Sequoll: a Framework for Model Checking Binaries

Bernard Blackham and Gernot Heiser

NICTA and University of New South Wales
Sydney, Australia
Email: {bernard.blackham,gernot}@nicta.com.au

Abstract

Multi-criticality real-time systems require protected-mode operating systems with bounded interrupt latencies and guaranteed isolation between components. A tight WCET analysis of such systems requires trustworthy information about loop bounds and infeasible paths.

We propose sequoll, a framework for employing model checking of binary code to determine loop counts and infeasible paths, as well as validating manual infeasible path annotations which are often error-prone. We show that sequoll automatically determines many of the loop counts in the Mälardalen WCET benchmarks. We also show that sequoll computes loop bounds and validates several infeasible path annotations used to reduce the computed WCET bound of seL4, a high-assurance protected microkernel for multi-criticality systems.

Keywords

Real time systems; Operating system kernels; Software verification and validation;

1. Introduction

Multi-criticality real-time systems consolidate mission-critical with less critical functionality on a single processor, in order to reduce cost, weight and volume, and improve software re-use. Examples include the *integrated modular avionics* architecture [1], and the integration of automotive control and convenience functionality with Infotainment [2]. Such systems demand strong isolation between components, and thus an operating system (OS) which encapsulates applications in user-mode address spaces. In order to maintain real-time guarantees, such an OS must have bounded interrupt latencies under all circumstances.

We have previously proposed using the formally-verified seL4 microkernel [3] as a platform on which multi-criticality systems can be implemented, and consequently conducted a worst-case execution time (WCET) analysis of seL4 [4], [5].

A precise timing analysis of non-trivial programs running on modern processors is infeasible due to the need to consider the exponential number of states of both the program and the machine. A sound WCET analysis therefore requires that conservative over-approximations be made in order to reduce the number of states considered and make the problem tractable. However, the longer the program the more pessimistic the approximations tend to become.

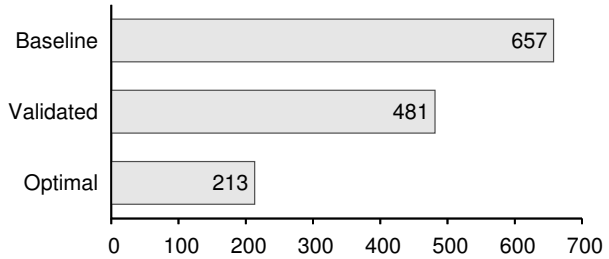

Fig. 1. Computed WCET (in thousands of cycles) of the seL4 microkernel on the ARM1136 CPU. The *baseline* figure uses no infeasible-path analysis, *optimal* is the best result achieved with manual annotations, while *validated* only eliminates paths confirmed infeasible by sequoll.

seL4 is a non-preemptible kernel, a design required to make formal verification feasible using current state-of-the-art verification tools [3]. This design is also preferred for better average-case performance. Therefore to bound interrupt response time in seL4, we must analyse much longer code paths than in the case of a fully-preemptible kernel, where only short sections of code run with interrupts disabled.

To mitigate the over-approximation in computing seL4's WCET, our previous analysis added manual annotations for excluding paths which were deemed infeasible. The improvements achieved by adding these annotations can be seen in Figure 1 – a 68 % reduction in the computed WCET. Additionally, we manually annotated the binary with bounds of the number of iterations of each loop. However, these manual annotations are tedious to construct and error-prone. Mistakes threaten the soundness of the analysis, and undermine the real-time guarantees desired from a high-assurance kernel.

We propose a framework called sequoll, which aims to fully automate this analysis and thus eliminate the need to trust hand-specified annotations. We show that sequoll can compute most of the loop bounds in seL4 automatically, detect some unreachable code paths (in the form of "0" loop bounds), and validate hand-specified infeasible path information.

Sequoll performs model checking on a control flow graph (CFG) derived from the binary. We restrict sequoll to analysing single-threaded code as our motivating application of non-preemptible kernels is inherently single threaded. Using a

symbolic model checker, we can validate properties of the code such as loop bounds, infeasible paths and more general invariants of functions. We utilise an existing high-fidelity specification of the ARM instruction set [6], which avoids the tedious and error-prone task of expressing semantics of instructions.

The main contribution of this paper is the use of symbolic model checking on binaries to automatically compute both simple and more complex loop bounds, as well as to verify infeasible path information without additional compiler assistance. We describe the approach in detail in Section 4. The second contribution is the demonstration that our approach is applicable to a real-world, highly-optimised yet non-preemptible kernel, where we show (Section 5.1) that sequoll determines the bound of the majority of loops, eliminates most manual interference in the WCET analysis, and improves the WCET estimate of seL4 by 27 % over the baseline (Figure 1). We also evaluate sequoll on the Mälardalen WCET benchmark suite [7], and show that it computes 64 % of the loop counts, without any source-level analysis or manual annotations (Section 5.2). We finally discuss the limitations which currently prevent the remaining loops and annotations from being analysed (Section 6).

2. Background

There is a large body of research related to the ideas behind sequoll – computing loop bounds, modelling instruction set architectures, reasoning about behavioural guarantees of systems, and reverse-engineering binaries to obtain control flow graphs. In this section, we will briefly highlight the state-of-the-art in each of these areas.

The specific problem of computing loop bounds on binaries has received much attention, particularly from WCET researchers for whom it is a fundamental hurdle. Manual annotation is a common but error-prone approach, where annotations are specified at the source code level, and compilers must ensure these annotations are carried through to the generated assembly [8]. Pattern matching on the binary can be used to search for common loop structures, but is fragile and compiler-specific. The aiT WCET analyser uses dataflow analysis to identify loop variables and loop bounds for simple affine loops in binary programs [9]. Abstract interpretation, polytope modeling and symbolic summation have also been used to compute loop bounds on high level source code [10], [11]. The SWEET toolchain for WCET computation 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 [12]. The r-TuBound tool uses pattern-based recurrence solving and program flow refinement to compute loop bounds, and also requires tight compiler integration [13].

Rieder et al. has shown that it is straight-forward to determine loop counts at the C source-code level through model checking [14]. However, attempting to automatically find a correspondence between source code and its compiled binary, in the presence of arbitrary compiler optimisations, is difficult and not fully solved by any single approach [15].

Using model checking on binaries is significantly harder than at the source level because there is less syntactic information available such as data types and structure layout, and there is limited information on what memory can potentially alias. Our work has similarities to Cassez's methods to compute WCET [16], but we do not attempt to compute the overall WCET using a model checker. Although seL4 is a small microkernel (∼8,000 LoC), using model checking for WCET computation does not (yet) scale to programs of this size.

Eliminating infeasible paths is also crucial for WCET analysis, as such paths may dramatically worsen the pessimism of WCET estimates. Several techniques have been proposed to detect infeasible paths, including abstract execution [12], conflict detection [17] and pattern matching [18]. We do not directly address the problem of detecting infeasible paths in this paper, but instead seek to validate manually-specified infeasible paths. This task is computationally less expensive in the general case, whereas many techniques for detecting infeasible paths do not scale for larger programs.

Model checkers can be used to perform static analysis of high-level languages, with a number of popular free and commercial tools available. For example, BLAST performs model checking on C sources using counter-example guided abstraction refinement (CEGAR) in order to check for desired safety properties [19].

Closely related to our work is that of Thakur et al. on the McVeto framework for directed proof generation [20]. They use model checking on an abstraction of arbitrary program binaries to determine if specific target instructions are reachable. Sequoll instead reasons about *paths* which may be infeasible in order to refine WCET analysis.

In addition to model checking, symbolic execution is becoming a popular option for exploring paths through a program. Symbolic execution groups together all inputs which may take the same path through a program. This technique is employed by TRACER [21] to analyse C code and also by S^2E [22], a selective symbolic execution platform. These techniques aim to analyse the properties and behaviour of a program or system under all possible input conditions.

Although symbolic execution would be a suitable alternative for our analysis, we still chose to write sequoll based upon model checking as we ultimately intend to integrate invariants based on formal proofs of the code. Model checking supports a more natural expression of such invariants, and has previously been used in conjunction with formal proof [23].

Conceptually, both loop bounds and infeasible paths could be proven within the verification of seL4. In practice, seL4's proofs heavily rely on a functional model of the code in which "paths" are ill-defined. Although proving loop bounds is possible, it is not immediately useful, as compiler optimisations can (and do) affect these in the binary.

To reconstruct control flow graphs from binaries, various solutions have been presented and implemented in binary static analysis frameworks such as Jakstab [24], BINCOA [25] and

```
int                          movs  r3, r0
popcount(uint32_t x)         moveq r0, r3
{                            bxeq  lr
  int c = 0;                 mov   r0, #0
  while (x != 0) {          loop:
    if (x & 1)               tst   r3, #1
      c++;                   addne r0, r0, #1
    x = x >> 1;              lsrs  r3, r3, #1
  }                          bne   loop
  return c;                  bx    lr
}
```

Fig. 2. A sample C function with no explicit loop variable, and the function compiled to ARM assembly.

MCVETO [20]. These frameworks implement an intermediate assembly language, for which translators are written for each architecture. To date, none exist for the ARM architecture.

3. The Problem

Sequoll was motivated by the desire to validate user-provided annotations about local code properties which were needed to perform a WCET analysis. Without independent (and, ideally, automatic) validation, such manual annotations significantly weaken the high degree of dependability we expect from the kernel.

Specifically, we seek to (a) automatically compute loop counts, and (b) verify claims of path infeasibility. We focus on binaries in order to avoid limiting compiler optimisations and to remain independent of any specific toolchain.

3.1. Loop bounds

The difficulty in determining loop counts in binaries depends heavily on the structure and invariants on the loops. Compilers may perform optimisations such as loop unrolling, rotation or reversal; loop variables may be saved to and restored from global memory; and loops may not have explicit or obvious loop counters. An example of this last issue can be seen in the code of Figure 2. This function counts the number of bits which are set in a word. It can be shown that the loop executes no more than 32 times, despite there being no explicit loop counter. We aim to compute loop bounds using only knowledge available from the binary.

3.2. Infeasible paths

Adding infeasible path information can improve the precision of WCET analyses considerably. By using knowledge that specific paths are infeasible, seL4's WCET estimate can be reduced by 68 %. In our previous analysis [5] we obtained this information by iteratively examining the worst-case path reported by our WCET toolchain manually. If we determined that it was infeasible, we constructed a constraint annotation to eliminate it. We repeated this until the worst-case path represented a valid execution.

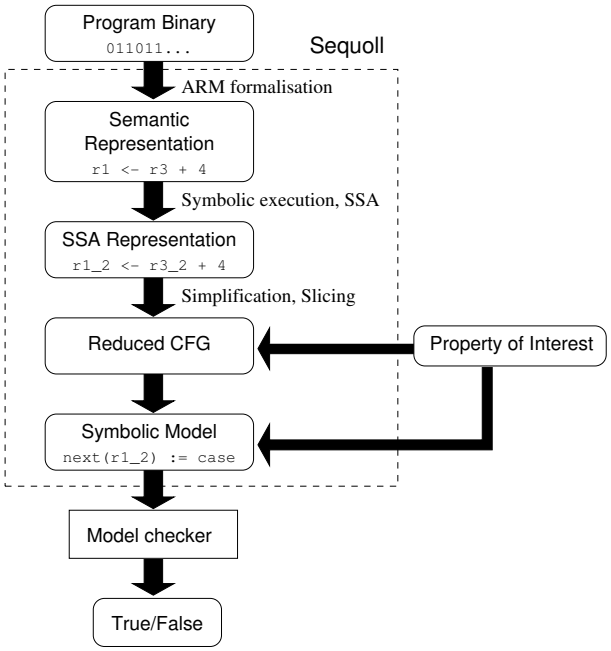

Fig. 3. An overview of the steps performed by sequoll.

The infeasible path constraints are used to augment the system of integer linear equations which is solved to find the WCET. As such, they take one of the following two forms:

- a **conflicts with** b **in** f: specifies that the instructions at addresses a and b are mutually exclusive, and will not both execute during an invocation of the function f. If f is invoked multiple times, a and b can each be executed under different invocations.
- a **is consistent with** b **in** f: specifies that the instructions at addresses a and b will execute the same number of times during an invocation of the function f.

The process of creating these annotations is error-prone, yet it is a trusted part of the WCET computation. Being able to validate these annotations will substantially increase the confidence in the WCET results, improving suitability for critical hard real-time applications.

4. Anatomy of Sequoll

The primary inputs to sequoll are a program binary and a property of interest such as a loop bound or an infeasible path constraint. From this we generate a model which is tested by a model checker. Figure 3 gives an overview of the key steps required to produce this model. In this section, we cover the techniques we use in sequoll to generate a suitable model from the binary.

Our motivation stems from WCET analysis of the seL4 microkernel, whose code has some restrictions that make verification tractable. We carry some of these restrictions to

Machine code:	E2813002
Disassembly:	add r3, r1, #2
Semantics:	r3 ← r1 + 2
	r15 ← r15 + 4

Machine code:	E8AD0028
Disassembly:	stmia r13!, {r3, r5}
Semantics:	mem r13 ← r3<7:0>
	mem (r13 + 1) ← r3<15:8>
	mem (r13 + 2) ← r3<23:16>
	mem (r13 + 3) ← r3<31:24>
	mem (r13 + 4) ← r5<7:0>
	mem (r13 + 5) ← r5<15:8>
	mem (r13 + 6) ← r5<23:16>
	mem (r13 + 7) ← r5<31:24>
	r13 ← r13 + 8
	r15 ← r15 + 4

Fig. 4. Example instruction semantics from the formalisation of the ARM ISA. Note that ARM maps the program counter onto general-purpose register `r15`.

sequoll, because they simplify our implementation. In particular, we assume that program binaries:

- do not contain recursive functions;
- do not contain self-modifying code;
- do not make use of function pointers; and
- are not affected by interrupts, signals, or other asynchronous control flow.

These hold for a large class of programs, particularly critical code running on top of the real-time OS. For example, the OS shields applications from the visible effects of interrupts (other than the rate of progress).

We have implemented sequoll in around 10,000 lines of code, excluding external tools such as the model checker.

4.1. Decoding instruction semantics

A prerequisite to performing static analysis on binary code is a representation of the semantics of instructions. Producing this is typically a complex, error-prone task and requires careful validation, as any inconsistencies can impact the soundness of our analysis.

We mitigate these issues and reduce engineering effort by reusing Fox & Myreen's formalisation of the ARM instruction set written in the HOL4 theorem prover [6]. This formalisation has been extensively tested against hardware, and is used in a number of other projects [26].

For a given instruction and system state, it can generate a precise set of semantics. We handle conditional instructions by adding any conditions to a set of preconditions under which the instruction may execute. We can obtain a simple representation even for quite complex instructions, as the example in Figure 4 shows.

Due to the use of this formalisation, we currently only support binaries for the ARM architecture. However, specifications of other architectures could be substituted, as all other concepts used in sequoll are architecture-independent.

4.2. Control flow graph reconstruction

Extracting the control flow graph of a program is a difficult task in the general case. However, the restrictions listed above considerably simplify the task.

Given the entry point to the program, sequoll explores all reachable instructions. Although we preclude the use of function pointers at the source level, the binary may still contain indirect branches (i.e. those via a register) which require extra work to resolve. Computing the possible destinations of function pointers is a much more challenging task as it requires reasoning globally across the entire binary, whereas the indirect branches generated by compilers can almost always be resolved locally.

We use a simple symbolic execution engine to determine the destination of indirect branches. It performs value analysis and tracks loads and stores into memory. This allows us to resolve:

- function returns – these typically involve storing the function's return address into stack memory and later reading it back either directly into the program counter, or via another register;
- indirect branches via literal loads – i.e. where the destination address is stored as data, interspersed within the instruction stream in the binary;
- switch statements – these presently require some compiler-specific pattern matching to decode, as in the general case this is also a difficult problem.

We expand function calls in the control flow graph through virtual inlining. This is necessary in order to compute loop bounds that are dependent on function arguments or calling context (e.g. in `memcpy`).

Although this analysis normally works for compiled programs, more sophisticated compiler optimisations or handcrafted assembler code can conceivably result in binaries where our simple symbolic execution engine fails. More comprehensive analyses exist and these could be implemented and substituted within sequoll if required [20], [27], [28].

Figure 5 shows the control-flow graph for our simple example from Figure 2.

4.3. Loop identification

Given the control flow graph of a program, we can identify loops and classify them as *reducible* or *irreducible*. A reducible loop has a single entry point from outside the loop body, whereas irreducible loops may have multiple entry points [29]. Irreducible loops are problematic for any WCET analysis that relies on specifying loop bounds relative to a unique entry point. They also lead to ambiguity in program structure, such as the relationship between nested loops [30].

We currently restrict sequoll to analysing reducible loops and failing when an irreducible loop is encountered. If required, irreducible loops could be handled by duplicating them

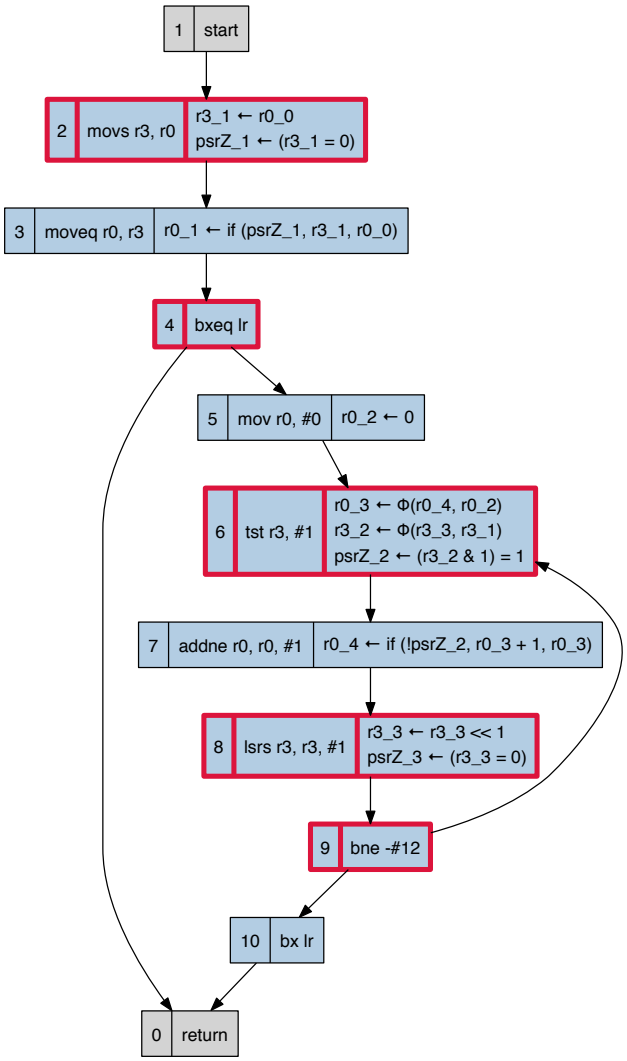

Fig. 5. The control flow graph of the assembly code in Figure 2, and its SSA representation. The nodes outlined in red are those in the computed slice to deduce the upper bound of the iteration count of the loop at node 6.

in the CFG – once for each entry edge. We have not done this in sequoll as it adds unnecessary complexity for our use cases.

Given a node E as a possible candidate for a loop entry point, we define a loop as the largest strongly connected component (SCC) that includes E, such that E dominates all other nodes within the loop. If no such SCC exists, then E does not induce a loop. This gives a one-to-one relationship between entry points and loops (under the reducibility criterion).

Sequoll finds reducible loops via a depth-first search, identifying for every instruction its inner-most loop, and reconstructing the loop nests of the program.

4.4. SSA transformation

We convert the program to single static assignment (SSA) form, as this simplifies later stages of our analysis. A program

$$\begin{array}{ll} sp_1 \leftarrow sp_0 + 8 & sp_1 \leftarrow sp_0 + 8 \\ sp_2 \leftarrow sp_1 + 16 & sp_2 \leftarrow sp_0 + 24 \\ sp_3 \leftarrow sp_1 + 16 \quad \Longrightarrow & sp_3 \leftarrow sp_0 + 24 \\ sp_4 \leftarrow \phi(sp_2, sp_3) & sp_4 \leftarrow sp_0 + 24 \\ sp_5 \leftarrow sp_4 - 24 & sp_5 \leftarrow sp_0 \end{array}$$

Fig. 6. Using constant propagation, sequoll can convert all stack pointer references to precise offsets relative to an initial stack pointer. This enables the stack to be treated independently from memory in the analysis of seL4.

in SSA form has the property that each variable is assigned to at most once. Using SSA makes it much simpler to track the dependencies between variables. Where program paths merge with potentially different values for a variable, a special function known as a ϕ function is used to represent the choice of values based on path.

The primary advantage of using SSA representation is that it can greatly reduce the number of states required for model-checking; we describe this further in Section 4.7. We use standard techniques to convert the program's representation into SSA form [31].

Figure 5 shows the SSA representation of our example from Figure 2 (with the irrelevant portions elided). Multiple edges reach node 6 in this diagram, and hence ϕ functions define the values of `r0` and `r3`.

4.5. Simplification

Once transformed to SSA form, there are many opportunities to simplify expressions within the analysis. For example, a program increments and decrements the stack pointer as it pushes or pops data on the stack. As Figure 6 shows, constant propagation often allows us to condense these chains of arithmetic to a simple expression, specifying the offset against the original stack pointer.

The ϕ functions generated by SSA transformation can also be simplified if all possible values are equivalent. This is particularly pertinent for the stack pointer, which generates a significant number of ϕ functions as its value is frequently modified. However, the relative offset of the stack pointer is typically consistent for any program point, regardless of execution history or any other program state.

If all pointer accesses were to known addresses, we could convert all accessed memory locations into simple variables, allowing us to track and simplify them further. Unfortunately, memory addresses cannot always be determined, and they frequently depend on function inputs. This impacts even on those memory addresses which we can determine, due to the possibility of pointer *aliasing*: a write to an unknown address may affect a later read from a known address.

However, there are cases where we may know that memory accesses do not alias. For example, C/C++ compilers provide the *restrict* keyword, which lets programmers hint that a memory region has no aliases. Similarly, the formal verification of seL4 guarantees that the C code will never take a pointer to a

local variable [3]. This ensures that stack memory will never alias with any other pointers.

Using this knowledge about seL4, we can treat each byte of stack memory as a local variable, thereby eliminating all accesses to stack memory from our model. This allows sequoll to track and analyse parameters passed via the stack. It also eliminates the pushing and popping of callee-saved registers, frequently emitted in function prologues and epilogues, from later analysis.

We also resolve accesses to read-only memory, such as constants loaded using PC-relative addressing.

4.6. Program slicing

Often, the property we wish to check involves a small portion of the entire program. We can speed up the analysis significantly by computing a smaller, equivalent program which preserves the property of interest, precisely what the technique of program slicing achieves [32]. Given a property of interest, known as the *slice criterion*, we recursively follow all data-flow and control-flow dependencies to compute an equivalent program with respect to this property.

We select the slice criteria based on the variables or control flow nodes relevant to our desired property. For instance, to compute loop bounds, we count the maximum number of times that the head of a loop may execute. Here the slice criterion simply consists of the loop head node (containing the first instruction in the loop). The slicing algorithm will begin by finding the control-flow dependencies for the loop head. This will include any conditional statements outside the loop which may prevent its execution, as well as any nodes inside the loop which may conditionally exit it.

It is possible that we over-approximate the slice, however it is guaranteed to be equivalent with regard to the slice criteria. Slices that are too large can make model checking prohibitively expensive. The simplification step described in Section 4.5 helps to minimise the size of the slice.

Some loops are bounded only because of earlier conditions on the path leading to them. Consider the code below – the loop is only bounded because it does not execute if the preceding condition fails. The slicing algorithm will identify the relevant parts of the execution history and preserve those in the slice.

```
if (c < 100) {
    for (i = 0; i < c; i++) {
        ...
    }
}
```

The trail of dependencies leading up to the head of the loop can be quite long, with much of it irrelevant for a tight bound. Consider a loop with two possible exit conditions – one exits the loop when an iteration count variable exceeds a hard-coded upper bound, while a second tests the iteration count variable against a complex expression computed before the loop. If we only care about the hard-coded upper bound then we can prune the slice to ignore the computation of the complex expression, significantly speeding up the analysis.

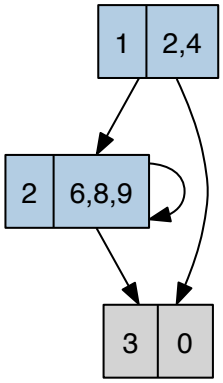

Fig. 7. The reduced control flow graph, equivalent to the slice shown in Figure 5. Each node also list the nodes of Figure 5 which it represents.

For this purpose, we allow the user to pass in a parameter specifying a *region of interest* – this is a subset of nodes of the control-flow graph that are considered by the slice, excluding statements outside the region. We convert variables that are modified outside the region of interest, but used inside, into non-deterministic inputs.

In order to preserve soundness, the chosen region of interest must maintain an important property: the region must have a unique entry node and this node must be a dominator of all other nodes in the region. If this property is not met, it may be possible to execute the region of interest without executing the entry node, invalidating any results established by the model checker.

We construct a *reduced* control flow graph representing only the nodes in the program slice, and collapse all consecutive nodes into one node where possible (i.e. any sequence of nodes with no branches originating from or destined for any nodes in between).

For loops with a fixed iteration count, the reduced control flow graph can have as few as three nodes. Figure 7 shows the reduced control flow graph of our example code from Figure 2; in this case, 11 nodes were reduced to three. More complex loop structures will obviously introduce more nodes, but our evaluation (Table 1) shows that in most cases the graph remains very small.

4.7. Translation to a symbolic model

The final step in processing the binary is to convert the reduced graph of our program slice into a symbolic model, which can be checked by a model checker. We use the NuSMV model checker which supports assertions expressed in temporal logic such as linear temporal logic (LTL) or computational tree logic (CTL). These logics are able to express useful properties such as "the value of some variable is always less than k" or "state A is not reachable until state

B has occurred".

We convert variables in our SSA representation into one of three types of expressions in the model:

- *stateful variables*: these variables are updated by transitions in the model and contribute to the state space which the model checker must explore;
- *frozen variables*: these variables retain their value throughout the program, and also contribute to the state space which must be explored; and
- *definitions*: these incur no state of themselves, and act merely as syntactic sugar for building larger models.

We translate the variables of the reduced control flow graph into a symbolic model as follows:

- Variables that are used but not defined anywhere in the SSA representation are inputs to the program. These are converted to frozen variables.
- Both ϕ functions and memory reads are converted to stateful variables in the model.
- All other SSA variables become definitions, as they themselves do not need to incur any state in the model.

The key observation here is that there are only two cases in which the program's execution may diverge, requiring the model checker to explore multiple states. The first is at memory reads when the result is unknown, either due to potential aliasing or otherwise being unable to identify a corresponding memory store.

The second case is when multiple paths can be taken through the control flow graph. Perhaps somewhat unintuitively, we only need to consider points where program paths *converge* rather than diverge. Given a node with multiple incoming edges, the choice of incoming edge implicitly defines which outgoing edges were taken earlier in the execution. This fits naturally with the SSA representation of ϕ functions which assign variables only at converging nodes.

For the example in Figure 5, we only need to create a single stateful variable for the expression $\texttt{r3\_2} \leftarrow \phi(\texttt{r3\_3}, \texttt{r3\_1})$. This is defined in the model checker as follows:

```
next(r3_2) := case
    n=1: r3_1;
    n=2: r3_3;
    TRUE: r3_2;
esac;
```

This intuitive syntax states that when visiting node 1 (in the reduced control flow graph of Figure 7), `r3_1` should be assigned to `r3_2`; similarly at node 2, `r3_3` should be assigned. The variable retains its value at all other nodes.

The input variable `r0_0` is specified as a frozen variable. All other information needed for the program slice is specified as definitions which incur no additional state.

```
psrZ_3 := r3_3 = 0;
psrZ_1 := r3_1 = 0;
r3_3 := r3_2 >> 1;
r3_1 := r0_0;
```

In addition to the SSA variables, we represent the reduced control flow graph of the program slice as an additional stateful variable. The transitions between nodes are represented in the model by conditional assignments to this stateful variable. For example, the reduced control flow graph in Figure 7 becomes:

```
init(n) := 1;
next(n) := case
    n=1 & !psrZ_1: 2;
    n=1 &  psrZ_1: 3;
    n=2 & !psrZ_3: 2;
    n=2 &  psrZ_3: 3;
    n=3: 3;
esac;
```

4.8. Loop bound checking

To solve the example loop bound from Figure 2, we assign a stateful variable C in the model checker which counts the number of times the loop entry node executes. The variable is reset to zero at nodes outside the loop that are immediate predecessors of the loop entry node.

Although the model checker cannot directly compute the loop bound for us, we can ask the model checker a statement of the form "is $C \leq k$?". We can then perform a binary search to find the smallest value of k for which the statement holds true. This value is the maximum number of times that the loop may iterate.

4.9. Path feasibility testing

As described in Section 3.2, we express the path infeasibility constraints in terms of pairs of instructions which either conflict with each other (are mutually exclusive), or are consistent with each other (both execute the same number of times).

These constraints translate naturally to expressions suitable for model checking, as follows. For conflict constraints, we create boolean flags a and b for the two instructions of interest. These flags are set to true when the respective node is visited. We can then express the infeasibility condition as the assertion that a and b are never simultaneously true. For consistency constraints, we use execution counters instead of flags, and assert that all paths finish with both counters equal.

5. Evaluation

5.1. seL4

Our primary motivation for developing sequoll was to validate annotations on our timing analysis of the seL4 microkernel, hence this is the obvious test case.

We use sequoll to automatically compute the loop bounds on all loops within the seL4 binary, which contains 32 loops in 11 functions. Regions of interest were specified on many of these loops – in particular, loops where there were exit conditions with long chains of data dependencies that have no effect on the worst-case iteration count. Sequoll succeeds in computing precise bounds on 18 loops (56 %). The analysis on the remaining loops presently fails for one of several reasons, which we discuss in detail in Section 6:

- one loop is only bounded thanks to an *invariant maintained by its environment*;
- one loop sequoll cannot analyse due to *complex exit conditions*;
- on 12 loops, all of identical structure, sequoll fails to determine a bound due to *poor memory aliasing analysis*;

Of the 18 loops, 13 loop bounds are computed within 10 seconds each, three further loops within one minute, and two more complex bounds in 28 minutes – 99% of which is spent performing SSA transformation and simplification. This step is slow primarily because of the expensive live variable analysis on the inlined control flow graph, which consists of over one million nodes. The analysis of seL4 used at most 6 GiB of memory.

Our best (manual) analysis eliminated 35 infeasible paths by manually adding appropriate constraint annotations, reducing the computed WCET bound from 657,000 cycles to 213,000 cycles, a 67 % improvement. Of these, sequoll validated 4, which was sufficient to reduce the bound to 481,000 cycles (a 26 % improvement), see Figure 1.

One of our manual constraints turned out to be wrong, as we found through sequoll! Fortunately, this somewhat embarrassing fact had no effect on the WCET estimate, but serves as a clear warning about the fickle nature of annotations derived from path inspections.

Of the remaining 30 constraints,

- 11 relate to infeasible paths which depend on values read from memory that can *potentially alias*. The loss of information makes it impossible to determine if these paths are truly infeasible.
- 19 depend on *invariants* which are not possible to ascertain from the binary.

5.2. WCET Benchmarks

We use the Mälardalen WCET benchmark suite [7] to further evaluate sequoll's ability to deduce loop bounds. We compile the C sources for the ARMv6 architecture, using gcc 4.4.1 and the `-O2` optimisation level. We omit benchmarks using floating-point arithmetic, as the ARM formalisation [6] does not presently support the instructions used for hardware floating point. We also omit the RECURSION benchmark, as our analysis does not presently support recursive functions. Finally, we do not analyse two programs containing irreducible (multiple-entry) loops, as "iteration count" is ill-defined on these structures. However, this does not preclude them from being checked for other properties.

Table 1 presents the results. Because of the compiler's aggressive optimisation, loops present in the C code were sometimes partially unrolled, completely unrolled or occasionally duplicated. Hence, the total number of loops listed in Table 1 differs in many cases from those evident in the C code.

The benchmark binaries we analysed have a total of 66 loops. Of these, sequoll computed the bounds of 41 automatically (62 %). Sequoll could compute the bound of one further loop (in EXPINT) after specifying a region of interest, as described in Section 4.6.

TABLE 1. Results of evaluating sequoll on programs from the Mälardalen WCET benchmark suite. Each benchmark lists the number of successfully computed loop bounds, and the maximum number of stateful variables and CFG nodes used.

Benchmark	# successful	max state vars	max CFG nodes
ADPCM	5/6	6	12
BS	1/1	10	10
BSORT100	2/3	17	21
CNT	1/1	6	5
COMPRESS	1/7	4	3
COVER	3/3	4	122
CRC	2/2	6	10
DUFF	irreducible loops		
EDN	irreducible loops		
EXPINT	2/3	4	8
FAC	2/2	8	12
FDCT	1/1	4	3
FIBCALL	1/1	4	3
FIR	1/2	4	4
INSERTSORT	1/2	4	3
JANNE_COMPLEX	0/2	4	3
JFDCTINT	2/2	4	3
LCDNUM	no loops (unrolled by gcc)		
MATMULT	5/5	9	7
NDES	6/6	9	12
NS	0/1		
NSICHEU	1/1	4	3
PRIME	0/4		
STATEMATE	0/1		
UD	5/9	5	7

Analysis of the remaining 24 loops fails for a number of reasons. We assume failure when the model checker either exhausts system memory or shows no signs of progress within 12 hours. Note that some loops fail for multiple causes, and we count those under each cause:

- *Memory accesses*: 9 failures are due to loop bounds subject to memory aliasing, as discussed above.
- *Complex early exits*: 7 loops clearly have fixed upper bounds, but can exit early in certain cases. The conditions for early exit depend on complex expressions, such as tests for divisibility in the PRIME benchmark. These expressions generate dozens of stateful variables, preventing the model checker from solving them in a timely fashion.
- *Complex loop bounds*: 8 loops have bounds which are complex expressions. Most of these are inner loops, where the state of the outer loop determines the iteration count. Our model checker fails to determine these loop bounds within several hours.
- *Software division*: 6 loops fail because their loop bounds depend on the result of an integer division – on our ARM platform, integer division is emulated in software, and the routines are far too complex for our model checker.

Analysis time: 28 of the 41 loops are computed by the model checker within five seconds each, and 39 within one minute. Two computations take significantly longer to run: analysis of one loop in BSORT100 takes 14 minutes, and one loop in

ADPCM requires 9 hours. Both of these cases are due to a loop exit condition which requires the model checker to explore the state space across two nested loops simultaneously. The loop in ADPCM also has an upper bound of 999 – the largest of any of our benchmark results.

As the loop bounds increase, the model checker requires more time to explore the state space, and the binary search for the loop bound requires more iterations. Thus, in general, smaller loop bounds will inherently be faster to compute.

Note that these durations do not include the time required by the ARM formalisation to generate the instruction semantics. This step is particularly slow in its current version, taking around one second per instruction. We mitigate this by caching the instructions as they are generated, so that subsequent analyses of the same binary are much faster.

The analysis phase of sequoll used at most 300 MiB of memory for all benchmarks, whilst the NuSMV model checker consumed up to 1.5 GiB.

6. Discussion

While the results so far are encouraging, we would ideally like to automate all loop analysis. Here we take a closer look at what we can do to improve the success rate.

- *Potentially aliasing memory accesses* affect loops and infeasible paths fairly frequently. In the problematic loops, the binary computes the upper bound and stores it into memory, where it vanishes from the view of our analysis. Improved alias analysis may resolve this and result in the automatic computation of the bounds of all affected loops (12 in seL4 and 11 in the WCET benchmarks). Applied to infeasible path detection on seL4, this alone would improve the WCET estimate from 27 % to 54 % over the baseline. Techniques such as those developed for MCVETO [20] are appropriate candidates, as they can be used to identify aliasing conditions relevant to specific properties of interest, and operate on compiled binaries.
- *Complex early exits* within single loop nests we could handle by considering each exit edge individually – the lowest result of any successfully analysed exit edge can be used as a safe upper bound. This works in single loop nests, as following any exit edge precludes the loop from iterating further. It is also suitable for simple inner loops where there are no dependencies on variables defined by outer loops, as the iteration count of an inner loop can be computed without regard to outer loop iterations. However, it is unsound for inner loops with dependencies on the outer loops, as the exit edge taken may vary with each execution of the loop.
- *Complex loop bounds* may be out of reach of model checkers for now, notwithstanding advances in model checking technology. Other techniques such as symbolic execution may be more appropriate to solve these.
- We could deal with *software division* by catching calls to the software emulation routines and replacing them with the model checker's native division operator.
- Loop bounds and infeasible paths which depend on *invariants maintained in the code's environment* cannot be computed by local static analysis. A good example of this (from seL4) is equivalent to the following C code:

```
uint32_t i = 1 << b;
if (i > 256)
   i = 256;
while (i != 0) {
   /* ... */
   i -= 4;
}
```

  Although there appears to be a loop variable with an upper bound, there exist values of the input b for which this loop will never terminate ($b \leq 1$ or $b \geq 32$). An invariant from the seL4 proof states that b is always in the range $[4, 31]$. Thus the loop is safe in its context of use, but our analysis tool is unable to deduce this fact. In fact, any sound method of detecting infeasible paths could not discover this without effectively reproducing a substantial part of the seL4 proof.
  In principle we can address such cases by importing invariants from the formal verification of the kernel into sequoll. Using such information is an interesting challenge for future research. On top of what is achievable by alias analysis, this would achieve a near-optimal WCET (68 % better than baseline) with a very strong level of confidence.

Finally we observe that there are very few infeasible paths in the Mälardalen benchmarks, as these are mostly single-path programs. However, sequoll in fact detected some infeasible paths as loops with an iteration count of 0 (due to virtual inlining, some loops became unreachable). This indicates that this approach could be extended to automatically detect some of the infeasible paths. To make this useful for reducing the overestimation of WCET, the search for infeasible paths should be guided by a WCET analysis.

7. Conclusions

We have presented sequoll, a framework for verifying local properties of binaries such as loop counts and path infeasibility information. Using symbolic model checking on program binaries can avoid a major source of human error, and in turn strengthens the assurances on a WCET analysis.

We evaluated sequoll on the seL4 microkernel, a challenging target due to its non-preemptible design, resulting in long code paths to be analysed. We were able to automatically compute the majority of loops and found that almost all failures could be avoided by the use of a more sophisticated memory aliasing analysis. We also validated several of the infeasible path constraints, gaining a 27 % improvement in WCET, without trusting any user annotations.

We also used the Mälardalen benchmark suite, where sequoll computes 64 % of the loop bounds from the binary alone, alias analysis being again the dominating limitation. However, some issues still remain on complex loops.

The speed of the ARM formalisation is problematic for larger programs, contributing to the majority of the computation time (although its results can be cached). Future versions of the ARM formalisation aim to improve its speed, however the construction within an LCF-style theorem prover inherently limits its performance. Work is under way to formalise the ARMv7 instruction set using a domain-specific language, which can be retargeted for different environments such as theorem provers and other analysis tools [33]. Once this work comes to fruition, we can easily substitute the results into sequoll, or indeed any other formalisation.

Future work will focus on eliminating further sources of human error, by improving memory alias analysis and incorporating formally-proven invariants of the source code into the model.

Combining seL4's machine-checked proof of correctness with a high-confidence timing analysis will create a foundation for mixed-criticality hard real-time systems with an unprecedented level of trustworthiness.

Acknowledgements

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

References

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

[2] A. Hergenhan and G. Heiser, "Operating systems technology for converged ECUs," in *6th Emb. Security in Cars Conf. (escar)*. Hamburg, Germany: ISITS, Nov 2008.

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

[4] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *32nd RTSS*, Vienna, Austria, Nov 2011, pp. 339–348.

[5] B. Blackham, Y. Shi, and G. Heiser, "Improving interrupt response time in a verifiable protected microkernel," in *7th EuroSys Conf.*, Bern, Switzerland, Apr 2012, pp. 323–336.

[6] A. Fox and M. Myreen, "A trustworthy monadic formalization of the ARMv7 instruction set architecture," in *1st ITP*, ser. LNCS, M. Kaufmann and L. C. Paulson, Eds., vol. 6172. Edinburgh, UK: Springer-Verlag, Jul 2010, pp. 243–258.

[7] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *10th WS Worst-Case Execution-Time Analysis*. Brussels, Belgium: OCG, Jul 2010, pp. 137–147.

[8] A. Metzner, "Why model checking can improve WCET analysis," in *Computer Aided Verification*, ser. LNCS, R. Alur and D. Peled, Eds. Springer-Verlag, 2004, vol. 3114, pp. 298–301.

[9] C. Cullmann and F. Martin, "Data-flow based detection of loop bounds," in *7th WS Worst-Case Execution-Time Analysis*, 2007.

[10] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models," in *7th IEEE Symp. Code Generation & Optimization*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 136–146.

[11] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács, "ABC: algebraic bound computation for loops," in *16th Int. Conf. Logic for Progr., Artificial Intelligence & Reasoning*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 103–118.

[12] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution," in *27th RTSS*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 57–66.

[13] J. Knoop, L. Kovàcs, and J. Zwirchmayr, "r-TuBound: Loop bounds for WCET analysis (tool paper)," in *Logic for Programming, Artificial Intelligence, and Reasoning*, ser. LNCS, N. Bjrner and A. Voronkov, Eds. Springer Berlin / Heidelberg, 2012, vol. 7180, pp. 435–444.

[14] B. Rieder, P. Puschner, and I. 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*, Jul 2008, pp. 1–7.

[15] I. Narasamdya and A. Voronkov, "Finding basic block and variable correspondence," in *Proceedings of the 12th international conference on Static Analysis*, ser. SAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 251–267.

[16] F. Cassez, "Timed games for computing WCET for pipelined processors with caches," in *11th Int. Conf. Applic. Concurrency to Syst. Design*. IEEE Comp. Soc., Jun 2011, pp. 195–204.

[17] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "Efficient detection and exploitation of infeasible paths for software timing analysis," in *43rd DAC*. New York, NY, USA: ACM, 2006, pp. 358–363.

[18] M. N. Ngo and H. B. K. Tan, "Detecting large number of infeasible paths through recognizing their patterns," in *6th ESEC*. New York, NY, USA: ACM, 2007, pp. 215–224.

[19] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST: Applications to software engineering," *Int. J. Softw. Tools for Technology Transfer*, vol. 9, no. 5, pp. 505–525, 2007.

[20] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps, "Directed proof generation for machine code," in *22nd CAV*, 2010, pp. 288–305.

[21] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "TRACER: A symbolic execution tool for verification," in *24th CAV*. Berlin, Heidelberg: Springer-Verlag, 2012.

[22] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Trans. Comp. Syst.*, vol. 30, no. 1, pp. 2:1–2:49, Feb 2012.

[23] M. Daum, S. Maus, N. Schirmer, and M. N. Seghir, "Integration of a software model checker into Isabelle," in *12th Int. Conf. Logic for Progr., Artificial Intelligence & Reasoning*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 381–395.

[24] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *20th CAV*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 423–427.

[25] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA framework for binary code analysis," in *23rd CAV*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 165–170.

[26] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "ARMor: fully verified software fault isolation," in *11th EMSOFT*. New York, NY, USA: ACM, 2011, pp. 289–298.

[27] J. Kinder, F. Zuleger, and H. Veith, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *10th Int. Conf. Verification, Model Checking & Abstract Interpretation*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 214–228.

[28] S. Bardin, P. Herrmann, and F. Védrine, "Refinement-based CFG reconstruction from unstructured programs," in *12th Int. Conf. Verification, Model Checking & Abstract Interpretation*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 54–69.

[29] R. E. Tarjan, "Testing flow graph reducibility," *J. Comp. & Syst. Sci.*, vol. 9, no. 3, pp. 355–365, 1974.

[30] P. Havlak, "Nesting of reducible and irreducible loops," *ACM Trans. Progr. Lang. & Syst.*, vol. 19, no. 4, pp. 557–567, Jul 1997.

[31] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Progr. Lang. & Syst.*, vol. 13, pp. 451–490, October 1991.

[32] M. Weiser, "Program slicing," *IEEE Trans. Softw. Engin.*, vol. SE-10, no. 4, pp. 352–357, Jul 1984.

[33] A. Fox, "Directions in ISA specification," in *3rd ITP*, ser. LNCS. Princeton, New Jersey: Springer-Verlag, Aug 2012.