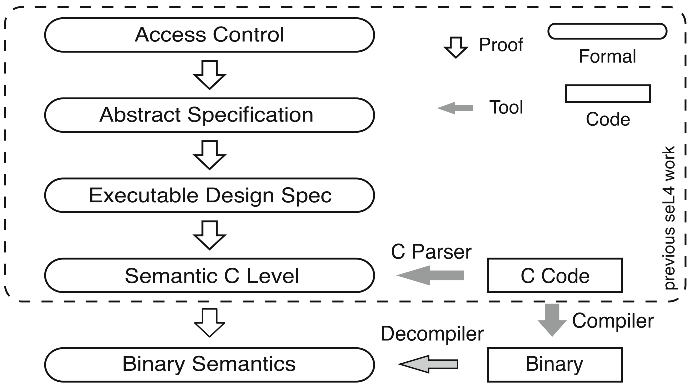

Fig. 1 The seL4 functional correctness stack

2.4 Decompilation of binary code

The decompiler of Fig. 1 is part of a collection of formal tools based on the Cambridge ARM ISA specification (Fox and Myreen 2010). The specification models the expected behaviour of various ARM processors in the theorem prover HOL4 (Slind and Norrish 2008). The key feature of these models is that they have been extensively validated by comparing their predictions to the behaviour of various real silicon implementations.

The decompiler builds on a tool which specifies what the effect of various instructions will be. This transformation also performs a HOL4 proof that the specification is implied by the CPU model. The decompiler stitches these instruction specifications together to produce a structured program which specifies the behaviour of a function in the binary. Crucially, the stitching process preserves the proofs. It results in a program specification, as well as a proof that the CPU would behave according to that specification, if it were to start executing the given binary at the given address.

In this project we use a variant of the decompiler which produces an output program in the graph-based language we describe below in Sect. 2.5. Each function in this program is structurally identical to the control-flow-graph of the relevant function in the binary, including sharing the same instruction addresses.

2.5 Translation validation

The proof of the correctness of the translation step from C to binary (Sewell et al. 2013)—the lowest level model of the seL4 functional verification—uses a *translation validation* toolset that builds on the decompiler introduced above. The proof process is sketched in Fig. 2. The starting point is the C program, parsed into Isabelle/HOL using the semantics of Tuch et al. (2007).

The overall TV approach is to transform both the C and the binary code into representations at the same abstraction level, i.e. a common intermediate language, and then prove correspondence function-by-function. The C program is transformed into

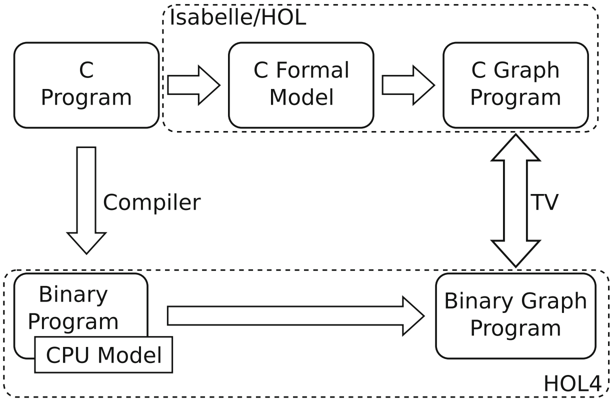

Fig. 2 Translation validation structure

a *graph language* with simpler types and control flow. The decompiler also transforms the binary into the same language. Both transformations construct proofs (in Isabelle/HOL and HOL4) that the semantics are preserved in the conversion.

Like machine code, statements in the graph language have explicit addresses and control flow may form an arbitrary graph. A program may manipulate an arbitrary collection of variables, with most programs having a “memory” variable in addition to variables representing registers or local variables. The graph language provides a mechanism for asserting a boolean property, which allows the correctness assertions (alignment etc.) made by the C-to-Isabelle parser and the decompiler to be expressed at this level. The C assertions, which have been proved in the previous verification work, become assumptions of the proof process, so the TV toolset may assume non-overflow conditions much like the compiler does. These assertions are also used by the decompiler to check key assumptions (such as alignment of various addresses). Assertions in the binary representation are proof obligations for the TV process.

The core of the TV process is a comparison of graph-language programs. For acyclic (loop-free) programs, this checks that the programs produce identical outputs (memory and return values/registers) given the same inputs (memory and argument values/registers). The calling convention specified by the ARM architecture defines the expected relationship between arguments and registers, etc. When loops are present, the tool must first search for an inductive argument which synchronises the loop executions, then check that the argument implies the same input/output relation. Both the check process and the search process use SMT solvers to do the heavy lifting. This process is described in detail elsewhere (Sewell et al. 2013).

2.6 The seL4 timing and preemption model

The majority of this paper is about 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 Sect. 2.7. 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.

2.7 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 (Sewell et al. 2011), 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. Real-time separation kernels (Rushby 1981), including Quest-V in separating mode (Li et al. 2013) and MASK (Martin et al. 2002), would enforce similar static restrictions. Virtualisation improves the usefulness of this configuration.
– A *closed* system, which is a use case that we evaluated in previous work (Blackham et al. 2011). 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.
– 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 Sect. 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.

2.8 Verifying preemptible seL4 operations

The abort style of preemption used in seL4 (see Sect. 2.6) 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 Sect. 5.2.

3 Related work

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. (2008), and we refer the reader to their summary for a more comprehensive overview.

Standard strategies for WCET analysis include hierarchical timing decomposition (Puschner and Koza 1989; Park and Shaw 1991), explicit path enumeration (Lundqvist and Stenström 1998; Healy et al. 1999), and implicit path enumeration (Li and Malik 1995; Burguière and Rochange 2006). We reuse the Chronos tool (Li et al. 2007) 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 (Wilhelm et al. 2008). In recent years, Rieder at al. have shown that it is straightforward to determine some loop counts at the C level though model checking (Rieder et al. 2008). Other authors use abstract interpretation, polytope modeling and symbolic summation to compute loop bounds on high-level source code (Lokuciejewski et al. 2009; Blanc et al. 2010). 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 (Cullmann and Martin 2007). The SWEET toolchain (Gustafsson et al. 2006) 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 (Knoop et al. 2011) 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 (Gustafsson et al. 2006; Ferdinand et al. 2001), with the same limitations as for loop-count determination. We earlier used binary-level model checking (Blackham and Heiser 2013) to automatically compute loop bounds and validate manually specified infeasible paths. We then used the CAMUS algorithm for automating infeasible path detection (Blackham et al. 2014). 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.

3.2 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 (Ermedahl et al. 2007; Kinder et al. 2009), 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. (2009), 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 Sect. 5.1). In our case the assertions are simple logical expressions, as used in Floyd or Hoare style program verification (Floyd 1967; Hoare 1969). 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 (2005). In their survey of WCET annotation styles, Kirner et al. (2011) 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 (Amadio et al. 2013; Ayache et al. 2012) 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 (Leroy 2009), 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 Verification

This paper discusses the design and verification of a simple preemptible algorithm (see Sect. 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 (Schlich 2010; Andronick et al. 2015) and the verification of fully concurrent algorithms is extremely challenging (de Roever et al. 2001; Feng et al. 2007; Cohen and Schirmer 2010; Turon et al. 2014; Gammie et al. 2015).

OS kernels are also an attractive target for formal verification, given their small size and critical importance. Starting with UCLA Secure UNIX (Walker et al. 2010) and the KIT OS (Bevier 1989), a number of OS-verification projects have been attempted, see Klein (2009) for detailed survey. In addition to seL4 (Klein et al. 2009), recent projects include the Verisoft project (Alkassar et al. 2010), the Verve kernel (Yang and Hawblitzel 2010) and the CertiKOS project (Gu et al. 2016). 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.

4 WCET analysis

The design of the WCET analysis process is shown in Fig. 3. We extend the TV framework 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 (Blackham et al. 2011), and generally reflects the “counterexample guided” approach to static analysis (Clarke et al. 2003; Henzinger et al. 2003). The approach has also previously been used by Knoop et al. (2013).

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

4.1 CFG conversion

In general, reconstructing a safe and precise binary CFG is difficult and error prone due to indirect branches (Bardin et al. 2011; Kinder et al. 2009). In previous work, we reconstructed the CFG from seL4’s binary using symbolic execution (Blackham et al. 2011). The soundness of the CFG so obtained, and thus the resulting WCET estimation, depended on the correctness of the symbolic execution 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*

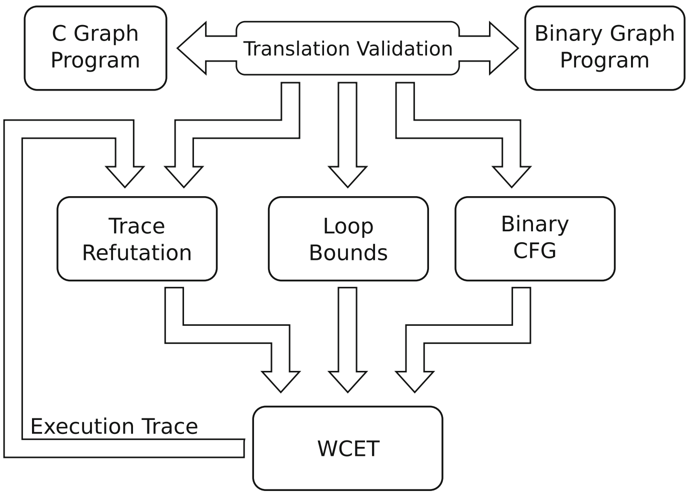

Fig. 3 Overview of dataflow in the analysis

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.

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 TV framework more complicated, and we leave this to future work. Instead we perform the simplification inside the TV framework 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.

4.2 Discovering and proving loop bounds

We employ two primary strategies for discovering loop bounds on the binary, both utilise features of the existing TV toolset. 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 TV toolset build 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 TV toolset generate an SMT model for loop 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. 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 induction done by the TV toolset for matching related loops in the source and binary.

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

```
e1a00004    mov    r0, r4
ebfffffe    bl     0 < f >
e2844004    add    r4, r4, #4
e3540c01    cmp    r4, #256    ; 0x100
1afffffa    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 a preexisting TV feature which discovers these linear series and sets up the inductive proofs.

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

[1] Here and later 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.

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 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 TV proof 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] The TV proof has already established this, 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

[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. The TV tool picks a particular concrete input relation for each binary function, and proves that this holds at all its call sites.

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 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 the TV framework, by asking it to relate the binary loop to some loop in the C program. If the TV toolset 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 the same language inside the TV framework, 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.

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. The TV toolset reasons 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 our previous work, where we consider much larger sets of path conditions and use the CAMUS algorithm to find all minimal conflicts (Blackham et al. 2014). 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.

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 our own (Blackham et al. 2014).

In Sect. 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 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 the TV toolset 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 (MISRA 2012), 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).

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. We have 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 (Blackham et al. 2012). Unfortunately some of these modifications increase the code 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 of January 2016. Secondly, we pick one of the preemption points added in our previous work (Blackham et al. 2012), 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.

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* (Liedtke 1994) 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.

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

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 (Shi et al. 2013).
   - 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 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.

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 we previously made to seL4 (Blackham et al. 2012) 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 Sect. 2.8). 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 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 previous work (Blackham et al. 2012) we adjusted this process 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 Fig. 4.

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

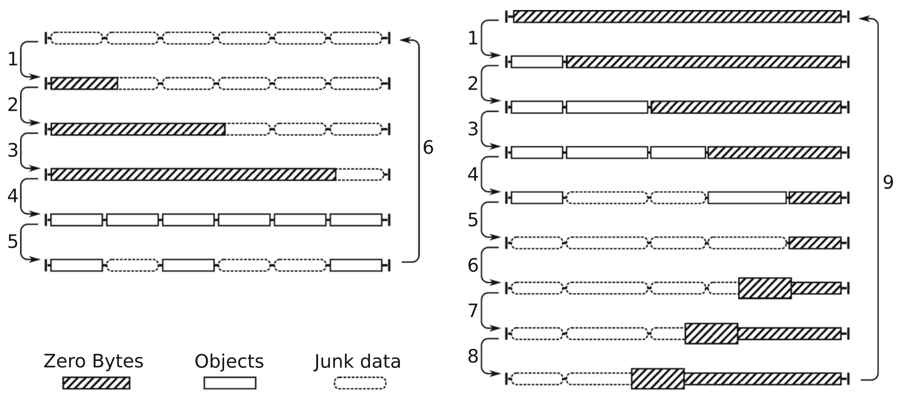

Fig. 4 Preemptible Retype designs from previous and current work. Previous steps (*left*): Starting state is junk data. *1–3*: Preemptibly zero the region. *4* Complete zeroing and create new objects. *5–6* Objects become unreachable over time. New steps (*right*): Starting state is zeroed region. *1–4* 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

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.

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 for a verification engineer with extensive 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.

6 Evaluation and discussion

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 our earlier work, which only succeeded on 18 of 32 loops[5] (56%) (Blackham and Heiser 2013). A further 5 loops in the binary contain preemption points

[5] Note that the total number of loops here is higher than in our earlier work. This results from this work targeting the verified kernel, and thus using preemption points less aggressively, see Sect. 2.2.

and have no relevant bound, these are bounded by the preemption limit, as discussed in Sect. 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 Sect. 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 Sect. 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 (Lyons and Heiser 2016), 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 reasonable 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.

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 Sect. 2.7). 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 many other functions into a single symbol that the resulting block runs for several thousand instructions and dominates the analysis time.

The success rates of the strategies discussed in Sect. 4.2 are listed in Table 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 Sect. 4.4. This bound is transferred across the TV 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 names of the binary functions in which the loops appear.

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.

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 2, with a small minority of the time measured spent preparing analysis problems in the TV framework 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.

[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.

Table 1 Loop bounds found by different strategies

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 2 Loop analysis time breakdown

	Strategies		Setup		Unaccounted		Total	
3.1.0-64K	4055 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%	1424 s	32.0%	4,444 s	(1:14:04)
Preempt-nodelete	1078 s	62.3%	106 s	6.1%	546 s	31.6%	1730 s	(0:28:50)
rt-branch-64K	12,014 s	58.5%	349 s	1.7%	8164 s	39.8%	20,527 s	(5:42:07)
rt-branch-static	9849 s	54.1%	248 s	1.4%	8101 s	44.5%	18,197 s	(5:03:17)

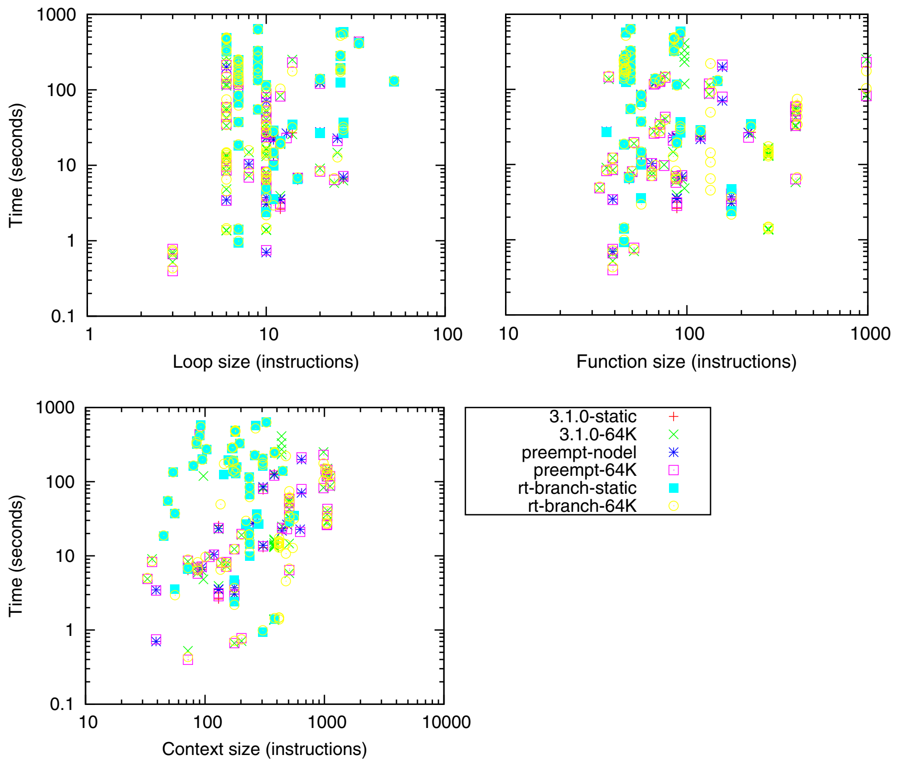

Fig. 5 Correlation of loop, function and context size to analysis time

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 Fig. 5 test these hypotheses. They compare loop analysis running time to the number of instructions in each loop, in 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 Fig. 6 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 TV transfer strategy is more

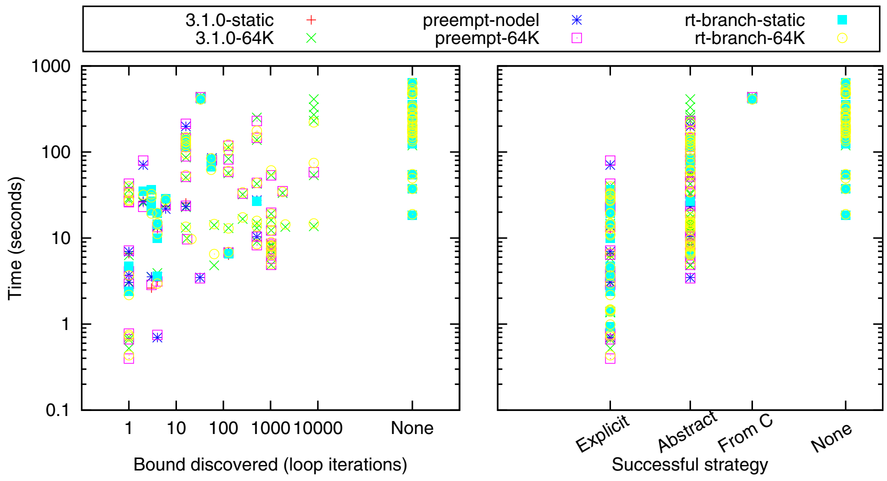

Fig. 6 Correlation of loop bound and successful strategy to analysis time

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 Fig. 6 as the relevant statistics contain timing for each of the subproblems instead. The most consistently expensive 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.

6.3 Binary-only analysis and comparison to previous work

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. 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-

Table 3 Loop bounds found without C-level information

	This work				Prior work	
	Full analysis		Binary-only		Blackham and Heiser (2013)	
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%

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*.

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.

6.5 Loop bounds in the Mälardalen suite

We use the Mälardalen WCET benchmark suite (Gustafsson et al. 2010) to further characterise the effectiveness of our approach. As in our previous such evaluation (Blackham and Heiser 2013), 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 Sect. 6.9).

The results are listed in Table 4. We must also omit a number of benchmarks which we attempted in our previous work. The current design depends on the C parser and TV toolset 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

Table 4 Mälardalen loop bounds

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

of which are in our verification C subset. We also skipped some tests which involve nested loops, which our TV toolset does not yet handle, or involve recursion[8]. The TV toolset also rejects 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 TV 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 the TV process.

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 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. (2012) have convincingly shown. We think this is a strong argument for the merits of pairing WCET and TV analysis with a source-level proof of safety [e.g. through static analysis, as required by MISRA-C (MISRA 2012)], as no safety-critical code should depend on invalid language constructs.

[8] The TV toolset handles some very simple cases of limited recursion.

Table 5 Infeasible path analysis statistics

seL4 version	3.1.0		Preemptible		RT	
Configuration	General	Static	General	Nodelete	General	Static
Initial est. (k cycles)	6894	1193	6888	1191	8256	781
Final est. (k cycles)	6349	532	6347	525	7397	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	708 s	488 s	642 s	443 s	2476 s	942 s
	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	5410 s	8002 s	5813 s	10,775 s	31,566 s	5588 s
	1:30:10	2:13:22	1:36:53	2:59:35	8:46:06	1:33:08

Note the "RT" statistics were impacted by hand workarounds

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 we evaluated in previous work (Blackham et al. 2011). 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, as we did in our previous work (Blackham et al. 2011), 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 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 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 Fig. 7, 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 (Kim et al. 2010; Pascal 2014). 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 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.

6.7 Performance

We studied the performance of the loop bound analysis in detail in Sect. 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

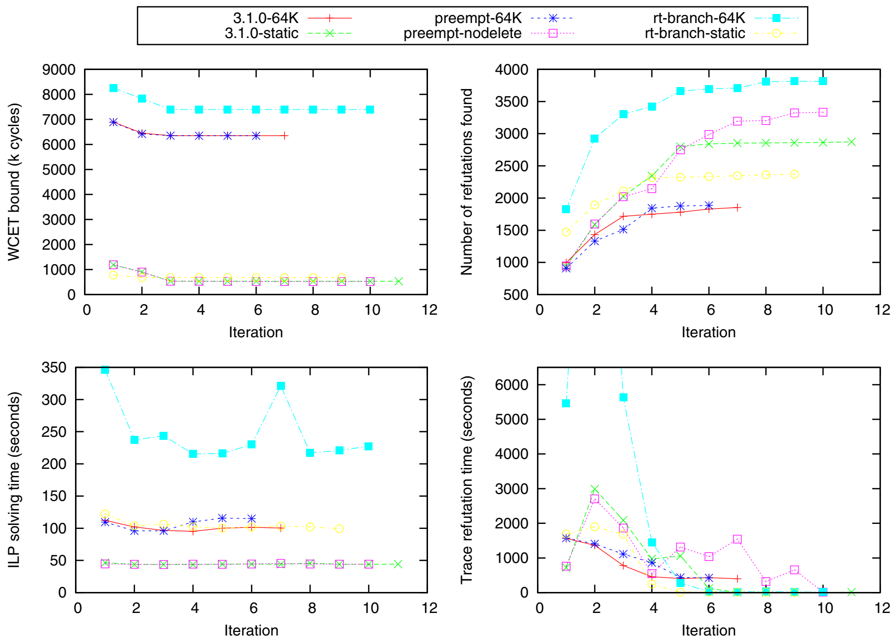

Fig. 7 WCET bound at each iteration, number of refutations discovered, ILP solving time, and trace analysis time. Note again that manual intervention in the "RT" branch prevents direct comparison

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 Fig. 7, 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 7 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 the 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 the TV apparatus.

6.8 API availability and future work

The final WCET estimates for the preemption modified kernel and seL4 3.1.0 were listed in Fig. 7. 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 Sect. 2.7, 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.

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). The TV framework does not discover loop relations for nested loops, though it handles loops with multiple 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 the TV framework extensively, we make relatively little use of the TV proofs themselves; we only use the loop relations for a few challenging loop bound problems. In principle, we could use the TV relation 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. As these are not C, our C-to-Isabelle parser cannot understand them. This creates a number of "blind spots" for the TV 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.

7 Conclusions

We propose a WCET analysis approach supported by the functional correctness apparatus used on the same program. In particular we build on a C source semantics used for manual verification and a translation validation framework used for checking the translation of the C source to the binary. 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 our present 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 have been 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 is a full-featured real-time operating system with complete verification and competitive WCET, this is a substantial step in that direction.

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

Availability

The present analysis extends, and is integrated with, the existing TV framework. The complete framework is available as open source (see https://ts.data61.csiro.au/software/TS/ and https://github.com/SEL4PROJ/graph-refine).

Acknowledgements This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorised to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S. Government.

References

Amadio RM, Ayache N, Bobot F, Boender JP, Campbell B, Garnier I, Madet A, McKinna J, Mulligan DP, Piccolo M, et al Certified complexity (CerCo) (2013) In: International workshop on foundational and practical aspects of resource analysis. Springer, Berlin, pp 1–18

Ayache N, Amadio R, Régis-Gianas Y (2012) 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

Andronick J, Lewis C, Morgan C (2015) Controlled owicki-gries concurrency: reasoning about the preemptible eChronos embedded operating system. In: van Glabbeek RJ, Groote JF, Höfner P (eds) Workshop on models for formal analysis of real systems (MARS 2015), Suva, Fiji, pp 10–24, Nov 2015

Alkassar E, Paul W, Starostin A, Tsyban A (2010) Pervasive verification of an OS microkernel: inline assembly, memory consumption, concurrent devices. In: O'Hearn P, Leavens GT, Rajamani S (eds) VSTTE 2010. LNCS, vol 6217, Edinburgh, UK. Springer, pp 71–85, Aug 2010

Avionics Application Software Standard Interface (2012) ARINC Standard 653

Barhorst J, Belote T, Binns P, Hoffman J, Paunicka J, Sarathy P, Scoredos J, Stanfill P, Stuart D, Urzi R (2009) A research agenda for mixed-criticality systems. http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/

Bevier WR (1989) Kit: a study in operating system verification. Trans Softw Eng 15(11):1382–1396

Bromberger AC, Peri Frantz A, Frantz WS, Hardy AC, Hardy N, Landau CR, Shapiro JS (1992) The KeyKOS nanokernel architecture. In: USENIX WS Microkernels & other Kernel Arch. Seattle, WA, US, pp 95–112

Blackham B, Heiser G (2013) Sequoll: a framework for model checking binaries. In: Tovar E (ed) RTAS, Philadelphia, USA, pp 97–106, Apr 2013

Blanc R, Henzinger TA, Hottelier T, Kovács L (2010) ABC: algebraic bound computation for loops. In: 16th international conference on logic programming, artificial intelligence & reasoning. Springer, pp 103–118

Bardin S, Herrmann P, Védrine F (2011) Refinement-based CFG reconstruction from unstructured programs. In: International conference on verification, model checking & abstract interpretation. Springer, Berlin, pp 54–69

Blackham B, Liffiton M, Heiser G (2014) Trickle: automated infeasible path detection using all minimal unsatisfiable subsets. In: West R (ed) RTAS, Berlin, Germany, pp 169–178, Apr 2014

Burguière C, Rochange C (2006) History-based schemes and implicit path enumeration. In: 6th WS worst-case execution-time analysis

Blackham B, Shi Y, Chattopadhyay S, Roychoudhury A, Heiser G (2011) Timing analysis of a protected operating system kernel. In: RTSS, Vienna, Austria, pp 339–348, Nov 2011

Blackham B, Shi Y, Heiser G (2012) Improving interrupt response time in a verifiable protected microkernel. In: EuroSys, Bern, Switzerland, pp 323–336, Apr 2012

Blackham B, Tang V, Heiser G (2012) To preempt or not to preempt, that is the question. In: APSys, ACM, Seoul, Korea, p 7, July 2012

Clarke E, Grumberg O, Jha S, Yuan L, Veith H (2003) Counterexample-guided abstraction refinement for symbolic model checking. J ACM 50(5):752–794

Cullmann C, Martin F (2007) Data-flow based detection of loop bounds. In: 7th WS worst-case execution-time analysis

Cohen E, Schirmer N (2010) From total store order to sequential consistency: a practical reduction theorem. In: Kaufmann M, Paulson L (eds) 1st ITP. LNCS, vol 6172. Springer, Edinburgh, UK, pp 403–418, July 2010

Dietz W, Li P, Regehr J, Adve V (2012) Understanding integer overflow in C/C++. In: Proceedings of the 34th international conference on software engineering, ICSE '12. IEEE Press, Piscataway, NJ, USA, pp 760–770

de Roever WP, de Boer F, Hanneman U, Hooman J, Lakhnech Y, Poel M, Zwiers J (2001) Concurrency verification: introduction to compositional and non-compositional methods. Cambridge Tracts in Theoretical Computer Science

Dennis JB, Van Horn EC (1966) Programming semantics for multiprogrammed computations. CACM 9:143–155

Ermedahl A, Sandberg C, Gustafsson J, Bygde S, Lisper B (2007) Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In: WS worst-case execution-time analysis

Feng X, Ferreira R, Shao Z (2007) On the relationship between concurrent separation logic and assume-guarantee reasoning. In: ESOP. Springer, pp 173–188

Ferdinand C, Heckmann R, Langenbach M, Martin F, Schmidt M, Theiling H, Thesing S, Wilhelm R (2001) Reliable and precise WCET determination for a real-life processor. In: EMSOFT. Springer, London, UK, pp 469–485

Floyd RW (1967) Assigning meanings to programs. Math Asp Comput Sci 19:19–32

Fox A, Myreen M (2010) A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann M, Paulson LC (eds) 1st ITP. LNCS, vol 6172. Springer, Edinburgh, UK, pp 243–258, July 2010

Gustafsson J, Betts A, Ermedahl A, Lisper B (2010) The Mälardalen WCET benchmarks—past, present and future. In: 10th WS worst-case execution-time analysis. OCG, Brussels, BE, pp 137–147, July 2010

Gustafsson J, Ermedahl A, Sandberg C, Lisper B (2006) Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: RTSS. IEEE Computer Society, Washington, DC, US, pp 57–66

Gammie P, Hosking T, Engelhardt K (2015) Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In: Blackburn S (ed) PLDI 2015: the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation. ACM, Portland, Oregon, United States, p 11, June 2015

Gu R, Shao Z, Chen H, Wu X, Kim J, Sjöberg V, Costanzo D (2016) An extensible architecture for building certified concurrent OS kernels. In: OSDI, CertiKOS

Healy CA, Arnold RD, Müller F, Whalley DB, Harmon Marion G (1999) Bounding pipeline and instruction cache performance. Trans Comput 48:63–70

Heiser G, Elphinstone K (2016) L4 microkernels: the lessons from 20 years of research and deployment. Trans Comput Syst 34(1):1:1–1:29

Hergenhan A, Heiser G (2008) Operating systems technology for converged ECUs. In: 6th embedded security in cars conference (escar), Hamburg, Germany, Nov 2008

Henzinger TA, Jhala R, Majumdar R (2003) Counterexample-guided control. In: 30th ICALP, Eindhoven, The Netherlands, pp 886–902, July 2003

Hoare CAR (1969) An axiomatic basis for computer programming. CACM 12:576–580

ISO (2011) ISO26262: road vehicles—functional safety

Klein G, Andronick J, Elphinstone K, Murray T, Sewell T, Kolanski R, Heiser G (2014) Comprehensive formal verification of an OS microkernel. Trans Comput Syst 32(1):2:1–2:70

Kim TH, Bang HJ, Cha SD (2010) A systematic representation of path constraints for implicit path enumeration technique. Softw Test Verif Reliab 20(1):39–61

Klein G, Elphinstone K, Heiser G, Andronick J et al (2009) seL4: formal verification of an OS kernel. In: SOSP, Big Sky, MT, US, pp 207–220, Oct 2009

Kirner R, Knoop J, Prantl A, Schordan M, Kadlec A (2011) Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. Softw Syst Model 10(3):411–437

Knoop J, Kovács L, Zwirchmayr J (2011) Symbolic loop bound computation for WCET analysis. In: International Andrei Ershov memorial conference

Knoop J, Kovács L, Zwirchmayr J (2013) WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In: Proceedings of the 21st international conference on real-time networks and systems, RTNS '13. ACM, New York, NY, USA, pp 161–170

Klein G (2009) Operating system verification—an overview. Sādhanā 34(1):27–69

Kinder J, Zuleger F, Veith H (2009) An abstract interpretation-based framework for control flow reconstruction from binaries. In: 10th International conference on verification, model checking & abstract interpretation. Springer, pp 214–228

Lokuciejewski P, Cordes D, Falk H, Marwedel P (2009) A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: 7th symposium code generation & optimization. IEEE Computer Society, Washington, DC, US, pp 136–146

Leroy X (2009) Formal verification of a realistic compiler. CACM 52(7):107–115

Lyons A, Heiser G (2014) Mixed-criticality support in a high-assurance, general-purpose microkernel. In: Davis R, Cucu-Grosjean L (eds) WS mixed criticality system, Rome, Italy, pp 9–14, Dec 2014

Lyons A, Heiser G (2016) It's time: OS mechanisms for enforcing asymmetric temporal integrity. arXiv preprint

Liedtke J (1994) Page table structures for fine-grain virtual memory. In: IEEE Technical Committee on Computer Architecture Newsletter, Oct 1994

Lisper B (2005) Ideas for annotation language (s). Technical report, Technical Report Oct. 25, Department of Computer Science and Engineering, University of Mälardalen

Li X, Liang Y, Mitra T, Roychoudhury A (2007) Chronos: a timing analyzer for embedded software. Sci Comput Program Spec Issue Exp Softw Toolkit 69(1–3):56–67

Li Y-T, Malik S (1995) Performance analysis of embedded software using implicit path enumeration. In: Proceedings of the 32nd ACM/IEEE design automation conference. ACM, pp 456–461, June 1995

Lundqvist T, Stenström P (1998) 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. Springer, Montreal CA, June 1998

Li Y, West R, Missimer ES (2013) The quest-V separation kernel for mixed criticality systems. In: WS mixed criticality system, pp 31–36, Dec 2013

MISRA (2012) Guidelines for the Use of the C language in critical systems

Murray T, Matichuk D, Brassil M, Gammie P, Bourke T, Seefried S, Lewis C, Gao X, Klein G (2013) seL4: from general purpose to a proof of information flow enforcement. In: S&P, San Francisco, CA, pp 415–429, May 2013

Martin WB, White PD, Taylor FS (2002) Creating high confidence in a separation kernel. Autom Softw Eng 9(3):263–284

Nipkow T, Paulson L, Wenzel M (2002) Isabelle/HOL—a proof assistant for higher-order logic. LNCS, vol 2283. Springer, Berlin

Puschner P, Koza C (1989) Calculating the maximum execution time of real-time programs. Real-Time Syst 1(2):159–176

Prantl A, Knoop J, Kirner R, Kadlec A, Schordan M (2009) From trusted annotations to verified knowledge. In: WS worst-case execution-time analysis, Dublin, IE, pp 35–45, June 2009

Park CY, Shaw AC (1991) Experiments with a program timing tool based on source–level timing schema. Trans Comput 24(5):48–57

Raymond P (2014) A general approach for expressing infeasibility in implicit path enumeration technique. In: Proceedings of the 14th international conference on embedded software. ACM, pp 8

Rieder B, Puschner P, Wenzel I (2008) Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement based WCET analysis. In: 2008 International Workshop on intelligent solutions in embedded systems, pp 1–7, Jul 2008

RTCA (1992) DO-178B: Software Considerations in Airborne Systems and Equipment Certification

RTCA (2011) *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*

Rushby J (1981) Design and verification of secure systems. In: SOSP, Pacific Grove, CA, USA, pp 12–21, Dec 1981

Shi Y, Blackham B, Heiser G (2013) Code optimizations using formally verified properties. In: OOPSLA, Indianapolis, USA, pp 427–442, Oct 2013

Schlich B (2010) Model checking of software for microcontrollers. ACM Trans Embed Comput Syst 9(4):36

Sewell T, Myreen M, Klein G (2013) Translation validation for a verified OS kernel. In: PLDI. ACM, Seattle, Washington, USA, pp 471–481, Jun 2013

Slind K, Norrish M (2008) A brief overview of HOL4. In: Mohamed OA, Muoz C, Tahar S (eds) TPHOLs. Springer, Montral, Canada, pp 28–32, Aug 2008

Sewell T, Winwood S, Gammie P, Murray T, Andronick J, Klein G (2011) seL4 enforces integrity. In: van Eekelen M, Geuvers H, Schmaltz J, Wiedijk F (eds) ITP. Springer, Nijmegen, The Netherlands, pp 325–340, Aug 2011

Tuch H, Klein G, Norrish M (2007) Types, bytes, and separation logic. In: Hofmann M, Felleisen M (eds) POPL. ACM, Nice, France, pp 97–108, Jan 2007

Turon A, Vafeiadis V, Dreyer D (2014) GPS: navigating weak memory with ghosts, protocols, and separation. In: ACM SIGPLAN notices, vol 49. ACM, pp 691–707

US National Institute of Standards (1999) Common criteria for IT security evaluation. ISO Standard 15408. http://csrc.nist.gov/cc/

Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra Tulika, Mueller Frank, Puaut I, Puschner P, Staschulat J, Stenström P (2008) The worst-case execution-time problem—overview of methods and survey of tools. Trans Embed Comput Syst 7(3):1–53

Walker BJ, Kemmerer RA, Popek GJ (1980) Specification and verification of the UCLA Unix security kernel. CACM 23(2):118–131

Yang J, Hawblitzel C (2010) Safe to the last instruction: automated verification of a type-safe operating system. In: 2010 PLDI. ACM, Toronto, Ont, CA, pp 99–110, Jun 2010

Thomas Sewell is a verification engineer at UNSW, where he has just completed his PhD. His research interests include formal methods, theorem proving, translation validation, high-assurance analysis and reasoning about concurrency.

Felix Kam is a software engineer and entrepreneur. He recently completed his bachelor's degree at UNSW, studying electrical engineering and law. His interests include smart contracts, game theory and operating systems.

Gernot Heiser is a Scientia Professor and the John Lions Chair for operating systems at UNSW. His research interests include operating systems, microkernels and building truly trustworthy embedded systems.