Complete, High-Assurance Determination of Loop Bounds and Infeasible Paths for WCET Analysis

Thomas Sewell, Felix Kam, Gernot Heiser
Data61 (formerly NICTA) and UNSW
Sydney, Australia
{thomas.sewell, felix.kam, gernot}@data61.csiro.au

***Abstract*—Worst-case execution time (WCET) analysis of real-time code needs to be performed on the executable binary code for soundness. Determination of loop bounds and elimination of infeasible paths, essential for obtaining tight bounds, frequently depends on program state that is difficult to extract from static analysis of the binary. Obtaining this information generally requires manual intervention, or compiler modifications to preserve more semantic information from the source program.**

We propose an alternative approach, which leverages 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.

I. Introduction

Critical real-time systems must be assured to both functional correctness as well as timely operation. Functional correctness is usually assured by traditional means such as testing, code inspection and controlled development processes [NIS99], [RTC92], [ISO11], or more recently by formal methods [RTC11]. The highest assurance is obtained by formal correctness proofs based on theorem proving, as was done with the seL4 microkernel [KEH+09]. Functional verification is generally performed on the *source-code* level (i.e. the C or other implementation language) and then translated into a binary using a trusted compilation tool chain or even a verified compiler [Ler09] guaranteed to produce correct code. Alternatively, we have recently shown that it is possible to prove the correctness of the translation without requiring a trusted compiler [SMK13].

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: leveraging the machinery we have developed for proving *functional correctness*, in order to enable a *high-assurance determination of loop bounds and infeasible paths* required for WCET estimation. Specifically, we reuse the *translation validation* (TV) apparatus we developed for proving that the compiler has correctly compiled the seL4 source code [SMK13]. It discovers a source-to-binary relation without requiring any compiler adjustments. We assume some safety and correctness conditions of the C code, which in the case of seL4 are implied by its formal verification, and in other cases result from safe programming practice.

Through the TV relation, our WCET analysis can make use of some source-level information missing in the binary, such as pointer aliasing information. We can also manually intervene in the process by annotating the source code with certain special comments. These are ignored by the compiler, but are picked up by the TV and WCET tools as additional assumptions. They are proved as additional obligations in the functional correctness proof.

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 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 framework's assumptions on the C subset, 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 make the following contributions:

- high-assurance construction of the binary control-flow graph, with a proof of correctness of all but the final simplification (Section III-A).
- WCET analysis supported by a translation-validation framework, allowing C-level information to be used in computing provable loop bounds and infeasible paths (Sections III-B–III-D);
- computation of all loop bounds needed for WCET of the seL4 kernel with one-off source-level assertions, but no manual inspection of the binary program (Section IV-A), and similarly elimination of infeasible paths (Section IV-C);
- demonstration that the approach is applicable to code that is not formally verified, by analysing a subset of the Mälardalen benchmarks (Section IV-B).

II. Background

A. Determining loop bounds and infeasible paths

Wilhelm et al. surveyed the wealth of WCET literature [WEE+08]. Since then, Rieder at al. have shown that it is straight-forward to determine some loop counts at the C level though model checking [RPW08]. Other authors use abstract interpretation, polytope modeling and symbolic summation to compute loop bounds on high-level source code [LCFM09], [BHHK10]. Attempting to automatically find a correspondence of source-level results in the compiled binary, in the presence of compiler optimisations, is difficult and makes the analysis dependent on compiler correctness, which is what we aim to avoid.

The aiT WCET analyser uses dataflow analysis to identify loop variables and loop bounds for simple affine loops in binary programs [CM07]. The SWEET toolchain [GESL06] uses abstract execution to compute loop bounds on binaries, and is aided by tight integration with the compiler toolchain, which improves the knowledge of memory aliasing, but this again implies relying on the compiler. The r-TuBound tool [KKZ11] uses pattern-based recurrence solving and program flow refinement to compute loop bounds, and also requires tight compiler integration.

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

B. Chronos

For WCET analysis we use the Chronos tool [LLMR07], which is based on the *implicit path enumeration technique* (IPET), to perform micro-architectural analysis and path analysis. The attraction of Chronos is its support for instruction and data caches, a flexible approach to modeling processor pipelines, and an open-source license. It transforms a simplified CFG, with loop-bound annotations, into an integer linear program (ILP). We solve this using an off-the-shelf ILP solver – IBM's ILOG CPLEX Optimizer – to produce the estimated WCET. Infeasible path annotations can generally be expressed as ILP constraints.

In earlier work [BSH12] 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, we target our analysis at the Freescale i.MX 31 for reasons enumerated before [BSH12], in particular its cache-pinning feature, which is unavailable in later ARM processors. The i.MX31 contains an ARM1136 CPU core clocked at 532 MHz, it has split L1 instruction and data caches, each 16 KiB in size and 4-way set-associative. Since the processor uses random cache-line replacement, we conservatively model the caches as direct-mapped caches of the size of one way (4 KiB). We also disable the L2 cache, since it significantly increases the L1 miss penalty and the WCET.

C. The seL4 operating-system kernel

seL4 is a general-purpose OS microkernel implemented mostly in C with a minimum of assembly code. In line with the tradition of high-performance L4 microkernels [EH13], seL4 only provides 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 [DVH66], [BFF+92] 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 cost, weight and volume, and improve software and certification re-use [BBB+09]. Examples include the integrated modular avionics architecture [ARI12], and the integration of automotive control and convenience functionality with Infotainment [HH08]. These systems require strong spatial and temporal isolation between partitions, for which seL4 is designed.

The 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 [SWG+11] and confidentiality [MMB+13], that the ABI is correctly implemented at the C level [KEH+09], and that the executable binary produced by the compiler and linker are a correct translation of the C code [SMK13]. This make it arguably the world's highest-assured OS. Its WCET analysis [BSC+11] is a step towards supporting mixed criticality systems, although more work remains to be done on its scheduling model [LH14].

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

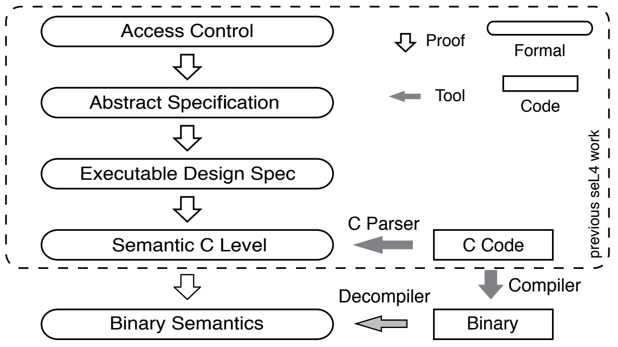

Fig. 1. The seL4 functional correctness stack.

[BSH12]. 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. Decreasing this limit improves the worst-case execution time of the system, at the expense of average case performance.

When a preemption point is triggered, seL4 checks for pending interrupts. If there are any, the current operation is exited. For verification reasons this is done by passing exceptional return values from all preemptible functions rather than by using a language exception mechanism like `exit` or `longjmp`. The interrupt is then handled, 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. Since the kernel execution ends on preemption, this makes the *worst-case response time* of seL4 equal to its *worst-case remaining execution time* from the point at which a pending interrupt is first raised.

This preemption mechanism is straightforward to encode as an ILP constraint. We can specify nonsense bounds (10^9) for all preemptible loops and instead restrict the overall number of visits to all preemption point sites to the configurable limit, i.e. we assume a pending interrupt will be detected.

Our previous work focussed on aggressively optimising the kernel for latency [BSC$^+$11], [BSH12]. 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*, and leave the optimisation of the WCET to future work. 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.

D. 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 [NPW02]. The proof contains four models of the behaviour of the kernel, as sketched in Figure 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 [TKN07]. 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.

E. Decompilation of binary code into logic

The decompiler of Figure 1 is part of a collection of formal tools based on the Cambridge ARM ISA specification [FM10]. The specification models the expected behaviour of various ARM processors in the theorem prover HOL4 [SN08]. 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 Section II-F. 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.

F. Translation validation

The proof of the correctness of the translation step from C to binary [SMK13] – the lowest level model of the seL4 functional verification – uses a *translation validation* (TV) toolset that builds on the decompiler introduced above. The proof process is sketched in Figure 2. The starting point is the C program, parsed into Isabelle/HOL using the semantics of Tuch et al. [TKN07].

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 a *graph language* with simpler types and control flow. The

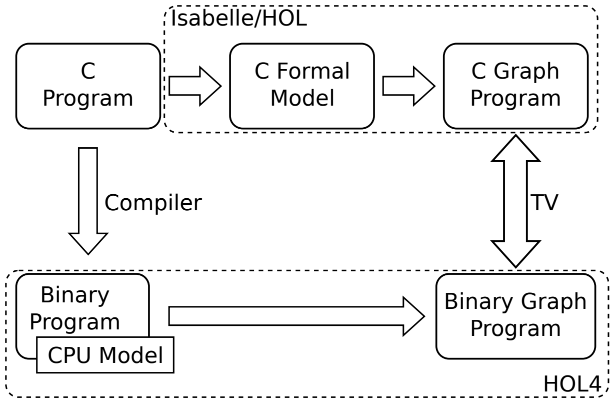

Fig. 2. Translation validation structure.

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. The assertions in the binary become proof obligations.

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 [SMK13].

III. WCET Analysis

The design of the WCET analysis process is shown in Figure 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 refutation approach is presented in detail by Knoop et al [KKZ13], who discuss it in detail.

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

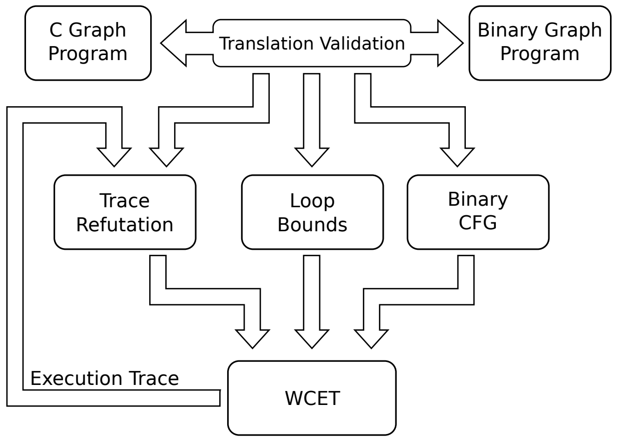

Fig. 3. Overview of dataflow in the analysis.

A. CFG conversion

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

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

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.

B. 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 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 onto 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 must be matched by a C trace in which f is called three times, with the register values r0, r1 matching the C values x, y at the respective calls. The TV proof has already established this, so the WCET analysis can consider this C execution trace simultaneously to the binary execution trace. Concretely this means that SMT problems will contain models of both binary f and the matching C f. The correctness conditions of the C f will be taken as assumptions. The x += 12 line in f above, for instance, tells us that adding 12 to either x or r0 must not cause a signed overflow.

The second extension, use of *call-stack information*, is useful in the case where the bound on a loop in a function is conditional on that function's arguments. Common examples include memset and memcpy, which take a size parameter, n, which determines how many bytes to loop over. To bound the

[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.

loop in `memset`, we must look at the values given to `n` at each of the call sites of `memset`. 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 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.

C. 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 [BLH14]. The trade-off is that, after eliminating refuted paths, we have to re-iterate the process on the next candidate ILP solution. We have not yet investigated which of these strategies is the most efficient and there is almost certainly room for significant optimisation.

D. 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 [BLH14].

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

For ordinary (application) programs, such as the Mälardalen benchmarks, we assume that the source conforms to the C standard, specifically that it is free of unspecified or undefined behaviour. This allows 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.

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 functional-verification approach. Here we reuse the annotation mechanism to explicitly assert information which we know will be of use to the loop-bound and infeasible-path modules. The assertions create proof-obligations in the functional-correctness chain, which we discharge by extending the hand-written Isabelle proofs about the kernel. The same mechanism can be used for application code, if the developer is certain about a certain assertion (eg. by having proved it through model checking).

With this manual (but safe) intervention, we can calculate and prove *all* loop bounds in the seL4 kernel binary, and effectively eliminate the most relevant infeasible paths. We add three kinds of assertions to achieve this.

1) We add an assertion that the "length" field of a temporary object is at maximum 16. In principle this information exists in the binary, and is unavailable to the WCET process only because we limit the extent to which we track information across function calls. In practice it was much simpler to assert this information, and do the equivalent propagation within the manual proofs.
2) We assert that each iteration of the capability-lookup process resolves at least one bit of the user's capability

descriptor. The kernel uses a *guarded page table* [Lie94] for storing capabilities, which allow the number of bits resolved to be user-configured. It is a proved kernel invariant that the user can never configure this value to zero, thus, the loop terminates. The assertion is trivial to prove from this invariant, and makes the invariant immediately available to the WCET apparatus.

3) We assert that certain capability cleanup operations cannot trigger any expensive object cleanups during the exchange of so-called reply capabilities. This is the same information that we have in previous work provided to the compiler to improve optimisation [SBH13].

We insert one further assertion to force a configurable limit on the size of objects in the present 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 inentionally fragmenting all large memory regions down to smaller chunks. Restricting the maximum object size might conceivably complicate the business of other resource managers, but has very little effect on application complexity or performance.

The clean way to resolve this issue would be to introduce preemption points into those long-running kernel operations, as we had done on a fork of the kernel in earlier work [BSH12]. However, this introduced significant complexity, which meant that the respective patches have to date not been included into the verified version of the kernel. In the present work we are targeting the *verified* kernel version, so we instead choose to enforce this slight restriction, and leave re-introducing further preemption points to future work.

This results in the following assertion:

4) We specify a maximum object size and assert that a number of creation, zeroing and cache-cleaning operations cover regions do not exceed this maximum size.

We presently set the maximum kernel-object size to 64 KiB. This object size limit is "ghost data", in the sense that it does not appear anywhere in the actual C program or in the binary, only in the Isabelle model of the C program. The kernel does not ensure that this limit ever comes into force, instead, it should be thought of as a key step in configuring a safe real-time system. However, we add a proof that this invariant, once true, will remain valid for the lifetime of the system.

Should the system 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. A possible exception is where the compiler's inlining changed (eg. by a new compiler version or different compilation options), which might move assertion information out of the necessary context. Clearly, the WCET analysis must be performed on the actually-deployed binary, it is obviously unsafe to assume that an analysis remains valid after re-compilation.

IV. Evaluation and Discussion

A. Loop Bounds in seL4

We successfully compute the bounds of all 67 bounded loops in seL4, which is in contrast to our earlier work, which only succeeded on 18 of 32 loops[3] (56%) [BH13]. A further 5 loops in the binary contain preemption points and have no relevant bound, these are bounded by the preemption limit, as discussed in Section II-C.

The seL4 version we study is the current mainline verified kernel, which includes our WCET assertions. We make one change to the kernel's standard build-time configuration, to adjust a configurable limit called the "fan-out limit" to the minimum. This avoids an issue involving a nested loop with a complex bounding condition. [4] We compile the kernel with `gcc-4.5.1` with optimisation setting `-O2`, which is the default for building the kernel.[5]

The success rates of the strategies discussed in Section III-B are listed in Table I. The explicit strategy discovers smaller bounds, the largest being 6, and the abstraction strategy finds all the higher bounds, which vary from 16 up to 8192. There is one outlier, a bound of 32 discovered by the explicit strategy on the C program. This is the capability lookup loop, manually annotated, which we discussed in Section III-D. This bound is transferred across the TV relation to bound the binary loop implicitly.

For comparison to previous work, we reran the analysis with all C-level information discarded, only using information available in the binary. The results are also in Table I. In this mode, some loop bounds are discovered by considering multiple possible call stacks individually.[6] This approach is available as a last resort in the analysis. We speculate the assertions we provide through the C code make this step redundant. In total we find 47 of 67 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%), probably because the abstract strategy can discover some large bounds more easily than our previous approach. The reason the binary-only strategies fail to find the remaining bounds are the same as in our earlier work: inability to perform a *memory aliasing analysis* on the binary and the lack of an *invariant maintained by a loop's environment*.

TABLE I. Loop bounds found by different strategies.

	This work				Prior work	
	Full analysis		Binary-only		[BH13]	
Explicit model	22	33%	13	19%	N/A	N/A
Abstraction	44	66%	28	42%	N/A	N/A
From C	1	1%	N/A	N/A	N/A	N/A
Call cases	0	0%	6	9%	N/A	N/A
Total found	67	100%	47	70%	18	56%
Not found	0	0%	20	30%	14	44%

[3]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 Section II-C.

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

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

[6]The different cases are solved by some combination of the two main strategies. Further breakdown is elided.

The advantages of source-level annotation became obvious when re-running the analysis after not touching it for about five months. During that time the kernel evolved, including a major maintenance patch which adjusts over 500 lines. Since the source level annotations were preserved, the automatic analysis immediately rediscovered all but one of the loop bounds in the kernel binary. The missing bound was because we had forgotten to adjust the kernel build parameters as mentioned above. This demonstrates the inherent safety of the present approach: the fully-automated analysis fails if changes to the code base invalidate previous assumptions.

B. Loop Bounds in the Mälardalen Suite

We use the Mälardalen WCET benchmark suite [GBEL10] to further characterise the effectiveness of our approach. As in our previous such evaluation [BH13], we compile the C sources for the ARMv6 architecture, with gcc (4.5.1) and the `-O2` optimisation setting, and omit benchmarks using floating point arithmetic. Floating point arithmetic is not presently supported by our C semantics nor the Cambridge processor model (see Section IV-D).

The results are listed in Table II. We must also omit a number of tests 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 of which are in our verification C subset. We also skipped some tests which use certain kinds of recursion or nested loops that our TV toolset does not yet handle. 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.

TABLE II. 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

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. [DLRA12] 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 model checking), as no safety-critical code should depend on invalid language constructs.

C. Eliminating infeasible paths

We evaluate infeasible path elimination on two use cases of seL4 as we did previously [BSC+11]: *open and closed systems*. In the open system scenario, all kernel operations are allowed. In the closed system, user tasks are never given capabilities that allow creation, deletion or recycling of kernel objects (such as address spaces or thread-control blocks) once the system is initialised.

The closed system forbids some expensive operations which otherwise dominate the WCET, which better exercises the trace refutation. In our previous work we made some address-space operations preemptible. In this work we do not make such kernel modifications, which unfortunately means we must forbid these operations in addition to those forbidden before. This means our current "closed" system is limited to static address space layouts. In future we will improve on this restriction.

We also forbid three particular operations for cancelling message sends which have no satisfactory WCET in the currently verified version of seL4. In our previous work we made these operations preemptible, however these changes adjust a number of function signatures and are still waiting to be verified. We hope to address this issue in the immediate future. For the time being we measure the open system as though these changes had already been included.

TABLE III. OPEN SYSTEM REFUTATIONS AND WCET

Iteration	# refutations	Estimated WCET (in thousands of cycles)	% diff against base case
Base Case	0	7,894	
1	57	7,411	-6.12%
2	82	7,307	-7.44%
...	...	...	...
9	120	7,306	-7.45%

In both cases, the automated process iteratively identifies the worst-case execution trace and eliminates paths within it, until no refutable paths are found. In both scenarios, a large number of infeasible paths are found. In the open-system case, they barely matter, as shown in Table III: this case is dominated by a `memzero` loop used to initialise a large object. Our previous fork of seL4 [BSC+11] made this step preemptible, and in future work we plan to merge more of these changes into the verified kernel and hopefully see a lower bound here.

The closed case has no such expensive operations. As Figure 4 shows, infeasible path elimination reduces the estimated WCET from 375 (thousand cycles) to 193, or 48.5% with 239 refutations. Each iteration refutes a significant number of paths, until the twelfth, which finds none, terminating the process. However, as Figure 4 shows, only the first two iterations have a significant effect on the WCET bound.

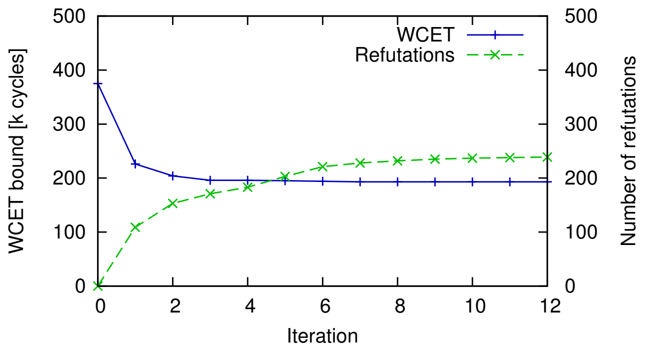

Fig. 4. Number of infeasible-path refutations of each iteration and resulting WCET bound (closed system).

D. 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 a 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, and handles only some irreducible 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 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 frequently 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.

E. Performance

The loop bound analysis can be run on all loops in the seL4 binary in 119 minutes 11 seconds, on an Intel i7-4790 based desktop machine running at 3.60GHz with 32 GiB RAM. Each iteration of the infeasible path search takes 2-3 hours for the first six iterations and less than 1 hour each afterwards, with a total time to run all the cycles of around 19 hours. This implementation is currently single threaded, and could be easily parallelised. There is certainly also room for other optimisation, especially in the refutation phase. Both analysis phases consider only a small number of functions at one time, however, so they should scale linearly to codebases with many more functions.

The seL4 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 eventually dominate the analysis. 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 TV apparatus.

V. 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 hand 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, 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 hand.

We have also shown 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.

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://ssrg.nicta.com.au/software/TS/).

Acknowledgements

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

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

[ARI12] *Avionics Application Software Standard Interface*, Nov 2012. ARINC Standard 653.

[BBB+09] James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russell Urzi. A research agenda for mixed-criticality systems. Available at http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/, Apr 2009.

[BFF+92] Alan C. Bromberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *USENIX WS Microkernels & other Kernel Arch.*, pages 95–112, Seattle, WA, US, Apr 1992.

[BH13] Bernard Blackham and Gernot Heiser. Sequoll: a framework for model checking binaries. In *RTAS*, pages 97–106, Philadelphia, USA, Apr 2013.

[BHHK10] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: algebraic bound computation for loops. In *16th Int. Conf. Logic for Progr., Artificial Intelligence & Reasoning*, pages 103–118, 2010.

[BHV11] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based CFG reconstruction from unstructured programs. In *Int. Conf. Verification, Model Checking & Abstract Interpretation*, pages 54–69, 2011.

[BLH14] Bernard Blackham, Mark Liffiton, and Gernot Heiser. Trickle: automated infeasible path detection using all minimal unsatisfiable subsets. In *RTAS*, pages 169–178, Berlin, Germany, Apr 2014.

[BSC+11] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *RTSS*, pages 339–348, Vienna, Austria, Nov 2011.

[BSH12] Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *EuroSys*, pages 323–336, Bern, Switzerland, Apr 2012.

[CM07] Christoph Cullmann and Florian Martin. Data-flow based detection of loop bounds. In *7th WS Worst-Case Execution-Time Analysis*, 2007.

[DLRA12] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 760–770, Piscataway, NJ, USA, 2012.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.

[EH13] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *SOSP*, pages 133–150, Farmington, PA, USA, Nov 2013.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, pages 469–485, London, UK, 2001.

[FM10] Anthony Fox and Magnus Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *1st ITP*, volume 6172 of *LNCS*, pages 243–258, Edinburgh, UK, Jul 2010.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In *10th WS Worst-Case Execution-Time Analysis*, pages 137–147, Brussels, BE, Jul 2010.

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, pages 57–66, Washington, DC, US, 2006.

[HH08] Andr Hergenhan and Gernot Heiser. Operating systems technology for converged ECUs. In *6th Emb. Security in Cars Conf. (escar)*, page 3 pages, Hamburg, Germany, Nov 2008.

[ISO11] ISO. *ISO26262: Road Vehicles – Functional Safety*, Nov 2011.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, et al. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, US, Oct 2009.

[KKZ11] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for WCET analysis. In *International Andrei Ershov Memorial Conference*, 2011.

[KKZ13] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Wcet squeezing: On-demand feasibility refinement for proven precise wcet-bounds. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 161–170, New York, NY, USA, 2013.

[KZV09] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *10th Int. Conf. Verification, Model Checking & Abstract Interpretation*, pages 214–228, 2009.

[LCFM09] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *7th Symp. Code Generation & Optimization*, pages 136–146, Washington, DC, US, 2009.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[LH14] Anna Lyons and Gernot Heiser. Mixed-criticality support in a high-assurance, general-purpose microkernel. In *WS Mixed Criticality Syst.*, pages 9–14, Rome, Italy, Dec 2014.

[Lie94] Jochen Liedtke. Page table structures for fine-grain virtual memory. *IEEE Technical Committee on Computer Architecture Newsletter*, Oct 1994.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming, Special issue on Experimental Software and Toolkit*, 69(1–3):56–67, Dec 2007.

[MMB+13] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *S&P*, pages 415–429, San Francisco, CA, May 2013.

[NIS99] US National Institute of Standards. *Common Criteria for IT Security Evaluation*, 1999. ISO Standard 15408. http://csrc.nist.gov/cc/.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. 2002.

[RPW08] Bernhard Rieder, Peter Puschner, and Ingomar Wenzel. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement based WCET analysis. In *Intelligent Solutions in Embedded Systems, 2008 International Workshop on*, pages 1–7, Jul 2008.

[RTC92] RTCA. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Dec 1992.

[RTC11] RTCA. *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*, Dec 2011.

[SBH13] Yao Shi, Bernard Blackham, and Gernot Heiser. Code optimizations using formally verified properties. In *OOPSLA*, pages 427–442, Indianapolis, USA, Oct 2013.

[SMK13] Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–481, Seattle, Washington, USA, Jun 2013.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *TPHOLs*, pages 28–32, Montral, Canada, Aug 2008.

[SWG+11] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *ITP*, pages 325–340, Nijmegen, The Netherlands, Aug 2011.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *POPL*, pages 97–108, Nice, France, Jan 2007.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. Emb. Comput. Syst.*, 7(3):1–53, 2008.