CHERI Formal Methods Report

Version 3.0, Interim Draft

Peter G. Neumann*, Robert N. M. Watson†, Simon W. Moore†
Nirav Davé*, Alexandre Joannou†, Matthew Naylor†,
Michael Roe†, Anthony Fox†, and Jonathan Woodruff†

*SRI International, †University of Cambridge

January 18, 2016

Approved for public release; distribution is unlimited. Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense. Additional support was received from the EPSRC REMS Programme Grant (EP/K008528/1).

Contents

Abstract

This document describes strategies suitable for employing formal methods in the design of SRI International and University of Cambridge's Capability Hardware Enhanced RISC Instructions (CHERI) Instruction-Set Architecture (ISA), its total-system architecture, and its hardware-software implementation. The document has evolved from capturing our thoughts on what seemed desirable early in the research and development cycle into what has actually been accomplished during the course of the main five-year project, and what might be done in the future.

We discuss the overall design and its desired assurance properties, known weaknesses, and some mitigations provided by our approach. We present an initial formal consideration of the CHERI ISA, and automated test-case generation capability to check the formal model against its Bluespec SystemVerilog (BSV) implementation. We also describe the application of formal methods to link design goals and a BSV-based prototype of the CHERI processor, requiring the development of new tools to support the formal validation of BSV-based designs. We conclude with a summary of subsequent analyses that remain to be done. Although the desired formal analyses of the CHERI ISA, hardware, and low-level software could not be completed within the scope, time-scale, and funding of this project, we hope to be able to subsequently pursue the detailed analyses in the future under other funding alternatives.

Acknowledgments

The authors of this report thank other members of the CTSRD team, and our past and current research collaborators at SRI and Cambridge:

Ross J. Anderson	Jonathan Anderson	Ruslan Bukin	Gregory Chadwick
David Chisnall	Brooks Davis	Anthony Fox	Paul J. Fox
Khilan Gudka	Jong Hun Han	Alex Horsman	Asif Khan
Myron King	Chris Kitching	Wojciech Koszek	Ben Laurie
Patrick Lincoln	Anil Madhavapeddy	Ilias Marinos	A. Theodore Markettos
Ed Maste	Andrew Moore	Will Morland	Alan Mujumdar
Prashanth Mundkur	Steven J. Murdoch	Robert Norton	Philip Paeps
Alex Richardson	Colin Rothwell	John Rushby	Hassen Saidi
Hans Petter Selasky	Peter Sewell	Muhammad Shahbaz	Natarajan Shankar
Stacey Son	Andrew Turner	Richard Uhler	Munraj Vadera
Philip Withnall	Bjoern A. Zeeb		

Ben Laurie has been at Google throughout our project. We are very grateful for his continuing participation. The work of Nirav Davé reported here was done entirely while he was at SRI, before he joined Google on 2 November 2015.

The CTSRD team also thanks past and current members of its external oversight group for significant support and contributions:

Lee Badger	Simon Cooper	Rance DeLong	Jeremy Epstein
Virgil Gligor	Li Gong	Mike Gordon	Steven Hand
Andrew Herbert	Warren A. Hunt Jr.	Doug Maughan	Greg Morrisett
Brian Randell	Kenneth F. Shotting	Joe Stoy	Tom Van Vleck
Samuel M. Weber			

We would also like to acknowledge the late David Wheeler and Paul Karger, whose conversations with the authors about the CAP computer and capability systems contributed to our thinking on CHERI.

Finally, we are grateful to Howie Shrobe, MIT professor and past DARPA CRASH program manager, who has offered both technical insight and support throughout this work. We are also grateful to Robert Laddaga and Stu Wagner, who succeeded Howie in overseeing the CRASH program, and to Daniel Adams and Laurisa Goergen, SETAs supporting the program.

Chapter 1

Introduction

Capability Hardware Enhanced RISC Instructions (CHERI) extends the commodity 64-bit MIPS Instruction-Set Architecture (ISA) with new security primitives in order to allow software to efficiently implement *fine-grained memory protection* and an *object-capability security model* within virtual address spaces. CHERI's extensions are intended to support incrementally adoptable, high-performance, formally supported, and programmer-friendly underpinnings for *vulnerability mitigation* and fine-grained *software compartmentalization* motivated by the principle of least privilege. CHERI is a *hybrid capability architecture* in that gradual deployment of CHERI features in existing software is possible, offering a more gentle technology adoption path. Detailed information on the CHERI architecture and its software may be found in the *CHERI Instruction-Set Architecture* [52] and the *CHERI Programmer's Guide* [51].

Throughout this project, we have applied formal methodology in order to avoid system vulnerabilities – whether with respect to the internal consistency of the ISA, the syntactic and semantic soundness of our BSV specifications, and in linkages between software and hardware designs. We have concretely (and judiciously) applied formal methodology in several areas:

- We formally modeled the CHERI (and underlying MIPS) ISA in order to reason about its internal consistency and expressivity properties.
- We have linked the formal ISA model to our hardware implentations via co-simulation (of an executable form of the model) and also automated test generation from the model.
- We have utilized existing formal models and formally generated tests to validate our coherent multicore cahe. We have also used these models are the starting for novel in-hardware fuzzing and automated test-case reduction.
- We have developed new techniques to that we hope to use in formally linking the CHERI design and actual implementation of the hardware prototype in the BSV Hardware Description Language (HDL).
- We have begun early work to model the impact of CHERI enforcement on the semantics of the C programming language.
- We have laid the groundwork for future proofs of key software Trusted Computing Bases (TCBs) such as our software object-capabilty exception handler, responsible for domain transition, memory allocator, and other low-level software constructs.

This report presents formal considerations in the design and implementation of the CHERI ISA and CHERI prototype. We drew on formal methodology wherever feasible to improve our confidence in the design and implementation of CHERI. This use was necessarily subject to real-world constraints of timeline, budget, design process, and prototyping, but was to help ensure that we avoid creating a system that cannot meet our functional and security requirements. This goal required us to not only perform research into CPU and software design, but also to develop new formal methodology. In particular, we describe the development of a formal approach to test-case generation to support the validation of the Bluespec SystemVerilog (BSV) implementation of CHERI, and a general approach to the validation of designs in BSV.

We have also found that formal modeling tools, such as those we have used in instruction-set modeling, but also for memory models, have proven invaluable in all phases of our implementation effort, vastly improving the effectiveness of our small research team. These modeling techniques have also been substantially enriched through their application to CHERI – for example, in improvements to the maturity of the L3 modeling language to capture non-userspace ISA functionality such as exceptions and the Memory Management Unit (MMU). These improvements have proven sufficient to boot and run the CheriBSD operating system, test tools, and applications on an executable version of the model, allowing both validation of the model and of the software stack running over it. Memory coherency models for multicore designs have also proven important: we have used (and extended) the Litmus ISA-level memory-model tests [1], and also applied the underlying models pragmatically through in-hardware fuzzing and test-case reduction via our Bluecheck system [34].

This document serves as a report on the formal techniques that can be applied to our implementation process.

1.1 Version history

This is the fourth version of the *CHERI Formal Methods Report*.

1.0 An initial version of the CHERI Formal Methods Report proposing various goals in applying formal methods in the CTSRD project. We consider in some detail the opportunities for applying formal methods, and its implications for the instruction set and architecture. An initial Z specification of our ISA is provided, and a model for verifying BSV architectural refinement is proposed.

1.1 Added formal definitions for CCall, CReturn, CLoadDoubleWord, CStoreDoubleWord, and CPU reset state.

1.2 Update formal specifications for CHERI instructions to reflect changes in version 1.4 of the *CHERI Architecture Document*.

2.0 Significant extensions to the report including new chapters on automatic architectural extraction from a Bluespec description of a processor, automatic generation of ISA tests using a model-checking-based representation of the ISA, and on modeling the MIPS and CHERI ISAs using Cambridge's L3 ISA modeling language and its application to trace validation.

3.0 A new chapter has been added on automated testing techniques used with the CHERI processor prototype: Cambridge's ISA-level Litmus test suite used to validate memory consistency; the Axe in-hardware memory-consistency checker deployed directly on the memory subsystem; the

Bluecheck in-hardware fuzz tester and test-case reduction system; and use of Brian Cambell's L3-based automatic ISA test-suite generation tools with the MIPS and CHERI L3 ISA models.

We now describe the L3 ISA models for MIPS and CHERI in greater detail, and describe use of the L3 models to validate test-suite coverage.

The *Future Work* chapter has been substantially extended.

The sample ISA-level properties have been updated to the most recent version of the CHERI ISA.

1.2 Document Structure

This document is an introduction to our goals, methods, and results in verifying the CHERI CPU architecture. It is best interpreted in the context of the *CHERI Instruction-Set Architecture*, *CHERI Programmer's Guide*, and *BERI Hardware Reference* – separately distributed documents that provide the context, ISA specification, and a description of our current hardware and software prototypes.

Chapter 1 introduces our overall goals in verifying aspects of CHERI, including the CHERI ISA and prototype implementation, mapping our BSV prototype implementation into formal tools, and developing automated formal support for the verification and validation of BSV-based designs.

Chapter 2 considers the set of properties that need to be rigorously defined and evaluated with respect to capability instructions and capability registers.

Chapter 3 considers some primary characteristic security and reliability weaknesses and how the CHERI system architecture, hardware/software implementation, and other relevant defenses are expected to prevent or mitigate those common weaknesses.

Chapter 4 describes a model of the MIPS ISA and the CHERI extensions in the L3 language.

Chapter 5 describes an initial test-case generation technique for CHERI ISA, using the SAL model checker.

Chapter 6 describes the problem of extracting an architectural design specification from a microarchitectural specification, a key technique we use to simplify the microarchitectural verification. We consider the issues of modularity and the complications associated with the verification of the CHERI design.

Chapter 7 discusses the encoding of the Guarded Atomic Action based semantics of Bluespec SystemVerilog into the lambda calculus to provide a natural translation to formal semantics analysis tools.

Chapter 8 discusses the Smten language and implementation. This language is used to express and execute the SMT queries required in the implementation of the architectural extraction described in Chapter 6.

Chapter 9 describes the automated testing techniques using executable specifications that are currently being used for CHERI.

Chapter 10 discusses future directions for our work, including carrying out the formal analyses of the CHERI hardware BSV prototype implementation as the system evolves under the continuing two-year extension of CTSRD for CHERI hardware and software technology transfer.

Chapter 2

Assurance Properties

2.1 System Assurance Considerations

We are seeking to provide a formally-based total-system holistic layered analysis of how the overall CTSRD system architecture can prevent or overcome many of the classical vulnerabilities that have been the causes of potential compromises in past systems. The layering of the analysis is important because each of many different vulnerabilities may exist at multiple layers of the architecture, and thus different exploits can arise at multiple layers of abstraction. In addition, many vulnerabilities tend to arise as a result of careless horizontal or vertical composition of different modules at any particular layer and composition of different layers [35, 49, 50], all of which require further analysis.

In this chapter we consider some properties that need to be rigorously defined and evaluated with respect to the capability instructions and capability registers.

In Chapter 3, we examine some characteristic weaknesses of existing systems and how the evolving CTSRD system expects to be able to combat them.

2.2 Hardware/Software Properties (Static/Dynamic)

2.2.1 Hardware Properties of Capability Instructions

This subsection suggests some properties of the capability instructions and capability registers defined in the *CHERI Instruction-Set Architecture* [52]. For each of the instructions listed there, it is desirable to show that desired properties are satisfied by the instruction specification and that the implementation is consistent with the specification.

(NOTE: This is a preliminary outline, primarily for illustrative purposes at this point in time — in that the augmented instruction set is still evolving. See the *BERI Hardware Reference* [53] for an overview of the hardware design, and the *CHERI Instruction-Set Architecture* for details of the instructions.)

CLC, CLLC Load (or Load Linked) Capability Register

- atomicity of capability and tag loads, tag check, bounds checks, permission checks, alignment checks
- exception handling

- inductive preservation of capability monotonicity

CSC, CSCC Store (or Store Conditional) Capability Register

- atomicity of capability and tag store, tag check, bounds checks, permission checks, alignment checks
- exception handling
- inductive preservation of capability monotonicity

CLW, CLLW Load (or Load Linked) Byte/HalfWord/Word/Double (CL[BHWD][U], CLL[BHWD][U])

- tag check, bounds checks, permission checks, alignment checks
- exception handling

CSW, CSCW Store (or Store Conditional) Byte/HalfWord/Word/Double (CS[BHWD][U], CSC[BHWD][U])

- atomicity of tag clearing, tag check, bounds checks, permission checks, alignment checks
- exception handling
- inductive preservation of capability monotonicity

CMove, CFromPtr, CIncOffset, CSetOffset Move Capability Register or Create Derived Capability with Identical Rights

- atomicity of the move including preservation of tag
- equivalence of rights
- in the presence of imprecision in which offset manipulation may clear the tag due to unrepresentable bounds, appropriate subsetting of rights
- inductive preservation of capability monotonicity

CJR Jump Capability Register

- tag check, bounds checks, permission checks, alignment checks
- exception handling

CJALR Jump and Link Capability Register

- tag check, bounds checks, permission checks, alignment checks
- atomicity and correctness in preserving the program-counter capability
- exception handling
- inductive preservation of capability monotonicity

CSeal Create an Sealed Capability from a Code or Data Capability

- tag checks, bounds checks, permission checks
- exception handling
- inductive preservation of capability monotonicity

CUnseal Create a Code or Data Capability from an Sealed Capability

- tag and seal checks, type checks, permission checks
- atomicity and correctness in deriving an unsealed capability
- exception handling
- inductive preservation of capability monotonicity

CCall, CReturn Secure domain transition modeled on call or return

- in-hardware vs. software implementation of tag and seal checks, type checks, permission checks, etc.
- exception handling
- inductive preservation of capability monotonicity
- support for higher-level nonforgeability, nonbypassability, nonalterability, atomic operation, indivisibility, integer of tags and basic types, ...
- hardware-enforced basic capability type enforcement (as opposed to software enforced extended types),
- controlled violation of inductive preservation of capability monotonicity

CAndPerm, CSetBounds, CSetBoundsExact Restrict Permissions or Bounds of Capability

- atomicity of the move including preservation of tag
- equivalence or appropriate subsetting of rights
- inductive preservation of capability monotonicity

Exception handling Safe transition of control when a processor exception fires

- (Not a capability instruction, but rather, a collective set of CPU and inductive behaviors relating to exception delivery)
- On entry, preservation of the program-counter capability in the exception program-counter capability
- On entry, installation of a suitable exception-vector capability in the program-counter capability
- On return, restoration of the program-counter capability from the exception program-counter capability
- In software, the safe transition to the privileged exception-handling domain, preservation and restoration of user or kernel general-purpose and capability registers, and the non-leakage of data or capabilities

Delete capability Overwrite a capability in a register or memory

- (Not a capability instruction, but rather, a collective set of inductive behaviors and memory-subsystem behaviors)
- Appropriate non-preservation of tags during capability register move or jump instructions

- Appropriate clearing of tags in memory during general-purpose register writes, or capability register writes without tags
- Appropriate non-preservation of tags during capability load from memory whose tag is clear, or that cannot hold tags
- Higher-level software object-capability properties, such as non-delegation of capabilities as required, and garbage collection if implemented

CPU reset Reset state of the CPU

- (Not a capability instruction, but rather, a collective set of behaviors on hard or soft reset)
- Initialization of capability registers with suitable initial ambient values
- Opportunity in the firmware or OS to configure tags for memory, and clear them prior to initial use

Inconsistencies that can arise from multiple capabilities for the same object will need to be resolved in software. However, some additional instructions that have not yet been specified may be helpful as well.

Formal analyses must also deal with certain aspects that are not fully modeled. For example, the soundness of the trap handler is fundamental to the security of the entire system, and flaws in its specification or bugs in its implementation could most likely be exploited.

Software-defined extended-type objects are considered in the following Section 2.2.2, along with other hardware-related software properties.

2.2.2 Hardware-Related Software Properties

This subsection considers some low-level software properties that need to be related to the capability hardware. Although not specifically hardware properties, it is the relationship between the hardware properties and the low-level software that will be important to model and analyze. Thus, this section is included here for the time being.

We refer to the CTSRD Architecture overview [37] and our paper on TESLA [2] for an initial view of how this all fits together.

- Integrity of extended type managers for extended type creation and capability invocation
- Type enforcement for extended-type capability operations
- Access control enforcement for extended-type capability operations
- Separation kernel properties relating specifically to capabilities
- TESLA mechanism: sound specification, e.g., terminating, deadlock free
- TESLA implementation: consistency with TESLA specification

2.3 Use of Formal Methods

The CHERI architecture provides a plethora of natural places for formal analysis: the Instruction-Set Architecture (ISA) defines an interface between hardware and software that defines both strong expectations for the hardware implementation, and a foundation on which software security properties must rest. We therefore take a strong interest in characterizing the ISA formally, using it as a foundation for proofs about software security, and linking it to the implementations in our hardware prototypes.

Our current efforts in understanding the ISA specification effort involve creating a model of the CHERI ISA specification and checking it formally. This initially was done in the Z representation, then ported to SAL to be mechanically tested by the SAL model checker, and finally to L3 – to both act as a "gold model" of the ISA, and also to verify that the implementation of the CHERI system corresponds to the specification. In the future, we hope to use the L3 ISA model to verify expressiveness properties of the ISA itself, and also as a foundation to verify software implemented using the ISA.

With respect to the hardware implementation, the parametric structuring of the Bluespec SystemVerilog (BSV) design provides a precise higher-level description of the implementations that we can use with any of the SRI formal verification tool suite (e.g. PVS, SAL, Yices2, Smten) integrated together within the Evidential Tool Bus (ETB) [20].

Initially, SAL and Yices2 are being included in the BSV development tool chain, with an intermediate lambda calculus representation providing an effective bridge to the use of those tools. In particular, we are interested in the analysis of BSV-based specifications and the consistency of its refinements into Verilog and the generated FPGA designs, as well as checking the critical properties of the capability-aware separation kernel as well as the compiler extensions. To this end, we hope to link our L3 ISA model with its practical pipeline implementation captured in Bluespec.

We have also explored other pragmatic verification techniques with formal grounding; for example, as described in Chapter 9, we are able to compile formal models for memory consistency into synthesizeable checkers that execute in FPGA for the purposes of high-performance unit testing using a tool called BlueCheck.

Chapter 3

Systemic Weaknesses and Their Remediation

This chapter considers the main characteristics of security/reliability weaknesses and what CT-SRD's system architecture, hardware/software implementation, and other relevant defenses are expected do to prevent or mitigate those common weaknesses.

3.1 Systemic Weaknesses

The taxonomy in this section is freely adapted from the ISI RISOS project reports in the mid-1970s – Bisbey, Carlstedt, Hollingworth, Popek, et al. ([8, 6, 7, 13, 12, 25]), the Neumann-Parker paper on misuse techniques ([36]), the 2010 Common Weakness Enumeration CWE SANS Top 25 Most Dangerous Programming Errors (http://www.sans.org/top25-software-errors/), our experience with design flaws and requirement errors, and more. It represents somewhat of a logical outside-in ordering (rather than attempting to capture the current frequency of exploitations, as is done in the list of the CWE Top 25) in which each set of weaknesses is applied to the different architectural layers. (The CWE list includes various software errors in categories such as insecure interaction between components, risky resource management, and porous defenses.)

For conceptual simplicity, the weaknesses have been organized into the following categories, which are expanded in the following section:

- Authentication Issues
- Authorization Issues
- Input Validation Issues
- Accountability Issues
- Cryptographic Issues
- Other Logic Errors
- Malware Issues (cross-cutting)

Some of the vulnerabilities are interdependent – that is, in some way related to one another. Also, exploitation of one can affect the exploitability of others. Furthermore, attacks may exploit multiple vulnerabilities. Some of the vulnerabilities can be triggered accidentally or intentionally, and thus can affect both security and reliability. Malware is in some sense cross-cutting, because it can involve combinations of the other issues. However, it is included here because of its pervasive potential for exploitation.

It is extremely unwise to try to place the blame in just one area. Exploits and accidental misuses often result from some combination of factors such as inadequate requirements, weak architectures, poor implementations, faulty modeling and faulty analysis, human frailty, willful malfeasance, bad management, misguided institutional cultures, and so on. Therefore, it is unwise to put all your eggs in one basket in attempting to confront a specific vulnerability. Today, everything is a potential weak link. In contrast, we seek to provide a layered architecture in which the strengths at each layer constructively enhance trustworthiness at higher layers.

Firm foundations are not sufficient for secure use: they go hand-in-hand with principled application construction, appropriate operation procedures, etc. However, they are required – no amount of best practice in operational security can make up for poor foundations.

3.2 Remediating Systemic Weaknesses

More generally: we are interested in providing a stronger computing substrate for TCBs – trustworthy computing bases – which have historically been large, critical, and vulnerable. TCBs will necessarily remain critical, but we will provide better tools for compartmentalization (disaggregation) and improve resistance to vulnerabilities, while improving programmability and performance. These tools will also work at all layers of the system, addressing the practical reality that security must be considered over the integrated whole: applications are part of the TCB with respect to the services they offer, and need the same fundamental primitives to operate reliably and securely that historic TCBs (kernels, language runtimes) have.

Figure 3.1 summarizes the hybrid system architecture. The following table identifies the constituent defensive elements, with three-letter mnemonics for easy reference.

Hierarchical layers or functions of hardware and software, from bottom to top

CHE CHERI hardware capability architecture

SEP Separation Kernel / VM Hypervisor to utilize CHE

TES TESLA

APP CTSRD Application or Applications (higher-layer software)

Development tools

VER Verilog, language of low-level specification of hardware, compiled into FPGA hardware simulations with Altera tools

BLU BSV, language and tools for high-level abstract structured hardware specification, compiled into structured Verilog by BSV tools

PrL Capability-aware programming language extensions to effectively utilize CHE

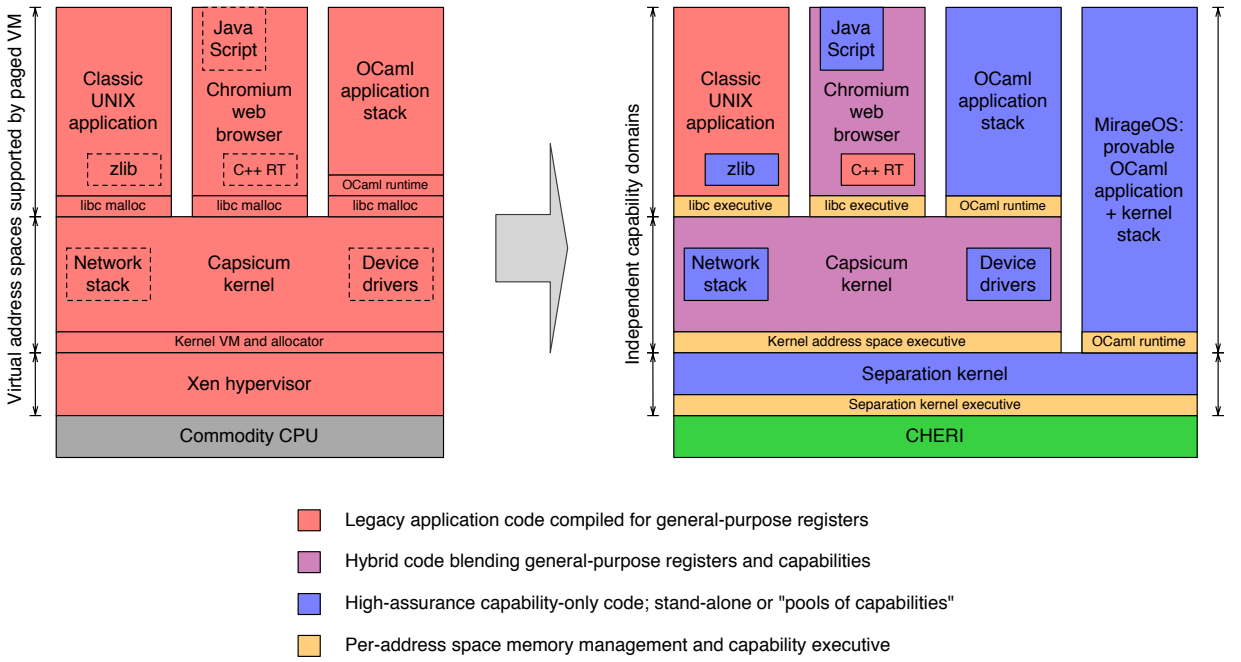

Figure 3.1: CHERI's hybrid capability architecture: initially, legacy software components execute without capability awareness, but security-sensitive TCB elements or particularly risky code bases are converted. In the long term, all packages are converted, implementing least privilege throughout the system.

COM Capability-aware compiler extensions to accommodate PrL compilations into capability-based CHE code executing within SEP.

Overarching concepts

PRI Principles such as Saltzer-Schroeder-Kaashoek

SWE Additional software engineering practices to be enforced

Many of the statements below need to be formalized in terms of BSV constructs for the hardware architecture CHE and other formal and semi-formal constructs for SEP, TESLA, PrL, COM. Further analysis can determine how PRI and SWE precepts are adhered to. In addition, TESLA can determine whether certain properties are satisfied – e.g., in the implementation, at APP installation, system configuration, or at runtime.

We have focused on integrating the formal and semi-formal desiderata for the hardware (CHERI) and system low-level software (separation kernel / hypervisor and TESLA) with this taxonomy. For example, the linkage between the BSV hardware specifications and our formal methods is expected to demonstrate some of the properties suggested below. In addition, the linkage between TESLA and our formal methods itself provides some interesting opportunities. TESLA will also provide us with some useful sources of information for compiler extensions within CTSRD and for research in dynamic system adaptability of our system, possibly by other CRASH performers as well as within CTSRD.

The subsequent text elaborates on illustrative types of weaknesses and for each type summarizes what CTSRD can contribute for each set of weaknesses.

3.2.1 Authentication

Authentication issues:

- Lack of user authentication
- Lack of component authentication
- Lack of system authentication
- Man in the middle attacks
- ToCtToU (Time of Check to Time of Use) problems
- Weak credential management

Authentication of course means different things to different people. It is relevant to purported user identities, but is also relevant internally to components, and externally to remote systems. It even includes communications and control among components even within a single application. That is not authentication in the usual sense (e.g., 'we share a secret'), but instead is derived from access control on namespaces and the authority to communicate in various ways. It also relates closely to type enforcement – for example, in the PSOS sense [38], in which every object in the system has a type strictly enforced by the hardware and software.

Authentication is primarily not just a hardware issue, although cryptography in the CHE hardware (and possibly crypto coprocessors) could help provide spoof-resistant logically unforgeable cryptographically based authentication. However, high assurance is required in the software for authentication and its system embedding – especially including the embedding of the crypto. PRI/SWE are essential here, and careful embedding of crypto and related software into CHE/SEP, particularly to resist insider misuse and spoofing. The underlying authentication mechanisms will all be part of the trustworthy capability-based core software, and the absence of access to those capabilities is designed to effectively prevent unauthorized access to those mechanisms and their data.

The issue of program compartmentalization is important here (as well as for authorization, below), allowing us to approximate least privilege at a far finer granularity than current commodity systems (or even research systems). Considering program structure and policies, this relates to the ability to map policies into tangible expression through protection and capabilities.

3.2.2 Authorization

Authorization issues:

- Access control permissions and privileges
- One-size-fits-all privileges (e.g., System High)
- Other excessive user privileges, including privileges to change other privileges

- Omnipotent root privileges and their subversion
- Reliance on untrustworthy third parties/clouds
- ToCtToU problems in access controls

CHE The capability mechanisms will provide strong access controls. Implicit hierarchical layering (a la Multics rings) can reduce root compromises by less trustworthy software components – that is, with a Biba-like lattice ordering of integrity.

SEP The separation kernel will provide controlled sharing among different virtual machine partitions, and otherwise strict isolation. The implicit hierarchical layering afforded by the capability mechanism sensibly used in the separation kernel can reduce root compromises by less trustworthy software components. The authorization mechanisms will all be part of the trustworthy capability-based core software, and the absence of access to those capabilities is designed to effectively prevent unauthorized access to those mechanisms and their data.

PrL One of the major challenges here will be to provide a relatively seamless transition for the programming language constructs into the access controls of the software and their implementation in capability based accesses. Consistent enforcement will rely on good programming language extensions, good software engineering, and security policies that are understandable, enforceable, and usable. Thus, usability is just one concern here. In particular, adding security-related primitives to the programming language means adding new semantics with respect to the underlying execution substrate, and not just new syntax or layering over it.

APP As always, for the application software there will be some reasonable assurance that higher-layer software cannot compromise the integrity of the lower layers (SEP, CHE).

3.2.3 Validation

Input validation issues:

Inadequate argument validation missing bounds checks, type inconsistencies, escape mechanisms/characters

Executable inputs cross-site scripting, SQL injection, buffer overflows

CHE The hardware is being designed to simplify the avoidance of these problems and to simplify the formal and semi-formal analysis that we have successfully done so. The presence of distinct segments, primitive and extended types, explicit bounds, and tagging of each capability should dramatically reduce the possibilities of many of these characteristic problems.

The segmentation implied by our capability model reduces the code-data confusion which leads to so many vulnerabilities, and encapsulation in an object system limits the scope of effective exploits through a least privilege model. By introducing type safety to the TCB, we should significantly improve its robustness and reduce programming errors.

SEP The separation kernel will have explicit properties that are satisfied architecturally, that will allow policy driven controlled sharing, with verifiable properties.

TES Static and dynamic checking for nasty situations will be very valuable.

PrL, COM Capability-aware programming language extensions can go a long way to avoiding many of these characteristics.

PRI, SWE Sensible software engineering concepts (e.g., abstraction, encapsulation, least privilege, minimization of what must be trusted) can also greatly reduce these vulnerabilities.

APP As always, for the application software there will be some reasonable assurance that higher-layer software cannot compromise the integrity of the lower layers (SEP, CHE).

3.2.4 Accountability

Accountability Issues:

- Inadequate monitoring and data collection
- Spoofable audit trails
- Lack of forensics worthiness

CHE The hardware will have explicit traceability facilities and will facilitate event logging.

SEP Each partition will necessarily have its own independent accountability, although some commonality of mechanisms should allow some system-wide monitoring with carefully controlled sharing. The audit trails need to be tamper resistant and tamper evident. The underlying accountability mechanisms will all be part of the trustworthy capability-based core software, and the absence of access to those capabilities is designed to effectively prevent unauthorized access to those mechanisms and their data.

It may be worth mentioning specifically that the implied compartmentalization of software introduces new controlled communication points with explicit protocols described by the programming language. This means that we can inject monitoring and perform forensics in a much more granular way than traditionally possible.

TES Monitoring potential violations of accountability policies

APP As always, for the application software there will be some reasonable assurance that higher-layer software cannot compromise the integrity of the accountability policies and mechanisms lower layers (CHE, SEP).

3.2.5 Cryptography

Cryptographic issues:

- Absence of encryption in authentication, communication, storage audit trails, ...
- Lack of integrity checks
- Weak algorithms for crypto, hashing

- Improper encapsulation of crypto
- Weak key management
- Key exposure
- Man in the middle attacks

CHE Complete isolation of certain hardware cryptographic operations and coprocessor embedded keys; tamper resistance and tamper evidence.

We've been pondering tamper resistance and tamper evidence. Although we do not presently have plans to do this in a hardware sense, we can do this via enforcement through capabilities. Note Saar Drimer's Cambridge PhD thesis [21] on FPGAs and security, which had a particular focus on tampering, protecting FPGA bit streams, and so on. While at Xilinx Research before coming to Cambridge, Drimer examined red-black issues in the FPGA-substrate and has a patent or two relating to that.

NOTE: We are still pondering what to do about logic errors, and particularly concurrency issues – and expect to have some concrete thoughts here later. We have been discussing a 'delegate and release' – or 'do not distribute' notion that might compose with capabilities to help with ephemeral/temporary delegations, but also help with concurrency. But this is still preliminary.

SEP Trustworthy embedding of crypto in the low-layer software. The core cryptographic mechanisms will be part of the trustworthy capability-based core software, and the absence of access to those capabilities is designed to effectively prevent unauthorized access to those mechanisms and their data.

APP As always, for the application software there will be some reasonable assurance that higher-layer software cannot compromise the integrity of the lower layers (CHE, SEP).

3.2.6 Logic Errors in Design/Implementation/Operation

Logic errors can affect security, reliability, program correctness, etc., generally. They were included long ago as a wide range catch-all for various other types of problems:

Poor resource management resource exhaustion, resource conflicts

Synchronization and timing Atomic primitives that aren't, Deadlocks, Race conditions, Other ToCtToU problems

Overt information leaks Undesired disclosure, Accidental disclosure

Exploitable covert channels (storage/timing channels)

CHE Must enforce sound synchronization properties. ...

SEP Must limit covert channels among partitions, Hierarchical locking strategies, ...

TES Ensure properties, especially in execution [See TESLA paper/report.]

COM, SWE Enforce detection, resilience, remediation.

APP As always, for the application software there will be some reasonable assurance that higher-layer software cannot compromise the integrity of the lower layers (CHE, SEP).

PRI Architecturally minimize what must be trusted in CHE, SEP, APPs.

3.2.7 Malware

The presence or insertion of malware can exploit all of the above. CTSRD countermeasures to malware include a combination of the capability-based hardware and separation kernel, finer-grain access controls, tightly controlled sharing, extensive assurance and formal analysis, the hybrid architecture that allows malware to co-exist suitably compartmentalized along with highly trustworthy applications, some assurances provided by capability-aware programming languages and corresponding compilers for software developed as highly trustworthy applications, and persistent layered assurance. These defenses provide resistance to all of the following: privilege escalation attacks, buffer overflows and network attacks, application-layer attacks, and even various types of insider attacks.

CHE The analysis of the BSV hardware specifications is expected to detect flaws in the hardware that might allow the insertion of embedded Trojan horses into the hardware during design and fabrication.

SEP The separation kernel/virtual machine hypervisor will be extensively analyzed formally.

TES See the TESLA paper/report.

APP The capability architecture in any trustworthy software silo should help protect the system from malware in applications that could affect the integrity of the lower layers.

PrL Wise choices of programming languages (e.g., typed abstractions, encapsulation, avoidance of buffer overflows) can help significantly to reduce the exploitable security flaws that can enable the introduction of malware.

PRI, SWE, COM Principled system and network architectures, good software engineering practice, and strong static checking in compilers could also assist in helping avoid the inadvertent construction of accidental malware and the insertion of intentional malware. Good configuration control and life-cycle oversight can help limit malware.

Chapter 4

ISA Model in the L3 language

For this part of the work, we used a formal model of the MIPS Instruction-Set Architecture (ISA) that was written by Anthony Fox, Matthew Naylor, and Alexandre Joannou for the EPSRC-funded REMS ("Rigorous Engineering for Mainstream Systems") project. This model of the MIPS ISA is written in the L3 [22] specification language, and can be automatically translated either into a format suitable for use by theorem provers such as HOL4, or into an executable model in the ML programming language.

The L3 model covers the core of the MIPS IV ISA specification, including privileged instructions, the TLB, software exceptions, and interrupts. It also includes BERI's nonstandard TLB, and some of BERI's I/O devices: the UART and the Programmable Interrupt Controller (PIC). At the time of writing, we are exploring extending the model to include the floating-point coprocessor.

The ISA model also includes the CHERI capability extensions described in the CHERI Instruction-Set Architecture [52]. These can be conditionally included when compiling the model to generate either a BERI or a CHERI model. Moreover, the model supports different configurations of the capability format (also described in the CHERI ISA specification), specifically the original 256-bit capability and the 128-bit compressed capabilities candidates. This feature proved particularly useful when used in conjunction with the fuzz_cap python script (Section 9.5) to assist with the development of the BSV counterpart of the compressed model.

The L3 model can handle nondeterminism in the MIPS ISA specification only in a very limited way: given a program whose result is "unpredictable" according to the MIPS specification (e.g., a 32-bit arithmetic operation on a 64-bit value that is not a sign extension of any 32-bit value), the simulation halts with an error message. This has proven especially valuable in identifying software reliance on unpredictable ISA behavior.

4.1 Co-validation of the model and test suite

We ran our test suite through the L3 model, confirming for each test that the expected result of the test agreed with the result predicted by the L3 model – and to identify areas of difference between a "consensus" version of MIPS implemented by the processor versus one derived from the ISA specification. In our test suite, each test is classified according to which components of the CPU it depends on (TLB, floating-point unit, capability unit etc.). Our main reason for doing this is that our processor can be built in a number of different configurations (with or without TLB, with or without floating point, with or without capabilities, and so on); thus, we need to customize the test

suite for the particular configuration under test. This approach also makes it easy to accommodate the limitations of the L3 model: we treat it as a minimal configuration of BERI or CHERI without the floating point, and then run the appropriate tests.

Our tests are also classified according to whether they are for standard MIPS ISA behavior, or for features specific to our implementation. Using this information, we could check whether the L3 model agreed with the expected test results on all implementation-independent tests, or stopped with the "unpredictable" error condition on all implementation-dependent tests.

Our BERI and CHERI processors pass our test suite, which confirms that they agree with the L3 model on all applicable tests: both the implementation and the model produce the same expected result. The process of implementation and model co-design led to improvements to both.

We used the *Jenkins* continuous integration system to automatically re-run our test suite against all configurations of the L3 model whenever either a test or the model was changed, and to re-run our test suite against BERI whenever a test or our hardware was changed. This provides an automatic and immediate notification if any source code change resulted in divergence between the hardware and the formal model.

4.2 Measuring manual test-suite coverage

We were able to use the L3 model to measure the coverage of our manually written test suite. The L3 language compiles to ML, and ML can be be compiled with an option to measure test coverage. We then ran our test suite through a CHERI emulator that had been compiled in this way. This automatically finds all branches in the L3 specification of CHERI that have not been exercised by the test suite.

A minor problem with this method is that the L3 compiler inserts some run-time checks that can never fail; however, the L3 compiler is unable to statically determine that they can't fail. Measurement of test coverage will then detect that these dead paths in the emulator are never exercised, but there is no automatic way to distinguish them from genuine omissions in the test suite.

Automatic measurement of test coverage was able to find some bugs in our test suite. For example, a test of the `branch likely` instruction used an ordinary branch instead of the intended `branch likely` instruction (a typo in the code). A branch instruction behaves in exactly the same way as `branch likely` (apart from performance), and so the test passed. This was detected by test coverage analysis, because the `branch likely` instruction was not being exercised.

4.3 Automatic Test Generation

Brian Campbell (at the University of Edinburgh) was able to automatically generate a test suite from our L3 model of the MIPS ISA, using techniques he had previously used for generating tests for the ARM architecture [11]. At the time of writing, a limitation of Brian's tests was that they covered only code paths for which a corresponding *step theorem* has been proved using the HOL theorem prover (e.g., they did not cover the cases where an instruction raises an exception), although work was underway to remove this limitation.

We ran Brian Campbell's tests against both the L3 model and a Bluesim simulation of the CHERI1 BSV, and automatically compared the traces.

An example of a CPU bug found using this method is that the MIPS ISA has a zero register that is supposed to be always zero; there was a hardware bug in CHERI1 that allowed it to be set to a nonzero value by a conditional move instruction. (The conditional move changes the register's value in a different pipeline stage from other instructions, and so bypasses a check on the register being the zero register that was performed for other instructions).

4.4 Booting FreeBSD in the L3 Model

Next, we ran the FreeBSD operating system in the simulator generated from the L3 model. The L3 model is complete enough that it is able to run FreeBSD: it includes privileged-mode instructions, the TLB, timer interrupts, the programmable interrupt controller, and the UART. It also successfully booted the CHERI-enabled version of FreeBSD with the capability ISA extensions.

4.5 Booting Multicore FreeBSD in the Model

We then ran FreeBSD in a simulation of a multicore processor derived from the L3 model. We simulated two different memory coherence models: (a) sequentially consistent memory; and (b) a relaxed memory model in which writes to cached memory can be reordered, but the `sync`, load-linked and store conditional instructions act as memory barriers. The model-derived simulator was capable of running FreeBSD under both memory models.

4.6 Trace Comparison

We were able to automatically compare traces of the L3 model running FreeBSD against traces of Bluesim simulating BERI running FreeBSD, for a single core CPU. The main challenge in comparing traces is nondeterminism in the specification: traces can differ and yet both be correct according to the ISA specification.

The main sources of nondeterminism in the MIPS ISA specification are as follows:

The UART

Clearly, the command line that a user types into the console of a FreeBSD system is not determined by the ISA specification, and yet the subsequent behavior of the system will vary greatly depending on which command is entered. For traces to match, exactly the same input string must be given to the two simulations that are being compared. Not only must the characters match, but also the exact time (instruction) at which a character enters the UART's input buffer must match. As the UART has flow control on output, the exact time at which the console device asserts "ready to send" (RTS) must match as well.

The TLB

In the MIPS ISA, the TLBWR instruction uses the contents of the "Random" register to choose which TLB entry to replace. According to the ISA specification, the Random register decrements each clock cycle, and so provides a very rudimentary pseudo-random number generator. The number of clock cycles that each instruction takes to complete is not determined by the ISA

specification, and so the contents of the Random register at a particular point in the program are also not determined.

BERI1's TLB uses a more deterministic algorithm that the one in the MIPS ISA: it decrements each time a "random" element of the TLB is chosen, so that TLB entries are replaced in a round-robin manner. This design choice in BERI1 was made in order to make trace comparison easier.

Timer interrupts

Coprocessor 0 will generate a timer interrupt when the contents of the Count register equals the contents of the Compare register. If the Count register increments once per clock cycle, then the exact instruction at which the timer interrupt fires is nondeterministic. Both BERI1 and BERI2 have options to make the Count register increment on each committed instruction, rather than each clock cycle, to make trace comparison easier.

"Unpredictable" behavior

The MIPS ISA identifies three kinds of undefined behavior: (a) "unpredictable result" – the result of the operation can be any bit pattern, but nothing else is nondeterministic; (b) "unpredictable operation" – the state of a user-space program can be anything after a user-mode program does an unpredictable operation, but kernel-mode state is guaranteed to be known; and (c) "undefined operation" – after which any state might change.

For security proofs, the distinction between "unpredictable" and "undefined" is important. A malicious program might deliberately do an "unpredictable" operation as part of an attack; the English-language text in the MIPS ISA specification implies that a malicious program can't break the system's security this way. The L3 model does not attempt to formally model this.

Instead, the L3 model halts with an error in most cases of "unpredictable" operations or results. (There is one exception, which we will describe below). In order to compare traces of the L3 model with BERI, we rely on there being no instances of unpredictable behavior in the traces being compared; the simulator will tell us if this assumption turns out to be wrong. In turn, this assumption relies on the operating system source code being sufficiently "clean" that it does not contain any unpredictable behavior.

Division by zero

Both gcc and clang generate object code in which a program does a divide instruction, and only afterwards checks if the divisor was zero. If the divisor was nonzero, it then picks up the result using the `mfhi` and `mflo` instructions. This compiler idiom takes advantage of the parallelism between the multiply/divide unit and the integer pipeline: the integer pipeline can be checking for equality to zero while the multiply/divide unit is starting work on the division. The correctness of this compiler idiom depends on the distinction between an "unpredictable result" and an "unpredictable operation". The result of division by zero is an "unpredictable result" in the HI and LO registers, but the rest of the state of the machine is guaranteed to not be affected, so (for example) the subsequent check for zero must function correctly.

To handle this common case, the L3 model makes a distinction between the "unpredictable result" of division by zero and "unpredictable operations": only the latter will cause the simulator to halt with an error message.

Context switch after multiply or divide

As described above, the result of division by zero is an "unpredictable result". Program operation is deterministic as long as this "unpredictable result" is not looked at. But what if you are simulating an entire operating system, and the operating system does a context switch after a user-space program does a divide by zero? The operating system will try to save the contents of the HI and LO registers, so that they can be restored later. The operating system has no way of knowing that these registers contain an "unpredictable result", and will try to save them anyway.

Again, the distinction between an "unpredictable operation" and an "unpredictable result" is important. A user-space program cannot crash the kernel by doing a division by zero, waiting for a context switch, and hoping that an unhandled exception will be raised in kernel space when the kernel tries to save its registers, because this is only an "unpredictable result", not an "unpredictable operation".

However, the L3 model does not have good mechanisms for modeling this kind of nondeterminism. To handle this case, we pass an option to the model that changes this case from being an error that stops the L3 simulation into deterministic behavior: we built into the L3 model knowledge of the implementation-specific way that BERI treats division by zero.

Multiplication into a general purpose register

In the MIPS III ISA, the result of an integer multiplication is always placed in the HI and LO registers. Later versions of the MIPS ISA add a new multiply instruction that places the result in a named register; the contents of HI and LO are "unpredictable results" afterwards. We added this instruction to BERI1 and BERI2 (against the protests of BERI1's hardware designer ...) because the gcc compiler and the FreeBSD kernel use it. The new multiply instruction presents a similar problem for the L3 model as division by zero. We deal with it in a similar manner, by modeling BERI's implementation-specific behavior (HI and LO are unchanged).

TLB Probe

If a TLBP (TLB Probe) instruction does not find a matching TLB entry, the MIPS R4000 ISA says that the high-order bit of CP0.Index is set. The ISA spec is somewhat unclear on what happens to the bottom bits in this case: they are presumably an "unpredictable result". BERI and the L3 model set them to zero.

4.7 Multicore Trace Comparison

In a multicore CPU, which core will be the next to complete an instruction is implementation dependent. It should theoretically be possible to compare traces if the memory is sequentially consistent: take a trace of Bluesim running a program; extract from it the relative order in which loads and stores from different cores interacted with the memory; run the L3 model with the cores scheduled to interleave instructions in the same order; and compare the results.

The L3 model and our trace comparison program have support for doing this. However, multicore BERI1 does not guarantee sequential consistency, so traces can fail to match because they contain out of order loads or stores.

The MIPS ISA does not define the block size for load linked/store conditional. A store conditional will fail if another CPU has written to the same block since the load linked, but it is implementation-defined how big these blocks are. We deal with this form of nondeterminism by making the L3 model use the same block size as the BERI implementation.

A third source of nondeterminism in a multicore CPU is interprocessor interrupts. There may be a delay of a few instructions between one core writing to the PIC and another core receiving an IPI, and the amount of delay is implementation-dependent.

4.8 Modeling BERI caches

The L3 model can also be built with BERI's caches. The cache hierarchy itself is made of private instruction L1 caches, private data L1 write-through caches, and a shared L2 write-back cache. Cache coherency is assured by a simple write-invalidate protocol in the L2's directory. Several counters of various events (hit, miss, eviction...) are also updated through the simulation and can produce useful cache statistics. The caches are parameterizable in terms of size, associativity, line size, replacement policy. They have been used to explore capability enabled pre-fetchers behaviors.

Chapter 5

ISA Model in the SAL language

(This chapter describes the initial method of generating automated tests for CHERI, and is included for completeness. The additional techniques used at present are documented in Chapter 9.)

During the implementation of the CHERI CPU, we found that it was very helpful to have a thorough set of tests for the instruction set. Our developers re-ran the test suite on each change to the Bluespec SystemVerilog (BSV) implementation before it was checked into revision control. In addition, we had a server (using the *Jenkins* package) that regularly checked out the current version of the software from revision control, compiled it, ran the tests, and produced a report of which tests failed.

Our development process was primarily an example of test driven development. For each new ISA change, we first wrote a specification, then the tests exercising those features, and finally modified our implementation. The tests and the implementation were usually written by different people, as an additional safeguard against the writers of the tests and the implementation both misunderstanding the specification in the same way.

Our basic set of tests were manually constructed based off our human-readable documentation. These tests primarily tested the standard MIPS instructions and an initial set of tests for the capability ISA extension. Generating hand-written tests has many disadvantages. It is both time-consuming and if a change is made to the specification of the instruction set architecture, then every test needs to be re-checked by hand to determine whether the change affects the expected result of the test. However, possibility of specification change was less of an issue for the standard MIPS instructions, which were effectively fixed, but more of a concern for the capability instructions, which were revised in the light of experience from writing the compiler and application programs.

In addition to the manual tests we automatically generated a second set of tests focused on the capability instructions. This was done by leveraging the formal specification written in SAL and the SAL Automatic Test Generator. This approach has the advantages that the entire test of tests can be automatically regenerated whenever a change is made to the specification, and it reduces the risk that a programmer might have misunderstood the specification when writing a test.

The Automatic Test Generator is supplied with a set of predicates defining the conditions to be tested, and it uses model-checking to find a set of sequence of instructions that will cause each of these test conditions to be become true at some point in the test run. We constructed these test predicates by taking every state transition in the formal model, and adding a test predicate that becomes true when that state transition is executed. In this way, the Automatic Test Generator produced a set of tests that exercised every branch in the formal model (branch coverage).

The output of the Automatic Test Generator is a trace of the formal model as it runs through the test sequence. We took these trace, extracted the "opcode" field from each state transition and disassembled it to produce a set of CHERI assembly language programs. Note that this approach relies on there being no branch instructions in the traces. These assembly language programs were then assembled into object code using the GNU assembler, linked, and run on a simulated CHERI CPU in the BSV simulator, Bluesim.

Our BSV implementation of CHERI is augmented to generate debug statements during simulation in Bluesim. These are triggered when certain important state transitions occur, such as write-back to CPU registers. We took this debug output from Bluesim, and converted it into the format of a SAL model-checking trace.

Finally, this trace of what actually happened when the test programs were run on a simulated CPU was compared against the SAL formal model, and each test was considered as having passed if the trace was consistent with the formal model.

In developing this approach, we encountered a number of issues:

- **Casting integers to bit vectors.** CHERI's capability registers are strongly typed. However, CHERI also has general purpose arithmetic registers that can contain integers, bit vectors, or indeed almost any data type that will fit in the number of bits available. The model checking approach to test generation can become computationally infeasible once you have registers that can be used as either integers or bit vectors. The problem is that optimizations used by the model checker to efficiently search integer or bit vector spaces cannot be used if there is a mixture of both types of operation, and the model checker is reduced to exhaustive search of a set of $O(2^{64})$ possible values.

- **Branches.** The approach described above, of taking a trace generated by a model checker and converting it into a test program by making the first instruction of the program the first instruction of the trace (and so on) does not work if trace contains branch instructions. For this reason, we were unable to use this method to test branches.

- **Exceptions.** Many of the CHERI instructions can generate exceptions. When an exception occurs, control branches to an exception handler, the CPU enters supervisor state and so on. The SAL model does not include the details of exception handling. Instead, a SAL trace will terminate when an exception occurs. The trace comparison program will conclude that traces match if the SAL trace terminates with an exception and the Bluesim trace contains an exception being raised at the corresponding point. Trace comparison does not extend into the behavior of the exception handler. In this way, we were able to test that CHERI instructions raise an exception when they are supposed to. This was an important part of the CHERI ISA to test, because many of the CHERI exceptions are security critical (i.e. it would be an exploitable bug in the CHERI security mechanism if some of these instructions did not raise an exception when they were supposed to).

- **State space explosion.**

5.1 Conversion from SAL to PVS

The SAL model checker is convenient for automatic generation of tests. However, model checking is not a powerful enough technique to verify some of security properties of CHERI that we are

interested in. For that, it would be better to use a theorem prover such as PVS. There is a problem: SAL and PVS are different languages, even though they have many features in common. Manually translating a specification from SAL into PVS is time-consuming and potentially error-prone: a proof about a property of the PVS translation might not be valid for the SAL model if a programmer made a mistake translating between the two.

To bridge the gap, we wrote an automatic translator from SAL into PVS. This effectively consists of a semantics for the SAL language that defines SAL in terms of PVS, and a compiler for SAL that outputs the PVS translation.

Chapter 6

Automated Extraction of Architectural BSV Descriptions from Microarchitectural Descriptions

While structures in a microarchitecture almost always have corresponding structures in the complete architectural definition, the correspondence between a reachable microarchitectural state and its corresponding reachable architectural state is not as simple as just eliding the microarchitecture-only state. This is because many of the tricks that designers do to achieve higher performance and efficiency require dramatically deconstructing and reordering the architectural operations.

Consider the example of a microprocessor design. While the ISA defines the operations of a processor with one instruction being completely executed before the next, in practice a typical implementation decomposes a single instruction into a sequence of pipeline stages. In effect, this means that the purely architectural state that is derived from the architectural specification cannot be neatly separated from the non-architectural implementation-level state in the microarchitectural elements. This may be further complicated by speculative operations, where the microarchitecture may optimistically perform work that it may later have to undo, *i.e.,* speculative instruction fetching via branch prediction.

When proving correctness of a microarchitectural implementation in relation to the architectural specification, an abstraction relationship relating state configurations in both systems must be given as an input. In the context of a design with complex speculation, caching, and pipelining, the relationship must effectively flush or unwind any partially completed operations; this can easily make the relationship's description require the full complexity of the design itself and in practice be substantially more complex than the proof itself. Further, defining this relationship is challenging for even for a formal verification specialist, and would be infeasible for a hardware designer to construct.

Shen and Arvind [3] noted that in the context of rule-based systems, much of the abstraction relationship can be mechanically extracted from the execution of rules themselves. For a simple processor example, all processor operations not starting a new instruction are executed until no more operations may be done, at which point a simple elision function can remove the microarchitecture-only state. This, in effect, leverages the design's natural execution capability to extract the abstraction function and allows the relationship between the implementation and the specification to be described in a simple and natural way. Using this approach, one is able to

sufficiently reduce the fundamental descriptive aspects of verification to a level where it may be possible for designers to verify their own designs.

Shen and Arvind used this approach to show bisimulation of microprocessor designs by first proving that the abstraction was a function, using standard bisimulation checks to establish the correctness of the design. However, this approach was limited in that one had to perform a significant set of proofs to show that the abstraction was both finitely bounded and a proper function. This limited the relationships that could be used and required manual intervention in the reasoning about the abstraction function to prove correctness, dramatically reducing the usability of the technique.

Davé et. al [17] removed many of these limitations by unifying the simulation check with the generation of the abstraction relationship. They further mechanized the verification process by reducing the check to a series of relatively simple SMT queries that can be checked in minutes for interesting systems in an initial implementation.

These techniques require that a specification exists before any verification can be done. While completely reasonable when the functionality is understood before development happens, or when the designers make a special effort to keep a valid specification, such a specification is generally not available. To handle this situation, we modify the scheme in [17] to provide an architecturally exposed view of an implementation that takes a rule-based system and a predicate function on the state and constructs a new system that moves *only* between states satisfying the predicate. When this predicate corresponds to a designer's notion of architectural specification, the constructed system corresponds to the architectural subset of the specification realized by the implementation. While this is not a complete specification (as some microarchitectural decisions will be reflected in it), it can serve as a starting point for extracting a complete description and as a simplified design to verify architecturally meaningful properties in the implementation.

6.1 Formalizing the Behavior of Architectural Extraction

To understand the approach, let us define a formal view of a BSV program. Let $P = (S, S_0, R)$ be a BSV program consisting of a set of states S, an initial set $S_0 \subseteq S$ of states, and a set of transition relations R on S where each element $\longrightarrow_a$ of R corresponds to the rule a. $(s, s') \in \longrightarrow_a$ if and only if the execution of rule a on state s results in the state s'. For brevity, we will denote this as:

$$s \rightarrow R\, s'$$

We use $\longrightarrow_R$ to denote the union of all elements of R and $\twoheadrightarrow$ to denote zero or more executions of a rule. For simplicity we assume that the null rule R_0 that takes s to s exists. This allows us to deal solely with infinite sequences of rules.

We are given a program P (the *microarchitecture*) and predicate p on the state S, $p : S \rightarrow \{0, 1\}$ (identifying *architectural states* of P) and we wish to find a new program $P' = (S, S_0, R')$ (the *architecture*) where the sequence of executions in P' correspond to the executions of P abstracted to elide non-architectural intermediate states. We call any execution between architectural states an *architectural execution* or *step*.

Stated more formally:

1. **All initial states are architectural:** $\forall\, s \in S_0.p(s) = 1$

2. **Any architecturally visible step in the microarchitecture exists in the architecture (that is, the architecture is a complete abstraction):** $p(s) \wedge p(s') \wedge s \twoheadrightarrow R\, s' \Rightarrow s \twoheadrightarrow R'\, s'$

3. **The architecture is sound:** $s \longrightarrow R'\, s' \Rightarrow s \twoheadrightarrow R\, s'$

4. **The architecture consists solely of architectural executions:** $s \longrightarrow R'\, s' \Rightarrow p(s) \wedge p(s')$

The set of possibilities for R' is infinite even if we restrict ourselves to compositions of the original rules of R. Consider a simple producer-consumer example with a two-element FIFO used for communication, with our notion of an architectural view corresponding to the state where the FIFO is empty. This is a very natural abstraction that a designer may choose as each of the two-step operations have either completed or not yet begun.

As each instance of the execution of a `consume` rule must always follow a unique instance of the execution of a `produce` rule, and we can have only two instances of `produce` before we fill the intermediate FIFO, the paths of R between architectural state can be fully described by the regular expression `produce (produce consume)* consume`. While this potential R' is infinite, we can further reduce this to the singleton set containing only `produce consume` by noting that `produce produce consume` is always equivalent to `produce consume produce` meaning we can reduce any element in the initial set to some finite sequence of `produce consume` that corresponds to our high-level idea of the "ISA"-level description.

To find a valid R', we consider the set of infinite executions $\mathcal{R}$ of the rules of R and consider a complete prefix cover C of $\mathcal{R}$; that is, a set of finite rule executions for which every element of $\mathcal{R}$ can be described as one of these prefixes followed by an element of $\mathcal{R}$. We can convert any such cover C into a set of rules R' by having each element of C (an element being a finite sequence of rule executions) correspond to new rule in R'. To meet the requirements of R', we need to have each element of R' start and end in architectural states. This leaves us with the problem that any matching prefix cover must be infinite, as in the produce-consume example described above. To resolve this, we observe that a prefix in our prefix cover C need not correspond directly to an element of R' as long as it corresponds to a line of reasoning explaining why that sequences with that prefix are covered by the set R'. This observation allows us to admit a prefix as covered by R' when we have proof that the prefix in question is equivalent to a "smaller" prefix already in R'.

Theoretically, as the state space for these programs are finite, one could enumerate all possible candidates for covers and check each for compliance. Given that for most practical checks, the size of the R' set is relatively small, it is potentially reasonable to try the approach of enumerating each candidate lexicographically, as we quickly reach a valid solution. Our approach is a refinement of this, allowing us to remove not only the particular cover candidate being considered but any that fail using the reasoning described above. In practice, this has been shown to require only a relatively small set of candidates before a valid one can be found.

6.2 The Algorithm: Finding a Prefix Cover

The architectural extraction algorithm operates on 3 objects, the prefix cover C that contains prefixes and the reasoning expressing why sequences starting with that prefix have been covered, U the set of unconsidered prefix sequences that we must consider, and R' the set of architectural rules. The algorithm proceeds as follows:

1. Start with $C := \emptyset$, $R' := \emptyset$. and $U := \{r_i \mid r_i \in R\}$, This corresponds to the cover where each prefix is a single rule step.

2. if $U = \emptyset$, we have no unconsidered prefixes and we have successfully found R' and the line of reasoning C for each sequence. If not, check if we have reached our iteration limit. If so, fail citing the elements of U as unresolved.

3. Select the smallest element σ of U:

   (a) Check if the execution of σ from an architectural state is ever valid (*non-degenerate*):

   $$\exists\, s \in S_0, s' \in \mathrm{Dom}(p).(s \overset{\sigma}{\twoheadrightarrow} R\, s')$$

   If no execution exists, then no sequences can follow this path and we can stop considering σ further: $C := \{\sigma \rightarrow \mathrm{INVALID}\}$.

   (b) Check if σ should be added to R'. That is, if some execution of σ reaches an architectural state.

   $$\exists\, s \in S_0, s' \in \mathrm{Dom}(p).(s \overset{\sigma}{\twoheadrightarrow} R\, s')$$

   If so, construct a restriction of σ, σ_{arch} that is valid only when it ends in an architectural state, and

   $$R' := R' \cup \{\sigma_{arch}\}, C := C \cup \{\sigma_{arch} \rightarrow \mathrm{ARCH}\}$$

   Construct a restriction of σ, $\sigma_{nonarch}$ valid only when it ends in a nonarchitectural state, and go to the next step.

   (c) For every execution where σ does not put us in an architectural state, we may be still be able to stop if we can show that any sequence with this prefix is equivalent to one we have already handled. To do this we establish a lexicographic well-ordering with the modification such that any execution sequence that touches an architectural state is smaller than all sequences of the same length that do not touch an architectural state. By construction, all smaller sequences are at most the same length as $\sigma_{nonarch}$. We can enumerate and check all smaller prefix sequences:

   $$\forall\, s \in \mathrm{Dom}(p), s' \in S.(s \overset{\sigma}{\twoheadrightarrow} R\, s') \Rightarrow \bigvee_{\sigma' \leq \sigma} .s' = \sigma'(s)$$

   If this query is true, we need not consider this prefix further. $C := C \cup \{\sigma_{nonarch} \rightarrow \mathrm{EQUIV}\}$ and we return to Step 2.

   If not, we partition all the prefix extensions into the $\mid R \mid$ sets of rules by extending $\sigma_{nonarch}$ by one rule execution:

   $$U := U \cup \{\sigma_{nonarch}.R_i \mid R_i \in R\}$$

We implement this algorithm by invoking a Smten-based solver. The lambda calculus input as described previously is extracted through an intermediate dump from the Bluespec SystemVerilog (BSV) compiler. To capture the predicate representing which state configurations are architecturally meaningful at the user-level, we manually represent an isArchitectural predicate in Smten on the transformed state. This could also be naturally represented in the BSV language as an "isArchitectural" rule which performs no state update and may fire only when the state is in the architectural state. Propositional queries are passed to the Yices SMT engine.

6.3 Leveraging Modularity: Decomposing the Problem

Our approach to architectural exploration is a relatively large task, requiring us to step through all sets of rule executions between two architectural states. This grows exponentially with the number of rules in the system. Thus, for practical reasons, it would be advantageous if we could split up the task of finding an architectural extraction for one subcomponent module of the design and use this simplified model in place for the architectural extraction.

The key issue here is that for any subset of rules in the whole program we must be able to put a bound on what modifications the outside world may do to the state that the rules read and write. This is difficult as the remainder of the design (the *context*) is nondeterministic and may interleave with any execution. In the situation of architecturally extracting a BSV module, we have the guarantee that all interactions are done through the finite set of interface methods, though any subset of methods may be called at once with any possible set inputs.

Rather than attempt to encode this nondeterminism in a new fashion, we can reduce the problem of architectural extraction of a single module to the previously resolved case of architectural extraction of a complete design by encoding the nondeterminism possible by the context as a "generic" BSV module. We construct an infinite input FIFO, containing "commands", each element of which can be any valid method call on the module. We also construct an infinite output record FIFO to hold every observation possible by the outside world, *e.g.,* method `m` was called with input `i` resulting in output `o`. The context has exactly one rule that takes the next input command, performs it on the module, and records the result in the output FIFO.

We construct an architectural predicate function for this new complete program consisting of the module in question and the generic context we've constructed such that it applies whenever the module is in an architectural state (as defined by the user's architectural state) and preserves both infinite FIFOs in the context.

For an architectural extraction to be considered complete, it must not generate sequences that mix the rule added to simulate usage with rules internal to the module. This corresponds to the assertion that method calls of the module cannot change whether or not the module is in an architectural state or not.

This technique works, but can be extremely limited in the general case. For instance, in the simple producer-consumer design with inputs and outputs going to outside the module, one would expect to find that we could replace the produce and consume rules with a single produce-consume rule. However, this choice effectively removes the buffering between the two rules from being used and as such the architectural design will not be able to have as many outstanding inputs as the original two rule module. If we use the module in a context that can observe this fact, for example entering N inputs and only then removing completed results, our generic context approach will not find this solution.

However, this kind of module usage is not common and so it is reasonable to not consider such contexts. Instead of our original generic context module, we can choose a context that represents all usages that cannot distinguish relative time between inputs and outputs. This can be done by having two pairs of input and output FIFOs, one pair for the input method and one for the output method. Since the records in the output FIFOs cannot tell the relative input-output timing, we can now find the architectural extraction needed.

In general, the selection of the correct generic context for a module is difficult and must be constructed by the verifier. The context must be generic enough to capture a reasonable subset of usage without being so general as to distinguish the microarchitectural input from the desired architectural extraction. When using an architectural module in place of the microarchitectural one for further extraction, it must be verified that the concrete context in the design falls within the behaviors of the generic context used.

6.4 Architectural Extraction of the CHERI Microprocessor

We are currently in the final integration phases for the architectural extraction tool. We have not yet successfully tested our extraction on the CHERI microarchitecture. However, we have spent significant time with simplified models to know what approach we should take. In this section we discuss the key aspects of the verification process for the complete CHERI hardware design.

The CHERI microarchitecture can most easily be decomposed into two major parts: the processor pipeline and the memory subsystem. Each of these has a relatively simple expected architectural view; an ISA-level processor and an uncached flat memory model. While these two components both have easily understood high-level behaviors, the details of their equivalences are very different and the challenges involved in constructing an architectural model are very different.

In the case of the pipeline, the designer has an extremely good understanding about the order in which operations should be executed in the architectural model, *e.g.,* fetch an instruction and then let it complete through the pipeline before repeating; but the designer would be hard pressed to be able to give an architectural description. Further, we expect the actual implemented pipeline to operate in a manner very close to the architectural (ISA) model, even reaching architectural states during exception handling. Thus as a natural consequence the nondeterministic BSV model is described by the designer such that it can always be immediately brought to an architectural (flushed) state. We expect no difficulties in the verification of the pipeline model with flattened caches; this approach most closely matches the initial examples used in BSV simulation checks on which this work is based.

In contrast, the model for the memory system, even one shared by many processors, is much easier to state. It should act as a flat uncached memory in a sequentially consistent manner; the individual request responses between processors may happen in any order. However, unlike the pipeline, determining what execution of rules should happen between any two architectural states is not easy. Nor is it obvious which states can be reached. Further, while the processor pipeline is described such that it can always reach an architectural state, cache systems are not described with the same level of independence of action; rarely does one think of a cache as spontaneously evicting a memory location or loading in a new value speculatively. These "voluntary" cache operations are not needed for a functional cache implementation [1]. However, they are necessary

[1]These voluntary rules, if used appropriately, may effectively allow more efficient memory sharing policies to be implemented. This is a major contribution of the CACHET cache coherence engine [45].

for our architectural extraction as they allow us to view operations through the lens of a flushed system. Without these rules our approach would need to reason about the infinite paths between cache flushes that will not terminate.

Our current cache hierarchy in the verifiable model does not include voluntary cache operations in its functionality. We plan to add these via a request wrapper that, observing the state of the cache, may choose to insert a fake cache request into the processor stream. This isolates all voluntary behaviors to the periphery of the cache model.

Chapter 7

Extracting BSV Semantics for Formal Analysis

While BSV's Guarded Atomic Action based model is closely related to term rewrite systems that are often used for formal analysis, the BSV code is not directly suitable for formal tools. A large part of this is because, like many hardware description languages, BSV provides a metaprogramming language to allow concise representation of high-level concepts such as tree-like computations and asymmetric folds. BSV's metaprogramming layer is far richer than the standard RTL languages (*e.g.,* Verilog, VHDL, allowing Turing-complete computations during hardware generation greatly enhancing the user's ability to express the desired design cleanly.

To keep the verification process amenable, we only consider the program once all metaprogramming has been completed, leaving only the system as expressed in the kernel of BSV (see Figure 7.1). This kernel is similar to the kernel of Bluespec Codesign Language (BCL), a hardware-software extension, save that it does not directly need the loop action connective, and must be augmented to allow the implementation of *actionvalues*, which represent both a state transition and the computation of a value. We represent actions as actionvalues with the empty result ().

7.1 Representing the BSV Semantic Model to Formal Tools

To provide semantics for the BSV kernel to be interpreted by formal tools we encode each action/rule into a lambda calculus term taking a state configuration of the system to a 3-tuple of a Boolean value representing validity of the rule to execute, a new state value, and the return type. A syntax-directed translation can be found in Figures 7.2, 7.3, and 7.4. As most techniques involving formal questions take special interest in how the non-determinism in rule selection is resolved, and exploit it in vastly different ways, the modelling of this top-level driving loop is left for each specific application.

Handling State Configurations

The above compilation step provides a translation from a BSV specification to a lambda calculus by translating the BSV module state in to appropriate structured data types. While this is is straightforward (each submodule is mapped directly to a field in the corresponds data type, to

program	::=	**Program** *name* [*m*] [**Rule** *R*: *a*]	// A list of Modules and Rules
m	::=	[**Register** *r* (v_0)]	// Reg with initial values
	‖	**Module** *name*	
		[*m*]	// Submodules
		[**AVMeth** $g = \lambda\, x.a$]	//ActionValue method
		[**VMeth** $f = \lambda\, x.e$]	//Value method
v	::=	*c*	// Constant Value
	‖	*t*	// Variable Reference
e	::=	*r*	// Register Read
	‖	*c*	// Constant Value
	‖	*t*	// Variable Reference
	‖	*e op e*	// Primitive Operation
	‖	*e* **?** *e* **:** *e*	// Conditional Expression
	‖	*e* **when** *e*	// Guarded Expression
	‖	(*t* **=** *e* **in** *e*)	// Let Expression
	‖	$m.f(e)$	// Value method call f of m
av	::=	*r* **:=** *e*	// Register update
	‖	**return** *v*	// ActionValue return
	‖	**if** *e* **then** *av* **else** *av*	// Conditional action value
	‖	$av >>\models \lambda\, x.av$	// Parallel composition
	‖	$av >>= \lambda\, x.av$	// Sequential composition
	‖	*av* **when** *e*	// Guarded action
	‖	(*t* **=** *e* **in** *av*)	// Let action
	‖	$m.g(e)$	// Action method call g of m
op	::=	&& | || | ...	// Primitive operations

Figure 7.1: Grammar of BSV Kernel. For simplicity, we do not include non-register primitives and assume all module and method names are unique.

TA :: BSV-Action-Value →
  (Translated-State → (Bool, Translated-State, Translated-ReturnValue))
TA [[r **:=** e_0]] =
 λs.(g, s{r = Reg{modified:True, value:e_1}}, ())
  where $(g, e_1) = (\mathbf{TE}\ [[\ e_0\ ]])\ \mathtt{s}$
TA [[**if** (p_0) **then** a_T **else** a_F]] =
 λs. if p_1 then $(g_1 \wedge g_T, s', v')$ else $(g_1 \wedge g_F, s'', v'')$
  where $(g_1, p_1) = (\mathbf{TE}\ [[\ p_0\ ]])\ \mathtt{s}$
  $(g_T, s', v') = (\mathbf{TA}\ [[\ a_T\ ]])\ \mathtt{s}$
  $(g_F, s'', v'') = (\mathbf{TA}\ [[\ a_F\ ]])\ \mathtt{s}$
TA [[a_0 **when** e_0]] =
 λs.$(g_1 \wedge g_2 \wedge e_1, s', v')$
  where $(g_1, s', v') = (\mathbf{TA}\ [[\ a_0\ ]])\ \mathtt{s}$
  $(g_2, e_1) = (\mathbf{TE}\ [[\ e_0\ ]])\ \mathtt{s}$
TA [[a_1 >>⊨ λ $x.a_2$]] =
 λs.let ns = newState s in
  let (g, s')=parMerge s_1 s_2 in $(g_1 \wedge g_2 \wedge \mathtt{g}, \mathtt{seqMerge\ s\ s'}, v_2)$
  where $(g_1, s_1, v_1) = (\mathbf{TA}\ [[\ a_1\ ]])\ (\mathtt{ns})$
  $(g_2, s_2, v_2) = (\mathbf{TA}\ [[\ a_2[v_1/x]\ ]])\ (\mathtt{ns})$
TA [[a_1 ¿¿= λ $x.a_2$]] =
 λs.$(g_1 \wedge g_2, s'', v'')$
  where $(g_1, s', v') = (\mathbf{TA}\ [[\ a_1\ ]])\ (\mathtt{s})$
  $(g_2, s'', v'') = (\mathbf{TA}\ [[\ a_2\ ]])\ (s'[v'/x])$
TA [[t = e_0 **in** a_0]] =
 λs.let tg = g_1 in (let tb = e_2 in (g_2, s', v'))
  where $(g_1, e_1) = (\mathbf{TE}\ [[\ e_0\ ]])\ \mathtt{s}$
  $(g_2, s', v') = (\mathbf{TA}\ [[\ a_0[(tb\ \mathbf{when}\ tg)/t]\ ]])\ \mathtt{s}$
  tg, tb are fresh names
TA [[m.$g(e_0)$]] =
 λs.let (g, s', v') = meth_g e_1 s in $(g_0 \wedge \mathtt{g}, \mathtt{s'}, \mathtt{v'})$
  where $(g_0, e_1) = (\mathbf{TE}\ [[\ e_0\ ]])\ \mathtt{s}$

Figure 7.2: Functionalization of Actions. `Fixed-width` text is concrete syntax of the lambda calculus expression

TE :: BSV-Expr →
  (Translated-State → (Bool, Translated-Expr))
TE [[r]] =λ`s.  (True, s.r.value)`
TE [[c]] =λ`s.  (True, c)`
TE [[t]] =λ`s.  (True, t)`
TE [[e_1 **op** e_2]] =
 λ`s.(`$g_1 \wedge g_2$`,` fe_1 `op` fe_2`)`
  where (g_1, fe_1) = (**TE** [[e_1]]) `s`
        (g_2, fe_2) = (**TE** [[e_2]]) `s`
TE [[e_b `?` e_t `:` e_f]] =
 λ`s.(`$g_b \wedge$ `(if` fe_b `then` g_t `else` g_f`),` `if` fe_b `then` fe_t `else` fe_f`)`
  where (g_b, fe_b) = (**TE** [[e_b]]) `s`
        (g_t, fe_t) = (**TE** [[e_t]]) `s`
        (g_f, fe_f) = (**TE** [[e_f]]) `s`
TE [[e **when** e_g]] =
 λ`s.(`$g_g \wedge fe_g \wedge g_e$`,` fe`)`
  where (g_e, fe) = (**TE** [[e]]) `s`
      (g_g, fe_g) = (**TE** [[e_g]]) `s`
TE [[t = e_1 **in** e_2]] =
 λ`s.(let` tg `=` g_1 `in (let` tb `=` fe_1 `in (`g_2`,` fe_2`)))`
  where (g_1, fe_1) = (**TE** [[e_1]]) `s`
        (g_2, fe_2) = (**TE** [[$e_2[(tb$ **when** $tg)/t]$]]) `s`
       tg, tb are fresh names
TE [[m.$f(e_0)$]] =
 λ`s.let (g,e) = meth_`f fe_0 `s in (`$g_0 \wedge$ `g, e)`
  where (g_0, fe_0) = (**TE** [[e_0]]) `s`

Figure 7.3: Functionalization of BSV Expressions. `Fixed-width` text is concrete syntax of the lambda calculus expression

```
TP :: BSV-Program → (λ-calculus declarations)
TP (Main ms rules) =
  (map TM ms)
  (map TR rules)
  seqMerge = λs0.λs1.
    (foreach Reg r.
     (λs.x{r=if s1.r.modified then s1.r else s0.r})) (True, s0)
  parMerge = λs0.λs1.
    (foreach Reg r.
     (λ(g, s).(g ∧ !(s0.r.modified ∧ s1.r.modified),
                s{r=if s0.r.modified then s0.r else s1.r})) (True, s0)
  newState =λs0.(foreach Reg r. (λs.s{r = s.r{modified = False}})) s0
TM :: BSV-Module → (λ-calculus declarations)
TM [[ Module mn ms ameths vmeths ]] =
  map TM ms
  map TVM vmeths
  map TAVM ameths
TR :: BSV-Rule → (λ-calculus declarations)
TR [[ Rule rn a ]] =
  rule_rn = λs0.TA [[ a ]]  s0
TVM :: BSV-Value-Method → (λ-calculus declarations)
TVM [[ VMeth f λx. e ]] =
  meth_f = λx.(λs.(TE [[ e ]]) s)
TAVM :: BSV-ActionValue-Method → (λ-calculus declarations)
TAVM [[ AVMeth g λx. a ]] =
  meth_g = λx.(λs.(TA [[ a ]]s))
```

Figure 7.4: Conversion of Top-level BSV Design to top-level definitions. `Fixed-width` text is concrete syntax of the lambda calculus expression.

fully encapsulate represent and understand all possible state configuration), we must also define a number of basic operations to set up and compare state values.

These are not always simple. Consider the implementation of a primitive bounded FIFO module in BSV. One can directly implement this via a finite array with pointers onto that function. While it is representationally easy to understand the meaning of any particular state configuration, multiple state configurations may represent the same state. Similarly, there are configurations that have no corresponding high-level representation (*e.g.,* the pointers in the FIFO correspond to an impossible size).

Rather than deal with these possibilities indirectly in the operations on our primitive BSV modules (*e.g.,* the FIFO dequeue operator), it is far more effective to separate these cases. As a result we provide three base operations to evaluate states: an initial state constructor that provides the translated initial state, a valid predicate that represents that a configuration has a high-level meaning, and an equality operator that returns true if any two configurations represent equivalent states. These operations are naturally point-wise, in that we can always define the behavior for a module using the results of the same functions recursively called on its submodules, and must only consider it specifically for primitive modules.

Chapter 8

High-level Orchestration of SMT queries in Smten

Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) have long been used for formal analysis in important verification applications like automatic test generation [10], model checking [5, 31, 44], and program synthesis [46]. The applications that benefit most significantly from SAT and SMT can be viewed as NP-hard combinatorial searches where the problems become significantly more difficult as the problem size increases. This scaling is the key factor in deciding whether an application is useful for practical problems or only for toy examples.

In principle, a SAT/SMT solver allows designers to sidestep all of the work associated with developing a sophisticated custom search technique for their specific problem, and instead translate only their problem to and from the general SAT/SMT predicate formula, reducing the time and effort required to develop a design of sufficient performance. However, in practice the translation of problems into SMT/SMT queries is non-trivial, and the efficiency this translation has significant impact on total performance. As a result, designers are forced to spend non-trivial effort on optimizing query generation. What may be a conceptually natural translation from a high-level problem to a SAT/SMT query can often require development of a large custom computational framework for constructing and interpreting an optimized query, requiring many man-months of effort to implement and debug. Consequentially, SMT solvers are rarely used for one-off or otherwise low-usage projects, as developers cannot justify the initial time costs versus a (likely less performant) hand implementation. To see this more clearly, one may refer to Appendix A which details the complications and construction of a SMT-based CFG-matching string solver.

In essence, while SMT has succeeded in allowing designers to share computational cores across many disparate tools, it does not simplify the tasks of query orchestration and generation necessary to leverage the shared computational core of SMT effectively. As a result, despite providing significant potential value, SMT technologies are used only by those with tasks that merit enormous amounts of developer effort and who have sufficient development experience to build a framework of sufficient complexity: a small set of experts. This directly affects our CHERI verification efforts as by nature of this being the first verification of this sort at this scale it is expected that we will need to make significant algorithmic modifications to our verification procedure and tool infrastructure.

To provide a resolution to the challenges of effective interfacing of SMT, we have developed Smten (pronounced *Smitten*), a high-level language for orchestrating and constructing satisfiability-based search queries. Smten is based on the functional programming language Haskell and allows

	v	$\in$	*Boolean Variable*
Boolean	b	::=	`true` $\mid$ `false`
Formula	ϕ	::=	$b \mid v \mid \neg\,\phi$
		$\mid$	$\phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$
Abbr	`ite` $\phi\ \phi_1\ \phi_2$	=	$(\phi \wedge \phi_1) \vee (\neg\,\phi \wedge \phi_2)$

Figure 8.1: Syntax of a boolean formula.

a developer to concisely specify the orchestration of search queries, while automatically optimizing query construction. In our examples we have seen that the use of Smten results in a 10-1000x reduction in lines of code and comensurate reduction in development time while maintaining comparable performance to hand-coded designs.

8.1 Understanding and using SAT/SMT solvers

A general approach used in state-of-the-art tools for solving satisfiability-based search applications is to build a significant computational framework to reduce the high-level search problem to flattened monolithic instances of satisfiability queries and leverage a SAT or SMT solver to efficiently solve them. A lightweight approach to solving satisfiability-based search applications is based on non-strict evaluation and the list monad in Haskell, which provides a direct, compositional, modular, and concise way to describe search problems, but which has a fixed-order evaluation and suffers from combinatorial explosion. The Smten language can be viewed as a combination of these two approaches, providing the linguistic benefits of natural modular composition in the description of search problems while providing the efficiencies of leveraging an SAT/SMT solver.

To properly discuss the Smten language and its translation to satisfiability queries, it is important first to establish what a satisfiability query is and how it relates to the problem of monadic composition, on which we base the Smten language.

The problem of Satisfiability is to determine if there exists an assignment to variables of a given boolean formula under which the formula is satisfied. The syntax for a boolean formula is shown in Figure 8.1. We use the abbreviation `ite` $\phi\ \phi_1\ \phi_2$ for the commonly occurring if-then-else construct. We will often use μ to represent a satisfying assignment to a boolean formula. An assignment μ is a mapping from a variable in the formula to a boolean value.

A SAT solver takes a boolean formula as input and returns a satisfying assignment for the formula if one exists, or otherwise indicates that the formula is unsatisfiable. Specifically, given boolean formula ϕ, a SAT solver returns `Sat` μ if the formula is satisfiable with an arbitrary satisfying assignment μ, or `Unsat` if the formula is unsatisfiable.

Though the problem of satisfiability is NP-complete [14], SAT solvers leverage efficient heuristic search algorithms based on DPLL [19, 18] for satisfiability that scale well for many practical problem instances due to decades of research and implementation efforts.

Satisfiability Modulo Theories (SMT) is an extension to the problem of Satisfiability with additional background theories for which there exist decision procedures [4]. For instance, an SMT solver may support formulas with integer variables and linear arithmetic operations, or bit-vector variables with bit-vector operations. SMT solvers leverage higher-level information when solving the formula, *e.g.,* the commutativity and associativity of the operator `+` which can significantly improve the solver's performance.

Reducing Search to Satisfiability

To understand the effort in taking a high-level search and reducing it to an SMT query, consider the string-constraint solving problem described earlier as an example. An SMT query can be constructed for the problem of synthesizing a string from the template `ab??cd` matching the regular expression `(ab)*(cd)*`. At a high level, the desired query is:

$$(\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"ababab"}\ ) \vee (\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"ababcd"}\ ) \vee$$
$$(\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"abcdcd"}\ ) \vee (\texttt{"ab}x_1x_2\texttt{cd"} = \texttt{"cdcdcd"}\ )$$

Here the free variables x_1 and x_2 have been used to represent the value of the unknown characters. The solver is responsible for finding values of x_1 and x_2 that satisfy the formula.

Unfortunately, SMT solvers do not support these high-level operations on character strings. Instead, the high-level query on character strings must be encoded indirectly using a background theory supported by the solvers, such as a theory of bit vectors. Using ASCII to encode characters as bit vectors, the characters `a`, `b`, `c`, and `d` map to bit-vector values `97`, `98`, `99`, and `100` respectively; the free variables x_1 and x_2 are treated as bit-vector variables; equality of strings is broken down into element-wise character equality; and equality of characters is mapped to equality of bit vectors. With this encoding, we arrive at the following SMT query:

$$(97 = 97 \wedge 98 = 98 \wedge x_1 = 97 \wedge x_2 = 98 \wedge 99 = 97 \wedge 100 = 98) \ \vee$$
$$(97 = 97 \wedge 98 = 98 \wedge x_1 = 97 \wedge x_2 = 98 \wedge 99 = 99 \wedge 100 = 100) \ \vee$$
$$(97 = 97 \wedge 98 = 98 \wedge x_1 = 99 \wedge x_2 = 100 \wedge 99 = 99 \wedge 100 = 100) \ \vee$$
$$(97 = 99 \wedge 98 = 100 \wedge x_1 = 99 \wedge x_2 = 100 \wedge 99 = 99 \wedge 100 = 100)$$

Under the assumption that the number of characters in the string is fixed, it is relatively straightforward to automate the construction of this query by implementing a regular-expression match algorithm that operates on strings of encoded characters instead of strings of normal characters. The result of running the regular-expression match will be an SMT encoding of a boolean where the equality operator, logical AND, and logical OR construct a boolean formula instead of evaluating to a boolean value.

Note that this query can be simplified before being sent to the solver, resulting in the simpler query:

$$(x_1 = 97 \wedge x_2 = 98) \vee (x_1 = 99 \wedge x_2 = 100)$$

While applying these optimizations does not affect the result in principle, in practice it is quite common for the size of the query to be reduced by orders of magnitude, significantly reducing the cost of communication to the solver.

If the number of characters in the string is not fixed, this simple encoding will not work as we must be able to represent the different possible lengths. There are many possible representations that we can use. If there is a non-zero lower bound on the number of characters in the string, it may make sense to implement a hybrid of these two approaches to use the most efficient encoding of the regular-expression match for both the fixed-size prefix of the string and the variable-size suffix. Regardless of our choice, it is clear that the regular-expression match algorithm must change to deal with the new representation of a string and not just the fixed list of characters we worked with before.

Once we have constructed an SMT query, we pass it directly to the SMT solver and it provides a satifying assignment or a statement that no solution exists. In this example one possible result would be $(x_1, x_2) = (97, 98)$. We must translate this back to the level of the high-level search that we were interested in ("ababcd").

SMT User Challenges

Aside from the challenges of encoding high-level queries as SMT queries and optimizing the construction of the SMT queries, there is a fair amount of tedious work required to use an SMT solver. This includes deciding which solver to use, installing the solver, understanding whether to use a text-based or native interface to the solver, learning the syntax of queries or the proper sequence of API calls to pose the query and interpret the model, knowing how to correctly configure the solver, and understanding memory management requirements for the solver.

For users who have limited or no knowledge of SMT, the startup costs of using SMT is prohibitively high. Expert users of SMT with enough incentive to overcome the tedium of using SMT solvers face the problem that their tools are optimized for specific solvers, encodings, and query orchestrations, making it impractical to reuse code for different applications and domains.

8.2 Alternative Search Approach: Monadic Composition

In contrast to the SMT approach to satisfiability search which relies on significant reduction, we can describe satisfiability searches directly in a natural way via Haskell's List monad. This generally provides worse performance as in evaluating a list-monad based search expression, one enumerates all possible candidate solutions to the search computation and filters out invalid cases one-by-one until a satisfactory solution is found: in essence, a simple backtracking search.

The value added by the functional programming language Haskell is its non-strict evaluation semantics, and special `do`-notation for monadic programming that can be used with the list type. Non-strict semantics mean functions are evaluated only when they are needed for the result of the computation. As a consequence, in Haskell a candidate solution need not be checked in its entirety before discovering it is an invalid solution, and evaluation stops as soon as the first valid solution is found, if only the first solution is inspected. Coupled with Haskell's `do`-notation and standard operations on the list type, non-strict evaluation provides an elegant way to represent search problems.

For example, Figure 8.2 shows a sketch of Haskell code that could be used to search for a string from the template `ab??cd` that matches the regular expression `(ab)*(cd)*`.

The function `candidates` produces a list of all concrete strings described by the template in our example. The notation `['a'..'z']` is an arithmetic sequence that produces the list of all characters from `'a'` to `'z'`.

The `do`-notation can be thought of as a set comprehension:

```
{ ['a', 'b', x1, x2, 'c', 'd'] | x1 ∈ ['a'..'z'],
                                 x2 ∈ ['a'..'z'] }
```

Technically `do`-notation desugars into calls of the function `map`, which applies a function to every element of a list, and `concat`, which flattens a list of lists into a single list of elements:

```
concat (map (λx1 →
    concat (map (λx2 →
       [['a', 'b', x1, x2, 'c', 'd']]
       ) ['a'..'z'])
    ) ['a'..'z'])
```

The `solutions` function uses `guard` to filter out those candidate strings that match the given regular expression.

```
match :: RegEx → String → Bool
match = ...

candidates :: [String]
candidates = do
  x1 ← ['a'..'z']
  x2 ← ['a'..'z']
  return ['a', 'b', x1, x2, 'c', 'd']

solutions :: [String]
solutions = do
  candidate ← candidates
  guard (match "(ab)*(cd)*" candidate)
  return candidate

main :: IO ()
main = case solutions of
    [] → putStrLn "No Solution"
    (x:_) → putStrLn ("Solution: " ++ x)
```

Figure 8.2: Haskell list monad approach to searching for a string from `ab??cd` matching `(ab)*(cd)*`.

Note from the type signature of the function `match` that its implementation is given for ordinary character strings, even though it is applied to a set of possible strings. The particular set of strings being applied is not built into the implementation of `match` as it was when encoding SMT queries. In this sense, the list monad approach for describing search problems is more modular than the SMT approach. Of course, the list approach is also terribly inefficient for problems of non-trivial complexity as it directly evaluates the query given and does not recognize any opportunities for sharing that could reduce the cost of evaluation, resulting in exploding run times.

The idea behind Smten is to have users express search problems using the more modular list monad approach, but rather than naively evaluating the search, recast the search into an SMT query which can be answered efficiently by the SMT solver and automatically recast back to the level of the search query. This lowers the barrier to entry for new users with limited experience using SMT.

Expert SMT users with existing SMT-based tools in different frameworks can also benefit from Smten, but not in as immediate a fashion. Our experiments suggest porting an existing SMT-based tool to Smten requires less effort than the original implementation of the tool, based on orders of magnitude code size reduction, but this effort is still non-trivial. The real benefits of Smten for expert SMT users come after their tools have been ported to Smten. Smten simplifies maintenance of these tools and design exploration for future tool improvements. Most importantly, Smten's modularity makes it feasible to construct and reuse libraries for search across a broad range of domains and applications.

Space	List	Set Interpretation
`empty`	`[]`	$\emptyset$
`single` e	`[`e`]`	$\{e\}$
`union` s_1 s_2	s_1 `++` s_2	$s_1 \cup s_2$
`map` f s	`map` f s	$\{f(e) \mid e \in s\}$
`join` s_s	`concat` s_s	$\{e \mid e \in s_i, s_i \in s_s\}$

Figure 8.3: Interpretation of Smten search space primitives.

8.3 The Smten Language

Smten can be viewed as a linguistic extenstion to Haskell whose goal is to provide an abstraction with the same descriptive benefits as the list monad, but which is also suited to efficient evaluation using SAT/SMT.

In Smten, instead of representing a search space as a list of candidate solutions, we introduce a primitive search space type analogous to the list type in Haskell. The key differences between Smten search spaces and Haskell's monadic list approach that enable us to perform search intelligently by reduction to SMT queries are:

- We leverage the property that we need only a single solution during evaluation. This allows extraneous computations to be skipped once a solution is found.
- We exploit the fact that we may accept any solution, not just the first possible satisfying solution. This flexibility allows us to concurrently search multiple subareas of the search spaces more easily and also avoid having to exhaustively search a computationally hard subspace for a solution before moving on.
- We disallow nesting searches. That is we do not allow a search to performed within a search as efficient evaluation of this would require dealing with multiple levels of quantification in the search, an ability that most solvers cannot do either at all or efficiently.

8.3.1 The Smten Search Interface

Our linguistic extension provides the following abstract data type and operations:

```
data Space a = ...

empty  :: Space a
single :: a → Space a
union  :: Space a → Space a → Space a
map    :: (a → b) → Space a → Space b
join   :: Space (Space a) → Space a

search :: Space a → IO (Maybe a)
```

Conceptually, an expression of type `Space a` describes a set of elements of type `a`. However, for our purposes, it is helpful to think of expressions of type `Space a` as describing a search space for elements of type `a`, because in general Smten does not construct the entire set during its evaluation.

Figure 8.3 shows the interpretation of the primitives for describing search spaces along with the corresponding Haskell list operation. The primitive `empty` is the empty search space and `single` e

is a search space with a single element e. The primitive `union` s_1 s_2 is the union of two search spaces, and differs from the corresponding list operation s_1 `++` s_2 in that it does not place the restriction that elements in s_1 are searched before elements in s_2. The `map` primitive applies a function to each element in the search space. The primitive `join` collapses a search for search spaces into a search for elements of those search spaces. As with `union`, `join` is different from its corresponding list operation `concat` in that it does not specify an order of the elements.

The `search` primitive is used to search a space and is the only new primitive operation not used to describe search spaces. The meaning of the `search` primitive, given a search space corresponding to a set of expressions s, is:

$$\texttt{search}\ s = \begin{cases} \texttt{return Nothing} & \text{if } s = \emptyset \\ \texttt{return (Just } e\texttt{)} & \text{for some } e \in s \end{cases}$$

The `search` primitive is not a pure function, because it is non-deterministic and may return any possible e. It is standard practice in Haskell to represent non-pure functions like `search` as IO computations to preserve referential transparency. This also implicitly prohibits nested uses of `search` as IO computations are also not allowed to be nested.

A concrete example of the construction of a Smten search can be found in Appendix A.

8.3.2 The Strictness of Search

Smten allows arbitrary expressions in a search space computation. This means the complete construction of a `Space` expression may be non-terminating. A natural conclusion is to state that such a computation would always result in non-termination ($\perp$). This has major impacts on our code construction as many important representational constructors are unbounded (*e.g.,* for loops or recursion using a unconstrained variable) in general but are restricted to be finite in their context (*e.g.,* a top-level constraint to that variable bounding it to a finite set of values). It is not possible for us to have perfect knowledge of whether the computation will terminate much less if contextual knowledge would allow us to prove termination.

This leaves us with two possible solutions. First, we may restrict ourselves only to computations where the bounds are directly obvious in the computation. It is generally straightforward to transform any such computation to have a strict bound if one is known; so this is possible in principle. However, this means that the form of our search computations will be very different from those of normal computations which partially invalidates the code-sharing we want.

Alternatively we could adopt a more fine-grained view. Since `search` returns only a single element of the search space, we may return a result even though the complete search space cannot be constructed.

To understand when we can potentially find a solution to a non-terminating example, consider the following usage from our string constraint solver. We may not wish to put an upper bound on the string length. The following example demonstrates a `Space` expression describing a search space that includes strings of unbounded length:

```
search (union (single "b") (union aStr (single "c")))
 where
  aStr :: Space String
  aStr = union (single "") (map (λs → 'a':s) aStr)
```

$$
\begin{array}{lrcl}
 & x & \in & \textit{Variable} \\
\text{Type} & T & ::= & T_1 \to T_2 \quad | \\
 & & & \texttt{Unit} \quad | \quad T_1\texttt{*}T_2 \quad | \quad T_1\texttt{+}T_2 \\
 & & & \quad | \quad \texttt{IO}\ T \quad | \quad \texttt{Space}\ T \\
\text{Term} & e, f, s & ::= & x \quad | \quad \lambda x_T\ .\ e \quad | \quad f\ e \quad | \quad \texttt{unit} \\
 & & & \quad | \quad (e_1, e_2) \quad | \quad \texttt{fst}\ e \quad | \quad \texttt{snd}\ e \\
 & & & \quad | \quad \texttt{inl}_T\ e \quad | \quad \texttt{inr}_T\ e \quad | \quad \texttt{case}\ e\ f_1\ f_2 \quad | \quad \texttt{fix}\ f \\
 & & & \quad | \quad \texttt{return}_{io}\ e \quad | \quad e\ \texttt{>>=}_{io}\ f \quad | \quad \texttt{search}\ s \\
 & & & \quad | \quad \texttt{empty}_T \quad | \quad \texttt{single}\ e \quad | \quad \texttt{union}\ s_1\ s_2 \\
 & & & \quad | \quad \texttt{map}\ f\ s \quad | \quad \texttt{join}\ s \\
\text{Abbr} & \multicolumn{3}{l}{\texttt{Maybe}\ T = \texttt{Unit + }T, \quad \texttt{Just}\ e = \texttt{inr}\ e, \quad \texttt{Nothing} = \texttt{inl unit}, \quad \texttt{err}_T = \texttt{fix}\ (\lambda x_T\ .\ x)}
\end{array}
$$

Figure 8.4: Syntax of KerSmten types and terms.

In this example, `aStr` is a search space defined recursively that represents the infinite set of strings whose every character is an `'a'`. The top-level search happens in a space including the string `"b"`, all the strings from `aStr`, and the string `"c"`. This is a search for elements of the set $\{\texttt{"b"}\} \cup \{\texttt{""}, \texttt{"a"}, \texttt{"aa"}, \ldots\} \cup \{\texttt{"c"}\}$. Operationally, constructing the entire space of strings in `aStr` will never terminate. Because we have not specified an order of elements searched in the space, however, it is not clear whether, for example, `"c"` is a valid result of this search, or if the search may fail to terminate.

Because it can lead to more natural descriptions of problems and better modularity, we chose to take the most non-strict view of search. In this case we allow either `"b"`, `"c"`, or any string from `aStr` to be found, and we require the search terminates eventually.

If there are no valid solutions to a search space for which the complete construction would lead to non-termination, then the entire space must be searched, and the search will fail to terminate. For example, using the `aStr` search space, the following search will fail to terminate:

```
search $ do
  s ← aStr
  if (elem 'b' s)
    then single s
    else empty
```

Understanding that the set of strings described by `aStr` can never contain the character `'b'` requires a deeper understanding of the internal construction than is reasonable to expect. As such we would expect that this search would continue to expand `aStr` and never find a solution. If the character `'b'` had been replaced with the character `'a'` in this example, however, we would expect the search would terminate with any one of the strings from `aStr`.

8.4 Compilation of Smten Search Expressions to SMT

In this section we present a syntax-directed approach for constructing a SAT formula from a Smten description of a search problem.

A key insight for producing SAT formulas from Smten search-space descriptions is to introduce two new forms of expressions to the Smten language. The first new form of expression is a

$$
\begin{array}{rcl}
e, f, s & ::= & x \;\mid\; \lambda x_T \,.\, e \;\mid\; f\ e \;\mid\; \texttt{unit} \;\mid\; \cdots \\
& \mid & \phi\ ?\ e_1\ :\ e_2 \;\mid\; \{e \mid \phi\}
\end{array}
$$

Figure 8.5: Augmented syntax of KerSmten terms.

ϕ-conditional expression that parameterizes an expression by a boolean formula, allowing arbitrary expressions to be dependent on the assignment of SAT variables. The second is a set expression representing a set of expressions by a combination of a boolean formula and an assignment-parameterized expression. The augmented syntax of KerSmten with the ϕ-conditional expression and set expression is shown in Figure 8.5. The syntax of boolean formulas ϕ was given in Figure 8.1.

ϕ-Conditional Expression (ϕ ? e_1 : e_2) The ϕ-conditional expression is a conditional expression parameterized by the value of the boolean formula ϕ. The value of the expression is e_1 for all boolean assignments under which ϕ evaluates to `true`, and e_2 for all assignments under which ϕ evaluates to `false`.

The expression $\phi\ ?\ e_1\ :\ e_2$ is well-typed with type T if both e_1 and e_2 are well-typed with type T.

An example of a ϕ-conditional expression is the expression $(v \wedge w)\ ?\ 1\ :\ 2$, which is an expression with value 1 for any assignment where both boolean variables v and w are `true`, and 2 otherwise.

We call an expression containing ϕ-conditional sub-expressions *partially concrete* in contrast to normal expressions that are fully concrete.

We use the notation $e[\mu]$ to refer to the concrete value of a partially concrete expression e under given boolean assignment μ, where all ϕ-conditional sub-expressions have been eliminated. For example, if $\mu_1 = \{(v, \texttt{true}), (w, \texttt{true})\}$ and $\mu_2 = \{(v, \texttt{false}), (w, \texttt{true})\}$, then we have that $(v \wedge w\ ?\ 1\ :\ 2)[\mu_1] = 1$ and $(v \wedge w\ ?\ 1\ :\ 2)[\mu_2] = 2$.

Set Expression $\{e \mid \phi\}$ The set expression is a canonical form for expressions of type `Space` a:

$$\{e \mid \phi\} = \phi\ ?\ \texttt{single}\ e\ :\ \texttt{empty}$$

In this form, each satisfying assignment, μ, of the boolean formula ϕ corresponds to a different element, $e[\mu]$, of the search space. The search space is empty exactly when ϕ is unsatisfiable. The set of expressions represented by set expression $\{e \mid \phi\}$ is the set of possible values of e for satisfying assignments of the boolean formula ϕ: $\{e[\mu] \mid \phi[\mu] = \texttt{true}\}$.

For example, the set expression $\{(v \wedge w)\ ?\ 1\ :\ 2 \mid v \vee w\}$ represents the set $\{1, 2\}$, because both μ_1 and μ_2 from above are satisfying assignments of the formula $(v \vee w)$. In contrast, the set expression $\{(v \wedge w)\ ?\ 1\ :\ 2 \mid v \wedge w\}$, with conjunction in the formula instead of disjunction, represents the singleton set $\{1\}$, because μ_1 is the only satisfying assignment of the formula $(v \wedge w)$.

The type of a set expression is `Space` T, where T is the type of expression e. We sometimes refer to e as the body of the set expression $\{e \mid \phi\}$.

Each of the primitives `empty`, `single`, `union`, `map`, and `join` are evaluated at runtime to construct a set expression representing the appropriate set of elements. The primitive (`search` s) is then implemented via a SAT solver as follows:

1. Construct Evaluate the expression s resulting in the construction of a set expression $\{e \mid \phi\}$.

$$
\begin{array}{llll}
\text{Pure Value} & v & ::= & \lambda x_T \,.\, e \mid \texttt{unit} \mid (e_1, e_2) \mid \texttt{inl}_T\ e \mid \texttt{inr}_T\ e \\
& & \mid & \texttt{return}_{io}\ e \mid e_1\ \texttt{>>=}_{io}\ e_2 \mid \texttt{search}\ e \\
& & \mid & \texttt{empty}_T \mid \texttt{single}\ e \mid \texttt{union}\ s_1\ s_2 \mid \texttt{map}\ f\ s \mid \texttt{join}\ s \\
& & \mid & \phi\ ?\ e_1\ :\ e_2 \mid \{e \mid \phi\}
\end{array}
$$

$$
\begin{array}{lrcl}
\textit{st-beta-phi} & (\phi\ ?\ f_1\ :\ f_2)\ e & \rightarrow_e & \phi\ ?\ (f_1\ e)\ :\ (f_2\ e) \\
\textit{st-fst-phi} & \texttt{fst}\ (\phi\ ?\ e_1\ :\ e_2) & \rightarrow_e & \phi\ ?\ (\texttt{fst}\ e_1)\ :\ (\texttt{fst}\ e_2) \\
\textit{st-snd-phi} & \texttt{snd}\ (\phi\ ?\ e_1\ :\ e_2) & \rightarrow_e & \phi\ ?\ (\texttt{snd}\ e_1)\ :\ (\texttt{snd}\ e_2) \\
\textit{st-case-phi} & \texttt{case}\ (\phi\ ?\ e_1\ :\ e_2)\ f_1\ f_2 & \rightarrow_e & \phi\ ?\ (\texttt{case}\ e_1\ f_1\ f_2)\ :\ (\texttt{case}\ e_2\ f_1\ f_2)
\end{array}
$$

Figure 8.6: KerSmten pure evaluation with new expressions.

2. Solve Run the SAT solver on the formula ϕ. If the result is `Unsat`, then the set s is empty and we return `Nothing`. Otherwise the solver gives us some assignment μ, and we return the result `Just` $e[\mu]$, because $e[\mu]$ belongs to the set s.

More precisely, we augment pure evaluation to work in the presence of ϕ-conditional and set expressions, and we define a new evaluation strategy for search space computations that reduces them to set expressions.

Augmenting Pure Evaluation Figure 8.6 shows the augmented values and rules for KerSmten pure evaluation. Both the ϕ-conditional expression and set expression are considered values with respect to pure evaluation. For the set expression, this is consistent with the rest of the search-space primitives. For the ϕ-conditional expression, because it can be of any type, this introduces a new kind of value for every type. Consequently, we need to augment pure evaluation with rules to handle this new kind of value.

The effect of rules *st-beta-phi*, *st-fst-phi*, *st-snd-phi*, and *st-case-phi* is to push primitive operations inside of the ϕ-conditional expression. The reason duplicating primitive operations is not as severe as the duplication of functional calls when using the list monad is because functions whose control flow is independent of their arguments can be executed once, instead of once for every argument.

For example, the expression $(\lambda\ x\ .\ (x, x))(\phi\ ?\ e_1\ :\ e_2)$ reduces with standard beta substitution (*st-beta*) to:

$$(\phi\ ?\ e_1\ :\ e_2,\ \phi\ ?\ e_1\ :\ e_2)$$

The approach of the list monad would be to re-evaluate this function for e_1 and e_2 separately. If e_1 and e_2 are themselves partially concrete, the function would need to be evaluated for each of the possibly exponential number of concrete arguments using the list monad approach, but just once using our approach. This is the key idea behind why Smten performs better than the list monad in Haskell.

We discuss in Sec. 8.4.2 how we can canonicalize partially concrete expressions to avoid duplication in the primitive operations as well.

Search-Space Evaluation Figure 8.7 gives a new set of rules for evaluating search-space expressions to set expressions. Note that the rules here do not properly describe the strictness properties

$$
\begin{array}{lrcl}
\textit{sx-empty} & \texttt{empty} & \to_{sx} & \{\bot \mid \texttt{false}\} \\
\textit{sx-single} & \texttt{single}\ e & \to_{sx} & \{e \mid \texttt{true}\} \\
\textit{sx-union} & \texttt{union}\ \{e_1 \mid \phi_1\}\ \{e_2 \mid \phi_2\} & \to_{sx} & \{v\ ?\ e_1\ :\ e_2 \mid \texttt{ite}\ v\ \phi_1\ \phi_2\}, \quad v \text{ fresh} \\
\textit{sx-map} & \texttt{map}\ f\ \{e \mid \phi\} & \to_{sx} & \{f\ e \mid \phi\} \\
\textit{sx-join} & \texttt{join}\ \{s \mid \phi\} & \to_{sx} & \phi\ ?\ s\ :\ \texttt{empty} \\
\textit{sx-phi} & \phi\ ?\ \{e_1 \mid \phi_1\}\ :\ \{e_2 \mid \phi_2\} & \to_{sx} & \{\phi\ ?\ e_1\ :\ e_2 \mid \texttt{ite}\ \phi\ \phi_1\ \phi_2\}
\end{array}
$$

$$
\textit{sx-union-a1}\quad \frac{s_1 \to_{sx} s_1'}{\texttt{union}\ s_1\ s_2 \to_{sx} \texttt{union}\ s_1'\ s_2} \qquad \textit{sx-union-a2}\quad \frac{s_2 \to_{sx} s_2'}{\texttt{union}\ s_1\ s_2 \to_{sx} \texttt{union}\ s_1\ s_2'}
$$

$$
\textit{sx-phi-a1}\quad \frac{s_1 \to_{sx} s_1'}{\phi\ ?\ s_1\ :\ s_2 \to_{sx} \phi\ ?\ s_1'\ :\ s_2} \qquad \textit{sx-phi-a2}\quad \frac{s_2 \to_{sx} s_2'}{\phi\ ?\ s_1\ :\ s_2 \to_{sx} \phi\ ?\ s_1\ :\ s_2'}
$$

$$
\textit{sx-map-a}\quad \frac{s \to_{sx} s'}{\texttt{map}\ f\ s \to_{sx} \texttt{map}\ f\ s'} \qquad \textit{sx-join-a}\quad \frac{s \to_{sx} s'}{\texttt{join}\ s \to_{sx} \texttt{join}\ s'} \qquad \textit{sx-pure}\quad \frac{s \to_{e} s'}{s \to_{sx} s'}
$$

Figure 8.7: SAT-Based search-space evaluation.

we need for search. In this section we focus on what set expressions are constructed from search-space descriptions, and later we discuss how the runtime should generate these expressions to properly preserve strictness of search.

To understand why these rules produce correct set expressions, it is often helpful to view set expressions as the canonical form of partially concrete search spaces instead of as a set of expressions.

sx-empty The primitive `empty` reduces to $\{\bot \mid \texttt{false}\}$ by the rule *sx-empty*. The boolean formula `false` has no satisfying assignments, so $\{\bot[\mu] \mid \texttt{false}[\mu] = \texttt{true}\}$ represents the empty set.

Interpreting the set expression as the canonical form of a partially concrete search-space expression gives:

$$\{\bot \mid \texttt{false}\} = \texttt{false}\ ?\ \texttt{single}\ \bot\ :\ \texttt{empty}$$

We use $\bot$ for the body of the set expression, but any value with the proper type could be used instead, because the body is unreachable. (We leverage this fact for an important optimization discussed in Sec. 8.4.2).

sx-single The primitive `single` e reduces to $\{e \mid \texttt{true}\}$ by the rule *sx-single*. The boolean formula `true` is trivially satisfiable. If e is a concrete expression, this represents a singleton set, because $e[\mu]$ is the same for all μ.

As with `empty`, treating the set expression as a canonical form makes sense:

$$\{e \mid \texttt{true}\} = \texttt{true}\ ?\ \texttt{single}\ e\ :\ \texttt{empty}$$

Note that the argument e to `single` does not need to be evaluated to put the expression `single` e in set expression form. The expression e may describe a non-terminating computation, but that has no effect on whether e may be returned as a result of a search.

sx-union The expression (`union` $\{e_1 \mid \phi_1\}$ $\{e_2 \mid \phi_2\}$) reduces to the set expression $\{v \ ?\ e_1 : e_2 \mid \texttt{ite}\ v\ \phi_1\ \phi_2\}$ by *sx-union*, where v is a fresh boolean variable.

The variable v represents the choice of which argument of the union to use; an assignment of $v = \texttt{true}$ corresponds to choosing an element from the first set, and $v = \texttt{false}$ corresponds to choosing an element from the second set. The formula $\texttt{ite}\ v\ \phi_1\ \phi_2$ is satisfied by satisfying assignments of ϕ_1 with $v = \texttt{true}$ and satisfying assignments of ϕ_2 with $v = \texttt{false}$.

For example, consider the following `Space` expression representing the set $\{1, 5\}$:

```
union (single 1) (single 5)
```

This evaluates to the set expression:

$$\{v \ ?\ 1 : 5 \mid \texttt{ite}\ v\ \texttt{true}\ \texttt{true}\}$$

If this were the top-level search space, the SAT solver would be free to assign the variable v to either `true` or `false`, selecting between values 1 and 5 respectively.

In contrast, consider the following `Space` expression:

```
union (single 1) empty
```

This evaluates to:

$$\{(v \ ?\ 1 : \bot) \mid \texttt{ite}\ v\ \texttt{true}\ \texttt{false}\}$$

The only satisfying assignment of the $\texttt{ite}\ v\ \texttt{true}\ \texttt{false}$ is with $v = \texttt{true}$, so the value 1 must be selected.

This `sx-union` rule is the primary means of introducing partially concrete expressions during evaluation.

sx-map The expression `map` f $\{e \mid \phi\}$ reduces to $\{f\ e \mid \phi\}$ by *sx-map*. The map reduction applies the function f to the body of the set expression, leaving the formula unchanged.

The `map` primitive can lead to arbitrary application of functions to partially concrete expressions in the body of a set expression.

sx-join The expression `join` $\{s \mid \phi\}$ reduces by the rule *sx-join* to $\phi \ ?\ s : \texttt{empty}$. This reduction is easy to understand interpreting the set expression $\{s \mid \phi\}$ as the canonical form $\phi \ ?\ \texttt{single}\ s : \texttt{empty}$. Applying the `join` operator to both branches of the ϕ-conditional expression leads to:

$$\phi \ ?\ \texttt{join}\ (\texttt{single}\ s) : \texttt{join}\ \texttt{empty}$$

Both branches trivially reduce, resulting in:

$$\phi \ ?\ s : \texttt{empty}$$

This expression must be further reduced before reaching canonical form.

sx-phi A ϕ-conditional `Space` expression of the form ϕ ? $\{e_1 \mid \phi_1\}$: $\{e_2 \mid \phi_2\}$ is reduced to the set expression $\{\phi$? e_1 : $e_2 \mid$ `ite` $\phi\ \phi_1\ \phi_2\}$ by *sx-phi*. As with `join`, using intuition about the meaning of a partially concrete set expression helps to understand why this is correct if it is not immediately clear.

The term ϕ ? $\{e_1 \mid \phi_1\}$: $\{e_2 \mid \phi_2\}$ is equivalent to:

$$\phi \text{ ? } (\phi_1 \text{ ? } \texttt{single}\ e_1 \text{ : } \texttt{empty}) \text{ : } (\phi_2 \text{ ? } \texttt{single}\ e_2 \text{ : } \texttt{empty})$$

This can be compressed to the form:

$$\texttt{ite}\ \phi\ \phi_1\ \phi_2 \text{ ? } \texttt{single}\ (\phi \text{ ? } e_1 \text{ : } e_2) \text{ : } \texttt{empty}$$

which is equivalent to $\{\phi$? e_1 : $e_2 \mid$ `ite` $\phi\ \phi_1\ \phi_2\}$.

8.4.1 Avoiding Spurious Non-Termination

The reduction rules *sx-phi-a1*, *sx-phi-a2* are used to evaluate both branches of a ϕ-conditional set expression to set expressions before the rule *sx-phi* can be applied. It is somewhat surprising that these rules can result in non-termination for some search spaces that are completely finite. Consider the following search space:

```
search $ do
  x ← union (single True) (single False)
  if (x || not x)
     then single x
     else let y = y in y
```

This describes a search for a boolean value x that is either `True` or `False`. In either case, the condition for the `if` is `True`, so the search should always follow the `then` branch. The `else` branch, which here is an infinite recursion, is unreachable.

This search is entirely finite. The list monad can produce the exhaustive list of both solutions: `True` and `False`. In our implementation, this evaluates to an expression such as:

$$(x \vee \neg\, x) \text{ ? } (\texttt{single}\ (x \text{ ? } \texttt{True} \text{ : } \texttt{False})) \text{ : } (\texttt{fix}\ (\lambda y.y))$$

Because we rely on the SAT solver to evaluate the condition in this expression, the implementation does not look to see that in every case the condition is `true`, so it evaluates both branches of the condition, one of which will never reduce to a set expression.

It is not unrealistic to expect that we could determine the second branch is unreachable with some simplifications to the condition in this example. In general, though, determining whether a branch is unreachable is as hard as determining whether a boolean formula is satisfiable.

One simple approach to avoid evaluating set expressions on unreachable paths is to call the SAT solver for every condition to determine whether a branch is reachable or not. There are two downsides to this approach:

- The SAT solver is called for every condition, which we expect to be prohibitively expensive
- This approach does not help any with searches involving reachable non-terminating paths. An implementation using this approach would be overly strict in evaluation of `search` as described in Sec. 8.3.2

We take an alternative approach that heuristically detects long running computations and uses an abstraction-refinement procedure to search for elements of the search space that do not require the long running computations be completed. This approach leads to many fewer calls to a SAT solver, and results in an implementation of `search` with our desired strictness properties.

In our implementation, we evaluate set expressions non-strictly (contrary to the rules presented in Figure 8.7); set expressions are forced only when we need to compute the formula part of the set expression. A consequence of this is we have to concern ourselves only with potential non-termination when looking at the formula being constructed, and not in the constructions for set expressions.

Our goal, then, is to solve the problem of satisfiability for a boolean formula where sub-terms may not be fully computable. The high-level approach is to search first for satisfying assignments to the formula that are not affected by the non-computable parts of the formula.

For example, consider the boolean formula:

$$\phi = \texttt{ite}\ x_0\ \hat{\phi}\ x_1$$

where x_0 and x_1 are boolean variables and $\hat{\phi}$ represents a part of the formula that appears to be uncomputable. Even without knowing the value of formula $\hat{\phi}$, it is possible to find a satisfying assignment for ϕ: take x_0 to be `false` and x_1 to be `true`.

Our runtime heuristically detects sub-terms of formulas that are long-running computations. Rather than compute their values, the runtime treats these sub-terms as black boxes. In the previous example, $\hat{\phi}$ is an example of a black-box sub-term. Given a formula ϕ annotated with black box sub-terms, we generate three new formulas: p, a, and b.

The formula p corresponds to the condition under which the black boxes of ϕ do not affect its value. In the above example, $p = \neg\ x_0$, because if x_0 is `false`, the value of $\hat{\phi}$ is irrelevant.

The formula a is the same as formula ϕ under the assumption all the black box sub-terms are irrelevant. The black box sub-terms can be replaced with any value, including `true`, `false`, or any other value leading to a simple boolean formula for a. In the above example, if $\hat{\phi}$ is unreachable in ϕ, then $\phi = x_1$, so we take $a = x_1$.

The formula b contains the black box sub-terms and is equivalent to ϕ for all assignments where p is `false`. In the above example, $b = \hat{\phi}$.

The meaning of each of these terms can be summarized with the logical equality $\phi = \texttt{ite}\ p\ a\ b$. In other words, a is a finite approximation to ϕ, the approximation is exact when p holds, and b is used to refine the approximation.

The following abstraction-refinement semi-decision procedure makes use of this construction to progressively refine the black box parts of a boolean formula ϕ until a solution is found. If there exists a solution to the search, it will be found eventually. The procedure is:

1. Construct the terms p, a, and b for ϕ.
2. Check if $p \wedge a$ is satisfiable. If so, the assignment is also a satisfying assignment for ϕ and we are done.
3. Otherwise, check if $\neg\ p$ is satisfiable. If not, then ϕ is unsatisfiable and we are done. In this case, the formula b is unreachable, so none of the black box sub-terms need to be computed.
4. Otherwise, b may or may not have a solution and we refine our abstraction, but may restrict it with the knowledge that the cases we have considered previously may be ruled out. That is, we repeat this procedure with $\neg\ p \wedge b$.

8.4.2 Optimizing Smten Compilation

Many of the optimizations required to make a tool developed with Smten work well in practice are specific to the tool and can be expressed in the user's code. There are a handful of important optimizations, however, which are built into Smten. Whereas user-level optimizations lead to changes in the high-level structure of the generated queries, the optimizations built into Smten focus on reducing the cost of generating those queries. This is achieved primarily by exploiting sharing and pruning parts of the query that have no effect.

Characterizing the impact of these optimizations is difficult. Combinatorial search problems, by their nature, are sensitive to scaling: small changes to the implementation can effect performance by orders of magnitude. In our experience, this is the difference between a tool that works well in practice and one that fails to work entirely. Nevertheless, we attempt to describe the broad impacts of our optimizations in this section and present some empirical results in Sec. A.3.

Normal Forms for Partially Concrete Expressions

The rules *st-beta-phi*, *st-fst-phi*, *st-snd-phi*, and *st-case-phi* for pure evaluation of partially concrete expressions push their corresponding primitive operations inside the branches of a ϕ-conditional expression. This duplicates work in query generation and leads to redundancy in the generated query. We can eliminate this duplication by normalizing partially concrete expressions to share structure in subexpressions, so that the primitive operations are evaluated only once. The consequence of normalization is that none of the rules that duplicate primitive operations will ever be applicable.

Products The normal form for a partially concrete product is a concrete product with partially concrete components:

$$\begin{aligned} & \phi \ ? \ (e_{11}, e_{12}) \ : \ (e_{21}, e_{22}) \\ \rightarrow \quad & (\phi \ ? \ e_{11} \ : \ e_{21} \ , \ \phi \ ? \ e_{12} \ : \ e_{22}) \end{aligned}$$

This eliminates the need for *st-fst-phi* and *st-snd-phi*.

Booleans The normal form for a boolean expression is a ϕ-conditional expression whose left branch is `True` and right branch is `False`: ϕ ? `True` : `False`. As a short-hand, we use ϕ to represent the normal form of the boolean expression, and apply the following rewrite:

$$\phi \ ? \ \phi_1 \ : \ \phi_2 \rightarrow \texttt{ite} \ \phi \ \phi_1 \ \phi_2$$

In effect, Smten boolean expressions are translated to boolean formulas and handled directly by the SAT solver. An analogous approach can be used for types with direct support in the SMT backend, as discussed in Sec. 8.4.2.

Sums Sum types can be viewed as a generalization of the boolean case. They are reduced to the canonical form ϕ ? `inl` e_l : `inr` e_r using:

$$\begin{aligned} & \phi \ ? \ (\phi_1 \ ? \ \texttt{inl} \ e_{l1} \ : \ \texttt{inr} \ e_{r1}) \ : \ (\phi_2 \ ? \ \texttt{inl} \ e_{l2} \ : \ \texttt{inr} \ e_{r2}) \\ \rightarrow \ & \texttt{ite} \ \phi \ \phi_1 \ \phi_2 \ ? \ (\texttt{inl} \ \phi \ ? \ e_{l1} \ : \ e_{l2}) \ : \ (\texttt{inr} \ \phi \ ? \ e_{r1} \ : \ e_{r2}) \end{aligned}$$

This allows us to replace the *st-case-phi* rule with one that does not duplicate the work of the case expression:

$$\texttt{case}\ (\phi\ ?\ \texttt{inl}\ e_1\ :\ \texttt{inr}\ e_2)\ f_1\ f_2 \to \phi\ ?\ (f_1\ e_1)\ :\ (f_2\ e_2)$$

Functions The normal form for functions are functions:

$$\phi\ ?\ f_1\ :\ f_2 \to \lambda x\ .\ \phi\ ?\ (f_1\ x)\ :\ (f_2\ x)$$

This eliminates the need for *st-beta-phi* and creates additional opportunities for sharing when the same function is applied multiple times.

Leverage Unreachable Expressions

There are many places in the implementation where unreachable expressions are created. For example, the rule *sx-empty* reduces `empty` to $\{\bot \mid \texttt{false}\}$, where $\bot$ could be any value because it is unreachable. By explicitly tracking which expressions are unreachable, we can simplify our queries before passing them to the solver. For instance, $\phi\ ?\ e\ :\ e_x$ can be simplified to e if e_x is known to be unreachable, because if e_x is unreachable, ϕ must be `true` for every assignment.

Recall that our scheme to avoid spurious non-termination (Sec. 8.4.1) introduces arbitrary values for black box subterms. Using explicitly unreachable expressions instead has the effect of selecting the value that simplifies the approximation the most.

Exploit Theories of SMT Solvers

Smten can naturally leverage SMT solvers by expanding the syntax of formulas for types directly handled by the solver and using a formula as the canonical representation for these types as we do for booleans. For example, consider the following boolean expression involving integers:

$$(\phi_1\ ?\ 1\ :\ 2) < (\phi_2\ ?\ 2\ :\ 3)$$

Without support for integers, the `<` operator must be pushed into the branches of the ϕ-conditional for evaluation:

$$\begin{aligned}\phi_1\ ?\ (\phi_2\ ?\ (1 < 2)\ &:\ (1 < 3))\\ :\ (\phi_2\ ?\ (2 < 2)\ &:\ (2 < 3))\end{aligned}$$

This exponential duplication of work can be avoided by representing the integers and operations on them in an SMT formula. In this case, we translate `<` to the SMT solver's `lt`:

$$\texttt{lt}\ (\texttt{ite}\ \phi_1\ 1\ 2)\ (\texttt{ite}\ \phi_2\ 2\ 3)$$

To fully exploit an underlying theory we must have direct access to free variables of the corresponding types. Our search primitives encapsulate this for boolean variables:

```
free_Bool :: Space Bool
free_Bool = union (single True) (single False)
```

There is no direct way to introduce a free variable of a different type, however. For this reason, we add new search primitives that create free variables directly if supported by the SMT solver. For example, for theories of integers and bit vectors, we introduce the primitives:

```
free_Integer :: Space Integer
free_Bit     :: Space (Bit n)
```

Consider the query generated for the search space:

```
do x ← free_Bit2
   guard (x > 1)
```

The `free_Bit2` function could be implemented in terms of existing `Space` primitives:

```
free_Bit2 :: Space (Bit 2)
free_Bit2 = do
  x0 ← union (single 0) (single 1)
  x1 ← union (single 0) (single 1)
  single (bv_concat x0 x1)
```

This would lead to a generated SMT query such as:

$$(\texttt{bv\_concat}\ (\texttt{ite}\ x_0\ 0\ 1)\ (\texttt{ite}\ x_1\ 0\ 1)) > 1$$

If `free_Bit` is provided as a primitive, however, then `free_Bit2` can use it directly as the implementation. This leads to a simpler generated SMT query: $x > 1$, where x is a free bit-vector variable in the SMT query.

Providing `free_Bit` and `free_Integer` requires modifying the implementation of Smten. This is required only for primitive Smten types. Theories that can be expressed in terms of currently supported SMT theories in Smten can be expressed directly in the Smten language by composition of the Smten primitive theories.

Formula Peep-Hole Optimizations

We perform constructor oriented optimizations to simplify formulas as they are constructed. In each of the following examples, the evaluation of lazily applied functions for constructing the formula ϕ_x can be entirely avoided:

$$\phi_x \wedge \texttt{false} \rightarrow \texttt{false}$$
$$\texttt{ite false}\ \phi_x\ \phi_y \rightarrow \phi_y$$
$$\texttt{ite}\ \phi_x\ \phi_y\ \phi_y \rightarrow \phi_y$$

Sharing in Formula Generation

Consider the user-level Smten code:

```
let y = x + x + x
    z = y + y + y
in z < 0
```

It is vital we preserve the user-level sharing in the generated query, rather than generating the query:

$$((x + x + x) + (x + x + x) + (x + x + x)) < 0$$

To prevent this exponential growth, we are careful to maintain all of the user-level sharing in the query when passed to the solver, including dynamic sharing that occurs due to user-level dynamic programming, memoization, and other standard programming techniques.

Chapter 9

Automated testing

This chapter outlines a number of automated testing techniques that we have applied to CHERI. A central theme of this work is the use of *executable specifications*: not only do such specifications help us to better understand the problem at hand, but they provide oracles to rigorously test against, as well as the ability to search for simple failure cases during debugging.

9.1 CHERI-Litmus

A memory consistency model, provided as part of a multi-processor's ISA, defines the allowed behaviors of a set of program threads acting on shared memory, and is a prerequisite for the development of well-defined concurrent programs. Memory consistency models are important because they can enable efficient implementations of cache-coherent shared memories while still providing certain guarantees needed by the programmer.

Litmus tests [1] are a way of testing whether certain behaviors (perhaps disallowed by a memory consistency model) are actually observable in hardware, thereby exposing possible bugs. To illustrate, here is a litmus test known as the *store buffer* test (or "SB" for short).

```
MIPS SB
{ 0:r2=x; 0:r4=y;
  1:r2=y; 1:r4=x; }
 P0              |    P1          ;
 li r1,1         |    li r1,1     ;
 sb r1,0(r2)     |    sb r1,0(r2) ;
 lb r3,0(r4)     |    lb r3,0(r4) ;
exists (0:r3=0 /\ 1:r3=0)
```

On the first line, we have a name for the test followed by a precondition stating: (1) registers `r2` and `r4` on thread 0 are initialized to the addresses of variables `x` and `y` respectively; (2) registers `r2` and `r4` on thread 1 are initialized to the addresses of `y` and `x` respectively. After the precondition, there are six MIPS instructions running on two threads (three instructions per thread). Thread 0 writes value 1 to `x` and then reads `y` into `r3`. Thread 1 writes value 1 to `y` and then reads `x` into `r3`. Finally, the postcondition states that the value of `r3` on each thread is 0. This particular test looks to see if stores are non-blocking, i.e., that store instructions can complete before the values

being stored are visible to other threads. This is the hallmark of the TSO (total store order) model typically found on X86 machines.

Once we have a litmus test, we can use the standard litmus tool [1] to automatically convert it to a multi-threaded executable program that repeatedly (1) establishes the precondition, (2) synchronizes all threads, and (3) executes the instructions. On each iteration, variability is introduced by various means, e.g., randomizing the addresses of shared variables and putting random delays between instructions. On termination, the test reports whether or not the postcondition was ever observed.

The standard litmus tool generates POSIX-compliant code and hence requires an operating system to be loaded in order to run tests. However, (1) booting an OS in a hardware simulator is slow; (2) debugging is simpler when the software running is small and self-contained; (3) the OS may not boot successfully due to bugs that the tests are aiming to expose!

This led us to the development of CHERI-Litmus (a variant of the standard litmus tool), which generates code for bare-metal (without the need to run an OS). We have used it to test CHERI in simulation using a wide-range of litmus tests produced by DIY (the litmus test generator tool). Although our tool is specific to CHERI, it should be straightforward to be ported to other architectures, and we plan to make it publicly available soon.

9.2 Axe

While Litmus tests check whether or not particular memory behaviors are observable in hardware, on their own they do not define a memory consistency model. For that, we developed a tool called Axe that can check arbitrary memory traces against a range of different models. Although Axe has been developed to test CHERI's memory subsystem, it can in principle be applied to any memory subsystem.

The specific problem solved by Axe is as follows. Given a trace consisting of one sequence of memory instructions per thread (including loads, stores, atomic read-modify-writes, and memory barriers), and the value returned by each load, determine whether this trace satisfies one of the SPARC consistency models: sequential consistency (SC), total store order (TSO), partial store order (PSO), or relaxed memory order (RMO).

Motivated by features present in CHERI's memory subsystem, we have generalized the SPARC consistency models in various ways, e.g., to permit cache hierarchies as well as caches that implement lazy invalidation and time-based invalidation. Axe also supports checking traces against these new models, and we are currently exploring the relationship between these new models and the popular POWER model [42]. Early results are encouraging, suggesting that the class of consistency models that can be checked using Axe is both rich and useful.

Axe is inspired by a previous tool called TSOtool [30], but has the following differences:

1. We support checking SC, PSO and RMO models as well as TSO. (We also support checking new models such as Lazy RMO and Hierarchical RMO.)

2. We use an existing general-purpose constraint solver rather than implementing our own custom solver. This simplifies the checker, making it easier to understand and extend.

3. We have developed both operational and axiomatic flavors of the models, and have checked them for equivalence. This gives us confidence that our models are indeed defining what we intend them to define.

4. Our checker is freely available and open source.

5. Early measurements suggest that the performance of our checker approaches the (impressive) published performance of TSOtool.

For details, we refer the reader to

```
http://github.com/CTSRD-CHERI/axe
```

9.3 BlueCheck

When developing large systems such as CHERI, it is helpful to test components *at a fine grain*: faults found in small self-contained modules are much easier to diagnose than whole-system faults. On the other hand, CHERI is comprised of hundreds of individual modules; writing test benches for each is time-consuming. This motivated us to develop a *generic* test bench called BlueCheck, which can assist in testing any hardware module written in the Bluespec SystemVerilog HDL. For details, we refer the reader to our MEMOCODE 2015 paper, "A Generic Synthesisable Test Bench" [34].

> *Writing test benches is one of the most frequently-performed tasks in the hardware development process. The ability to reuse common test-bench features is therefore key to productivity. In this paper, we present a generic test bench, parameterized by a specification of correctness, which can be used to test any design. Our test bench provides several important features, including automatic test-sequence generation and shrinking of counterexamples, and is fully synthesizable, allowing rigorous testing on FPGAs as well as in simulation. The approach is easy to use, cheap to implement, and encourages the formal specification of hardware components through the reward of automatic testing and simple failure cases.*

BlueCheck is available online at

```
https://github.com/CTSRD-CHERI/bluecheck
```

9.4 TEST_MEM

`TEST_MEM` involves the application of both BlueCheck and Axe to CHERI's memory subsystem: BlueCheck automatically stimulates the memory subsystem with random inputs, Axe checks the results, and BlueCheck shrinks any counterexamples found.

This test framework has proved extremely useful during a recent extensive refactoring of the memory subsystem by our implementation team, finding bugs on an almost daily basis. Most (but not all) of the bugs found were also exposable either by running our large suite of software unit tests, or by trying to boot FreeBSD. However, the main reason why our implementors prefer using `TEST_MEM` is that it gives far simpler counterexamples. Perhaps the most extreme example of this occurred in a version of the memory subsystem capable of booting FreeBSD, but which led to the following failure using `TEST_MEM`.

```
=== Depth 20, Test 82/10000 ===
setAddrMap(<13, 9, 3, 2>)
513: store(5,9)
516: load(8)
556: getResponse
557: load(9)
571: getResponse
571: Not equal: 0 v 5
```

It is quite surprising that an implementation containing a bug found by just five memory operations is capable of booting an OS involving millions of such operations! This particular bug arose from a cache prefetching feature. It is exposable only under certain conditions in which L2 cache lines are marked as invalid, and is hard to trigger once the L2 becomes populated. Typically, to diagnose such a bug, our implementation team will replay the test with cache debug messages enabled. This particular test leads to around ten debug messages. In contrast, one of our software unit tests that exposes the same bug leads to tens of thousands of debug messages. This highlights the value of searching for simple failures.

As another illustration of small failures found by `TEST_MEM`, here is a counterexample to sequential consistency:

```
=== Depth 10, Test 5/10000 ===
setAddrMap(<15, 11, 8, 5>)
Core 0: MEM[3] == 0
Core 0: MEM[7] := 8
Core 1: MEM[3] := 9
Core 1: MEM[7] == 0
Core 0: MEM[3] == 0
Not sequentially consistent
```

We believe this to be a minimal counterexample. Compared to software litmus testing, which can also tell us that our memory subsystem is not sequentially consistent, `TEST_MEM` does so while emitting far fewer cache debug messages – making it much easier to understand *why*.

Using `TEST_MEM`, we have checked various versions of our multi-threaded memory subsystem on millions of random test sequences, giving us a high degree of confidence in their observable behaviors.

9.5 `fuzz_cap`

`fuzz_cap` is a 600-line python script that generates random sequences of capability instructions (including capability manipulating instructions, capability field testing instructions and capability memory reading and writing instructions) and checks that the outputs of our BSV CHERI implementation match the outputs of our L3 CHERI model. It supports shrinking of counterexamples, helping to find concise failures. The script was used heavily during the recent addition of compressed capability support to CHERI, finding numerous errors both in the implementation and the model.

To illustrate, here is a counterexample found by the script:

```
dli $t0, 0x43f0
candperm $c0, $c0, $t0
ccleartag $c0, $c0
dli $t0, 0xfffffffffff1bae
ccheckperm $c0, $t0
```

This sequence of instructions led to a user-defined exception in BSV but a tag exception in L3. This turned out to be a bug in the implementation.

Other counter-examples include more capability instructions misbehaviors as well as some more subtle divergences such as in memory representation of capabilities.

Chapter 10

Future Directions

In our efforts to formally specify and verify the CHERI design, we have been active on multiple fronts that cover exploration of design choices, formal notations, and verification tools – as well as employing testing to validate our current implementation. We aim through these multiple efforts to prove various and diverse properties of the CHERI design and to produce an overall argument that allows us to gain confidence in the trustworthiness of our design and implementation, from the ground up. In this chapter, we consider potential directions to extend our work on full hardware verification via BSV, but also next directions for ISA-centered modeling and testing, language-level modeling and testing, as well as the potential for verification of larger software stacks grounded in these models.

10.1 A BSV Foundation for Hardware Verification

Our adoption of Bluespec SystemVerilog (BSV) as a basis for our development environment enables rapid prototyping through the use of modularization, abstraction, and automated support for synthesis, and simulation. It also naturally enables us to explore ways of applying automated tools and methodologies to validate designs written in the BSV language. Of critical importance to our trustworthiness argument is the verification of whether the ISA is implemented correctly, the microarchitecture is correct, and the capability unit behaves according to the specifications. These efforts require us to map the BSV language to a form that is amenable to formal analysis, that is, translating BSV into input languages for SRI's PVS formal tool suite – which itself encompasses model checkers (SAL and its successor SALLY) and SMT solvers (Yices2), as well as other formally based tools. Each of these tools has its own sweet spots, although all three have potential applications to formal analysis of hardware and software. Also, it is of particular importance to verify that the BSV compiler produces hardware that preserves the properties of the input description. (The entire SRI tool suite is source available: pvs.csl.sri.com .)

After a week-long meeting with Joe Stoy (Bluespec) earlier in the project, and in light of existing research (e.g., [41]), we determined that a strong link can be made between the BSV programming language and the SRI PVS suite of formal methods tools. PVS provides theorem proving capabilities that are very much suited for verifying refinement mappings between abstract hardware description representing a specification of our design, and concrete, more detailed, and complex descriptions representing an implementation of our design. The SAL/SALLY model checkers offer a direct evaluation of the operational semantics of the instruction set and the microarchitecture

and can be used for automated test case generation. It can be used to verify the correctness of the ISA implementation. Yices2 is also useful for identifying specific types of states that should normally be inaccessible. Our efforts in the future will involve translating the BSV implementation of the ISA and the specification described in both the MIPS documentation and the CHERI ISA reference manual to SAL/SALLY/Yices2 and proving their correspondence and satisfiability of desired properties. Our collaboration with Bluespec Inc. to enhance the intermediate output stages of the BSV compiler will ensure that these properties not only hold for the input BSV description but also for the hardware produced by the compiler.

We have made considerable progress in developing the desired tools and incorporating those tools specifically related to the analysis of BSV specifications for the hardware into the Bluespec SystemVerilog compiler chain. However, as the CTSRD project has progressed, we have not been able to perform the desired formal analyses, although we have made very productive inroads into formally generated test cases. Despite only early testing of the waters in proven hardware design, pragmatic applications of formal models of the ISA, memory models, and so on, in automated testing have had a substantial impact on the effectiveness of the overall project, and were directly supported by (and in some cases, entirely dependent on) use of BSV.

We are currently seeking additional funding sources to enable the desired formal analyses that could be conducted during the final two years of the CTSRD project, as the CHERI hardware and system architecture continue to mature – particularly in tracking the emerging changes for our hardware and software technology transfer efforts.

If this effort were successful in analyzing the CHERI instruction-set architecture, its implementation, and some of the low-level software, we believe that the existence of a well-documented open-source bottom-up formally analyzed clean-slate hardware-software system would provide a demonstrably trustworthy system with unprecedented assurance. We also believe that no previous effort has ever been completed with the desired scope of addressing the hardware, low-level software, compilers, and applications.

10.2 Attaining Layered/Compositional Assurance

The ability to carry out bottom-to-top layered and compositional assurance has progressed greatly in the past forty years since the formally specified PSOS hardware-software co-design [38]. Layered assurance was pursued in HDM as a fundamental part of the PSOS project, and more recently supported by *abstract interpretations* in SRI's PVS verification system. Further considerations extending layered assurance to more general notions of compositional assurance have also progressed (e.g., see [35]). Many years later, the methodology and tools are now ready for the proposed system assurance. Heavily aided by the CTSRD project, our team has established the methodologies and formal analysis tools that we hope to be able to apply to CHERI hardware and some of its software. This section summarizes our recent advances – and how they might relate to efforts making compositional assurance arguments for the CHERI architecture.

- **Formal analysis within the BSV toolchain.** The BSV hardware compiler derives a Verilog specification from the BSV specification. We have augmented the BSV compiler to include the SRI formal tools within the BSV compilation. This enables desired properties of the hardware instruction-set architecture specification to be formally evaluated without having to install the compiled Verilog into an FPGA and test it.

- **Architectural extraction.** The SRI Architectural Extraction tool (Chapter 6 and [48]) will be fundamental to the analysis. It enables the extraction of a coherent subspecification (i.e., a proper subset) of a complete hardware specification in BSV in such a way that the subspecification is precise and sufficiently self-standing for the analysis of certain desired properties. The net effect is that the desired analysis becomes perhaps orders of magnitude simpler than a comparable analysis of the entire system. In this way, architectural extraction loses none of the critical dependencies, and also enhances hierarchically constructed assurance where the intermodule dependencies are explicitly covered by the analysis.

- **Modular Composable Proof Infrastructure.** Generally the modular decomposition of a hardware design's implementation is different than the decomposition of the formal proof used to show its correctness. This means that any design refinements completely invalidate the entirety of the proof destroying our ability to effectively do proof-based verification anywhere but post-design. However, by careful initial construction of theorems, proof designers can decompose their proof requirements to match the natural design modularity [48]. Once done, the designer has free reign to refine her design for performance and incur the module-level proof cost only when doing full verification. Our initial examples of this capture distributed shared-memory systems; we are planning on applying this to microprocessor, interconnects and other common highly-tuned hardware components.

- **Smten.** Smten [47] provides user-focused abstractions for SMT queries, allowing rapid modular development of formal tools. This occurs as a precursor to the use of the analysis tools embedded in the BSV chain. Smten is a higher-level query language that greatly simplifies direct invocation of the SRI formal analysis tools noted above, as an intrinsic part of the BSV-to-Verilog development process. It is directly applicable to analyses of the CHERI ISA and its extensions.

- **BCL co-design language**. BCL [15] involves an extension of the BSV language and a corresponding BCL compiler. The BCL language enables executable hardware and software specifications to be defined in a single language, and also enables the specification of a movable boundary between hardware and software. The BCL compiler accordingly enables compilation of the hardware specification into Verilog and the software specification directly (into C or C++). By deferring decisions about the exact placement of the hardware-software boundary until the implications for performance and security are better understood and formally analyzed, enormous flexibility can be gained in the overall system architecture.

- **The L3 ISA model language.** The L3 ISA domain-specific language (DSL) allows the declarative specification of an ISA to be compiled into an executable reference interpreter for program binaries targeting that ISA, as well as instruction and processor state definitions for a theorem prover (currently HOL4). We would like to extend the L3 language tool to target other theorem provers (specifically, the SRI theorem prover PVS, in which effort we would leverage the substantial interest of and input from the formal methods team at SRI) as well as BSV. The latter target would allow us to automatically generate simple reference single-cycle in-order hardware processors that can function as sources for equivalence checks against the highly optimized manually designed custom processors that implement the same ISA. We also take a strong interest in the successor Sail modeling language, which promises

closer links to memory concurrency models that will be important in understanding strong atomicity properties embedded in the CHERI ISA (e.g., as relates to tags).

- **Verifying the BSV Specification Language and Compiler**

  We would like to develop a formal model of the core of the BSV hardware description language, and its provably correct compilation to Verilog (following the example of CompCert for the C programming language). The objectives are two-fold: in addition to the valuable artifact of a provably correct hardware compiler, we expect the formal language model itself to prove valuable in the static analysis of hardware designs. Initial work towards this goal has been done by Rishiyur Nikhil from Bluespec, Inc, in the form of an executable specification in Haskell of the core BSV language dynamic semantics[39]. We would like to extend this model to a full formal model for the language as well as its compilation.

- **Verified Compilers.** *CakeML* [28] is a compiler for a subset of the ML language that has been formally verified using the HOL theorem prover. In version 1, CakeML produces code for Intel-compatible CPUs. The correctness proof of CakeML depends on the assumption that those CPUs do in fact behave in the way that is described by the formal model used in the proofs. It is planned that a future version of CakeML will also produce code for BERI (using just the MIPS ISA compatible instruction of BERI, not the capability instructions). When this is done, the formal model of the ISA that is used for the correctness proof of the compiler will be the same L3 model that we have used for automated testing of BERI/CHERI, giving a high degree of confidence that the model used in the proofs is an accurate description of how the CPU behaves.

- **Floating Point.** The L3 model of BERI/CHERI currently does not include floating point, although the CHERI1 Bluespec implementation does include an optional hardware floating point unit, and we have a hand-written test suite for the MIPS III and MIPS IV floating point ISAs. The L3 specification language only has limited support for floating point, and the CakeML language has no floating point at all. Previous work elsewhere has formally modelled IEEE-754 (or IEEE-854) floating point arithmetic in a theorem prover, for example Paul Miner in PVS [33] and Victor Carreño in HOL [40]. Future work in this area could extend the L3 language to have more complete support of IEEE-754 floating point; map the L3 language primitives to a HOL model of IEEE-754:2008; formally model BERI's floating point unit in L3; validate the L3 model of BERI's FPU against our existing floating point test suite; extend our trace comparison tool to be able to compare traces of floating point programs; extend Brian Campbell's automatic test generation so that it can generate floating point tests; extend the CakeML language to include floating point types; extend the CakeML backend to target BERI's hardware floating point unit; and finally extend the correctness proofs of the CakeML compiler to include floating point.

- **Multicore Trace Comparison.** Our current trace comparison tool can compare traces of the multicore L3 model and a BSV implementation if the BSV implementation has a sequentially consistent memory model. However, CHERI1 uses a more relaxed memory model for perfomance reasons; this is also the case for many commercial multicore CPUs. A research question is whether it is possible to combine the L3 model of the instruction semantics with a model of multicore memory coherence.

- **Trace comparison with interrupts and I/O**. Our current trace comparison tool can compare traces of FreeBSD booting up until the point at which interrupts are turned on. Trace comparison cannot continue beyond that point because the interrupts introduce non-determinism: for example, the exact clock cycle at which the CPU will receive a character typed by the user on the system console depends on components outside the CPU (most notably, the human user). Trace comparison with interrupts is theoretically possible by examining a trace of the hardware or BSV Bluesim simulation to find out when an interrupt occurred, and then replaying the interrupt at the same point inside the L3 model (i.e. the L3 model uses a hardware trace as an *oracle* for when the interrupts arrives). We have not implemented this yet.

- **Formalized Security Proofs.** During the initial design of the CHERI capability mechanism, security proofs were carried out by hand. A machine-checked proof would offer a higher level of assurance. There are several routes by which this might be done. (a) The SAL model (which includes only the capability instructions, and does not model the core MIPS ISA) can be tested against the BSV implementation and automatically translated into PVS using tools that were developed by the CTSRD project. This PVS specification could then be used as the basis of a fully formalized proof. Security proofs could also be carried out directly in SAL, but would have the limitation that the SAL model checked is less expressive than a full theorem prover such as PVS. (b) The L3 model can be automatically translated into HOL, and security proofs could be carried out in HOL.

- **Bluecheck.** The University of Cambridge has developed the Bluecheck tool [34]. to improve assurance of the CHERI architecture under MRC2. It is a generic synthesizable test bench parameterized by a formal specification of correctness. It can be used to test any hardware design implemented with BSV. This tool encourages the formal specification of hardware components and provides the rewards of automated testing, avoiding various failure cases, creating concise counterexamples, and providing simplified debugging.

- **Axe.** The University of Cambridge has also developed Axe, a tool that checks the implementation of the memory consistency model from the memory traces generated by the processors of a particular architecture. Axe incorporates the axiomatic and operational semantics of several common memory consistency models. It includes an automatic test-generation framework that can generate test instruction sequences, and analyze the resulting memory traces to check compliance against its axiomatic models. This tool uses SRI's Yices SMT solver to check constraint satisfaction.

- **RISC-V.** We have developed an L3 specification of the RISC-V ISA developed by UC Berkeley. Our specification is being used as a reference oracle in the verification of a hardware implementation of the ISA. We are now in the early stages of using this ISA specification along with processor hardware implementations in BSV provided by Bluespec Inc. as a platform for some of the work described above (specifically, architectural extraction, Smten, and L3 extensions). We also hope to port the capability instruction set from the CHERI architecture to RISC-V as an illustration of the essential portability of the CHERI design.

- **Capability-based C-language extensions.** The CHERI project has introduced extensions to the C language for capability-based memory-safety and compartmentalization, in order to

take advantage of the CHERI hardware facilities. We would like to formally specify these C-language extensions and integrate them into existing formal models of the C language (e.g., [32, 27]), to ensure that their semantics meshes well with the complex C datatype and memory model.

Cerberus implements formal models of both the strict ISO C semantics, including detecting uses of undefined behavior, and a "de-facto" model, which provides concrete instantiations of undefined and implementation defined behavior based on programmer expectations. We have begun working with Kayvan Memarian and Peter Sewell (the authors of Cerberus) to add a model of the CHERI C semantics, which defines trapping behavior for a number of cases that are undefined in both the ISO and *de facto* models.

We also participated in the design of the *de facto* model based on our work exploring idioms in existing C code. As an initial step towards modeling CHERI C behavior, we have been able to run the Charon test suite, developed as part of the Cerberus work, using the CHERI C compiler. Charon provides a number of test cases that highlight difference between standard-specified and programmer-expected behavior. A small number of the Charon tests are derived from the set of idioms that we identified during our exploration of CHERI C. We have identified a number of issues with the CHERI C compiler (and fixed many of them) as a result of this, and expect to use the test suite along with the CHERI-expected behavior to test the formal model.

We hope to be able to use this model, in conjunction with formal models of the ISA to validate two things. First, that a given set of CHERI-like extensions to a base ISA can express the same abstract machine to programmers in C and higher-level languages. Second, that a given compiler is performing valid translations, by bisimulating the ISA model and the C-level model and ensuring that the same pattern of memory accesses and traps occur.

These are both vital to technology transfer, providing confidence that other ISA designs that aim to provide the same benefits as CHERI actually do.

10.3 Targets for Formal Analysis

This section suggests some of the illustrative types of properties that would be appropriate for formal analysis relating to the CHERI ISA, and eventually to the CHERI low-level system architecture. Many of the properties that might be considered are discussed in the CHERI Protection Model chapter of the latest version of the CHERI Instruction-Set Architecture report, and reflected here.

10.3.1 CHERI Capability Mechanism

The value of the CHERI capability mechanism hinges on ISA level operation faithfully enforcing key security properties. These properties (listed below) are all of the form of preventing certain behaviors from appearing. These are naturally understood as safety properties and solved via model checkers and SMT solvers rather than strict-sense formal verification of correctness.

- Nonbypassability of the capability mechanism
- Nonforgeability of capabilities

- Integrity of the tags
- Consistency and integrity of metadata protected by the tag mechanism, which adds metadata to pointers:
  - Bounds checks
  - Permission checks
  - Capability flow
  - Ephemerality and revocation
- Integrity of the compression mecahnism used for the 128-bit capability CHERI ISA variant
- Integrity of virtual-memory separation
- Integrity and type consistency of object capabilities
- Integrity, noncompromisibility, and other properties above that relate to multicore CHERI, and similarly to computers with multiple microprocessors, input-output devices, and other active devices including many with direct memory access

We would like to ensure that the semantics of the individual instructions, as well as their combinations as used in low-level calling conventions and application binary interfaces generated by compilers and linkers, preserve these safety properties.

10.3.2 CHERI Hardware and Infrastructure

As with the specification, we are interested in the preservation of the safety properties of the capability mechanism, but this time by the processor hardware. In theory, the same ISA-level verification checks would apply naturally. However, as the complexity of the implementation dwarfs that of the specification, it is far more practical to show a correspondence between the implementation and the specification. This can be decomposed as follows:

- The Memory Subsystem – This subsystem contains the majority of the nondeterminism in the system. By virtue of the parameterized extensibility of the whole system organization, we think it is more practical to verify the safety properties via a theorem-proving approach, and then replace the complex system with an idealized model capturing the memory's nondeterminism. This would allow us to maintain synchronization between design modularity and proof modularity while compartmentalizing the details of the CPU and memory implementations using designer-oriented modularization [48], and enable architectural changes with only incremental proof work.
- The CHERI CPU pipeline – Showing that a single pipeline is faithful to the ISA-level model is tantamount to showing that the microarchitectural techniques to increase parallelism and improve circuit quality are implemented correctly. This is exactly what is captured by our architectural extraction efforts, as they would provide an instruction-level model of the design.

- Interrupt Handling – Due to its implementation-specific variations, interrupt handling is generally represented as natural-language text in specifications, and not specified in a formally verifiable way. Even with our verified simplifications of the processors and memory subsystems, we still must verify that interrupts are handled correctly. Given the difficulty in generating inputs, a practical solution would use an automated test suite to exhaustively check all behaviors.

- Test-Suite Infrastructure – For practical microarchitectural exploration, it is important that during rapid design changes we can quickly get an effective model of the system behavior. To this end, it is valuable to have a coverage-aware test-suite generation. This can be done both at the ISA-level (as we have done with our fuzz-based test suite) and via concolic test generators [24, 43, 9, 29] – whereby symbolic executions and concrete executions run in parallel.

- Bluespec-to-Verilog Compilation – The CHERI hardware implementation representation with high-level rules is automatically compiled down into a Verilog-based FSM. While significant effort has gone into the compiler to verify that the output FSM is a faithful restriction of the "one-rule-at-a-time" semantic model, it is possible that the compiler may introduce an error, either in the metaprogrammatic de-sugaring of rules to the bit-level or in the scheduling-level restriction where rules are narrowed down. The former requires a full certification of the Bluespec compiler front-end. The latter is relatively straightforward, and requires showing only that the FSM cycle step is bisimilar to the steps of the rule-level schedule [16].

10.3.3 Low-Level Software

In general, asynchrony is a pervasive problem in hardware, but seemingly much more difficult to manage in operating systems. With respect to applications of formal methods, it appears that analysis of software is even more challenging than analysis of hardware. Here are just a few of the issues relating to operating systems and compilers.

- Integrity and correct separation of compartmentalization in operating systems and application software, based on CHERI hardware properties

- Correctness of LLVM transformations and compiler limitations, such as Spatial Safety, Temporal Safety, software compartmentalization, and data-pointer protection – particularly as dependent on the CHERI hardware properties

- Portability of the CHERI C abstract machine across different ISA implementations

- Correctness of the auxiliary tools, such as the BSV compiler (e.g., consistency of the BSV spec with Verilog with no hidden hidden unspecified functionality), and Bluecheck

10.3.4 Total-system Properties

Bringing to bear the notions of layered and compositional assurance noted in Section 10.2, we believe the time is ripe for future attempts to carry out analysis of total-system properties as well as the hardware and the low-level software.

System security, human safety, and other emergent properties would have to addressed as the results of lower-level compositions of the hardware and software and analysis of the compositions involving the application software, from the bottom up. Correctness of the formally based tools (e.g., theorem provers, model checkers, SMT solvers, and proof checkers), compilers, and so on must also be considered – as they also present opportunities for compromise of the desired overall total-system properties.

Appendix A

Using Smten

In this section, we provide a case study considering the challenge of implementing HAMPI [26], a string-constraint solving tool implemented using a conventional SMT solver, and then compare the design task for a corresponding tool implemented using Smten.

A.1 A String Constraint Solving Algorithm

For this string constraint problem, the goal is to synthesize a bounded-size string based on a template that belongs to the language of a given context-free grammar (CFG) or regular expression. For instance, given the template `ab??cd`, where `?` stands for an arbitrary character, and given the regular expression `y(z*(ab)*)* | (((ab)*(cd)*)*)`, the tool may synthesize either the string `ababcd` or `abcdcd`. Intuitively, HAMPI constructs an SMT query by unrolling the regular expression match computation over a symbolic string input. This involves encoding the notion of strings and regular expressions at the level of the data types and operations the SMT solver understands, *e.g.,* booleans and bitvectors, and translating the result back to the high-level string.

To make this practical, the implementation must do two major things. First, it must avoid unrolling obviously false paths. In the above example, the first alternative can never match as every string from the template necessarily starts with an `a`, not `y`. Capturing this property frequently reduces the size of queries by an order of magnitude. Second, the implementation must effectively leverage any sharing in the CFG-match. In this example, the substring match for `cd`, the last two characters in the string template, is the same whether the outermost Kleene star in the second alternative is repeated once or twice. Effectively noting these sharing points and leveraging this in the construction of a SMT problem can cause an exponential reduction in the query size for complex query matches.

The original implementation of HAMPI requires about 20K lines of Java and uses the STP [23] SMT solver. Much of this code deals with the intermediate query representation and optimizations on it. Given its size, it is not surprising that the resulting code, while well written, is relatively hard to understand and modify.

A.2 Implementing String Matching in Smten

In contrast to the original HAMPI implementation, the Smten-based reimplementation, called SHAMPI, is around 1000 lines of code including lexing and parsing for the string constraint lan-

guages. This is an order of magnitude reduction in size. Most of this reduction comes from having the low-level optimizations automatically done in Smten, allowing us to focus on representing the core algorithmic steps in creating a solution. The full code can be found at:

`https://github.com/ruhler/smten-apps/tree/master/shampi`

As short as it is, the code is still too long to display here. Instead, we show in Figure A.1 a simplified subset of the code which leaves out the details of parsing and the sharing optimization phase. The latter partially reduces the CFG-match against a fixed-sized string to a regular expression match, using a shared hash to capture all sharing in the search of submatches, where submatches have the form of a CFG-match against a concrete fixed substring (*e.g.,* the third character to the seventh character of the string template). The omitted code is relatively straightforward and does not provide significant additional insight into the Smten design process, as it can be done as a purely non-search optimization and is equivalent to a pure Haskell implementation.

As is natural in a functional language, regular expressions are represented using an algebraic data type (Line #1). Here, we implicitly make use of sum, product, and recursive types.

The `match` function (Line #4) uses Haskell's pattern matching and higher-order functions to test whether a string matches a regular expression. Note that the description for `match` is exactly the same as one written in standard Haskell. Further, the Smten `match` can be used directly with *concrete* strings, whose length and characters are all fully known, with no additional cost due to its ability to be used symbolically.

The string template is described using an algebraic data type `Template` (Line #15). A template is either `Str` for a concrete string, `Cat` for the concatenation of strings formed from two templates, or `Free` for a variable string whose length ranges between the given bounds and whose characters may be any value.

The function `candidates` (Line #29) takes a template and returns a search-space computation of all candidate strings defined by that template. The first clause of the `candidates` function handles the `Str` case and produces a singleton search space with the concrete string `s`. The second clause handles the `Cat` case by concatenating the two search spaces defined by the sub-templates `ta` and `tb`. The final clause instantiates a free string via the helper functions `choose` and `replicateM`. The `choose` function (Line #18) converts a list of elements into a search space by recursively constructing a union of singleton spaces of each list element. The function `replicateM` (which is part of the standard library in Haskell) replicates its arguments the given number of times and returns the cross product of all possible results (Line #22). For example, the search space described by the expression `replicateM 2 [‘a’.. ‘c’]` represents the set of 9 strings `"aa"`, `"ab"`, `"ac"`, `"ba"`, `"bb"`, `"bc"`, `"ca"`, `"cb"`, and `"cc"`. Thus, the `do`-notation for the third clause of `candidates` can be interpreted as meaning: for each possible sequence `chars` of `h` characters, each character of which is one of the characters from `‘a’` to `‘z’`, and for each possible width `w` between `l` and `h`, take `w` characters from `chars` and include that string in the search space.

The function `solutions` (Line #40) takes a string template and a regular expression and produces the search space of all candidate strings from the template that match the regular expression. It does this by first calling `candidates` to produce the search space containing all strings generated by the template, then uses `do`-notation, which desugars into a `join` of a `map`, to map each non-matching string into the empty set and each matching string into a singleton set space, and join the results. This results in a unified search space which contains only those strings that matched the regular expression.

The function `solve` (Line #45) calls `search` to evaluate the search, and prints the solution

found, if any.

An important point about this implementation of string constraint solving is that the description of regular expressions and regular expression matching is direct, reusing standard Haskell code that operates on concrete regular expressions and strings to describe search over a space of strings (and possibly a space of regular expressions, though that is not shown here).

A.3 Evaluating the Implementations

Shampi is a Smten-based reimplementation of Hampi, a string constraint solver whose constraints express membership of strings in both regular languages and fixed-size context-free languages.

A Hampi input consists of the definitions for regular expressions and context free grammars, bounded-size string variables, and predicates on these strings referencing the regular expressions and grammars. The output from Hampi is a string that satisfies the constraints or a report that the constraints are unsatisfiable.

The Hampi tool has been applied successfully to testing and analysis of real programs, most notably in static and dynamic analyses for SQL injections in web applications and automated bug finding in C programs using systematic testing. Hampi's success is due in large part to preprocessing and optimizations that recognize early when a CFG cannot match a string, regardless of the undetermined characters the string contains.

The original implementation of Hampi is implemented in about 20K lines of Java and uses the STP [23] SMT solver. In contrast, our implementation of Shampi is around 1K lines of Smten code, including the lexer and parser for the string constraint language. This is an order of magnitude reduction in code size.

We capture Hampi's algorithmic sharing using a straight forward memoization technique. The numbers presented here are an improvement of previously published work [47] in that the whole of Shampi is now implemented in the Smten language.

Our initial development of Shampi required approximately three weeks of effort, including the effort required to understand the tool and implement the important optimizations required to run effectively. We revisited the implementation a year after the initial development, and were happy to find we could still understand the code, and we were able to identify and implement some additional optimizations easily.

We ran both Hampi and Shampi on all benchmarks presented in the original Hampi paper. We experimented with two different backend solvers for Shampi: STP, which is the same backend solver used by Hampi, and the Yices2 [54] SMT solver, which performs notably better. Figure A.2 compares the performance of Hampi and Shampi for each of the benchmarks. Those points below the 45 degree line are benchmarks on which Shampi out-performs the original Hampi implementation. For both Shampi and Hampi, we took the best of 8 runs. Shampi was compiled with GHC-7.6.3. We ran revision 46 of a single Hampi server instance for all runs of all tests on Hampi to amortize the JVM startup cost; this also allows code specialization to happen in later instances of each input, which biases the results slightly in favor of Hampi.

Assuming our implementation includes the same algorithms and optimizations for generating the queries, as we believe to be the case, the left graph in Fig. A.2 represents the overheads associated with using the more general Smten framework for solving the string constraint problem. The right graph represents the benefits the framework provides in requiring a trivial amount of effort to experiment with different backend solvers.

```
1  data RegEx = Empty | Epsilon | Atom Char
2   | Star RegEx | Concat RegEx RegEx | Or RegEx RegEx

4  match :: RegEx → String → Bool
5  match Empty        _  = False
6  match Epsilon      s  = null s
7  match (Atom x)     s  = s == [x]
8  match r@(Star x)   [] = True
9  match r@(Star x)   s  = any (match2 x r) (splits [1..length s] s)
10 match (Concat a b) s  = any (match2 a b) (splits [0..length s] s)
11 match (Or a b)     s  = match a s || match b s
12 match2 a b (sa, sb)   = match a sa && match b sb
13 splits ns x           = map (λn → splitAt n x) ns

15 data Template =
16   Str String | Cat Template Template | Free Int Int

18 choose :: [a] → Space a
19 choose [] = empty
20 choose (x:xs) = union (single x) (choose xs)

22 replicateM :: Int → Space a → Space [a]
23 replicateM 0 s = single []
24 replicateM n s = do
25   hd ← s
26   tl ← replicateM (n-1) s
27   single (hd:tl)

29 candidates :: Template → Space String
30 candidates (Str s) = single s
31 candidates (Cat ta tb) = do
32   a ← candidates ta
33   b ← candidates tb
34   single (a ++ b)
35 candidates (Free l h) = do
36   chars ← replicateM h (choose [‘a’..‘z’])
37   w ← choose [l..h]
38   single (take w chars)

40 solutions :: Template → RegEx → Space String
41 solutions t r = do
42   s ← candidates t
43   if (match r s) then single s else empty

45 solve :: Template → RegEx → IO ()
46 solve t r = do
47   result ← search (solutions t r)
48   case result of
49      Nothing → putStrLn "No Solution"
50      Just x → putStrLn ("Solution: " ++ show x)
```

Figure A.1: String constraint solver implemented with Smten.

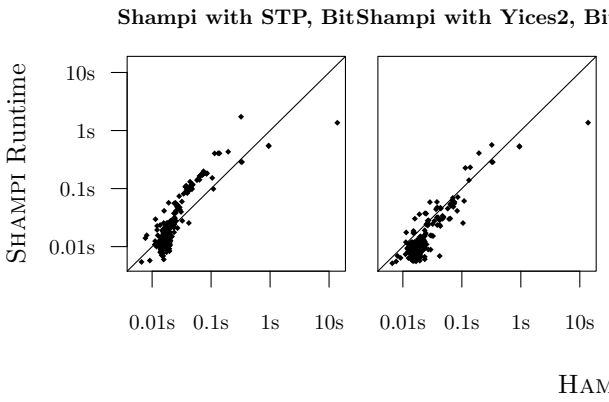

Figure A.2: Hampi compared to Shampi.

Bibliography

[1] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, TACAS'11/ETAPS'11, pages 41–44, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Jonathan Anderson, Robert N. M. Watson, David Chisnall, Khilan Gudka, Brooks Davis, and Ilias Marinos. TESLA: Temporally Enhanced System Logic Assertions. In *Proceedings of The 2014 European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, April 2014.

[3] Arvind and Xiaowei Shen. Using Term Rewriting Systems to Design and Verify Processors. *IEEE Micro*, 19(3):36–46, May 1999.

[4] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 825–886. IOS Press, 2009.

[5] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 1999.

[6] R. Bisbey II, J. Carlstedt, and D. Chase. Data dependency analysis. Technical Report ISI/SR-76-45, USC Information Sciences Institute (ISI), Marina Del Rey, California, February 1976.

[7] R. Bisbey II and D. Hollingworth. Protection analysis: Project final report. Technical report, USC Information Sciences Institute (ISI), Marina Del Rey, California, 1978.

[8] R. Bisbey II, G. Popek, and J. Carlstedt. Protection errors in operating systems: Inconsistency of a single data value over time. Technical Report ISI/SR-75-4, USC Information Sciences Institute (ISI), Marina Del Rey, California, December 1975.

[9] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *LNCS*, pages 2–23. Springer, 2013.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[11] Brian Campbell and Ian Stark. Randomised testing of a microprocessor model using SMT-solver state generation. In *Formal Methods for Industrial Critical Systems (FMICS 2014)*, number 8718 in LNCS. Springer, 2014.

[12] J. Carlstedt. Protection errors in operating systems: Validation of critical conditions. Technical Report ISI/SR-76-5, USC Information Sciences Institute (ISI), Marina Del Rey, California, May 1976.

[13] J. Carlstedt, R. Bisbey II, and G. Popek. Pattern-directed protection evaluation. Technical Report ISI/SR-75-31, USC Information Sciences Institute (ISI), Marina Del Rey, California, June 1975.

[14] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[15] Nirav Davé. *A Unified Model for Hardware/Software Co-design.* PhD thesis, MIT, Cambridge, Massachusetts, 2011.

[16] Nirav Davé, Arvind, and Michael Pellauer. Scheduling as Rule Composition. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.

[17] Nirav Davé, Michael Katelman, Myron King, Arvind, and José Meseguer. Verification of Microarchitectural Refinements in Rule-based Systems. In *Proceedings of MEMOCode 2011*, Cambridge UK, 2011. `ctsrd/website/cam/pdfs/201107-memocode2011-verification.pdf`.

[18] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[19] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[20] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, and Natarajan Shankar. Integrating verification components. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, and Experiments*, number 4171 in LNCS, 2008.

[21] Saar Drimer. *Security for Volatile FPGAs.* PhD thesis, University of Cambridge, 2010.

[22] Anthony Fox. Directions in ISA specification. In *Interactive Theorem Proving*, 2012.

[23] Vijay Ganesh and David Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer Berlin / Heidelberg, 2007.

[24] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI 05: Programming Language Design and Implementation*, pages 213–223. ACM, 2005.

[25] D. Hollingworth and R. Bisbey II. Protection errors in operating systems: Allocation/deallocation residuals. Technical report, USC Information Sciences Institute (ISI), Marina Del Rey, California, June 1976.

[26] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *Proceedings of the 18th international symposium on Software testing and analysis*, ISSTA '09, pages 105–116, New York, NY, 2009. ACM.

[27] Robbert Krebbers and Freek Wiedijk. A typed C11 semantics for interactive theorem proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 15–27, New York, NY, USA, 2015. ACM.

[28] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191, 2014.

[29] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE)*, pages 416–426. ACM, 2007.

[30] C. Manovit. *Testing memory consistency of shared-memory multiprocessors.* PhD thesis, Standford University, 2006.

[31] K.L. McMillan. Interpolation and SAT-based model checking. In Jr. Hunt, WarrenA. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 2003.

[32] Kayvan Memarian, Kyndylan Nienhuis, Justus Matthiesen, James Lingard, and Peter Sewell. Cerberus: towards an executable semantics for sequential and concurrent C11. In *HCSS 2015: Fifteenth Annual High Confidence Software and Systems Conference*, 2015.

[33] Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Report 110167, NASA Langley Research Center, June 1995.

[34] Matthew Naylor and Simon Moore. Bluecheck: A generic synthesisable test bench. In *13th ACM-IEEE International Conference on Formal Methods and Models for System Design – MEMOCODE*, Austin, TX, September 2015. ACM-IEEE. `www.cl.cam.ac.uk/research/`.

[35] Peter G. Neumann. Principled assuredly trustworthy composable architectures. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, December 2004. http://www.csl.sri.com/neumann/chats4.html, .pdf, and .ps.

[36] Peter G. Neumann and Donn B. Parker. A summary of computer misuse techniques. In *Proceedings of the Twelfth National Computer Security Conference*, pages 396–407, Baltimore, Maryland, 10–13 October 1989. NIST/NCSC.

[37] Peter G. Neumann and Robert N. M. Watson. Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In *Fourth Layered Assurance Workshop*, Austin, Texas, December 2010. U.S. Air Force Cryptographic Modernization Office and AFRL. http://www.csl.sri.com/neumann/law10.pdf.

[38] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980. 2nd edition, Report CSL-116.

[39] Rishiyur Nikhil. BSV Formal Semantics, 2015.

[40] Victor A. Carre no. Interpretation of IEEE-854 floating-point standard and definition in the HOL system. Technical Report 110189, NASA Langley Research Center, September 1995.

[41] Dominic Richards and David Lester. A monadic approach to automated reasoning for bluespec systemverilog. *Innovations in Systems and Software Engineering*, pages 1–11, 2011. 10.1007/s11334-011-0149-0.

[42] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. ACM.

[43] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE 05: Foundations of Software Engineering*. ACM, 2005.

[44] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125, London, UK, UK, 2000. Springer-Verlag.

[45] Xiaowei Shen, Arvind, and Larry Rudolph. Cachet: an adaptive cache coherence protocol for distributed shared-memory systems. In *International Conference on Supercomputing*, pages 135–144, 1999.

[46] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 404–415, New York, NY, USA, 2006. ACM.

[47] Richard Uhler and Nirav Davé. Automatic translation of high-level symbolic computations into SMT quer ies. In *Proceedings of the 25th International Conference on Computer-Aided Verification (CAV 2013)*, St. Petersburg, Russia, July 2013. `http://people.csail.mit.edu/ndave/Research/2013_cav_smten.pdf`.

[48] Muralidaran Vijayaraghavan, Arvind, Adam Chlipala, and Nirav Davé. Modular deductive verification of multiprocessor hardware designs. In *Proceedings of the 27th International Conference on Computer-Aided Verification (CAV 2015)*, San Francisco, July 2015. `http://people.csail.mit.edu/ndave/Research/cav_2015_seqconsistency.pdf`.

[49] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *WOOT Workshop*, Gaithersburg, Maryland, 2007. USENIX Security. http://www.watson.org/ robert/2007woot/20070806-woot-concurrency.pdf.

[50] Robert N. M. Watson. New Approaches to Operating System Security Extensibility. Technical report, Ph.D. Thesis, University of Cambridge, Cambridge, UK, October 2010.

[51] Robert N. M. Watson, David Chisnall, Brooks Davis, Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, and Jonathan Woodruff. Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide. Technical Report UCAM-CL-TR-877, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, November 2015.

[52] 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. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-876, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, November 2015.

[53] Robert N. M. Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, Robert Norton, and Michael Roe. Bluespec Extensible RISC Implementation: BERI Hardware reference. Technical Report UCAM-CL-TR-868, University of Cambridge, Computer Laboratory, April 2015.

[54] http://yices.csl.sri.com/index.shtml, August 2012.