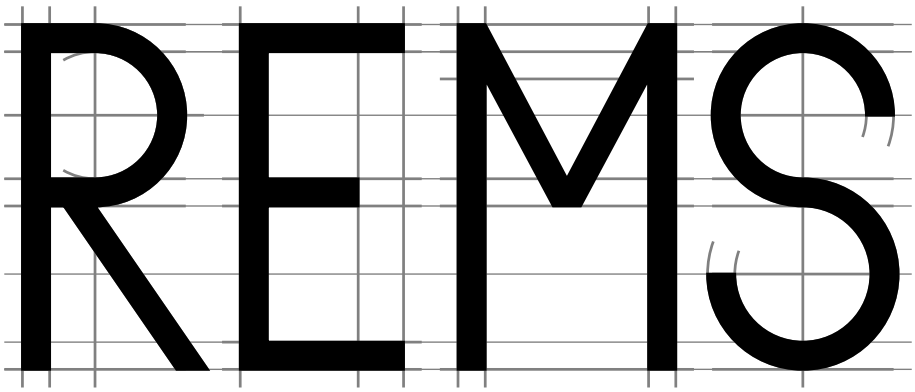


REMS: Rigorous Engineering for Mainstream Systems

EPSRC Programme Grant EP/K008528/1, £5.6M, 2013–2020

http://www.cl.cam.ac.uk/users/pes20/rems

Summary, 2013-03 – 2019-01

September 30, 2019

Principal Investigator: Peter Sewell (Computer Laboratory, University of Cambridge)
Co-Investigators: Jon Crowcroft, Neel Krishnaswami, Mike Gordon, Anil Madhavapeddy, Andrew Pitts, Simon Moore, Robert Watson (Cambridge); Philippa Gardner (Imperial); Ian Stark (Edinburgh)
Industrial Partners: ARM, IBM, Microsoft, FreeBSD Foundation
Academic Partners: INRIA, University of Pennsylvania, Purdue University, U. Texas Austin

Contents

Abstract

REMS, *Rigorous Engineering for Mainstream Systems*, is an EPSRC-funded Programme Grant (2013–2020, £5.6M, Cambridge, Imperial, and Edinburgh) to explore how we can use rigorous mathematics to improve engineering practice for mainstream computer systems, to make them more robust and secure: broadly, to *"create the intellectual and software tools to apply semantics and verification at the heart of mainstream system engineering"*. REMS brings together an unusual combination of researchers to achieve this: in architecture, operating systems, security, and networks, and in semantics, programming languages, and mechanised reasoning. Our work is in close collaboration with a range of industrial partners, including ARM, IBM, and the standards committees for C, JavaScript, WebAssembly, and TLS.

We are building accurate full-scale mathematical models of some of the key computational abstractions (processor architectures, programming languages, concurrent OS interfaces, and network protocols), studying how this can best be done, and investigating how such models can be used for new verification research and in new systems and programming language research. For many of these abstractions, our work has exposed and clarified fundamental questions about what the abstractions are, and provided tools to let them be explored. Supporting all this, we are also developing new specification tools. Most of our models and tools are publicly available under permissive open-source licences.

This note summarises REMS progress to date. Highlights include models and tools for the production ARMv8-A architecture (including user-mode concurrency and the full ISA), the RISC-V and research CHERI-MIPS architectures, the C, JavaScript, and WebAssembly programming languages, ELF linking, POSIX filesystems, the TLS and TCP protocols, and extensive work on concurrency verification.

Introduction: the REMS project

The mainstream computing infrastructure on which we all rely works well enough to support much of modern society, but it is neither robust nor secure, as demonstrated over the years by untold numbers of system failures and successful malicious attacks. Fundamentally, this is because the engineering methods by which computer systems are built are not up to the task. Normal industry practice is based on a test-and-debug engineering process, with triumvirates of code, manually curated tests, and prose specification documents; it relies also on the division of labour that is enabled by common interfaces, allowing systems to be composed of subsystems built by different groups. But prose specification documents do not provide an oracle that one can test a system against, leaving code, tests, and specifications only weakly related; they are inherently ambigous, lacking mathematical precision; and the existing specifications of many key interfaces leave much to be desired. Moreover, some of the legacy interfaces that we still rely on, such as the C programming language, were simply not designed to protect against the accidental errors and malicious attacks that are now so common.

REMS (Rigorous Engineering for Mainstream Systems), is an EPSRC-funded Programme Grant (2013–2020, £5.6M, Cambridge, Imperial, and Edinburgh) to explore how we can use rigorous mathematics to improve engineering practice for mainstream computer systems, to make them more robust and secure: broadly, to *"create the intellectual and software tools to apply semantics and verification at the heart of mainstream system engineering"*. Addressing this problem needs close interaction between the normally disjoint cultures of systems and semantics research, and REMS brings together an unusual combination of researchers in both to achieve this: in architecture, operating systems, security, and networks, and in programming languages, automated reasoning, and verification. Building on many years of research worldwide on mathematical semantics and verification, we are finally in a position to scale that up, to apply mathematically rigorous semantics to mainstream systems. REMS aims:

- to develop rigorous semantics for key mainstream abstractions; and
- to improve those, and to develop new systems research, using rigorous semantics and semantics-based tools.

The project is focussed on lightweight rigorous methods: precise specification (post hoc and during design) and testing against specifications, with full verification only in some cases. It thus provides a relatively smooth adoption path from conventional engineering, and the project emphasises building useful (and reusable) semantics and tools. We are building accurate full-scale mathematical models of some of the key computational abstractions (processor architectures, programming languages, concurrent OS interfaces, and network protocols), studying how this can be done, and investigating how such models can be used for new verification research and in new systems and programming language research. Supporting all this, we are also working on new specification tools and their foundations.

REMS is organised into five interlinked tasks, shown in Fig. 1.Three tasks are focussed on the three key abstractions: Task 2 on multicore processor architectures, Task 3 on programming languages, and Task 4a on concurrent systems software interfaces. In each of these, some subtasks build models and specifications of key parts of today's mainstream interfaces or of major research proposals, while other subtasks go on to apply, extend, and relate those models. For all three, good *semantic tools*, built in Task 1, are needed to work with the large mathematical definitions of our models. Task 4b addresses higher-level abstractions and the theory of how we can reason about these concurrent systems (Task 4 of the original proposal has been split into Task 4a and Task 4b for clarity).

Much of this needs collaboration not just between systems and semantics researchers within the project, but also with academic partners elsewhere and with industry vendors and groups; some of the latter are shown in Fig. 1.We have substantial interactions with ARM, IBM, the ISO C and C++ standards committees (WG14 and WG21), the RISC-V Foundation *Memory Model* and *ISA Formal* Task Groups, the IETF, the ECMAScript committee, and the WebAssembly Committee. We have also engaged with Amazon, POSIX, the FreeBSD

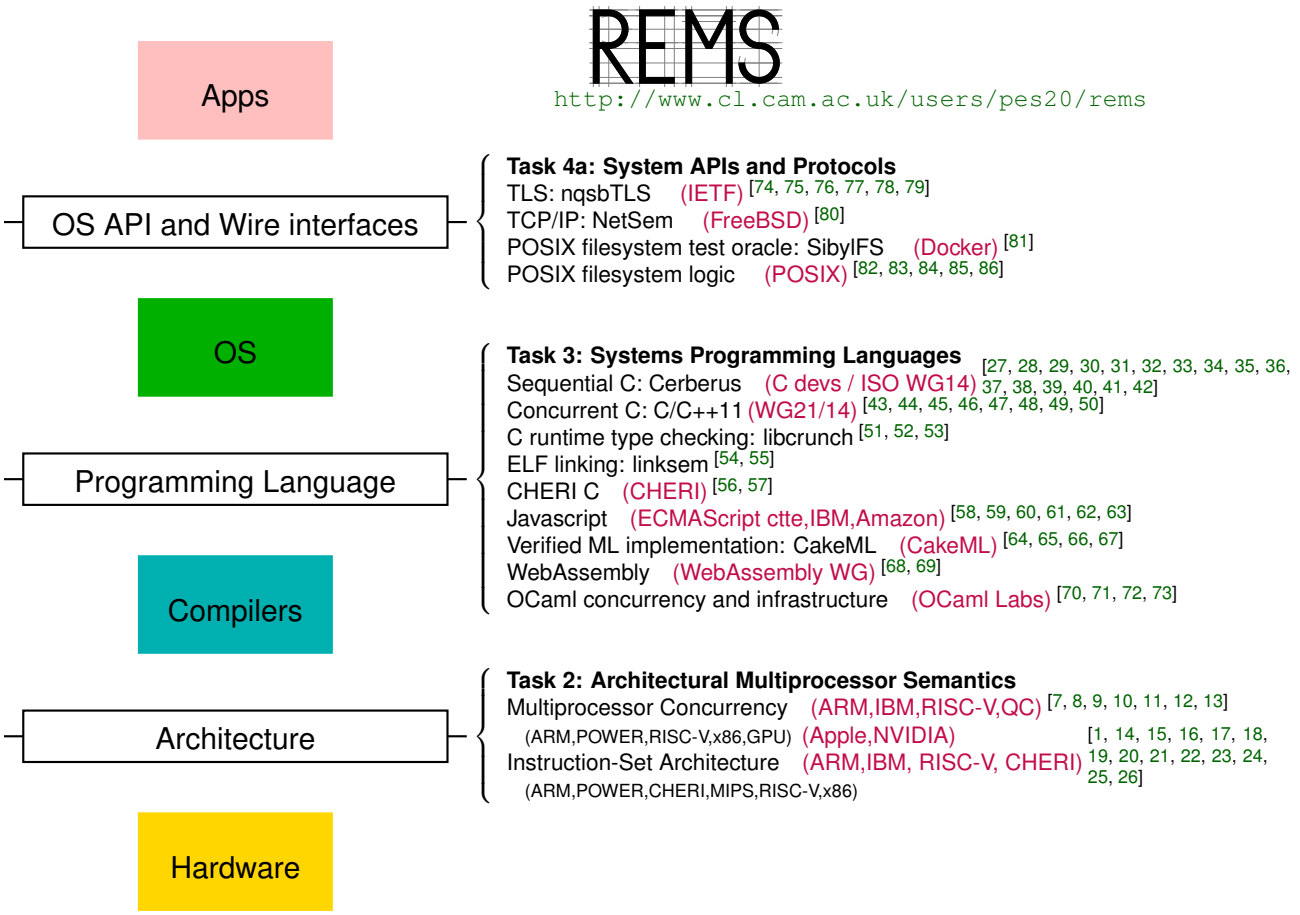


Figure 1: REMS Overview

development community, the Linux concurrency development community, Qualcomm, Apple, and Nvidia. Also very important are close working links with other major academic projects, especially:

- CTSRD: Rethinking the hardware-software interface for security, including CHERI (Moore, Neumann, Watson et al.; Cambridge and SRI International), https://www.cl.cam.ac.uk/research/security/ctsrd/
- CakeML: A Verified Implementation of ML (Fox, Kumar, Myreen, Norrish, Owens, Tan; Chalmers, Kent, Cambridge, Data61), https://cakeml.org/
- OCaml Labs and Mirage OS (Madhavapeddy et al.; Cambridge), http://www.cl.cam.ac.uk/projects/ocamllabs/, https://mirage.io/
- SibylFS: A Filesystem Test Oracle (Ridge et al.; Leicester), http://sibylfs.github.io/

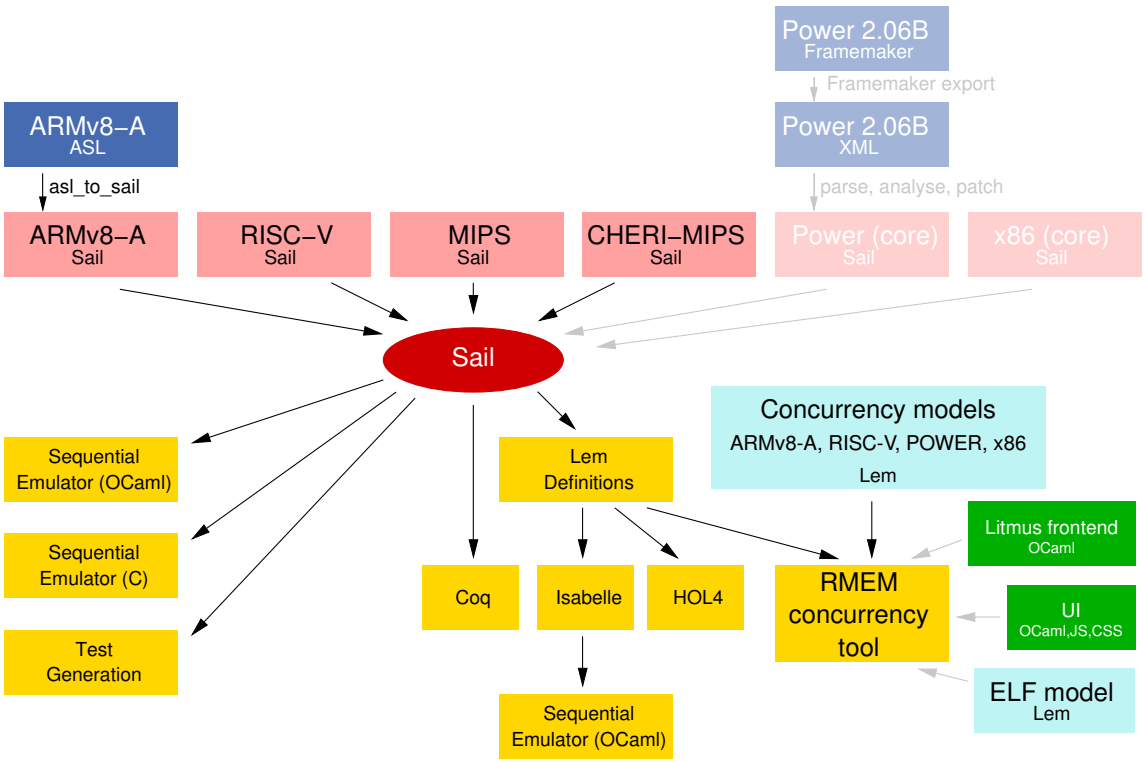
Academic collaborators include colleagues at St Andrews, Kent, Leicester, INRIA Paris, Chalmers, NICTA Australia, and the DeepSpec NSF Expedition in Computing in the USA.

In this document we summarise progress in the project as a whole, from its start in 2013-03 to date (2019-01). The current state of the project, including links to all publications to date and to our main models and software, is presented on the REMS web page, http//www.cl.cam.ac.uk/users/pes20/rems.

1 Task 1: Semantic tools

The mathematical models of mainstream-system abstractions that we need have to be expressed in well-defined formal languages, for the sake of mathematical precision, and those languages need good software tool support, to let us handle large and complex definitions. In this task we address the foundational question of what formal languages are suitable for this purpose; the pragmatic question of what facilities good tool support should provide; and the engineering question of actually building these tools to a sufficient quality to be usable by other researchers and by industry staff. In REMS we are developing and using the following tools.

- **Sail** is a domain-specific language for concurrent and sequential processor instruction-set architecture (ISA) definition, to define the architected behaviour of processor instructions [1, 7, 8, 10, 12]. Sail is a first-order functional/imperative language, with a lightweight dependent types for vector lengths and indices and for integer ranges. We have Sail models for ARMv8-A, RISC-V, MIPS, and CHERI-MIPS, each complete enough to boot an OS, and for smaller fragments of IBM POWER and x86. Given a Sail ISA definition, the tool will automatically generate emulators in C and OCaml; theorem-prover definitions in Isabelle, HOL4, and Coq (via Lem for the first two); and versions of the semantics integrated with our RMEM concurrency model tool (both a deep embedding, in a form that can be executed by a Sail interpreter written in Lem, and shallow embeddings to Lem and OCaml). Sail is available under a permissive open-source licence.



Sail web page: http://www.cl.cam.ac.uk/users/pes20/sail/.
Sail github repo: https://github.com/rems-project/sail.
Sail is also available as an OPAM package.

- **Lem** is a tool for large-scale portable semantic specification [2, 6]. Lem supports the definition language of a typed higher-order logic, for type, function, and inductive-relation definitions, in an OCaml-like syntax. Given such a definition, the tool will automatically generate (for various fragments) OCaml, LaTeX, HTML, Isabelle, HOL4, and Coq versions. Lem is used in our work on Sail, concurrent architecture specification, CakeML, ELF linking, Concurrent C semantics, Sequential C semantics, and SibylFS POSIX filesystem semantics. Lem is available under a permissive open-source licence.

  Lem web page: http://www.cl.cam.ac.uk/users/pes20/lem/.
  Lem github repo: https://github.com/rems-project/lem/.

- **Ott** is a tool for programming language definitions, supporting user-defined syntax and inductive relations over it, and compiling to LaTeX and to theorem-prover definitions for Coq, HOL4, and Isabelle; it can also generates some OCaml infrastructure, e.g. concrete substitution functions and simple parser and pretty-printer code. Ott is available under a permissive open-source licence.

  In 2017, the original Ott paper

  > Ott: Effective Tool Support for the Working Semanticist. Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. ICFP 2007

  was awarded the ACM SIGPLAN *Most Influential ICFP Paper Award* (the papers are judged by their influence over the past decade).

  Ott web page: http://www.cl.cam.ac.uk/users/pes20/ott/.
  Ott github repo: https://github.com/ott-lang/ott.
  Ott is also available as an OPAM package.

- **L3** is an earlier domain-specific language for sequential processor instruction-set architecture (ISA) semantic definition [4, 64]. L3 exports to HOL4 and Isabelle/HOL, including Hoare-triple generation, and to SML, providing reasonably fast emulation. L3 is used in our work on CakeML, CHERI, and instruction testing. L3 is available under a permissive open-source licence.

  L3 github repo: `https://acjf3.github.io/l3`.

In Task 1, we have also worked on:

- A semantic meta-language enabling multiple interpretations, for concrete, abstract, and flow-sensistive analysis intepretations [3].
- The foundations needed for future semantic tools, focussed on Nominal Type Theory [5].

2 Task 2: Architectural Multiprocessor Semantics

The first key interface we address is the architectural interface between hardware and software, specifying the multiprocessor behaviours that processor implementations guarantee and that systems code relies on. Normal industrial practice is to specify this interface in informal-prose architecture reference manuals, sometimes with pseudocode descriptions of the sequential instruction semantics, but these descriptions cannot be used directly to test either the hardware below the interface or software above it, or to build verification tools, let alone as a basis for proof. Compounding the problem, architectures must be nondeterministic loose specifications, to accommodate microarchitectural variation, so one cannot simply characterise them with reference implementations.

There are two linked challenges: handling the subtlety of the concurrency architectures, and handling the scale of the instruction-set descriptions. We are addressing these for multiple architectures: the ARM A-class production architecture, which is the basis for the processors used in the majority of mobile phones and many other devices; the IBM POWER production architecture, used for high-performance server machines, among other areas; the open RISC-V architecture; the x86 architecture of Intel and AMD processors, used in the majority of laptop, desktop, and server machines; and the research CHERI architecture, of which more below.

2.1 Multiprocessor Concurrency: Rigorous Concurrency Architecture

The *concurrency architecture* of a family of multiprocessors defines the range of allowed behaviours of concurrent programs. It is the interface between multiprocessor hardware vendors, who endeavour to ensure that all their hardware implementations have behaviour within that range, and concurrent programmers (of high-performance concurrent algorithms, operating system kernels, etc.), who can thereby assume that range as their programmer's model. Ten years ago, in 2008, most architectures, including ARM, x86, and IBM POWER, did not have well-defined concurrency architectures; they had only non-rigorous and highly confusing prose descriptions of the allowed behaviour, sometimes augmented by a few examples. These did not provide a usable basis for either hardware testing or concurrent programming.

Our work before REMS, in collaboration with other academic colleagues and with senior IBM staff, established rigorous concurrency models for modest fragments of the x86 and IBM POWER architectures. These covered the fundamental "user-mode" non-privileged concurrency instructions (loads, stores, dependencies, memory barriers, read-modify-write and load-linked/store-conditional instructions), with ad hoc descriptions of the intra-instruction behaviour and assuming no mixed-size accesses.

Within REMS, we have successfully moved to a world in which these aspects *of production architectures* are mathematically defined, based on rigorous engineering processes including formal modelling, proof about those models, making the models executable as test oracles, and testing against them.

ARMv8-A We worked closely with ARM staff and our INRIA partners to clarify the concurrency model of their new ARMv8-A architecture, with a combination of rigorous formal modelling, experimental testing, test generation, and mathematical proof. This contributed to ARM's 2017 switch to a "multi-copy-atomic" model – a substantial simplification. Together we have developed rigorous axiomatic and operational models, principally by ARM and by our academic group respectively, but deeply informed by each other and proved equivalent for the intersection of their scope. The ARM architecture definition now contains the former, replacing its previous unusable prose [7, 8, 10].

Along the way, our experimental testing discovered a number of hardware errata, in core designs from four major vendors, and our test suite is being used within ARM.

RISC-V Most major architectures (x86, ARM, MIPS, IBM POWER, Itanium) are proprietary. In contrast, the RISC-V architecture is an open-source architecture, gathering increasing academic and industry interest. Its concurrency architecture is broadly similar to the simplified ARMv8 model, but with a number of subtle differences. We have contributed substantially to the RISC-V Memory Model Task Group and the resulting specification [14], which again has (experimentally) equivalent axiomatic and operational models.

We have also provided a permissive open-source library of approximately 7 000 concurrency litmus tests for RISC-V, including the reference results from the two models.

RISC-V Litmus Tests github repo: `https://github.com/litmus-tests/litmus-tests-riscv`

Mixed-size and instruction-semantics integration Scaling up from the modest fragments of the instruction set of our pre-REMS work, we have extended all of these models – and also our operational IBM POWER (working closely with senior IBM staff) and x86 models – to cover *mixed-size* accesses (all the subtleties of overlapping accesses to parts of memory and registers), and integrated them with principled descriptions of the intra-instruction semantics. This finally suffices to define the behaviour of concurrent algorithm code, not just the litmus tests used in concurrency model development.

Our RMEM exploration tool Combining all the above with extensive search and user-interface engineering, we provide a web-interface tool, RMEM, that allows one to explore, either interactive, randomly, or exhaustively, the behaviour of concurrent code for ARMv8, RISC-V, IBM POWER, and x86. This supports both litmus tests and small ELF binaries (e.g. compiled by gcc or clang), using our semantics for ELF linking and DWARF debug information from Task 3. The model-allowed behaviour results can then be compared with experimental data, obtained using the `litmus` tool of the diy tool suite (`http://diy.inria.fr/doc/`), from production silicon or simulation. The sources for our operational models and RMEM tool are available under a permissive open-source licence.

RMEM tool web interface: `http://www.cl.cam.ac.uk/users/pes20/rmem`.
RMEM github repo: `https://github.com/rems-project/rmem`.

GPU Concurrency We also studied the concurrency semantics of GPUs, with colleagues elsewhere – conducting a large empirical study of their behaviour and proposing a model of Nvidia GPU hardware [11].

2.2 Instruction-Set Architecture (ISA)

The second important aspect of architecture specification is the sequential intra-instruction behaviour. At least for the non-privileged part of an architecture, and neglecting floating point, this is usually less semantically involved than the concurrency behaviour, but the scale of modern ISAs makes it a non-trivial problem, particularly when one wants definitions that are both readable and usable for mathematical proof. We have developed two domain-specific languages for describing ISA behaviour: first L3 and then Sail, as introduced in Task 1.

ARMv8-A We are working closely with ARM to translate their internal ("ASL") description of their ARMv8-A architecture into Sail [1]. The resulting Sail model is complete enough to boot Linux, and to pass most of the relevant ARM-internal Architecture Validation Suite tests (in the C emulator generated from Sail). We have approval from ARM to release this under a permissive open-source licence. This will be the first modern production architecture to have a public formal definition complete enough for an OS boot; it should provide a foundation for much future work on system software verification. As a first step towards this, we have an Isabelle machine proof of a characterisation of ARMv8-A address translation.

The first version of this was for a fragment based on the ARM public XML release for ARMv8.3-A, excluding system registers. A more complete version, updated to ARMv8.5-A, is now available under a permissive open-source licence (BSD 3 Clause Clear).

ARMv8.5-A ASL-derived Sail ISA model, in a github repo: https://github.com/rems-project/sail-arm.
ARMv8.3-A ASL-derived Sail ISA model, in the Sail github repo: https://github.com/rems-project/sail/tree/sail2/aarch64.

We also have an earlier hand-written Sail ISA model for a modest user-mode ARM fragment; this model is integrated with our RMEM tool to compute the set of all allowed behaviour for ARMv8 concurrency litmus tests or small ELF files.

ARMv8 hand-written Sail ISA model, in the Sail github repo: https://github.com/rems-project/sail/tree/sail2/arm.

ARMv6, ARMv7, and ARM Cortex-M0 We also developed sequential L3 models for the ARMv6-A and ARMv7-A architectures and the ARM Cortex-M0 implementations, the former including floating-point and some system-level instructions, and the latter equipped with accurate cycle counts for work on resource-aware compilation.

RISC-V We are working closely with our SRI International partners to hand-write a formal RISC-V ISA specification. This is also complete enough to boot an OS, here Linux, FreeBSD, and seL4, and the model is integrated with our RMEM tool to compute the set of all allowed behaviour for the RISC-V concurrency litmus tests mentioned above.

RISC-V Sail ISA model github repo: https://github.com/rems-project/sail-riscv

IBM POWER We worked closely with IBM to automatically translate their ISA pseudocode, for a modest user-mode non-vector non-floating-point fragment, from an XML version of their ISA documentation (not previously parsed or type-checked) into a Sail formal model (albeit currently only for an older version of Sail) [12]. This too is integrated with the RMEM tool. We tested the ISA model by generating single-instruction test cases and comparing their behaviour in the model and on POWER 7 hardware; for the 154 user-mode branch and fixed-point instructions, we generated 6984 tests.

IBM POWER Sail ISA model, in the Sail github repo: https://github.com/rems-project/sail/tree/sail2/power.

x86 We have hand-written a Sail model for a modest user-mode fragment of x86, again integrated with RMEM to support concurrency exploration.

x86 Sail ISA model, in the Sail github repo: https://github.com/rems-project/sail/tree/sail2/x86.

2.3 CHERI Architecture Formal Engineering

Turning to large-scale systems research, the CHERI project, led by Moore and Watson at Cambridge and Neumann at SRI International, largely DARPA-funded from 2010 to date, is developing new architectural mechanisms for security protection, revisiting the hardware-software security interface for general-purpose CPUs to fundamentally improve security. CHERI includes an architecture design, processor implementations, and a software stack adapting Clang/LLVM and FreeBSD.

In contrast to ARM and IBM POWER, this is a research architecture, not a mainstream production architecture, but it is being developed at scale, e.g. with a software stack including a variant of a production OS.

CHERI was originally developed using traditional systems and computer architecture methods, with prose and pseudocode descriptions of the architecture, hand-written test suites, and hardware and software prototypes – though with an unusual hardware/software co-design process. REMS created the opportunity for new collaborations between the original CHERI team, spanning systems, security, and computer architecture, and REMS researchers in semantics, verification, and programming languages. This gave us an ideal context to develop and try new rigorous engineering methods in practice:

- Since 2014, CHERI architecture research has used formal models of the architecture as central design tools – first in our L3 language, and now in our Sail language. The Sail version is used as the principal definition in the current working version of the CHERI ISA specification (to be released as version 7 of the CHERI architecture).
- An SML emulator (running at several hundred KIPS) automatically generated from the L3 formal model has been invaluable, e.g. to bring up a bootable CHERI version of FreeBSD above the model, and as an oracle for testing the CHERI hardware implementation.
- We have augmented the hand-written CHERI test suite with an automatic test generator: it creates randomized instruction sequences paired with precise descriptions of their required behaviour, machine-checked against our model by HOL4/SMT provers [26, 25, 24]. We have also produced a generic synthesisable test bench, BlueCheck [19]. This has been used to check various microarchitectural properties of CHERI, finding some bugs. It has moreover been used for teaching in MIT (in the 6.175 Computer Architecture course).
- As a first exercise in developing mathematically rigorous *proof* of specific security properties about an architecture description, we have generated theorem-prover definitions (in the Isabelle and HOL4 provers), discussed and clarified some of the fundamental security properties the CHERI designers intended the architecture to guarantee, and produced machine proofs (in Isabelle) of some of these.
- We have proved (mechanised proof, in HOL4) correctness properties of the "CHERI-128" compressed capability scheme.
- We have further contributed to the CHERI hardware implementation by building an efficient checker for its memory consistency model, our Axe tool [9], which has also been used for RISC-V.

The theory/practice interactions with the CHERI team have also been fruitful at higher levels in the stack, for the C language, linking, and OS interfaces, as we return to below. This has only been possible with intertwined groups that cover both systems and semantics at multiple layers of the stack, from hardware to systems software.

2.4 CakeML ISA models

We have used L3 to define the user-mode instruction semantics for each of the target architectures of the CakeML verified compiler, and to generate proof infrastructure for them [64, 65, 66, 67]. CakeML uses the above ARMv6-A and ARMv8-A L3 models, together with MIPS and x86-64 models developed here and a RISC-V model developed by our partners at SRI International.

3 Task 3: Systems Programming Languages

Here our main focus is on abstractions used for real-world systems programming, looking at the C language (both sequential and concurrent aspects), runtime typechecking for systems code (in the libcrunch system), linking (specifically at the ELF format), CHERI C JavaScript, WebAssembly, and OCaml. We have also contributed to the CakeML verified ML compiler with the ISA modelling work described in Task 1.

3.1 Sequential C: Cerberus

The C language, originating around 1970, remains at the heart of a great deal of security-critical systems software, including operating systems, hypervisors, and embedded and Internet-of-Things devices – and many security flaws ultimately come down to C code. Ambiguities and differences of opinion about what the language is thus have considerable potential for harm – but it has never been well-defined. In earlier (pre-REMS) work, we collaborated closely with the ISO WG21 C++ and WG14 C committees to clarify their concurrency models, with our mathematical models (in prose form) incorporated into those international standards.

Our work on the sequential aspects of C semantics proceeds on several fronts: investigating the de facto standards of C as it is implemented and used; addressing and contributing to the ISO standard produced by the ISO WG14 C committee; and building an executable formal model for a substantial fragment of C, Cerberus [27, 28]. Our initial paper on this [27] was awarded a PLDI 2016 Distinguished Paper Award. We intend to release Cerberus under a permissive open-source licence in due course.

Cerberus web page `https://www.cl.cam.ac.uk/users/pes20/cerberus/`
Cerberus web interface: `http://cerberus.cl.cam.ac.uk/cerberus`

We are participating in the ISO WG14 C standards committee: we attended the April 2016 (London), October 2016 (Pittsburgh), April 2018 (Brno), and October 2018 (Pittsburgh) committee meetings, producing committee discussion papers ([29, 30, 31, 32], [33, 34, 35], [36, 37, 38, 39, 40], [41]), and instigated the WG14 C Memory Object Model Study Group; we are also engaging with the various compiler and developer communities, with talks at EuroLLVM 2018 and the GNU Tools Cauldron 2018, to Red Hat, and at 35c3.

3.2 CHERI C

At the C language level, just as at the architecture level, we have another strong collaboration with the CHERI work, with their experience in systems software (and specifically in porting it to the C supported by CHERI) informing our semantic understanding, and with this discussion identifying issues with the CHERI design. This made substantial contributions to the two Cerberus papers cited above [27, 28], and we have also contributed to work on the CHERI ABI, enforcing pointer provenance and minimising pointer privilege in a POSIX C environment [57].

We also developed a conceptual framework for application compartmentalization, a vulnerability mitigation technique employed in programs such as OpenSSH and the Chromium web browser, which decomposes software into isolated components to limit privileges leaked or otherwise available to attackers. This is embodied in an LLVM-based tool: the Security-Oriented Analysis of Application Programs (SOAAP), that allows programmers to reason about compartmentalization using source-code annotations (compartmentalization hypotheses) [56].

3.3 C runtime type checking: libcrunch

We are developing a system for run-time type checking in C, libcrunch, extending it to additionally check array/pointer bounds [51, 52, 53]. Using run-time type information, we have found it possible to tolerate real C idioms not supportable by software fat-pointer schemes. Early indications suggest that the available run-time performance is comparable or better than competing approaches.

We have continued to refine the run-time type infrastructure underlying libcrunch into a more general-purpose infrastructure with a variety of other applications [55] including debugging, interfacing native code with dynamic languages, and dealing with memory-mapped files.

Libcrunch github repo: https://github.com/stephenrkell/libcrunch/

3.4 ELF linking: linksem

Systems software in reality is written not just in more-or-less conventional programming languages, like C and C++, but also using a complex menagerie of techniques to control the linking process. This *linker speak* is often poorly understood by working programmers and has largely been neglected by language researchers. We have surveyed the many use-cases that linkers support and developed the first validated formalisation of a realistic executable and linkable format, ELF, together with an executable specification of linking with substantial coverage [54]. This also incorporates a semantics for DWARF debug information. The model is written in Lem, which generates an executable OCaml version. This ELF model is used in our Task 2 RMEM architectural emulation tool. The model is available under a permissive open-source licence.

ELF semantics github repo: https://github.com/rems-project/linksem.

3.5 Concurrent C/C++

Turning to the concurrency semantics of C-based languages, in earlier (pre-REMS) work, we collaborated closely with the ISO WG21 C++ and WG14 C committees to clarify their concurrency models, with our mathematical models (in prose form) incorporated into those ISO C/C++11 international standards. (Batty et al., POPL 2011). We have built on those, and also considered potential future models that would fix the so-called 'thin-air problem' that C/C++11 suffers from.

Batty's 2014 Cambridge PhD thesis *The C11 and C++11 Concurrency Model* was awarded the 2015 ACM SIGPLAN John C. Reynolds Doctoral Dissertation Award, presented annually to the author of the *"outstanding doctoral dissertation in the area of Programming Languages"*, and was the winner of the 2015 CPHC/BCS Distinguished Dissertations competition, selected as *"the best British PhD/DPhil dissertations in computer science"*. The WG14 C committee also approved his remaining defect reports w.r.t. the C11 concurrency model.

We contributed to work in LLVM implementing the C/C++11 concurrency model [49, 50].

We developed a C11 operational semantics that is proved equivalent (in Isabelle/HOL) to the Batty et al. axiomatic model [43]. This was also integrated into our Cerberus sequential C model to allow litmus tests to be executed, though that integration is not currently maintained, to simplify Cerberus development.

We showed that no straightforward adaptation of the C/C++11 axiomatic model, phrased as a consistency predicate over candidate executions, can exclude thin-air reads while also admitting some standard optimisations, identified additional difficulties with undefined behaviour and with mixtures of atomic and nonatomic accesses. We also proved (in HOL4) that the C/C++11 model actually provides the DRF-SC guarantee [45].

We proposed a model that admits compiler optimisation while avoiding the thin-air problem [44, 47].

Along with the extension of our ARM and IBM POWER hardware models to support mixed-size accesses, described above, we proposed a corresponding extension to the C/C++11 axiomatic model to support mixed-size non-atomic accesses [8].

We discussed a new problem with C/C++11, uncovered by groups in Princeton and MPI-SWS/SNU, that contradicts the soundness of compilation schemes from C/C++11 to ARM and POWER (pre-REMS, we had published hand proofs of those that now turn out to be flawed); this required additional changes to C/C++11, developed by the latter group.

We proposed a safe, portable, and efficient implementation of co-routines [48].

3.6 Verified ML implementation: CakeML

We continue to contribute to the CakeML development of a verified ML implementation, especially w.r.t. the backend ISA models and proof, as discussed in Task 2.

3.7 JavaScript

We have developed JSExplain [60], a reference interpreter for JavaScript that closely follows the specification and that produces execution traces. These traces may be interactively investigated in a browser, with an interface that displays not only the code and the state of the interpreter, but also the code and the state of the interpreted program. In that respect, JSExplain is a double-debugger for the specification of JavaScript, useful both for standard developers and JavaScript programmers.

We have developed a system for logic-based verification of JavaScript programs [61] and a symbolic evaluation system for JavaScript [58].

We have developed JaVerT [59, 63], a semi-automatic JavaScript Verification Toolchain, based on separation logic and aimed at the specialist developer wanting rich, mechanically verified specifications of critical JavaScript code. JaVerT is the first verification tool based on separation logic that targets dynamic languages. To specify JavaScript code, we provide abstractions that capture its key heap structures (for example, prototype chains and function closures), allowing the developer to write clear and succinct specifications with minimal knowledge of the JavaScript internals. To verify JavaScript programs, we develop a trusted verification pipeline consisting of: JS-2-JSIL, a well-tested compiler from JavaScript to JSIL, an intermediate goto language capturing the fundamental dynamic features of JavaScript; JSIL Verify, a semi-automatic verification tool based on a sound JSIL separation logic; and verified axiomatic specifications of the JavaScript internal functions. JaVerT has so far been used to verify functional correctness properties of: data-structure libraries (key-value map, priority queue) written in an object-oriented style; operations on data structures such as binary search trees (BSTs) and lists; examples illustrating function closures; and test cases from the official ECMAScript test suite. The verification times suggest that reasoning about larger, more complex code using JaVerT is feasible.

This work has been presented in keynotes in CADE 2017 and FSEN 2017, and at an invited talk at *Emerging Technologies*, the 2017 New York ECMAScript meeting.

JavaScript semantics web page: `http://psvg.doc.ic.ac.uk/research/javascript.html`
JavaScript semantics (jscert) repo: `https://github.com/jscert/jscert`

3.8 WebAssembly

WebAssembly is a new low-level language currently being implemented in all major web browsers. It is designed to become the universal compilation target for the web, obsoleting previous solutions such as asm.js and Native Client. We have developed a mechanised Isabelle specification for WebAssembly, including a verified executable interpreter and type-checker, and a mechanised proof of type soundness [68]. This exposed several issues with the official WebAssembly specification and influenced its development. We have also worked on constant-time computation in WebAssembly [69].

WebAssembly AFP Isabelle development: `https://www.isa-afp.org/entries/WebAssembly.html`

3.9 OCaml concurrency and infrastructure

Working with our OCaml Labs colleagues, we have proposed a concurrency semantics for OCaml [70] and contributed to various OCaml infrastructure: a generic approach to defining partially-static data and corresponding operations [72]; *conex* [73], a tool which allows developers to cryptographically sign their released software, and users to verify them, ensuring integrity and authenticity without a centralised trust authority; and a modular foreign function interface for OCaml [71].

4 Task 4a: System APIs and Protocols

4.1 POSIX Filesystem Specification and Reasoning

We have pursued two directions here: operational modelling, validated by extensive testing against production filesystem implementations, and program-logic work oriented towards reasoning about filesystems.

Operational Modelling SibylFS is a semantics and filesystem test oracle for POSIX filesystems, developed in a collaboration between semantics and system researchers led by Ridge at Leicester [81]. SibylFS has been used to validate and identify bugs in existing production filesystems and in two verified filesystems (FSCQ and Flashix) from academia. The SibylFS infrastructure was used by Docker, following its acquisition of Unikernel Systems (a spin-out from Cambridge). The SibylFS model has been integrated into our Task 3 Cerberus C semantics, providing an integrated semantic model for C programs that use a filesystem. SibylFS is available under a permissive open-source licence.

SibylFS web page: `http://sibylfs.github.io/`
SibylFS github repo: `https://github.com/sibylfs/`

Program-logic reasoning In parallel, we have developed a separation-logic style program logic for a sequential specification of a core subset of POSIX file systems and for modular reasoning about client programs using file systems.

We developed a separation logic style program logic for path-addressable structured data, and applied this logic to develop a sequential specification of a core subset of POSIX file systems. This work demonstrated the modularity and scalability of reasoning about file-system client programs using simplified paths that allow updates to appear local [86]. We then developed fusion logic, a file-system specific program logic based on separation logic, for reasoning about non-local properties of arbitrary paths using symbolic links or dot-dots. We extended the previous sequential specification of core POSIX file systems to arbitrary paths. The extended specification and fusion logic were then used to reason about implementations of the `rm` utility, discovering bugs in popular implementations due to mishandling of symbolic links and dot-dots in paths [84].

We have extended concurrent program logics with reasoning about how programs are affected by, and recover from, host failures (crashes), as in file systems and databases [85]. This extends the Views framework, which acts as a basis for many concurrent program logics, with host failure and recovery semantics; we instantiated this with a Fault-tolerant Concurrent Separation Logic and used it to verify properties of an ARIES recovery algorithm used in databases.

4.2 TLS Security

In another collaboration between our systems and semantics groups, we have developed a clean-slate specification and implementation, nqsb-TLS, of the Transport Layer Security (TLS) protocol. This is fundamental to internet use, underlying https secure web access and ssh interaction, but its implementations have a history of security flaws [77, 76]. This is available under a permissive open-source licence. We engaged in the TLS 1.3 design process [75], providing extensive testing facilities for other implementations; and we presented an outreach talk at 31C3 [78].

We developed `libnqsb-tls1` [74], a drop-in replacement for the C `libtls` library that wraps our pure OCaml TLS implementation `ocaml-tls`. We have also demonstrated unmodified OpenBSD system software running against our replacement library.

nqsbTLS web page: `https://nqsb.io/`
nqsbTLS github repo: `https://github.com/mirleft/ocaml-tls`
libnqsb-tls1 github repo: `https://github.com/mirleft/libnqsb-tls/`.

4.3 TCP/IP

We have resurrected the NetSem network semantics from 2000–2009, which defined an executable-as-test-oracle specification for the TCP and UDP network protocols and their Sockets API, using HOL4. In collaboration with the FreeBSD project, and with the DARPA CADETS project at BAE/Cambridge/Memorial, we now collect traces from a current FreeBSD system using DTrace[1]; CADETS is extending DTrace to support this work. Exploiting DTrace lets us substantially simplify the instrumentation and to instrument in more places, allowing nondeterminism to be resolved early; together with Moore's-law advances in checking speed, this let us produce a more usable test oracle. We have published a J.ACM journal paper on the entire NetSem project, including this [80].

The model and infrastructure are available under a permissive open-source licence.

NetSem web page: https://www.cl.cam.ac.uk/users/pes20/Netsem/
NetSem github repo: https://github.com/PeterSewell/netsem.

5 Task 4b: Concurrency Reasoning

Here we focus on challenging problems of concurrent systems that include correctness properties, such as atomicity and termination, abstract specifications for concurrent modules and modular reasoning, and memory models. We have approached these problems from varying yet complementary directions, by developing new program logics models and type systems, algebraic reasoning, linearisability correctness conditions and proof techniques, reduction theories and automated verification tools.

5.1 Program logics, models, type systems

We have proposed a logic, iCAP-TSO, for reasoning about programs running on a TSO memory model. The logic supports direct reasoning about the buffering observable on TSO machines. It also supports a fiction of sequential consistency that hides this buffering behaviour from clients for libraries that contain sufficient synchronisation. This allows us to recover standard separation logic reasoning for well-synchronised clients and still support linking with code that requires explicit reasoning about buffering. [105]

We have proposed a refinement of the mathematical technique of step-indexing [102]. Step-indexing is used to break circularities that arises when constructing models of advanced program logics and type systems. Our refinement provides a more flexible way of breaking these circularities that increases compositionality when reasoning about program equivalence. We are currently investigating whether the same is true for program logics.

We have defined a relational semantics for an expressive type-and-effect system to validate effect-based program transformations. The type-and-effect system combines effect tracking with a lightweight mechanism for tracking state ownership. This combination is intended to simplify concurrent programming by allowing a compiler to automatically parallelise sub-computations based on inferred effect types. We use the semantics to prove that such automatic parallelisation is sound. [93]

We have studied abstract local reasoning for concurrent libraries [108]. There are two main approaches: provide a specification of a library by abstracting from concrete reasoning about an implementation; or provide a direct abstract library specification, justified by refining to an implementation. Both approaches have a significant gap in their reasoning, due to a mismatch between the abstract connectivity of the abstract data structures and the concrete connectivity of the concrete heap representations. We demonstrate this gap using structural separation logic (SSL) for specifying a concurrent tree library and concurrent abstract predicates (CAP) for reasoning about a concrete tree implementation. The gap between the abstract and concrete connectivity emerges as a mismatch between the SSL tree predicates and CAP heap predicates. This gap is closed by an interface

[1] https://wiki.freebsd.org/DTrace

function I which links the abstract and concrete connectivity. We generalise our SSL reasoning and results to arbitrary concurrent data libraries.

We show how to verify four challenging concurrent fine-grained graph-manipulating algorithms, including graph copy, a speculatively parallel Dijkstra, graph marking and spanning tree. We develop a reasoning method for such algorithms that dynamically tracks the contributions and responsibilities of each thread operating on a graph, even in cases of arbitrary recursive thread creation [99]. We demonstrate how to use a logic without abstraction (CoLoSL) to carry out abstract reasoning in the style of iCAP, by building the abstraction into the proof structure rather than incorporating it into the semantic model of the logic.

We have developed TaDA, a separation logic for fine-grained concurrency which introduces atomic triples for specifying the observable atomicity exhibited by operations of concurr ent modules. The atomic triples can be used to specify the atomicity of operations at the granularity of a particular data-abstraction, such as an abstract queue, and under restricted interf erence from the concurrent environment [109]. This approach leads to strong specifications of concurrent modules, such as locks, double-compare-and-swap, and concurrent queues, that do not depend on particular usage patterns.

We have demonstrated the scalability of TaDA reasoning by studying concurrent maps. In particular, we defined two abstract specifications for a concurrent map module, one focusing on the entire abstract map and the other on individual key-value pairs, which we proved to be equivalent by applying TaDA. We have applied the TaDA program logic to produce the first partial correctness proof of core operations of `ConcurrentSkipListMap` in `java.util.concurrent` [96].

We have extended TaDA with well-founded termination reasoning. This led to the development of Total-TaDA, the first program logic for reasoning about total correctness of fine-grained concurrent programs [101].

We have surveyed a range of verification techniques to specify concurrent programs, from Owicki-Gries to modern logics. We show how auxiliary state, interference abstraction, resource ownership, and atomicity are combined to provide powerful approaches to specify concurrent modules in CAP and TaDA [103].

A key difficulty in verifying shared-memory concurrent programs is reasoning compositionally about each thread in isolation. Existing verification techniques for fine-grained concurrency typically require reasoning about either the entire shared state or disjoint parts of the shared state, impeding compositionality. We have introduced the program logic CoLoSL, where each thread is verified with respect to its subjective view of the global shared state [106]. This subjective view describes only that part of the state accessed by the thread. Subjective views may arbitrarily overlap with each other, and expand and contract depending on the resource required by the thread. This flexibility gives rise to small specifications and, hence, more compositional reasoning for concurrent programs. We demonstrate our reasoning on a range of examples, including a concurrent computation of a spanning tree of a graph.

5.2 Algebraic reasoning

We formalise a modular hierarchy of algebras with domain and antidomain (domain complement) operations in Isabelle/HOL that ranges from modal semigroups to modal Kleene algebras and divergence Kleene algebras. We link these algebras with models of binary relations and program traces. We include some examples from modal logics, termination and program analysis [111].

We describe the application of Modal Kleene algebra in program correctness. Starting from a weakest precondition based component, we show how variants for Hoare logic, strongest postconditions and program refinement can be built in a principled way. Modularity of the approach is demonstrated by variants that capture program termination and recursion, memory models for programs with pointers, and trace semantics relevant to concurrency verification [112, 113].

We have summarised the progress in the research towards the construction of links between algebraic presentations of the principles of programming and the exploitation of concurrency in modern programming practice [110]. The research concentrates on the construction of a realistic family of partial order models for Concurrent Kleene Algebra (aka, the Laws of Programming). The main elements of the model are objects and the events in which they engage. Further primitive concepts are traces, errors and failures, and transferrable

ownership. In terms of these we can define other concepts which have proved useful in reasoning about concurrent programs, for example causal dependency and independence, sequentiality and concurrency, allocation and disposal, synchrony and asynchrony, sharing and locality, input and output.

5.3 Linearizability and reduction theory

We generalized Lipton's reduction theory for TSO programs. We developed a methodology with which one can check whether a given program allows non-SC behaviours and transform any given program P to another program P' such that P under TSO is observationally equivalent to P' under SC [104].

Linearisability of concurrent data structures is usually proved by monolithic simulation arguments relying on the identification of the so-called linearization points. Regrettably, such proofs, whether manual or automatic, are often complicated and scale poorly to advanced non-blocking concurrency patterns, such as helping and optimistic updates. In response, we propose a more modular way of checking linearisability of concurrent queue algorithms that does not involve identifying linearization points [107]. We reduce the task of proving linearisability with respect to the queue specification to establishing four basic properties, each of which can be proved independently by simpler arguments. As a demonstration of our approach, we verify the Herlihy and Wing queue, an algorithm that is challenging to verify by a simulation proof.

We developed local linearizability, a relaxed consistency condition that is applicable to container-type concurrent data structures like pools, queues, and stacks. While linearizability requires that the effect of each operation is observed by all threads at the same time, local linearizability only requires that for each thread T, the effects of its local insertion operations and the effects of those removal operations that remove values inserted by T are observed by all threads at the same time. We investigate theoretical and practical properties of local linearizability and its relationship to many existing consistency conditions. We present a generic implementation method for locally linearizable data structures that uses existing linearizable data structures as building blocks. Our implementations show performance and scalability improvements over the original building blocks and outperform the fastest existing container-type implementations [100]

5.4 Automated verification

We have developed Caper, a tool for automating poofs in concurrent separation logic with shared regions, aimed at proving functional correctness for fine-grained concurrent algo rithms. The tool supports a logic in the spirit of concurrent abstract predicates taking inspiration from recent developments such as TaDA [109]. Caper provides a foundation for exploring the possibilities for automation with such a logic [95].

We have created an axiomatic specification of a key fragment of the Document Object Model (DOM) API using structural separation logic. This specification allows us to develop modular reasoning about client programs that call the DOM [62].

5.5 Recent concurrency reasoning

More recently, we have published several papers on reasoning about different aspects of concurrency:

- Separation logic for Promising Semantics [90] [ESOP 2018]
- Algebraic laws for weak consistency [91] [CONCUR 2017]
- Verifying strong eventual consistency in distributed systems [92] [OOPSLA 2017, Distinguished Paper Award]
- Transactions with relaxed concurrency (largely non-REMS work) [89] [VMCAI 2018]
- Modular concurrent verification [88] [J.LAMP 2018]
- Concurrent specification of POSIX file systems [82] [ECOOP 2018]

Other

We have also published a few papers that do not fit exactly into the above task structure:

Verified trustworthy systems We have published an opinion piece [87] with the goal of raising awareness about the pressing issues surrounding reliability of complex modern software systems and the importance of specification and verification in the industrial design process. This publication is part of a broader journal publication on Verified Trustworthy Software Systems, aimed at the general public.

Confidentiality verification We have published a journal paper on earlier (largely non-REMS) work on formal verification of confidentiality in a social media system [114].

Programming and proving with classical types We have published the following on programming and proving with classical types, building on an MPhil dissertation of the first author, supervised by REMS staff [115].

6 People

6.1 Investigators

We very sadly lost Mike Gordon, who died suddenly in August 2017. Mike made an enormous contribution to REMS, which builds on much of his work. See `http://www.cl.cam.ac.uk/misc/obituaries/gordon/` for his obituary.

Neel Krishnaswami, who joined the Cambridge Computer Laboratory as a faculty member in 2016, is contributing substantially to REMS, now as an investigator.

There were various movements of staff between the original REMS proposal and the project start. Steve Hand moved from his position at the University of Cambridge to industry research positions in silicon valley, while we added Robert Watson and Anil Madhavapeddy, both of who were co-authors of the proposal and have since taken up faculty positions at Cambridge, as investigators.

Four other postdoctoral researchers who were co-authors of the proposal have now taken up faculty positions elsewhere Scott Owens at the University of Kent, Susmit Sarkar at the University of St Andrews, Mike Dodds at the University of York (since moved to Galois), and Magnus Myreen at Chalmers, Sweden.

6.2 Postdocs, PhD students, visitors, and interns

The research staff who have been funded by or otherwise substantially involved in REMS are listed below. This includes postdoctoral researchers, PhD students, visitors, interns, and MPhil and undergraduate project students. REMS alumni are marked with a star*.

Cambridge Alasdair Armstrong, Mark Batty*, Thomas Bauereiss, David Chisnall*, Shaked Flur, Anthony Fox*, Jon French, Kathryn E. Gray*, Victor Gomes, Chung-Kil Hur (SNU, South Korea) (sabbatical visit Jul 2017 – Feb 2018)*, Alexandre Joannou, Ohad Kammar*, Stephen Kell*, Gabriel Kerneis*, Ramana Kumar*, Stella Lau (Part III project and summer intern, summer 2018)*, James Lingard (MPhil project)*, Justus Matthiesen, Hannes Mehnert*, Kayvan Memarian, David Kaloper Mersinjak, Dominic Mulligan*, Matt Naylor*, Kyndylan Nienhuis, Robert Norton-Wright, Jean Pichon-Pharabod, Christopher Pulte, Linden Ralph (undergraduate intern, summer 2017)*, Mike Roe, Simon Ser (intern, summer 2018)*, Ali Sezgin*, David Sheets*, Ben Simner, Kasper Svendsen*, Thomas Tuerk*, Mark Wassell, Conrad Watt.

Imperial Andrea Cerone, Thomas Dinsdale-Young*, Emanuele D'Osualdo, Petar Maksimovic, Gian Ntzik, Azalea Raad*, Pedro Miguel da Rocha Pinto*, José Fragoso Santos, Julian Sutherland, Jules Villiard*, Adam Wright*, Thomas Wood*, Shale Xiong.

Edinburgh Brian Campbell.

7 Summary of REMS publications by calendar year

This shows the venues of the main REMS peer-reviewed publications in each year, together with (below the line) technical reports, presentations not associated with a paper, and suchlike.

2014	2015	2016	2017	2018	2019
Onward!	MICRO	OOPSLA	ASPLOS	POPL	POPL
ICFP	APLAS	OOPSLA	ESOP	POPL	POPL
FMICS	CCS	OOPSLA	ESOP	ESOP	POPL
ECOOP	Onward!	FM	POPL	CPP	POPL
MFPS	SOSP	APLAS	POPL	WWW Comp.	POPL
ESOP	OOPSLA	APLAS	CPP	VMCAI	ASPLOS
RAMiCS	MEMOCODE	FMCAD	ICCD	DLS	
PEPM	ITP	FMCAD	CONCUR	PPDP	
POPL	USENIX Security	IEEE Micro	CADE	J. ACM	
J. Funct. Program.	VSTTE	OCaml	OOPSLA	J. LAMP	
Batty PhD	MFPS	OCaml	APLAS	ECOOP	
31C3 talk $\times 2$	Security and Privacy	CONCUR	Sci. Comp. Prog	PLDI	
Clement MPRI	LSFA	PLDI	Onward!	Pichon PhD	
	ESOP	ESOP	J. Aut. Reas.	UCAM-CL-TR-907	
	ESOP	ESOP	Phil. Trans. A	ARW$\times 3$	
	ESOP	Sci. Comp. Prog.		WG14 $\times 6$	
	LMCS	POPL		35c3 $\times 2$	
	ASPLOS	POPL		EuroLLVM	
	ASPLOS	Ntzik PhD		GNU Cauldron	
	UCAM-CL-TR-876	da Rocha Pinto PhD		RISC-V ISA manual	
	UCAM-CL-TR-877	Raad PhD			
		AFP $\times 2$			
		WG14 $\times 4$			
		WG14 $\times 3$			
		TLS v1.3 TRON			
		TyDe			

8 All REMS publications

Note that this does not include our main released software and models, which are linked to above. It contains a few talk presentations that did not have associated papers. It is ordered by topic, following Fig. 1.

[1] ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. In **POPL 2019**. Proc. ACM Program. Lang. 3, POPL, Article 71. [project page]. [pdf].

[2] Lem: Reusable Engineering of Real-world Semantics. Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. In **ICFP 2014**. [pdf].

[3] Skeletal Semantics and their Interpretations. Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. In **POPL 2019**. [pdf].

[4] Improved Tool Support for Machine-Code Decompilation in HOL4. Anthony C. J. Fox. In **ITP 2015**. [pdf].

[5] A Dependent Type Theory with Abstractable Names. A. M. Pitts, J. Matthiesen, and J. Derikx. In **LSFA 2014**. [pdf].

[6] Running programming language specifications, Basile Clement, August 2014. MPRI report.

[7] Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. In **POPL 2018**. [project page]. [pdf].

[8] Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. In **POPL 2017**. [pdf].

[9] A Consistency Checker for Memory Subsystem Traces. M. Naylor, S. W. Moore, and A. Mujumdar. In **FMCAD 2016**. [pdf].

[10] Modelling the ARMv8 architecture, operationally: concurrency and ISA. Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. In **POPL 2016**. [pdf].

[11] GPU Concurrency: Weak Behaviours and Programming Assumptions. Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. In **ASPLOS 2015**. [url].

[12] An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. In **MICRO 2015**. [pdf].

[13] Rigorous Architectural Modelling for Production Multiprocessors, S. Flur, K. Gray, G. Kerneis, D. Mulligan, C. Pulte, S. Sarkar, and P. Sewell, May 2015. Presentation at HCSS 2015: the Fifteenth Annual High Confidence Software and Systems Conference.

[14] The RISC-V Instruction Set Manual Volume I: Unprivileged ISA. Andrew Waterman and Krste Asanović, editors. December 2018. Document Version 20181221-Public-Review-draft. Contributors: Arvind, Krste Asanović, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, Roger Espasa, Shaked Flur, Stefan Freudenberger, Jan Gray, Michael Hamburg, John Hauser, David Horner, Bruce Hoult, Alexandre Joannou, Olof Johansson, Ben Keller, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin Manerkar, Luc Maranget, Margaret Martonosi, Joseph Myers, Vijayanand Nagarajan, Rishiyur Nikhil, Jonas Oberhauser, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Christopher Pulte, Jose Renau, Colin Schmidt, Peter Sewell, Susmit Sarkar, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Andrew Waterman, Robert Watson, Derek Williams, Andrew Wright, Reinoud Zandijk, and Sizhuo Zhang. [pdf].

[15] Efficient Tagged Memory. Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. In **ICCD 2017**. [pdf].

[16] Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6). Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, Stacey Son, and Hongyan Xia. Technical Report UCAM-CL-TR-907, University of Cambridge, Computer Laboratory, April 2017. [pdf].

[17] CHERI JNI: Sinking the Java Security Model into the C. David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward

Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. In **ASPLOS 2017**. [url].

[18] Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. Robert N.M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. *IEEE Micro*, 36(5):38–49, September 2016. [pdf].

[19] A generic synthesisable test bench. Matthew Naylor and Simon W. Moore. In **MEMOCODE 2015**. [pdf].

[20] Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, and Stacey Son. Technical Report UCAM-CL-TR-876, University of Cambridge, Computer Laboratory, September 2015. [pdf].

[21] Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide. Robert N. M. Watson, David Chisnall, Brooks Davis, Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, and Jonathan Woodruff. Technical Report UCAM-CL-TR-877, University of Cambridge, Computer Laboratory, September 2015. [pdf].

[22] CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. In **Security and Privacy 2015**. [pdf].

[23] Beyond the PDP-11: Processor support for a memory-safe C abstract machine. David Chisnall, Colin Rothwell, Brooks Davis, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Peter G. Neumann, and Michael Roe. In **ASPLOS 2015**.

[24] Extracting Behaviour from an Executable Instruction Set Model. Brian Campbell and Ian Stark. In **FMCAD 2016**. Full proceedings `http://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD16/proceedings/fmcad-2016-proceedings.pdf`. [pdf].

[25] Randomised Testing of a Microprocessor Model Using SMT-Solver State Generation. Brian Campbell and Ian Stark. ***Sci. Comput. Program.***, 118:60–76, March 2016. [pdf].

[26] Randomised Testing of a Microprocessor Model Using SMT-Solver State Generation. Brian Campbell and Ian Stark. In **FMICS 2014**. [pdf].

[27] Into the depths of C: elaborating the de facto standards. Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N.M. Watson, and Peter Sewell. In **PLDI 2016**. PLDI 2016 Distinguished Paper award. [pdf].

[28] Exploring C Semantics and Pointer Provenance. Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. In **POPL 2019**. Proc. ACM Program. Lang. 3, POPL, Article 67. [project page]. [pdf].

[29] Clarifying the C memory object model, Kayvan Memarian and Peter Sewell. ISO SC22 WG14 N2012, March 2016. [html].

[30] C memory object and value semantics: the space of de facto and ISO standards, David Chisnall, Justus Matthiesen, Kayvan Memarian, Kyndylan Nienhuis, Peter Sewell, and Robert N. M. Watson. ISO SC22 WG14 N2013, March 2016. [pdf].

[31] What is C in practice? (Cerberus survey v2): Analysis of Responses, Kayvan Memarian and Peter Sewell. ISO SC22 WG14 N2014, March 2016. [html].

[32] What is C in practice? (Cerberus survey v2): Analysis of Responses – with Comments, Kayvan Memarian and Peter Sewell. ISO SC22 WG14 N2015, March 2016. [url].

[33] Clarifying Unspecified Values (Draft Defect Report or Proposal for C2x), Kayvan Memarian and Peter Sewell. ISO SC22 WG14 N2089, September 2016. [html].

[34] Clarifying Pointer Provenance (Draft Defect Report or Proposal for C2x), Kayvan Memarian and Peter Sewell. ISO SC22 WG14 N2090, September 2016. [html].

[35] Clarifying Trap Representations (Draft Defect Report or Proposal for C2x), Kayvan Memarian and Peter Sewell. ISO SC22 WG14 N2091, September 2016. [html].

[36] N2223: Clarifying the C Memory Object Model: Introduction to N2219 – N2222, Kayvan Memarian, Victor Gomes, and Peter Sewell. ISO SC22 WG14 N2223, March 2018. [project page]. [html].

[37] N2219: Clarifying Pointer Provenance (Q1-Q20) v3, Kayvan Memarian, Victor Gomes, and Peter Sewell. ISO SC22 WG14 N2219, March 2018. [project page]. [html].

[38] N2220: Clarifying Trap Representations (Q47) v3, Kayvan Memarian, Victor Gomes, and Peter Sewell. ISO SC22 WG14 N2220, March 2018. [project page]. [html].

[39] N2221: Clarifying Unspecified Values (Q48-Q59) v3, Kayvan Memarian, Victor Gomes, and Peter Sewell. ISO SC22 WG14 N2221, March 2018. [project page]. [html].

[40] N2222: Further Pointer Issues (Q21-Q46), Kayvan Memarian, Victor Gomes, and Peter Sewell. ISO SC22 WG14 N2222, March 2018. [project page]. [html].

[41] N2263 Sewell,Clarifying Pointer Provenance (Q1-Q20) v4, Kayvan Memarian, Victor Gomes, and Peter Sewell. ISO SC22 WG14 N2263, March 2018. [project page]. [html].

[42] Cerberus: towards an Executable Semantics for Sequential and Concurrent C11, Kayvan Memarian, Kyndylan Nienhuis, Justus Matthiesen, James Lingard, and Peter Sewell, May 2015. Presentation at HCSS 2015: the Fifteenth Annual High Confidence Software and Systems Conference.

[43] An operational semantics for C/C++11 concurrency. Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. In **OOPSLA 2016**. [pdf].

[44] A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. Jean Pichon-Pharabod and Peter Sewell. In **POPL 2016**. [pdf].

[45] The Problem of Programming Language Concurrency Semantics. Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. In **ESOP 2015**. [pdf].

[46] The C11 and C++11 Concurrency Model. Mark John Batty. **PhD thesis**, University of Cambridge Computer Laboratory, November 2014. Winner of 2015 ACM SIGPLAN John C. Reynolds Doctoral Dissertation Award and 2015 CPHC/BCS Distinguished Dissertation competition.

[47] A no-thin-air memory model for programming languages. Jean Pichon-Pharabod. **PhD thesis**, University of Cambridge, 2018.

[48] QEMU/CPC: static analysis and CPS conversion for safe, portable, and efficient coroutines. Gabriel Kerneis, Charlie Shepherd, and Stefan Hajnoczi. In **PEPM 2014**. [url].

[49] Efficient code generation for weakly ordered architectures, Reinoud Elhorst, Mark Batty, and David Chisnall. Presentation at the 4th European LLVM conference (EuroLLVM), April 2014. Slides and report available at `http://llvm.org/devmtg/2014-04/`.

[50] Lowering C11 Atomics for ARM in LLVM, Reinoud Elhorst, March 2014. [pdf].

[51] Some were meant for C: the endurance of an unmanageable language. Stephen Kell. In **Onward! 2017**. [pdf].

[52] Dynamically diagnosing type errors in unsafe code. Stephen Kell. In **OOPSLA 2016**. [pdf].

[53] In Search of Types. Stephen Kell. In **Onward! 2014**. [url].

[54] The missing link: explaining ELF static linking, semantically. Stephen Kell, Dominic P. Mulligan, and Peter Sewell. In **OOPSLA 2016**. [pdf].

[55] Towards a dynamic object model within Unix processes. Stephen Kell. In **Onward! 2015**. [pdf].

[56] Clean Application Compartmentalization with SOAAP. Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. In **CCS 2015**. [pdf].

[57] CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. In **ASPLOS 2019**. Best paper award. [pdf].

[58] Symbolic Execution for JavaScript. José Fragoso Santos, Petar Maksimovic, Théotime Grohens, Julian Dolby, and Philippa Gardner. In **PPDP 2018**. [url].

[59] JaVerT: JavaScript verification toolchain. José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. In **POPL 2018**. [pdf].

[60] JSExplain: a Double Debugger for JavaScript. Arthur Charguéraud, Alan Schmitt, and Thomas Wood. In **WWW**. [pdf].

[61] Towards Logic-Based Verification of JavaScript Programs. José Fragoso Santos, Philippa Gardner, Petar Maksimovic, and Daiva Naudziuniene. In **CADE 2017**. [url].

[62] DOM: Specification and Client Reasoning. Azalea Raad, José Fragoso Santos, and Philippa Gardner. In **APLAS 2016**. [url].

[63] JaVerT 2.0: Compositional Symbolic Execution for JavaScript. José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. In **POPL 2019**. [pdf].

[64] Verified compilation of CakeML to multiple machine-code targets. Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. In **CPP 2017**. [pdf].

[65] A New Verified Compiler Backend for CakeML. Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. In **ICFP 2016**. [pdf].

[66] CakeML: a verified implementation of ML. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. In **POPL 2014**. [url].

[67] Proof-producing translation of higher-order logic into pure and stateful ML. Magnus O. Myreen and Scott Owens. ***J. Funct. Program.***, 24(2-3):284–315, January 2014. [url].

[68] Mechanising and Verifying the WebAssembly Specification. Conrad Watt. In **CPP 2018**. [pdf].

[69] CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. In **POPL 2019**. [pdf].

[70] Bounding Data Races in Space and Time. Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. In **PLDI 2018**. [pdf].

[71] A modular foreign function interface. Jeremy Yallop, David Sheets, and Anil Madhavapeddy. ***Science of Computer Programming***, April 2017. [pdf].

[72] Generic Partially-static Data (Extended Abstract). David Kaloper-Meršinjak and Jeremy Yallop. In **TyDe 2016**. [pdf].

[73] Conex — establishing trust into data repositories. Hannes Mehnert and Louis Gesbert. In **OCaml 2016**. [url].

[74] OCaml inside: a drop-in replacement for libtls (extended abstract). Enguerrand Decorne, Jeremy Yallop, and David Meršinjak. In **OCaml 2016**. [url].

[75] Not-quite-so-broken TLS 1.3 mechanised conformance checking. David Kaloper-Meršinjak and Hannes Mehnert. In **TRON workshop 2016**. [pdf].

[76] Not-Quite-So-Broken TLS: Lessons in Re-Engineering a Security Protocol Specification and Implementation. David Kaloper-Mersinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. In **USENIX Security 2015**. [pdf].

[77] Transport Layer Security purely in OCaml. Hannes Mehnert and David Kaloper Mersinjak. In **OCaml 2014**. [url].

[78] Trustworthy secure modular operating system engineering, David Kaloper Meršinjak and Hannes Mehnert. Invited talk at 31st Chaos Communication Congress (31C3), January 2015.

[79] Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation, David Kaloper Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell, May 2015. Presentation at HCSS 2015: the Fifteenth Annual High Confidence Software and Systems Conference.

[80] Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API. Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. ***J. ACM***, 66(1):1:1–1:77, December 2018. [project page]. [pdf].

[81] SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. In **SOSP 2015**. [pdf].

[82] A Concurrent Specification of POSIX File Systems. Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. In **ECOOP 2018**. [pdf].

[83] Reasoning About POSIX File Systems. Gian Ntzik. **PhD thesis**, Imperial College London, September 2016. [url].

[84] Reasoning about the POSIX File System: Local Update and Global Pathnames. Gian Ntzik and Philippa Gardner. In **OOPSLA 2015**.

[85] Fault-tolerant Resource Reasoning. Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. In **APLAS 2015**. [pdf].

[86] Local Reasoning for the POSIX File System. Philippa Gardner, Gian Ntzik, and Adam Wright. In **ESOP 2014**. [url].

[87] Verified trustworthy software systems. Philippa Gardner. ***Philosophical Transactions of the Royal Society of London A***, 375(2104), September 2017. [pdf].

[88] A Perspective on Specifying and Verifying Concurrent Modules. Thomas Dinsdale-Young, Pedro da Rocha Pinto, and Philippa Gardner. ***Journal of Logical and Algebraic Methods in Programming***, 98:1–25, August 2018. [pdf].

[89] On abstraction and compositionality for weak-memory linearisability. Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. In **VMCAI 2018**. [url].

[90] A separation logic for a promising semantics. Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. In **ESOP 2018**. [pdf].

[91] Algebraic Laws for Weak Consistency. Andrea Cerone, Alexey Gotsman, and Hongseok Yang. In **CONCUR 2017**. [url].

[92] Verifying Strong Eventual Consistency in Distributed Systems. Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. In **OOPSLA 2017**. Distinguished Paper award. [url].

[93] A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic. Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. In **POPL 2017**. [url].

[94] Trace Properties from Separation Logic Specifications. Lars Birkedal, Thomas Dinsdale-Young, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. *CoRR*, abs/1702.02972, February 2017. [pdf].

[95] Caper: Automatic Verification for Fine-grained Concurrency. Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. In **ESOP 2017**. [pdf].

[96] Abstract Specifications for Concurrent Maps. Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. In **ESOP 2017**. [pdf].

[97] Reasoning with Time and Data Abstractions. Pedro da Rocha Pinto. **PhD thesis**, Imperial College London, September 2016. [url].

[98] Abstraction, Refinement and Concurrent Reasoning. Azalea Raad. **PhD thesis**, Imperial College London, September 2016. [url].

[99] Verifying Concurrent Graph Algorithms. Azalea Raad, Aquinas Hobor, Jules Villard, and Philippa Gardner. In **APLAS 2016**. [url].

[100] Local Linearizability for Concurrent Container-Type Data Structures. Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. In **CONCUR 2016**. [pdf].

[101] Modular Termination Verification for Non-blocking Concurrency. Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. In **ESOP 2016**. [project page]. [pdf].

[102] Transfinite Step-indexing: Decoupling Concrete and Logical Steps. Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. In **ESOP 2016**. [pdf].

[103] Steps in Modular Specifications for Concurrent Modules (Invited Tutorial Paper). Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. In **MFPS 2015**. [pdf].

[104] Moving Around: Lipton's Reduction for TSO - (Regular Submission). Ali Sezgin and Serdar Tasiran. In **VSTTE 2015**. [url].

[105] A Separation Logic for Fictional Sequential Consistency. Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. In **ESOP 2015**.

[106] CoLoSL: Concurrent Local Subjective Logic. Azalea Raad, Jules Villard, and Philippa Gardner. In **ESOP 2015**. [pdf].

[107] Aspect-oriented linearizability proofs. Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. ***Logical Methods in Computer Science***, 11(1):1–33, April 2015. [url].

[108] Abstract Local Reasoning for Concurrent Libraries: Mind the Gap. Philippa Gardner, Azalea Raad, Mark J. Wheelhouse, and Adam Wright. ***MFPS 2014****: Electr. Notes Theor. Comput. Sci.*, 308:147–166, June 2014. [url].

[109] TaDA: A Logic for Time and Data Abstraction. Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. In **ECOOP 2014**. [url].

[110] Developments in Concurrent Kleene Algebra. Tony Hoare, Stephan van Staden, Bernhard Möller, Georg Struth, Jules Villard, Huibiao Zhu, and Peter W. O'Hearn. In **RAMiCS 2014**. [url].

[111] Kleene Algebras with Domain. Victor B. F. Gomes, Walter Guttmann, Peter Höfner, Georg Struth, and Tjark Weber. *Archive of Formal Proofs*, 2016, April 2016. [url].

[112] Modal Kleene algebra applied to program correctness. Victor B. F. Gomes and Georg Struth. In **FM 2016**. [pdf].

[113] Program Construction and Verification Components Based on Kleene Algebra. Victor B. F. Gomes and Georg Struth. *Archive of Formal Proofs*, 2016, June 2016. [url].

[114] CoSMed: A Confidentiality-Verified Social Media Platform. Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. ***J. Automated Reasoning***, December 2017. [url].

[115] Programming and proving with classical types. Cristina Matache, Victor B. F. Gomes, and Dominic P. Mulligan. In **APLAS 2017**. [url].