AUTOMATED DEDUCTIVE VERIFICATION OF SYSTEMS SOFTWARE

BY

EDGAR PEK

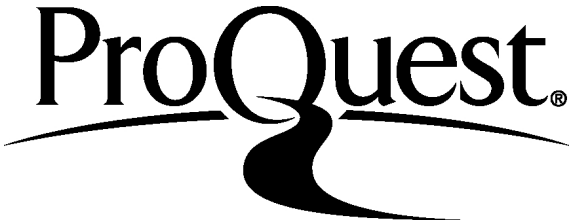
DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Professor P. Madhusudan, Chair and Director of Research
Dr. Thomas Ball, Microsoft Research
Professor Samuel Talmadge King
Professor Grigore Rosu



Abstract

Software has become an integral part of our everyday lives, and so is our reliance on his correct functioning. Systems software lies at the heart of computer systems, consequently ensuring its reliability and security is of paramount importance. This thesis explores automated deductive verification for increasing reliability and security of systems software. The thesis is comprised of the three main threads. The first thread describes how the state-of-the art deductive verification techniques can help in developing more secure operating system. We have developed a prototype of an Android-based operating system with strong assurance guarantees. Operating systems code heavily relies on mutable data structures. In our experience, reasoning about such pointer-manipulating programs was the hardest aspect of the operating system verification effort because correctness criteria describes intricate combinations of structure (shape), content (data), and separation. Thus, in the second thread, we explore design and development of an automated verification system for assuring correctness of pointer-manipulating programs using an extension of Hoare's logic for reasoning about programs that access and update heap allocated data-structures. We have developed a verification framework that allows reasoning about C programs using only domain specific code annotations. The same thread contains a novel idea that enables efficient runtime checking of assertions that can express properties of dynamically manipulated linked-list data structures. Finally, we describe the work that paves a new way for reasoning about distributed protocols. We propose *certified program models*, where an executable language (such as C) is used for modelling – an executable language enables testing, and emerging program verifiers for mainstream executable languages enable certification of such models. As an instance of this approach, concurrent C code is used for modelling and a program verifier for concurrent C (VCC from Microsoft Research) is used for certification of new class of systems software that serves as a backbone for efficient distributed data storage.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

We are finally rising to the grand challenge of Verified Software Initiative [94] thanks to 30 years of advances in logic, programming languages theory, automated deductive verifiers, proof-assistant software, decision procedures for theorem proving, and finally the in the "raw" computational power. One of the goals of this thesis has been to contribute to the Verified Software Initiative – a fifteen year scientific challenge of large-scale software verification. Software is complex and flawed, a recent study by University of Cambridge estimates that the cost of debugging to $312 billion per year [44]. Thus, we would like to reason about program correctness and ultimately certify that the behavior of a program adheres to formal specification. In this study we embark on an deductive reasoning approach proposed by Floyd and Hoare [81, 92]. The key ingredient in this approach are program logics, i.e., rules for reasoning about the behavior of programs. In this thesis we study how can extend, automate, and apply the logical foundations provided by Floyd and Hoare to systems software verification to facilitate development of reliable and secure systems.

Higher-level applications build on functionality provided by the lower-level systems software, such as operating systems and database management systems. So, we would like the systems software to be reliable and secure. Reliable systems software guards users from frustrations and potential data loss caused by the infrastructure systems crashes. Secure systems software defends against the network attacks that exploit flaws in lower-layer systems software for various nefarious activities. Despite the recent advances we cannot declare victory – systems software still requires steady supply of updates [1] to ensures reliable and secure functioning. In this thesis we focus on verification of systems because it is the cornerstone of any large-scale verification effort.

1.1 Code Annotations

The application of Floyd-Hoare's methodology relies on a well-annotated program code. Thus, in this section we study with *formal* code-level annotations that are used to describe *behavior* of system components. These annotations comprise of preconditions, postconditions, invariants, and program assertions that provide programmers with a way to express the intended behavior of the systems being developed. The focus of the work described in this thesis

[1] "Patch Tuesdays" ensure timely updates to Microsoft's systems software.

has been on modular verification which requires specification of the *interface* between program components and the specification of components' behavior.

Following the notation introduced in [88], we also refer to these annotations as behavioral interface specifications. Such specifications are useful by itself, for example when precisely documenting program behavior, for guiding implementation, to clarify interface between components developed by different (teams of) developers, etc. However, the full power of such specifications becomes evident when the code annotations are used in conjunction with automated analysis and program verification techniques. In this thesis the interface specification are used in conjunction with deductive program verification to capture application specific semantic properties (in Chapter 2 we focus on security invariants, and in Chapter 5 on consistency properties), and for full formal program verification (see Chapter 3 for specification and verification of program that manipulate heap allocated data structures). Also, we use the code annotations in conjunction with run-time checking for detection of code vulnerabilities (in Chapter 4 we show how to effectively and efficiently detect flawed linked-list manipulations).

Behavioral specification Behavioral specifications [88] provide a precise description of the planned behavior of a systems or its components. Some of the types of behavioral specifications are:

- preconditions – describe intended values that a components must handle,
- assertions – impose constrains of variable values at particular program points,
- postconditions – describe functionality of the component by relating input to output values,
- state machines – provide a high-level view of system states and transition between those states,
- sequence charts – summarize how a component interacts with other components or its environment,
- use cases – describe how users with different roles may interact with the system.

Modern software development relies on use of specifications. In particular, developers extensively uses existing component frameworks and libraries. For example, GLib [84] is low-level system library used by a variety of systems (e.g., Gnome database) that provides advanced data structures, such as doubly and singly linked list, balanced binary trees, etc. Specifying interfaces of these libraries is useful for declaring software contracts [130] that are precise definitions of the functionality provided by the libraries and calling conventions that client need to follow to ensure correct functioning. Software contracts are also useful for specifying intended functionality and facilitate better understanding, because the code itself is prone to various interpretations of what abstraction the software is supposed to implement.

Moreover, the large scale of the software systems requires functional decomposition to enable distributed development. Imprecise (or non-existent) interface specifications can hinder development and lead to bugs, cost overruns, or increased time to market. Clear interface specifications (through software contracts) provide a mechanism for clean system partitioning, precise definition of the intended behavior, effective division of labor and communication among teams, smoother integration of the system components, etc. Furthermore, the pervasiveness of software systems encourages extended use. However, vendors have to provide timely updates to keep-up with the pace of innovation. Advances in execution platforms, implementation languages, frameworks, or libraries can be hard to accommodate in a large system. Specifications can ensure implementation-independent descriptions of system behavior. They provide a high-level view of the system which can make development less dependant on the lower-level implementation details, and easier adoption of hardware and software innovations.

Safety critical systems, such as avionics, and systems supporting critical infrastructure, such as smart grid, increasingly rely on software to provide their functionality. Verification is an important mechanism to increase reliability and trustworthiness in such systems, and specifications provide a language to define the intended functionality whose implementation can be verified.

Formal specification language Specifications are particulary interesting when they are written in a mathematically precise notation – formal specification language. A formal specification languages is characterized with formal syntax and semantics. Formalizing the syntax of the specification languages is the first and often critical step towards its integration into a formal reasoning tool. Formalizing the semantics through mathematical logic is perhaps specification's raison d'être because it provides automated reasoning about specifications with respect to their associated code.

Formal specifications can be used throughout the software development life-cycle. In the design phase, a higher-level model can be automatically checked against a specification using automated deduction and model-checking techniques. This thesis presents an approach in which a properties of eventually consistent distributed data stores are specified, modeled and verified (see Chapter 5). Moreover, specification are critical for correct-by-construction process of stepwise refinement (see e.g., [20]) during which specifications are systematically refined into code.

During implementation phase, static analysis tools can be check implementations against specifications. For example, contract-based modular checkers can be used to assure that a method body correctly implements its contract, that it satisfies each callee's precondition, and that the assertions at each program point hold (see e.g., [80, 63, 58]). Modular checkers can have different design goals. One of the most important decision is whether the checker is sound (used for verification) or unsound (used for bug-finding). For example an early tool ESC/Java was focused on finding bugs. Tools such as, HAVOC and CodeContracts also aim at bug-finding, and improve expressiveness of properties they can check. For example, HAVOC focuses on finding bugs in heap-manipulating systems code (see [24, 63]).

The other point in the decision space are sound modular checkers [58, 118, 99]. The formal underpinnings of the contract-based modular checkers are described in section on Floyd-Hoare logic below (see section 1.3). This first part of this thesis (see chapter 2 and chapter 3) describes research contributions in the verification domain.

Specifically, an important facet of the work described in the first chapter is that *formal specifications do not have to specify full functional correctness to be useful*. As described in the chapter, light-weight annotations can be specify and verify important security properties of a large system software. On the other side, as described in chapter 3, for low-level software libraries, full functional verification is desirable because it can be used a building block for a wide range of clients.

Besides static analysis, we can check executable representations at run-time, (which is the topic of the chapter 4). Other synergies between specification and run-time analyses are test case generation, and model based testing in which implementations can be exercised directly from specifications. Also, the code generated from specifications can be used for run-time monitoring of system's execution, as well as a fault-recovery mechanism.

Formal specifications can be used in the various phases of the the software development process, such as, requirements, architecture, and implementation. The work described in this thesis deals exclusively with the specification languages that describe implementation decisions for the program modules. These kinds of specifications have been introduces as *interface specifications* [172]. The name reflects the characteristics of these specifications, namely the interface specification provides a client with the information needed to use the module without knowing its implementation. At the same time, it gives the developer of the module a precise description the functionality that the module needs to provide, without knowing possible clients. However, interface specification have also been used by Object Management Group to describe weak, informal specifications which describe module at the syntactic level (i.e., through method names, parameter types, etc.). Cheon and Leavens [53] introduced the term *behavioral interface specification* to emphasize crucial behavioral aspects such as precondition and postcondition.

1.2 Automating Program Verification

A range of methods have been used to reason about software correctness. A model checking method is based on temporal logics specification, abstract interpretation (cite Cousot, ASTREE) is the foundation of program analysis methodologies that represent program execution in a symbolic setting of abstract domains. Furthermore, CEGAR-based approaches combine model checking and abstract interpretation to yield a practical approach for increasing software reliability. This methodology was used in the first commercially successful and perhaps the most notable example of using program verification techniques to improve real-world world systems software: the SLAM approach [23]. The SLAM has been shipped as part of the Static Device Verifier (SDV), and SDV has been part of

the Microsoft's platform for developing device drivers: Window's Driver Kit (WDK) [132] (previously known as Windows Driver Development Kit – DDK).

Most of the approached summarized above are based on *operational reasoning*. The essence of this kind of reasoning is analysis of the execution traces of the given program. The problem with this class of approaches is that the number or size of the traces can be prohibitively large or unbounded (in the case of the program manipulating dynamically allocated data structures). The only way to effectively analyze all the traces is to generate an *abstraction* of the program execution traces (or its state space). An abstraction partitions the original program executions into the number of subsets that constitutes an abstract representation. However, the abstract representation may be too coarse to be useful, or the exploration of the abstraction might still be infeasible.

1.3 Floyd-Hoare Logic

A different approach is based on axiomatic reasoning (see [81, 92]). The essence of this approach is a representation using *logic*. A language of the logic provides a way to express or specify program executions or properties. A natural choice of the logic is the first-order logic, which was used in the earliest work based on Floyd-Hoare approach (e.g., [74]). The formulas of first-order logic used in the context of program verification are *assertions*. As detailed in section 1.1, code annotations are used to describe desirable system behaviors. The reasoning engine, i.e. a proof system in a Floyd-Hoare logic, consists of axiom and proof rules that provide basic building blocks used to prove that a given program satisfies the desired properties. A proof in this setting is based on an induction on the program structure.

A crucial step in the inductive Floyd-Hoare style reasoning is the provision of *loop invariants*. Intuitively, a loop invariant can be thought of as a formula in a logic (e.g., a quantity of some sort) that remains unchanged during the execution of the loop body. However, in the context of the Floyd-Hoare style reasoning the loop invariant means something more specific: an *inductive* invariant, informally, it is an invariant that can be proved correct using a proof by induction. Thus, an inductive loop invariant I satisfies two conditions: (a) I holds when first entering the loop, and (b) if I is true at the loop entry then it remains true across the loop body. When the two conditions hold, a terminating execution of the loop will result with the state in which the invariant I and the loop's exit condition hold.

Formally, the following proof rule defines the correctness requirement for a *while* loop (essentially any looping construct can be reduced to the *while* loop).

$$\frac{\{C \wedge I\}\ \text{body}\ \{I\}}{\{I\}\ \textbf{while}(C)\ \text{body}\ \{\neg C \wedge I\}}$$

The proof rule works as long as the loop terminates. Thus, the proof correctness is *partial correctness* proof. In thesis, we are *mostly* concerned with the partial correctness proofs. The exception is exposition in chapter 5, in which

the correctness conditions also require proofs of termination. We approach proofs of termination in a usual way, that is by introducing a *loop variant*. A notion of loop variant was introduced in Floyd's paper [81], and it essentially represents a quantity that decreases with each iteration of the loop. This quantity is defined using a *ranking function* whose range is is a well-founded set. Because there are *no* infinite strictly decreasing sequences in a well-founded set, a variant expression that representing a ranking function implies termination. In our case, ranking function ranged over the set of natural numbers.

Finding suitable loop invariant is often the most intellectually challenging step in verification, because it often requires coming up with the fundamental insight into the nature of the computation.

Besides loop invariants another important class of annotations are data structure (data type, object) invariants. These invariants are employed to capture structuring techniques often employed in well-written code and enable reasoning on a more abstract level. Advantages of the abstract specification over implementation-dependent specifications are: (a) abstraction specification are independent of implementation level changes, thus proof of correctness about clients relying on abstract specifications are unaffected, and (b) Abstract specifications are usually written using a mathematical notation (e.g. sets, sequences, etc.) which is both simpler to understand because it hides implementation level details and easier to reason because of lemmas and tactics already implemented in a verification engine. Essentially, the goal of data structure invariants is to represent a more higher-level view of the system components. The specification of the component can be defined using the abstract view, which needs to be connected to with the actual implementation.

1.3.1 Reasoning about Heap-allocated Data Structures

Floyd-Hoare logic provides a paradigm for reasoning about program, however it does not provide technical means to reason about realistic programs written in real programming languages. In particular, an important (and hard) problem is how to extend Floyd-Hoare logic to reason about heap manipulating programs. This problem has been studied in multiple contexts (see e.g., [37]). A more recent class of solutions are founded on separation logic (e.g. [31] and other works). Even though a lot of progress has been made, it remains unclear how practical is it to apply the current techniques for programs written in commonly used programming languages (such as C, Java, etc.). Thus, one of the research questions posed by this thesis: can we make reasoning about heap manipulating programs practical for C? We choose C because it is still one the most commonly used programming languages for development of systems software. Furthermore, heap allocated data structures, such as trees, and lists are prevalent in systems software.

Separation logic is an extension of Hoare's logic for reasoning about programs that access and manipulate heap-allocated data. The main goal of the logic is to simplify the reasoning about shared mutable data structures. The use of shared mutable data structures is prevalent in systems software [39]. For example, an implementation of dynamic

storage allocator requires complex manipulation of shared mutable data structure (for details see the paragraph below). Reasoning about heap manipulating have been studied since 1970s (see e.g., Burstall's work [48]. However, the devised methods are either limited or prohibitively complex, and scale poorly to programs of moderate sizes. One of the key reasons why shared mutable data structure manipulations are hard to handle with conventional logics is that by default everything is shared, while exactly the opposite holds for programming. Thus, declaring all of the instances where sharing does *not* occur is counterintuitive and oftentimes extremely tedious.

An example of systems code with mutable shared data structures: dynamic storage allocator

The basic interface to dynamic storage memory allocator consists of `malloc` and `free` functions. The implementation of the dynamic storage memory allocator maintains a list (e.g., singly-linked list) of free blocks that serve for handling memory requests. The list of free blocks is null-terminated list of possibly non-contiguous memory blocks. Each block in the list contains a header consisting of two words. The first word stores a pointer to the next free block, and the second stores the size of the block. The allocated block pointer returned to the user program points to the usable space in the block, and the header is always carried along with the allocated block (but it is not pointed to). The blocks in the list are stored in the order of increasing addresses. The requests for allocation are served according to the first-fit policy. If there is no big-enough block in the list if free blocks, or if the list of free blocks is empty, then `malloc` requests more memory from the operating system. When a user calls `free` on a memory block, the memory block is inserted into the appropriate position into the list of free blocks. [176]

The main primitive of separation logic is the *separating conjunction* $P * Q$, which denotes that P and Q hold in two *separate* portions of the heap. This primitive facilitates introduction of the most important program-proof rule for modular reasoning about the programs – the *frame rule*:

$$\frac{\{P\}\ \mathrm{C}\ \{Q\}}{\{R*P\}\ \mathrm{C}\ \{R*Q\}}\ FV(R)\cap MOD(C)=\emptyset$$

Using the frame rule we can extend reasoning about *local specification* that only consists of the variables and heap locations that may be modified by C (i.e., $MOD(C)$ – the *footprint* of C) by adding arbitrary predicates (R) about variables and heap locations ($FV(R)$) that are not modified by C. The frame rule capture the essence of local reasoning:

> "To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that program actually accesses. The value of any other cell will automatically remain unchanged." [141]

Many important data structures are defined recursively. Thus, when reasoning about programs that manipulate such data structures, we employ inductively-defined predicates that define the data structures.

1.4 Contributions of the Thesis

We study challenges in automated deductive verification of systems software. The three main aspects studied in this context are: (a) applying a program verifier while building of a realistic operating system, (b) a deductive program verification framework and a run-time checking prototype for reasoning about heap manipulating programs written in C programming language, and (c) a novel paradigm for reasoning about complex systems demonstrated on distributed systems protocols.

The key contributions of the work in this thesis are detailed below.

EXPRESSOS We have developed a new OS that provides high assurance security mechanisms for application security policies. Moreover, we have *verified* the implementation of the mechanisms that our OS uses to enforce security policies. The EXPRESSOS is designed to safely handle legacy hardware, it includes a programming language and run-time system for building our operating system, and a *verified* kernel implementation to assure the desired security invariants.

Our verification approach is pragmatically focused on a set of security invariants – *not* aimed on full functional correctness. Thus confirming that the verification does not have to be "all or nothing", i.e., one does not need to show full functional correctness to reap the benefits of formal methods. We focused on seven security invariants covering secure storage, memory isolation, user interface (UI) isolation, and secure IPC. The verified implementation of these invariants in EXPRESSOS assures that (a) processes can isolate their state and ensure its integrity, (b) achieve confidentiality to a certain degree, and (c) shield run-time events from malicious applications running on the same device. We do *not* show non-interference, in particular we cannot prevent side-channel attacks. For pragmatical reasons we combine deductive verification (we used BOOGIE-based tools) with abstract interpretation (we used Code Contracts static analysis). As expected, the workhorse of our verification effort was deductive approach. The power of this approach comes with the price having to provide ghost code annotations annotations that track and update mathematical abstractions and heap-related frame annotations. Overall, the verification of the invariants above requires only a modest annotation effort ($\sim$2.8% annotation overhead).

To evaluate the security and the performance of EXPRESSOS, we have built a prototype system that runs on x86 hardware and exports a subset of the Android/Linux system call interface. To evaluate security, we have examined 383 recent vulnerabilities listed in CVE [62]. Our OS successfully prevents 364 of them. To evaluate performance, we have used an ASUS Eee PC and run a web browser on top of EXPRESSOS. Our experiments show that the performance of our OS is comparable to Android with 16% overhead for the web browsing benchmark.

Automated Reasoning about Mutable Data Structures in C using Separation Logic. We extended *natural proof technique*, proposed by Qiu et al. [147]. The key technical contribution is the *lifting* of natural proof techniques from the verification-condition-generation level to the code-level: we automatically synthesize *annotations* at the code-level so that VCC interpreting these will automatically search for natural proofs of the properties it verifies. VCC is a tool for C programs manipulating data-structures where only annotations of pre/post conditions and loop invariants are required. The tool is designed to make most reasoning of data-structures completely automatic, without the use of programmer-provided tactics and lemmas. However, when a proof fails, the programmer can still see and interact with the DRYAD specifications, its translation to first-order logic, and the automatically generated tactics to help the proof go through, since all our effort explicitly resides at the code-level. The proof for any sound but incomplete system is in the pudding. Our main hypothesis going into the evaluation was that we can make natural proofs work for a complex language like C by exerting enough control through annotations at the code level. We give evidence to this hypothesis by developing the tool and experimenting with it on more than 150 C programs manipulating data-structures. The tool VCDRYAD was able to prove *all* the programs correct, automatically, without any user-provided proof tactics.

Abstraction-guided Runtime Checking of Assertions on Lists Following the previously describe work, we study the problem of expressing and efficiently run-time checking assertions over list-segments, lists, and the way they merge or remain separate, when dynamically allocated and manipulated by a program using pointers. Immediate motivation comes from the need to debug or understand the annotations or while they fail on the code being checked. A more general motivation is that mainstream programming languages lack any standard assertion logic for reasoning about program that manipulate heap allocated data structures. We develop efficient runtime checking for a *declarative and logical* specification language for expressing properties of list data-structures.

The main contribution is an idea of *abstraction-guided runtime assertion checking*. The idea is to maintain an abstraction of the concrete heap dynamically, as the program executes, which will maintain certain structural properties of the heap symbolically. This abstraction obviates the need to dynamically check crucial and expensive properties, like separation, on the concrete heap.

We implement both assertion checking techniques for specifications that express properties of lists, list-segments, and separation— the one that evaluates assertions on the concrete heap and the technique based on abstraction-based runtime checking. We evaluate both techniques on a suite of 25 programs, including both library code manipulating lists and client code calling these libraries. Our evaluation shows that cost of evaluating an assertion on the concrete heap grows with the size of the list, as expected, typically growing quadratically with the sizes of the lists. However, checking an assertion on the abstraction performs orders of magnitude faster, and seems to essentially take *constant time* in checking an assertion, when there are many assertions and when the sizes of the data structures are large.

Certified Program Models. We propose a new methodology – *certified programs models* – to advocate that the fairly complex protocols in distributed systems be modeled using *programs* (programs written in traditional systems languages, like C with concurrency), and shown to be correct against it's specifications. The idea is to model the entire distributed system in software, akin to a software simulator of the system. The model captures the distributed processes, their memory state, the secondary storage state, the communication, the delays, and the failures, using non-determinism when necessary. We have explored the certified model paradigm for modeling, testing, and formally proving properties of core distributed protocols that underlie eventually consistent distributed key-value/NoSQL stores.

The key contribution is reasoning about the guarantees of eventual consistency that real implementations of key-value stores provide. We model two core protocols in key-value stores as programs, the *hinted-handoff protocol* and the *read-repair* protocol, which are anti-entropy mechanisms first proposed in the Amazon Dynamo system [70], and later implemented in systems such as Riak [2] and Cassandra [4].

We build certified program models— program models for these protocols written in concurrent C and that are verified for eventual consistency. In the case of the *hinted-handoff protocol*, we prove that this protocol working alone guarantees eventual consistency provided there are only transient faults. For the *read-repair protocol*, prove a property that at any point, if a set of nodes are alive and they all stay alive, and if all requests stop except for an unbounded sequence of reads to a key, then the live nodes that are responsible for the key will eventually converge.

Chapter 2

Verifying Security Invariants in an Android-based Operating System

In this chapter we present ExpressOS, a new OS for enabling high-assurance applications to run on commodity mobile devices securely. The thesis contributions described in the chapter are a new OS architecture and use of deductive formal verification techniques for proving key security invariants about our implementation. The focus is on lightweight formal verification, i.e., proving that our OS implements the security invariants correctly, rather than striving for full functional correctness, requiring significantly less verification effort while still proving the security relevant aspects of our system.

We built ExpressOS, subject it to a vulnerability analysis, and tested its performance. Our evaluation shows that the performance of ExpressOS is comparable to an Android-based system. In one test, we ran the same web browser on ExpressOS and on an Android-based system and found that ExpressOS adds 16% overhead on average to the page load latency time for nine popular web sites.

2.1 Introduction

Modern mobile devices have put a wealth of information and ever-increasing opportunities for social interaction at the fingertips of users. At the center of this revolution are smart phones and tablet computers, which give people a nearly constant connection to the Internet. Applications running on these devices provide users with a wide range of functionality, but vulnerabilities and exploits in their software stacks pose a real threat to the security and privacy of modern mobile systems [52, 104].

Current work on secure operating systems has focused on formalizing UNIX implementations [78, 170] and on verifying microkernel abstractions [96, 107]. Although these projects approach or achieve full functional correctness, they require a large verification effort (the seL4 paper claims that it took 20 man-years to build and prove) and detailed knowledge of low-level theorem-proving expertise.

In this chapter we detail ExpressOS, a new OS we designed to provide high assurance security mechanisms for application security policies, and a verified implementation of the mechanisms that ExpressOS uses to enforce security policies. Our design includes an OS architecture for coping with legacy hardware safely, a programming language and

run-time system for building our operating system, and a set of proofs on our kernel implementation that provide high assurance that our system can uphold the security invariants we define.

We focus on a set of security invariants of the code *without attempting full-blown functional correctness*. For example, we have verified that the private storage areas for different applications are isolated in ExpressOS. We *do not* derive this security invariant by showing the every aspect of all involved components, like the file systems and the device drivers, is correct. Instead, ExpressOS isolates the above components as untrusted system services. The proofs show that ExpressOS encrypts all private data of applications before sending it out to the system services, and ExpressOS places security checks correctly so that only the application itself can access its private data.

The proofs focus on seven security invariants covering secure storage, memory isolation, user interface (UI) isolation, and secure IPC. By proving these invariants, ExpressOS enables sensitive applications to provably isolate their state (mostly ensuring integrity of state; confidentiality is also ensured to a certain degree, but side-channel attacks are not provably prevented), and run-time events from malicious applications running on the same device.

We achieve the proofs of these invariants by annotating source code with formal specifications written using mathematical abstractions of the properties. We have applied deductive verification techniques, which required, writing ghost-code annotations that track and update these mathematical abstractions according to the code's progress, and by discharging verification using either abstract interpretation or automatic theorem provers (mainly SMT solvers [68]). A thorough verification of the invariants above requires only a modest annotation effort ($\sim$ 2.8% annotation overhead).

To evaluate the security and the performance of ExpressOS, we have built a prototype system that runs on x86 hardware and exports a subset of the Android/Linux system call interface. To evaluate security, we have examined 383 recent vulnerabilities listed in CVE [62]. The ExpressOS architecture successfully prevents 364 of them. To evaluate performance, we have used an ASUS Eee PC and run a web browser on top of ExpressOS. Our experiments show that the performance of ExpressOS is comparable to Android: ExpressOS shows 16% overhead for the web browsing benchmark.

2.2 ExpressOS Overview

The primary goal of ExpressOS is to be a practical, high assurance operating system. As such, ExpressOS should support diverse hardware and legacy applications. Also, security invariants of ExpressOS should be formally verified.

This section provides an overview to the architecture, the verification approach, and system components of ExpressOS. Section 2.4 discusses the security implications of our architecture and the detailed design of the ExpressOS kernel.

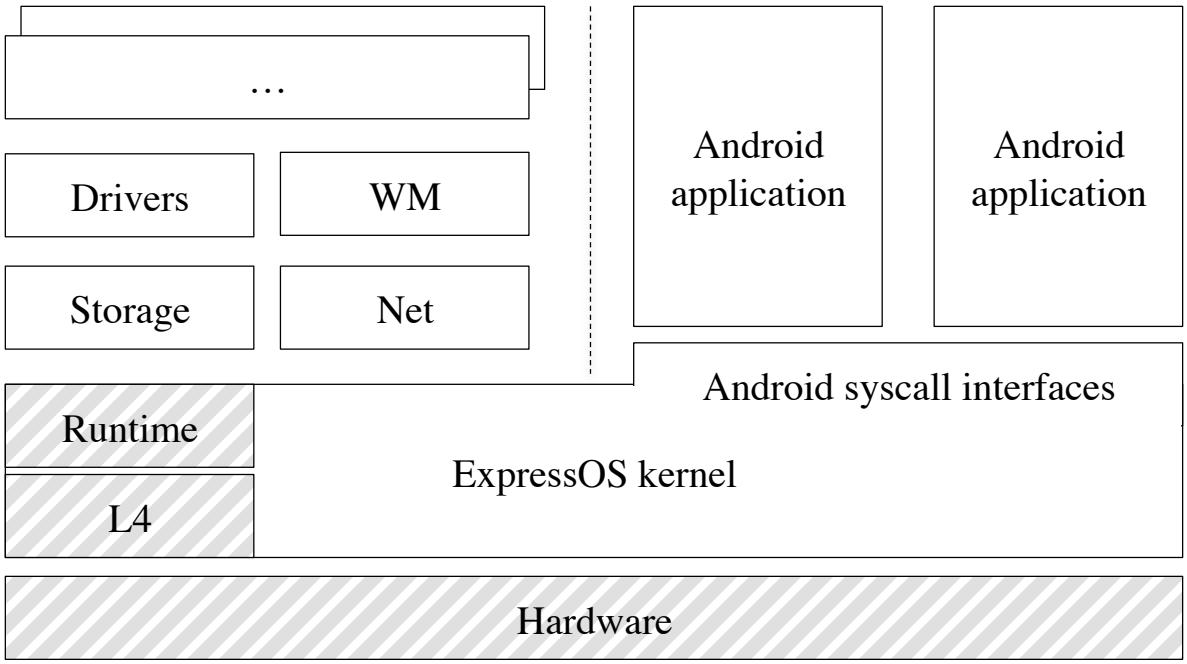


Figure 2.1: Overall architecture of ExpressOS. Android applications run directly on top of the ExpressOS kernel. Boxes that are left to the vertical dotted line represent system services of ExpressOS. Shaded regions show the trusted computing base (TCB) of the system.

2.2.1 Architecture for Verification

Figure 5.3 describes the architecture of ExpressOS. The architecture includes the ExpressOS kernel, system services, and abstractions for applications.

ExpressOS uses four main techniques to simplify verification effort. First, ExpressOS pushes functionality into microkernel services, just like traditional microkernels, reducing the amount of code that needs to be verified. Second, it deploys end-to-end mechanisms in the kernel to defend against compromised services. For example, the ExpressOS kernel encrypts all data before sending it to the file system service. Third, ExpressOS relies on programming language type-safety to isolate the control and data flows within the kernel. Fourth, ExpressOS makes minor changes to the IPC system-call interface to expose explicitly IPC security policy data to the kernel. By using these techniques, ExpressOS isolates large components of the kernel while still being able to prove security properties about the abstractions they operate on.

Current techniques to isolate components and manage the complexity of a kernel include software-fault isolation [169], isolating application state through virtualization [18], and microkernel architectures [78, 170, 96, 107]. These techniques alone, however, are insufficient for verifying the implementation of a system's security policies – the correctness of the security policies still relies on the correctness individual components. For example, an isolated, but compromised, file system might store private data with world-readable permissions, compromising the confiden-

tiality of the data.

2.2.2 Design Principles

In order to meet the overall goal of ExpressOS, three design principles guide our design:

- *Provide high-level, compatible, and verifiable abstractions*. ExpressOS should provide high-level, compatible abstractions (e.g., files) rather than low-level abstractions (e.g., disk blocks), so that existing applications can run on top of ExpressOS, and developers can express their security policies through familiar abstractions. More importantly, the security of these abstractions should be formally verifiable to support secure applications.

- *Reuse existing components*. ExpressOS should be able to reuse existing components to reduce the engineering effort to support production environments. Some of the components will have vulnerabilities. ExpressOS isolates these vulnerabilities to ensure they will not affect the security of the full system.

- *Minimize verification effort*. Fully verifying a practical system requires significant effort, (e.g., the seL4 microkernel takes 20 person years to build and to verify [107]), thus the verification of ExpressOS focuses *only on security invariants*. This design choice also enables combining lightweight techniques like code contracts (which get verified using abstract interpretation techniques) with Dafny annotations [119] (which get verified using logical constraint solvers) to further reduce verification effort.

2.2.3 Verification Approach

Our modular design limits the scope of verification down to the ExpressOS kernel. The ExpressOS kernel is implemented in C# and Dafny [119], both of which are type-safe languages. Dafny is a research programming language from Microsoft Research that researchers use to teach people about formal methods. Like SPIN [32] and Singularity [96], type-safety ensures that the ExpressOS kernel is free of memory errors, and provides fine-grain isolation between components within the kernel. A static compiler compiles both C# and Dafny programs to object code for native execution.

We verify security invariants in ExpressOS with both code contracts and Dafny annotations. Code contracts have low annotation overhead but are unable to reason about complicated properties, like properties about linked lists. In contrast, Dafny is capable of reasoning about complicated properties, but requires a heavy annotation burden and deep expertise in formal methods to use. Based on these charactertics, we verified simpler security invariants using code contracts, and more complex ones (like manipulation of linked lists) in Dafny, where we use ghost variables (i.e., variables that aid verification) to connect the proofs of both techniques. Such a combination enables high productivity

for verification using code contracts, yet still retaining the the full expressive power of Dafny to verify complicated properties needed to prove security invariants.

ExpressOS verifies security invariants that involve asynchronous execution contexts by translating them into object invariants of relevant data structures (Please see Section **??** for more about object invariants). Verification on asynchronous execution contexts becomes challenging due to the separation of the control and data flows. Verifying only the security invariants, however, has a simple solution. In particular, ExpressOS expresses security invariants in terms of ghost variables, and the object invariants of relevant data structures, so that the object invariants imply the original security invariant. The object invariants can be reasoned about locally, which is significant simpler than verifying full functional correctness about those asynchronous execution contexts.

2.2.4 The ExpressOS Kernel

The ExpressOS kernel is responsible for managing the underlying hardware resources and providing abstractions to applications running above. The ExpressOS kernel uses L4 to access the underlying hardware. L4 provides abstractions for various hardware-related activities, such as context switching, address space manipulation, and inter-process communication (IPC). L4 resides in the TCB of ExpressOS, although a variant of L4, seL4 [107], has been fully formally verified.

2.2.5 System Services

ExpressOS separates subsystems as system services running on top of the ExpressOS kernel. These services include persistent storage, device drivers, networking protocols, and a window manager for displaying application GUIs and handling input. ExpressOS reuses the existing implementation from L4Android [114] to implement these services.

All these services are untrusted components in ExpressOS. They are isolated from the ExpressOS kernel. The isolation combined with other techniques (discussed in Section 2.4) ensures that the verified security invariants remain valid even if a system service is compromised.

2.2.6 Application Abstractions

The ExpressOS kernel exports a subset of the Android system call interface directly to applications. These abstractions include processes, files, sockets, and IPC. ExpressOS supports unmodified Android applications like a Web Browser to run on top of it directly.

The ExpressOS kernel provides an alternative set of IPC interfaces that is more amenable to formal verification, but still enables applications to perform Android-like IPC operations. First, it exposes all IPC interfaces as system calls rather than using `ioctl` calls. Second, it requires all IPC channels to be annotated with permissions, so that

the ExpressOS kernel can perform access control on IPC operations. This design choice enables us to verify IPC operations in ExpressOS.

2.3 Threat Model

An untrusted Android application runs directly on ExpressOS. Such an application contains arbitrary code and data, along with a manifest listing all its required permissions. The user grants all permissions listed on the manifest to the application once he or she agrees to install it into the system. ExpressOS must be able to confirm that all activities of the application conform to its permissions, and all security invariants defined by ExpressOS (discussed in Section 2.4) must hold during the lifetime of the system.

ExpressOS should be able to isolate multiple applications. An application must not be able to compromise any security invariants of other applications.

The TCB of ExpressOS includes the hardware, the L4 microkernel, the compilers, and the language run-time. None of the system services are trusted by ExpressOS. ExpressOS should be able to maintain its security invariants for all applications even if the system services execute arbitrary code.

The work described in this chapter focuses only on confidentiality and integrity; availability is out of the scope of the work described in the chapter.

2.4 Proving Security Invariants

This section describes that how we apply the techniques described in Section 2.2 to verify security invariants on the implementation of storage, memory management, user interface, and IPC in ExpressOS.

2.4.1 Secure Storage

The storage system of ExpressOS provides guarantees of access control, confidentiality, and integrity for applications, yet offering the same sets of storage APIs as Linux. The storage system is implemented as an additional layer on top of the basic storage APIs (e.g., `open()`, `read()`, and `write()`), which are provided by the untrusted storage service.

The first security invariant for secure storage that the ExpressOS kernel enforces is:

SI 1. *An application can access a file only if it has appropriate permissions. The permissions cannot be tampered with by the storage service.*

An application can implement its security policy directly on top of SI 1. For example, an application might restrict the permission of a file so that only the application itself has access to it.

```
static SecureFSInode Create(Thread current,
                            ByteBufferRef metadata,
                            ...)
{
  ...
  var ret = InitializeAndVerifyMetadata(...);
  ...

  var metadata_verified = ret == 0;
  ...

  var access_permission_checked
    = Globals.SecurityManager.CanAccessFile(current,
                                            metadata);
  ...

  // Verify both Property 1 and Property 2
  Contract.Assert(metadata_verified
                  && access_permission_checked);

  return ...;
}
```

Figure 2.2: Relevant code snippets in C# for Property 1 and Property 2.

Since other compromised components such as the storage service and device drivers can affect SI 1, the first step of verifying SI 1 is to isolate the effects of these services. The ExpressOS kernel uses the HMAC algorithm [28] to achieve this goal. The ExpressOS kernel prepends several pages to all files managed by the secure storage system. These pages store metadata such as permissions, the size of the file, as well as an HMAC signature of these pages. The ExpressOS kernel checks the HMAC signature to ensure the integrity of the metadata when loading the file into the memory.

Now SI 1 can be reduced to the following two lower level properties:

Property 1 (Integrity-Metadata)**.** *The signature of a file's metadata is always checked before an application can perform any operations on it.*

Property 2 (Access Control)**.** *An application can only access a file when it has appropriate permissions.*

Figure 2.2 shows relevant code of Property 1 and Property 2. The `Create()` function calls `InitializeAndVerifyMetadata()` to verify the HMAC signature of the meta data. Then it calls `SecurityManager.CanAccessFile()` to determine whether the current process has appropriate permissions to open the file. Finally, the annotation of `Contract.Assert()` instructs the verifier to prove that both the integrity of the meta data, and the access permission of the file have been checked.

The reduction is a trade-off between formal verification and practicality. We argue that pragmatically the conjunction of Property 1 and Property 2 implies SI 1. The reduction captures the important fact that ExpressOS misses no se-

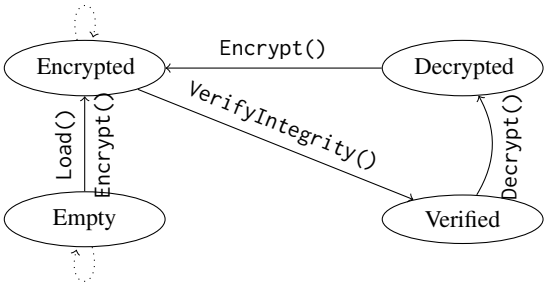


Figure 2.3: State diagram for the `Page` class. Ellipses represent the state. Solid and dotted arrows represent the successful and failed transitions.

curity checks in access controls, which is the main point of verifying ExpressOS. It does assume the implementation of relevant libraries like AES / SHA-1, and the one of `InitializeAndVerifyMetadata()` and `SecurityManager.CanAccessFile()` is correct. These components can be verified independently, and a verified implementation can be plugged into the system to further strengthen the proof.[1]

The ExpressOS kernel also enforces integrity and confidentiality of its secure storage system:

SI 2. *Only the application itself can access its private data. The data cannot be accessed or tampered with by other applications, or by system services.*

Similar to SI 1, the ExpressOS kernel uses encryption to defend against compromised system services. From a high level, it partitions a file into multiple pages, and then it encrypts each page with AES for confidentiality. To ensure integrity, it signs each encrypted page with the HMAC algorithm, and packs the results to the metadata of the file. The overall implementation is similar to Cryptfs [177].

ExpressOS assigns each application a different private key during installation; therefore SI 2 can be reduced to the following properties:

Property 3 (Confidentiality)**.** *Every page is encrypted before sending to the storage service.*

Property 4 (Integrity)**.** *Each page loaded from the storage service has the appropriate signature.*

The idea behind verifying Property 3 and Property 4 is to use the ghost variable `CurrentState` to record the current state of the page. Figure 2.3 shows the state transition diagram for a `Page` object. A page can be in the state of `Empty`, `Verified`, `Decrypted`, and `Encrypted`, meaning that (1) the page is empty, (2) its integrity has been verified, (3) its contents have been decrypted, and (4) its contents have been encrypted. To verify these properties, we specify

[1]The `InitializeAndVerifyMetadata()` function parses binary data read from the disk and verifies its integrity, which is easier implemented in C. To demonstrate the feasibility of this approach, we have implemented `InitializeAndVerifyMetadata()` in C, and verified its correctness with VCC [59].

```
class CachePage {
  enum State { Empty, Verified, Decrypted, Encrypted }
  [Ghost] State CurrentState;

  void Encrypt(...) {
    Contract.Requires(CurrentState == State.Empty
                      || CurrentState == State.Decrypted);
    Contract.Ensures(CurrentState == State.Encrypted);
    ...
  }

  void Decrypt(...) {
    Contract.Requires(CurrentState == State.Verified);
    Contract.Ensures(CurrentState == State.Decrypted);
    ...
  }

  // Verify the integrity of the page,
  // Returns true if the page is authentic.
  bool VerifyIntegrity(...) {
    Contract.Requires(CurrentState == State.Encrypted);
    Contract.Ensures(!Contract.Result<bool>()
                     || CurrentState == State.Verified);
    ...
  }

  // Load the content of the page from
  // the storage service.
  bool Load(...) {
    Contract.Requires(CurrentState == State.Empty);
    Contract.Ensures(!Contract.Result<bool>()
                     || CurrentState == State.Encrypted);
    ...
  }

  void Flush(...) {
    Contract.Requires(CurrentState == State.Encrypted);
    ...
  }
}
```

Figure 2.4: Relevant code snippets in C# for Property 3 and Property 4. Contract.Requires() and Contract.Ensures() specify the pre- and post-conditions of the functions.

the valid state transitions as the pre- and post-conditions of the relevant functions. For example, the specifications of `Decrypt()` state that the page should has its integrity verified before entering the function, and its contents are decrypted afterwards.

Figure 2.4 shows the relevant code snippets. Notice that Property 3 can be specified as a pre-condition of the function `Flush()`: the function can be called only if the page is in the `Encrypted` state.

For compatibility reasons, ExpressOS does allow an application to create unencrypted, public readable files. ExpressOS, however, does not provide additional security invariants for these files.

2.4.2 Memory Isolation

The ExpressOS kernel enforces proper access control and isolation for all memory of the applications:

SI 3. *If a memory page of an application is backed by a file, the pager can map it in if and only if the application has proper access to the file.*

SI 4. *An application cannot access the memory of other applications, unless they explicitly share the memory.*

The challenge of verifying SI 3 is that there is insufficient information available for verification at the point of assertions (i.e., in the pager). This is because the security checks are executed in different contexts, where both the control and data flows are separated in these two contexts.

ExpressOS addresses this challenge by connecting the information indirectly through the *object invariants* of relevant data structures. It strengthens these object invariants to contain information about the security checks, so that the object invariants can derive the desired security invariants.

Figure 2.5 and Figure 2.6 show the relevant implementation of the pagers. From a high level, the ExpressOS kernel organizes the virtual memory of a process with a series of `MemoryRegion` objects. A `MemoryRegion` represents a continuous region of the virtual memory, which has information on its access permissions, location, and a reference to the backing file (i.e., the `File` field in the `MemoryRegion` class) if it maps to a file. Since we have verified that the ExpressOS kernel properly checks access to the backing file in Section 2.4.1, SI 3 can be reduced to the following property:

Property 5. *When the page fault handler serves a file-backed page for a process, the file has to be opened by the same process.*

To verify Property 5, we use a ghost variable to record the *ownership* of the relevant objects, and to specify object invariants based on the ownership. The first assertion in Figure 2.5 specifies Property 5, which gets verified through a series of object invariants.

```
static uint HandlePageFault(Process proc,
                            uint faultType,
                            Pointer faultAddress,
                            Pointer faultIP)
{
  Contract.Requires(proc.ObjectInvariant());

  ...
  AddressSpace space = proc.Space;
  var region = space.Regions.Find(faultAddress);

  if (region == null || (faultType & region.Access) == 0)
    return 0;

  ...
  var shared_memory_region = IsSharedRegion(region);
  var ghost_page_from_fresh_memory = false;

  if (shared_memory_region)
  {
    ....
  }
  else
  {
    page = Globals.PageAllocator.AllocPage();
    ghost_page_from_fresh_memory = true;
    ...

    if (region.File != null)
    {
      // Assertion of Property 5
      Contract.Assert(region.File.GhostOwner == proc);
      var r = region.File.Read(...);
      ...
    }
    ...
  }

  Contract.Assert(shared_memory_region
                  ^ ghost_page_from_fresh_memory);
  ...
}
```

Figure 2.5: Code snippets for the page fault handler.

```
class MemoryRegion {
  ...
  var File: File;
  var GhostOwner: Process;

  function ObjectInvariant() : bool ... {
    File != null ==> File.GhostOwner == GhostOwner
    ...
  }
}

class Process {
  var space: AddressSpace;

  function ObjectInvariant() : bool ... {
    space != null && space.GhostOwner == this
    ...
  }
}

class AddressSpace {
  var GhostOwner: Process;
  var Head: MemoryRegion;
  // Ghost variable to record the set of its owned
  // MemoryRegion
  ghost var Contents: seq<MemoryRegion>;

  function ObjectInvariant() : bool ... {
    forall x :: x in Contents ==>
      x != null && x.ObjectInvariant()
      && x.GhostOwner == GhostOwner;
    ...
  }

  method Find(address: Pointer) returns (ret: MemoryRegion)
  requires ObjectInvariant();
  ensures ObjectInvariant();
  ensures ret != null ==>
    ret.ObjectInvariant() && ret.GhostOwner == GhostOwner;

  method Insert(r: MemoryRegion)
  requires r != null && r.GhostOwner == GhostOwner;
  requires ObjectInvariant() && r.ObjectInvariant();
  ...
  ensures ObjectInvariant();
}

class File { ... var GhostOwner: Process; }
```

Figure 2.6: Reduced code snippets in Dafny for the `MemoryRegion` and the `AddressSpace` class.

First, the AddressSpace class represents the virtual address space of a process by a sequence of MemoryRegion objects. Intuitively the AddressSpace object "owns" all MemoryRegion objects in the sequence, and the Find() method looks up and returns the corresponding MemoryRegion object for the virtual address. Since the Find() method can only return MemoryRegion objects that are owned by the AddressSpace object, the object invariant of the AddressSpace class should lead to the following post-condition:

```
ret != null → ret.ObjectInvariant()∧
              ret.GhostOwner == GhostOwner;
```

At the first assertion in Figure 2.5, this is simplified down to:

```
region.File.GhostOwner == region.GhostOwner
                       == space.GhostOwner
```

The object invariant of the proc object ensures that

```
proc.space.GhostOwner == proc
```

This leads to the assertion of Property 5.

Property 5 is strictly weaker than the property that ensures full functional correctness. For example, Property 5 does not enforce that the file used in page fault handler has to be the exact same file that was requested by the user. The property, however, maintains isolation and can be verified through object invariants.

We combine code contracts and Dafny to verify this property. Dafny verifies the MemoryRegion and AddressSpace class, because Dafny is able to reason about the linked lists in their implementation. The verification results from Dafny are expressed as properties of ghost variables (i.e., the GhostOwner fields). These properties are exported to code contracts as ground facts with the Contract.Assume() statements. Using Contarct.Assume() is a simple way let code contracts know about proofs about Dafny code. Additionally, we annotate the GhostOwner fields as read-only fields in C# to ensure soundness.

SI 4 can be expressed in a slightly different way to ease the verification.

Property 6 (Freshness)**.** *The page fault handler maps in a fresh memory page when the page fault happens in non-shared memory.*

The idea is to ensure that the page fault handler always allocates a fresh memory page, i.e., a memory page that are not overlapped with any allocated pages. ExpressOS adopted the verified memory allocator from seL4 [165] for

this purpose. ExpressOS dedicates a fixed region to this allocator in order to implement this property. The reduction allows specifying this property as the second assertion in Figure 2.5.

2.4.3 UI Isolation

The ExpressOS kernel enforces the following UI invariant:

SI 5. *There is at most one currently active (i.e., foreground) application in ExpressOS. An application can write to the screen buffer only if it is currently active.*

To write to the screen, an application requests shared memory from the window manager, and writes screen contents onto the shared memory region. In ExpressOS, this can only be done through an explicit API so that the ExpressOS kernel knows exactly which memory region is the screen buffer.

```
class UIManager
{
  Process ActiveProcess;

  [ContractInvariantMethod]
  void ObjectInvariantMethod() {
    Contract.Invariant(ActiveProcess == null
                       || ActiveProcess.ScreenEnabled);
  }

  void OnActiveProcessChanged(Process next) {
    Contract.Requires(next != null);
    Contract.Ensures(ActiveProcess == next);
    ...
  }

  void DisableScreen() {
    Contract.Ensures(ActiveProcess == null);
    Contract.Ensures(
      !Contract.OldValue(ActiveProcess).ScreenEnabled);
    ...
  }

  void EnableScreen(Process proc) {
    ...
    Contract.Ensures(ActiveProcess == proc);
    Contract.Ensures(ActiveProcess.ScreenEnabled);
    ...
  }
}
```

Figure 2.7: Relevant code snippets for SI 5. `[ContractInvariantMethod]` annotates the method that specifies object invariant.

The ExpressOS kernel enforces SI 5 by explicitly enabling and disabling the write access of screen buffers when changing the currently active application.

The object invariant of `UIManager` in Figure 2.7 specifies SI 5. The boolean ghost variable `ScreenEnabled` denotes whether the application has write access to the screen. The initial value of `ScreenEnabled` is set to false for each application. Since it is the only API to manipulate the screen, the object invariant implies that only the current application has write access to the screen buffer.

2.4.4 Secure IPC

To simplify verification, ExpressOS provides an alternative, secure IPC (SIPC) interface over the Android's IPC interface. First, SIPC exposes all IPC functionality explicitly through system calls to eliminate the implementation and verification efforts on complex logic in `ioctl()` of Android's IPC. Second, the ExpressOS kernel enforces proper access controls for SIPC, compared to relying on the receiver of Android's IPC for proper access control. This design moves the access control logic of SIPC into the ExpressOS kernel.

SIPC provides basic functionality to the applications, including creating SIPC channels, connecting to SIPC channels, and sending and receiving messages over the channel. Applications can still perform Android-like IPC operations using the SIPC interface.

The ExpressOS kernel enforces the following security invariants for SIPC:

SI 6. *An application can only connect to SIPC channels when it has appropriate permissions.*

SI 7. *An SIPC message will be sent only to its desired target.*

SI 6 and SI 7 can be verified with similar approaches described above.

SI 6 is an access control invariant, thus the strategy of proving SI 6 is similar to the one of SI 1. We use a ghost variable to indicate whether the process has properly checked the permissions when opening a new SIPC channel.

We follow the proving strategy of Property 5 to verify SI 7. The idea is to create a `SIPCMessage` object for each IPC message, and to introduce a ghost variable `target` to record the target process of the message, which provides sufficient information to verify SI 7.

2.4.5 Verification Experience

Overall, we found the verification effort in terms of annotations practical for the properties that were proven correct. While we developed the system and wrote code, we came up with the relevant security properties at the level of the module we were writing, and formalized it using appropriate annotations. Combining code contracts and Dafny reduced the code to annotation ratio down to about 2.8% (implementing the specification defined by the annotations). There were some instances where the code we wrote was not actually correct, and using the verification tools to

prove the property led us to find the error. Equally importantly, formalizing the specification at the level of the code crystallized the vague properties we had in mind, and helped us write better code as well.

One lesson that we had in ExpressOS was to refine the shapes and aliasing information of the objects through redesigning data structures, and implementing ownership using ghost variables. Simpler shapes eased the verification. For example, we have reimplemented the the `MemoryRegion` object as a singly-linked list instead of a doubly-linked one, since verifying the manipulations of the linked list in the latter case requires specifying reachability predicates that are difficult to reason about within SMT-based frameworks. The verification of heap structure properties in Dafny was achieved sometimes using further ghost annotations in the style of natural proofs [124, 146].

In simpler cases we used ownership to constrain the effect of aliasing. For example, the ghost field `GhostOwner` in the `AddressSpace` object specified which `Process` had created the object. The information was used in proving that each process creates its own `AddressSpace` object, effectively forbidding aliasing between `AddressSpace` objects of different processes.

One potential drawback we found during verification was that the specification using ghost code is sometimes too intimately interleaved with the implementation. Consequently, the specification gets strewn all across the code, and it is our responsibility that this actually is correct. Though the mathematical abstractions do help to some extent to distance the specification from the code, ghost updates to these abstractions are still intimately related. To illustrate this, consider a programmer implementing the code `Encrypt()` in Figure 2.4, and consider the scenario where the actual encryption fails for some reason, and yet the programmer puts the page into the `Encrypted` state. The verifier will go through even though the implementation is incorrect with respect to what the developer wanted; the onus of writing the correct specification is on the developer.

While we did not encounter any case where we noticed we made errors in inadvertently formulating too weak a specification, we did spend time double-checking that our *specifications* were indeed correct. We think an alternate mechanism for writing specifications that are a bit more independent from the code, resilient to code changes, and yet facilitates automated proving would make the developer's work more robust and productive.

2.5 Implementing ExpressOS

The implementation of ExpressOS consists of two parts: the ExpressOS kernel and ExpressOS services. The ExpressOS kernel is a single-thread, event-driven kernel built on top of L4::Fiasco. We have implemented the kernel in C# and Dafny, which is compiled to native X86 code with a static compiler.

The implementation of ExpressOS kernel includes processes, threads, synchronization, memory management (e.g. `mmap()`), secure storage, and secure IPC, so that it is sufficient to limit the scope of verification entirely within the

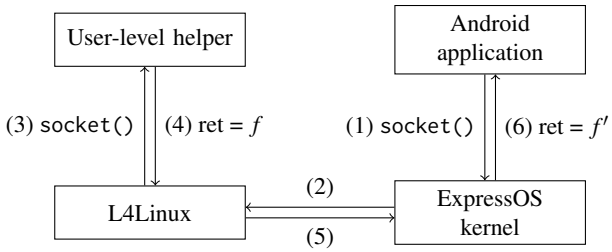


Figure 2.8: Work flow of handling `socket()` system call in ExpressOS. The arrows represent the sequences of the work flow.

kernel. It implements a subset of Linux system calls to support Android applications like the Android Web browser. The kernel contains about 15K lines of code. The source code is available at `https://github.com/ExpressOS`.

The ExpressOS kernel delegates the implementation of system calls to ExpressOS services whenever it does not affect the soundness of the verification. These services include file systems, networks, device drivers, as well as Android's user-level services like window manager and service manager. ExpressOS reuses L4Android to implement these services. L4Android is a port of Android to a Linux kernel that runs on top of L4::Fiasco (i.e., L4Linux).

The rest of the section describes (i) how to dispatch a system call to ExpressOS services, (ii) how to bridge Android's binder IPCs between ExpressOS and Android's system services, and (iii) how to support shared memory between an application and ExpressOS services.

Dispatching a system call to ExpressOS services. The ExpressOS kernel forwards system calls with IPC calls to L4-Android. Figure 2.8 shows the workflow of handling the `socket()` system call in the ExpressOS kernel. When (1) the application issues a `socket()` system call to the ExpressOS kernel, (2) the kernel wraps it as an IPC call to the L4Linux kernel. The L4Linux kernel executes the system call (which might involve a user-level helper like the step (3) & (4)), and (5) returns the result back to the ExpressOS kernel. The ExpressOS kernel (6) interprets the result and returns it to the application.

It is important for the ExpressOS kernel to maintain proper mappings between the file descriptors (fd) of the user-level helper and those of the application. In Figure 2.8, it maps between the fd f and f' so that subsequent calls like `send()` and `recv()` can be handled correctly. This workflow mirrors the implementation of the Coda file system inside Linux [105].

Bridging Android's binder IPC. Android applications communicate to Android system services (e.g., the window manager) through the Android's binder IPC interface. The ExpressOS kernel extends the mechanism in Figure 2.8 to bridge the binder IPC. The user-level helper in Figure 2.8 acts as a proxy between the Android application and Android system services. Both the user-level helper and the ExpressOS kernel transparently rewrite the IPC messages to support advanced features like exchanging file descriptors.

Supporting shared memory. To support shared memory between Android applications and Android services, the ExpressOS kernel maps all physical memory of L4Linux into its virtual address space. It maps the corresponding pages to the address space of the application when sharing happens. We have modified L4Linux to expose its page allocation tables so that the ExpressOS kernel is able to compute the address. Both ExpressOS kernel and L4Linux use the L4 Runtime Environment (L4Re) to facilitate this process.

2.6 Evaluation

This section describes our evaluation of ExpressOS. To evaluate to what extent that the ExpressOS architecture can prevent attacks, we studied 383 relevant real-world vulnerabilities to analyze the security of the system. Then, we present the performance measurements of ExpressOS.

2.6.1 Vulnerability Study

To understand to what extent ExpressOS is able to withstand attacks, we studied 742 real-world vulnerabilities (from Jun, 2011 to Jun, 2012) listed in CVE. 383 out of 742 are valid vulnerabilities, and they affect different components used in Android. We manually examined each of them, and classified it into one of the four categories based on its location:

In the core of the kernel. The vulnerability exists in the core of the Linux kernel, which means that the same functionality is implemented in the ExpressOS kernel. The proofs ensure that such a vulnerability cannot affect any security invariants discussed in Section 2.4. If the vulnerability is irrelevant to the security invariants, the language run-time ensures it cannot subvert the control flow and data flow integrity to circumvent the proofs.

In the libraries used by applications. The vulnerability exists in the libraries used by the applications, like the Adobe Flash Player and libpng. In the worst case, the attacker can gain full control of applications by exploiting the vulnerability. ExpressOS ensures that the compromised application must adhere to its permissions, which prevents it from accessing other applications' private data, effectively protecting sensitive applications from compromised applications.

In system services. The vulnerability exists in the system services of ExpressOS, including the file system, the networking stack, device drivers, and Android user-level services. ExpressOS combines three techniques to contain the vulnerability.

First, ExpressOS uses end-to-end security mechanisms and the protections provided by the ExpressOS kernel to protect file system and network data. For example, both the confidentiality and integrity of any private data remain intact when the storage service is compromised, because SI 2 ensures that the attacker cannot access or tamper with

private data. Similarly, the attacker cannot eavesdrop or tamper with any TLS/SSL/HTTPS connections, even if he or she compromises the networking service.

Second, ExpressOS isolates applications and system services, and restricts updates to the screen (i.e., SI 5) to contain compromises of the window manager service. An attacker with a compromised window manager takes full control of the physical screen. Successful attacks, however, still require information about UI widgets in the targeted application, and the ability to provide timely visual feedback to the user. For example, the attacker might steal the user's input by overlaying a malicious application running in background on top of the targeted application. The isolation mechanism prevents the window manager from accessing the memory of the targeted application to retrieve the exact locations of UI widgets, and SI 5 prevents the malicious application running in background updating the screen to timely react to the user's input.

Third, the L4 layer isolates the system services and the ExpressOS kernel. An attacker can potentially compromise the L4Android kernel with a vulnerability. However, the ExpressOS kernel remains intact because the L4 layer manages allocation of physical memory and the IOMMU, ensuring that the ExpressOS kernel's memory is isolated from all system services.

ExpressOS currently does not prevent compromises where a service acts as a privileged deputy that allows the attacker to use its permissions to attack the system. For example, "the Bluetooth service in Android 2.3 before 2.3.6 allows remote attackers within Bluetooth range to obtain contact data via an AT phone book transfer." (CVE-2011-4276), and "the HTC IQRD service for Android ... does not restrict localhost access to TCP port 2479, which allows remote attackers to send SMS messages." (CVE-2012-2217).

In sensitive applications. If an application is exploited, there is not much that ExpressOS can do for that application. Although we proved our implementation of several security policies in ExpressOS, if an application configures its policy incorrectly or its application logic leads to a security compromise, there is little the ExpressOS kernel can do to protect it.

Location	Example	Num.	Prevented
The core of the kernel	Logic errors in the futex implementation allow local users to gain privileges.	9	9 (100%)
Libraries of applications	Buffer overflow in libpng 1.5.x allows attackers to execute arbitrary code.	102	102 (100%)
System services	Missing checks in the vold daemon allows local users to execute arbitrary code.	240	226 (93%)
Sensitive applications	The BoA application stores a security question's answer in clear text which allows attackers to obtain the sensitive information.	32	27 (84%)
Total		383	364 (95%)

Figure 2.9: Categorization on 383 relevant vulnerabilities listed in CVE. It shows the number of vulnerabilities that ExpressOS prevents.

Figure 2.9 summarizes our analysis of 383 vulnerabilities. ExpressOS is able to prevent 364 (95%) of them.

A large portion of vulnerabilities are due to memory errors in application libraries or in the system services. The verified security invariants and the protection from the ExpressOS kernel protect the data of sensitive applications from being compromised.

Out of the 383 vulnerabilities, there are two vulnerabilities related to covert channels. For example, the Linux kernel before 3.1 allows local users to obtain sensitive I/O statistics to discover the length of another user's password (CVE-2011-2494). These types of vulnerabilities are beyond the scope of our verification efforts and something ExpressOS is unable to prevent.

2.6.2 Performance

We evaluate the performance of ExpressOS by measuring the execution time of a variety set of benchmarks. We compared the performance of benchmarks running on ExpressOS, unmodified L4Android, and Android-x86. All experiments run on an ASUS Eee PC 1005HA with an Intel Atom N270 CPU running at 1.60 GHz, 1GB of DDR2 memory, and a Seagate ST9160314AS $5,400$ RPM, 160G hard drive. Our Eee PC connects to our campus network through its built-in Atheros AR8132 Fast Ethernet NIC.

Both ExpressOS and L4Android run the L4Linux 3.0.0 kernel. The Android-x86 runs on top of Linux 2.6.39. All three systems run the same Android 2.3.7 (Gingerbread) binaries in user spaces.

We evaluate the performance of the Android web browser on real network, and microbenchmarks evaluating different aspects of system performance, including IPC, the file system, the graphics subsystem, and the networking stack. All numbers reported in this section are the mean of five runs.

Page load latency in web browsing. We measure the page load latency for nine popular web sites to characterize the overall performance of ExpressOS compared to L4Android and Android-x86. The page load latency for a web site is the latency from initial URL request to the time when the browser fires the DOM `onload` event. The app clears all caches between each run.

Figure 2.10 shows the page load latency for nine web sites of all three systems. L4Android has 2% overhead on average, suggesting that the microkernel layer added by ExpressOS adds little overhead in real-world web browsing.

ExpressOS shows 14% and 16% overhead on average compared to L4Android and Android-x86.

IPC performance. We uses a simple ping-pong IPC benchmark to compare the performance of the SIPC mechanism in ExpressOS against the Android's Binder IPC mechanism. There are two entities (the server and the client) in this benchmark. For each round, the client sends a fixed-size IPC message to the server. The server receives the message and sends an IPC message back to the client which has the same content. Then the client receives the reply and

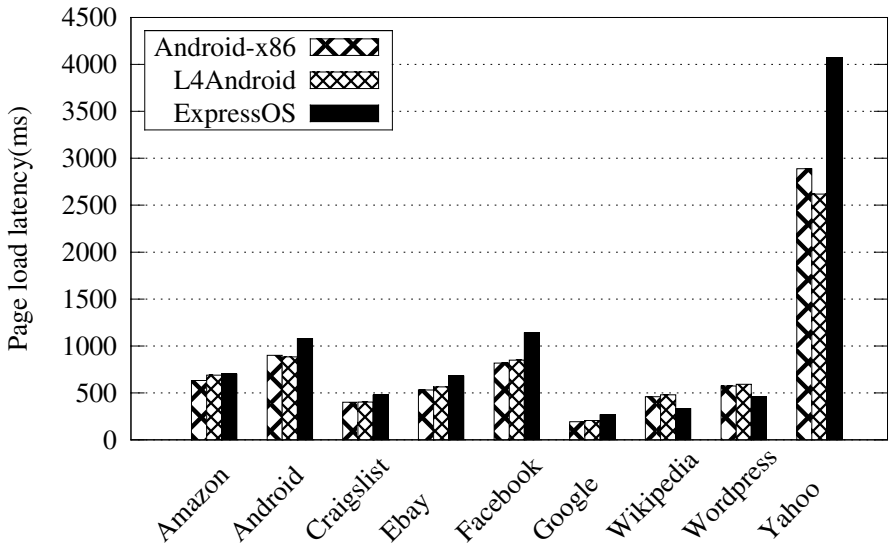


Figure 2.10: Page load latency in milliseconds for nine web sites for the same browser under Android-x86, L4-Android, ExpressOS over a real network connection.

continues to the next round.

We measured the total execution time for 10,000 rounds of this benchmark on Android-x86, L4Android and ExpressOS. We measured the performance with different message sizes, including four bytes, 1KB, 4KB, 8KB, and 16KB. Figure 2.11 describes the results of this benchmark. These numbers show that it is possible to implement a verifiable IPC mechanism in the ExpressOS architecture without sacrificing efficiency.

Other microbenchmarks. We further evaluate the performance of ExpressOS with three microbenchmarks, including (1) a SQLite benchmark (SQLite) which creates a new database, then inserts 25,000 records in one transaction, and writes all data back to the disk. (2) A network benchmark (Netcat), which receives a 32M file from local network. (3) A graphics benchmark (Bootanim), showing a PNG image, and adding light effects with OpenGL, which is derived from the boot animation program from Android.

These microbenchmarks help to categorize different aspects of the ExpressOS's performance. First, the SQLite benchmark manipulates the heap heavily, thus it is used to evaluate the performance of memory subsystem. Second, the Netcat benchmark helps to quantify the effects of microkernel servers, because ExpressOS delegates all networking operations to L4Android. Finally, the Bootanim benchmark helps to identify the cost of shadow processes. Manipulating the screen heavily relies on Android's Binder IPC and shared semaphores, both of which are forwarded back and forth between the ExpressOS kernel and the user-level shadow process in L4Android.

Figure 2.12 describes the results of all three microbenchmarks above. For the SQLite benchmark, both ExpressOS and L4Android are about 10% slower than Android-x86. For the Netcat benchmark, Android-x86, L4Android, and

ExpressOS perform almost the same. ExpressOS is about 10% slower than L4Android in the Bootanim benchmark, but surprisingly, Android-x86 performs significantly worse than both L4Android and ExpressOS. We suspect that it might due to some subtle differences between the kernel of L4Android and Android-x86, since all three systems are using the same user-level binaries.

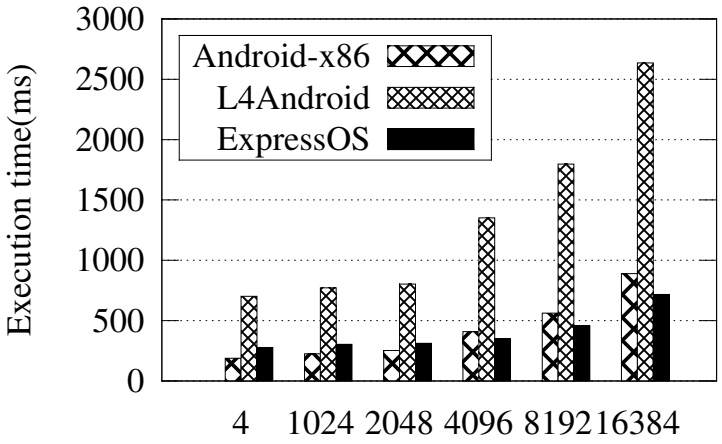


Figure 2.11: Performance of the ping-pong IPC benchmark of Android-x86, L4Android, and ExpressOS. X axis: the size of IPC messages, Y axis: the total execution time of running $10,000$ rounds of ping-pong IPC.

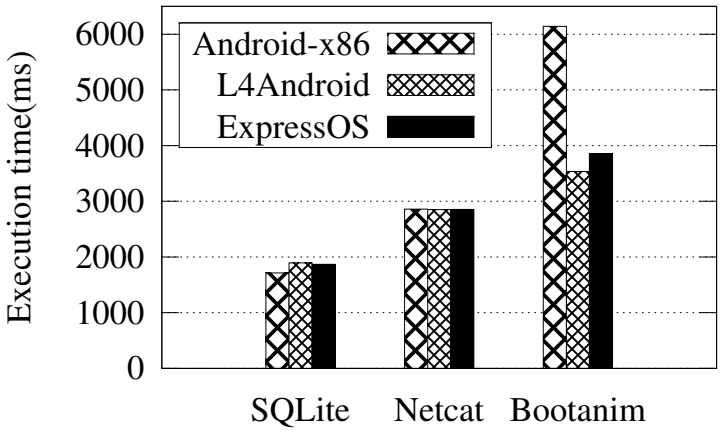


Figure 2.12: Performance results of the SQLite, Netcat and the Bootanim microbenchmark. X axis shows the type of the benchmark. Y axis shows their total execution time in milliseconds.

2.7 Related Work

Attempts to eliminate defects of operating systems with full formal verification date back to the late 1970s. Dealing with all details of real-world operating systems has been a challenge for heavyweight full formal verification methods.

Verifying OSes that provide UNIX abstractions has been cumbersome (e.g., UCLA Secure Unix [170], PSOS [78], and KSOS [143] provide partially verified UNIX abstractions). Hypervisors and microkernels have lower-level abstractions that are more amenable for verification [33, 153, 95, 59, 107], but they provide lower-level abstractions such as IPC, interrupts and context switches, which are not immediately meaningful to applications.

The key difference between ExpressOS and the above work is that the verification of ExpressOS only focuses on security invariants rather than achieving full functional correctness. ExpressOS ensures that defects in unverified parts of the system cannot subvert the security invariants. As a result, ExpressOS provides high-level abstractions (e.g., files) with verified security invariants, and verifying these security invariants requires only $\sim 2.8\%$ annotation overhead.

Alternative approaches to improve security of operating systems include controlling information flows in operating systems [76, 179], separating application state through virtualization [18, 66], intercepting security decisions in reference monitors [17], and exposing browser abstractions at lowest software layer [162]. These techniques reduce the TCB dramatically down to the implementation of themselves. There are two differences between them and ExpressOS. First, ExpressOS does not requires the applications to pervasively adopt new APIs, instead it provides Android/Linux system calls so that it can run legacy applications directly. Second, ExpressOS provides formally verified abstractions to the applications, where other techniques trust their implementation.

Implementing OS in safe languages has several benefits, such as avoiding memory errors, and isolating control and data flows in a finer granularity [32, 86, 96, 136]. ExpressOS inherits these benefits, and further verifies that the security invariants always hold in ExpressOS using code contracts and Dafny.

The current implementation of ExpressOS trusts the language run-time and the L4 microkernel. Verve [173] and seL4 [107] have verified the language run-time and the L4 microkernel. They are complementary to ExpressOS: ExpressOS can plug them in to further reduce the size of TCB. ExpressOS might also benefit from potential hardware support [161, 164].

2.8 Conclusions

In this chapter we have presented ExpressOS, a new OS architecture that provides formally verified security invariants to mobile applications. The verified security invariants cover various high-level abstractions, including secure storage, memory isolation, UI isolation, and secure IPC. By proving these invariants, ExpressOS provides a secure foundation for sensitive applications to isolate their state and run-time events from malicious applications running on the same device.

The verification effort on ExpressOS focuses on the most important properties from a system builder's perspective

rather than full functional correctness. Also, the main verification effort was in applying and adapting deduction verification techniques, ExpressOS combines several verification techniques to further reduce the verification effort. The approach is relatively lightweight and has about $\sim 2.8\%$ annotation overhead.

Our evaluation shows that ExpressOS is effective in preventing existing vulnerabilities from different attack surfaces. Besides its strong security guarantees, ExpressOS is a practical system with performance comparable to native Android.

The experience suggests that the verification technique describe in this chapter is mature enough to be broadly used by systems developers in order to obtain lightweight proofs of safety and security by focusing on a small but crucial subset of properties.

Chapter 3

Automated Reasoning about Mutable Data Structures in C using Separation Logic

The natural proof technique for heap verification developed by Qiu et al. [147] provides a platform for powerful sound reasoning for specifications written in a dialect of separation logic called Dryad. Natural proofs are proof tactics that enable automated reasoning exploiting recursion, mimicking common patterns found in human proofs. However, these proofs are known to work only for a simple toy language [147].

In this chapter, we describe how we developed a framework called VCDRYAD that extends the VCC framework [58] to provide an automated deductive framework against separation logic specifications for C programs based on natural proofs. We develop several new techniques to build this framework, including (a) a novel tool architecture that allows encoding natural proofs at a higher level in order to use the existing VCC framework (including its intricate memory model, the underlying type-checker, and the SMT-based verification infrastructure), and (b) a synthesis of ghost-code annotations that captures natural proof tactics, in essence forcing VCC to find natural proofs using primarily decidable theories.

We evaluate our tool extensively, on more than 150 programs, ranging from code manipulating standard data structures, well-known open source library routines (Glib, OpenBSD), Linux kernel routines, customized OS data structures, etc. We show that all these C programs can be fully automatically verified using natural proofs (given pre/post conditions and loop invariants) without *any user-provided proof tactics*. VCDRYAD is perhaps the first deductive verification framework for heap-manipulating programs in a real language that can prove such a wide variety of programs automatically.

3.1 Introduction

As we described in section 1.3, a promising mostly-automated yet scalable approach to program verification is the paradigm of *automated deductive verification*. To apply this approach orogrammers develop and annotate the code. These annotations not only capture the specification of the software, but also provide invariants that chop up the reasoning of the program into Hoare triples involving finite loop-less code. Using a logical semantics of the programming language, program verification reduces to reasoning purely about logic validity. Finally, these validity checks can be

automated, for the large part, using automated theorem provers and SMT solvers. The tools VCC [58], DAFNY [118], HAVOC [63], etc. that compile into BOOGIE [25] (which in turn generates verification conditions to the SMT solver Z3) and several other tools such as VeriFast [99], jStar [75], Smallfoot [31], etc. fall in this category. Several large software have been wholly or partly verified using such tools, including Microsoft Hypervisor [61], Verve [174] (an OS), and ExpressOS [126] (an Android platform verified for a small set of security properties).

We briefly explained in section 1.3.1 one of the main drawbacks of state-of-the-art mostly-automated deductive verification tools today. Recall that when verifying properties of the dynamically manipulated heap, the verification condition expressed in logic is typically not in a decidable theory, and hence the proof is hardly ever automatic. Tools that sit over BOOGIE (like VCC) and others such as VeriFast [99] and Bedrock [56] give the programmer the ability to write code-level proof tactics to make the proof go through. For example, VCC allows programmers to give *triggers* that essentially give terms that the underlying SMT solver should try substituting universally quantified variables with in order to prove the negation of a verification condition unsatisfiable. In VeriFast, the programmer often works with recursive definitions and can at the code-level ask for such definitions to be unfolded, prove lemmas that help the proof, where these lemmas themselves are guided by such high-level proof tactics. Needless to say, this is very hard for programmers to furnish, requiring them to not only understand the code and the specification language, but also the underlying proof mechanisms and tactics. Programmers with formal methods background, however, can typically take the verification through for simple properties of the software (see [126] and [144] for such practical case-studies).

The *natural proof technique*, proposed by Qiu et al. [147], suggests a way to alleviate this trigger/tactic annotation problem by identifying natural proof tactics for heap verification that are commonly used in manual proofs and deploying them automatically on code. Effective natural proofs need to have two properties (a) they should embody a set of tactics that are powerful enough to take the proofs of many verification tasks through, and (b) the search for proofs with the proof tactics must be efficient and predictable (preferably searchable using decidable logical theories and SMT solvers). In order to develop a set of tactics that is effective, the specification logic itself may need to be co-designed with natural proofs. Qiu et al. [147] provide a dialect of separation logic called DRYAD that forces the user to write specifications using primarily recursion (shunning unguarded quantification, and even tweaking the semantics of the separation logic mildly to ensure that it doesn't introduce arbitrary quantification), and develop a set of natural proof tactics that involve unfolding recursive definitions across the footprint of the program segment verified followed by an *uninterpreted* abstraction of recursive definitions. Furthermore, they encode these tactics using decidable SMT-solvable theories to build fast automatic proof techniques.

While the natural proof technique is very promising (Qiu et al. [147] describe 100+ data-structure programs automatically verified using their methodology), the tool they develop is only for a basic toy programming language consisting of heap manipulating commands. Consequently, it is not at all clear how the technique will fare in a realis-

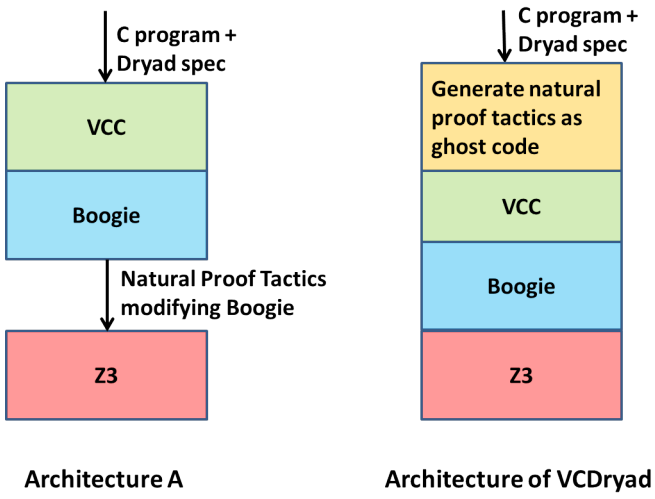


Figure 3.1: Choices of architecture for implementing natural proofs

tic programming language, with a complex semantics and a complex memory model with a varied set of underlying types. In particular, handling the memory model itself often requires quantification, and it is highly unclear how the natural proof technique will work in such a setting.

In this chapter, we describe an automated deductive verification tool, called VCDRYAD for C programs against DRYAD separation logic specifications, where proofs are automated using natural proofs, developing the ideas in Qiu et al. [147] to a real programming language. The tool extends VCC, the C program verifier from Microsoft Research [58] by augmenting the deductive verification tool with DRYAD and natural proofs.

The technical novelty in our work is the *lifting* of natural proof techniques from the verification-condition-generation level (as suggested in Qiu et al. [147]) to the code-level: we automatically synthesize *annotations* at the code-level so that VCC interpreting these will automatically search for natural proofs of the properties it verifies. As we describe below, this involves several intricate mechanisms in order to (a) describe heaplets and separation logic semantics using a custom-defined theory of sets in VCC that exploit decidable array-theories, (b) synthesize annotations so as to place important consequences of destructive heap updates (including function calls) so that VCC can recover properties after such a destruction using the local reasoning provided by separation logic, and (c) careful ways to write precise annotations, sometimes directly writing at the underlying BOOGIE level, so as to side-step VCC's complex modeling of the C semantics and memory model into BOOGIE.

We now describe the architecture of the tool and the issues that motivated it, the challenges in following this architecture, some details of the synthesis of annotations, and the evaluation of the tool.

Architecture: Lifting Natural Proofs to the Code-Level. There are two choices for building natural proof tactics into the VCC verification pipeline, as depicted in Figure 3.1. The obvious architecture suggested by Qiu et al. [147] is the first architecture (Architecture A), shown on the left in Figure 3.1. Here, we encode natural proof tactics involving unfolding recursive definitions and their uninterpreted abstraction, and the encoding of heaplet semantics and specification into decidable theories, while generating the verification conditions. This has obvious advantages— we have complete control of the logical formulas being generated, and seeing the entire formula allows us to exploit natural proofs to the widest extent possible. However, this architecture is incredibly hard to engineer for a real language. The biggest problem is that the specifications, now in DRYAD separation logic and not native VCC specifications, will have to be weaved through the entire stack. VCC uses a fairly complex translation from C programs to BOOGIE, using a *typed object model* (rather than the official untyped bit-based memory model for C). This typed object model allows simpler reasoning on types, memory accesses, etc., to achieve better scalability, less annotation overhead, and greater versatility in proving well-written C programs correct [60]. Weaving the new definitions through all this would be considerably hard. We also lose the ability for VCC/BOOGIE to handle specifications written in their syntax, including possible triggers sent to the SMT solver, the interaction of the specification with the memory model, etc. unless we carefully transform these layers as well. Furthermore, the various logical models for handling the C semantics and memory model by VCC/BOOGIE itself introduce quantified constraints [60] that fall in undecidable theories and VCC augments these with automatic trigger mechanisms (hundreds of such quantified formulas are generated even for very simple programs, with hundreds of triggers). It's unclear how the natural proof VC generation will handle these constraints and orchestrate their interaction with the other recursive constraints. We believe that incorporating natural proofs at the lowest level will be hard in any verification stack (not just VCC) that handles a complex programming language.

The approach we opted work in this work is the second architecture depicted in Figure 3.1, where we engineer natural proofs at the *code-level*, writing annotations at the VCC input level that force natural proofs to be discovered down the pipeline. We lose the advantage of being able to control the precise verification conditions generated to the SMT solver. However, as we show in this chapter, it is possible to do the translation of DRYAD specifications to first-order logic (using custom defined object sets to handle heaplet semantics and keep their manipulation within decidable theories) and encode the unfolding and abstractions of recursive definition tactics, all at the VCC level. We hence fully exploit the VCC/BOOGIE levels as black-boxes that manipulate our abstracted specifications, keeping our engineering entirely agnostic to the memory model handling and quantification trigger handling by these tools. While engineering the tool, we did occasionally look carefully at the Z3 constraints being generated and used this to formulate our VCC-level annotations in a way so as to ensure that the specification and reasoning that we add do not contribute to undecidability at the logic level, but this was minimal and the design is largely agnostic to the internals

of VCC and BOOGIE.

Annotation Synthesis. Our tool VCDRYAD hence purely synthesizes annotations using the VCC syntax, at the level of the C code. The tool interprets the C program and the specification written in DRYAD, and performs three main tasks:

- Translates the separation logic DRYAD to VCC's roughly first order syntax. It models heaplets defined by recursive DRYAD formulas as recursively defined sets of objects, where sets of objects and set-operations on them are modeled using point-wise functions on arrays that are amenable to automatic reasoning at the SMT level. It performs these translations for definitions, annotations and assertions throughout the code.
- The recursive definitions, including the recursive definitions of heaplets, are then modeled as entirely uninterpreted functions. However, the recursive definitions are captured logically and their precise definition is unfolded at every point of dereference in the program, capturing the two natural proof tactics. Our tool hence inserts these expansions throughout the code, doing some minimal static analysis to figure the right definitions to expand (based on the type of the variable being dereferenced, etc.).
- The most intricate part of the annotations are for function-calls and statements that perform destructive heap updates (these two are handled very similarly). Whenever there is a function call, VCC havocs all information it currently has, and we restore all information that we can derive from the heaplet semantics of the separation logic contract for the function being called.Intuitively, the pre/post condition for the called function implicitly defines the heaplet that could be modified, and we explicitly restore the field-pointers and the values of recursive definitions that were not affected by the function called.

Note that the annotations that we add are far removed from the *triggers* that VCC programmers are encouraged to use to aid proofs, which are meant to help provide instantiations of quantified constraints in order to prove them unsatisfiable. We write no triggers at all. The natural proof method that we employ seems to be a very different technique that tries to tie the recursion in the specification with the recursion/iteration in the code in order to extract simple inductive proofs of correctness.

VCDRYAD is hence a tool for C programs manipulating data-structures where only annotations of pre/post conditions and loop invariants are required. The tool is designed to make most reasoning of data-structures completely automatic, without the use of programmer-provided tactics and lemmas. However, when a proof fails, the programmer can still see and interact with the DRYAD specifications, its translation to first-order logic, and the automatically generated tactics to help the proof go through, since all our effort explicitly resides at the code-level.

Evaluation. The proof for any sound but incomplete system is in the pudding. Our main hypothesis going into the evaluation was that we can make natural proofs work for a complex language like C by exerting enough control through annotations at the code level. We give evidence to this hypothesis by developing the tool and experimenting with it on more than 150 C programs manipulating data-structures.

Our program suite consists of standard routines for data structure manipulation (i.e., singly-linked, doubly-linked list, and trees), real world programs taken from well-known open source projects (i.e., Glib and OpenBSD), custom OS kernel data structures, Linux kernel routines used in a software verification competition [34], and programs used to evaluate approaches based on decidable separations logic fragments reported in Piskac et al. [145] and Itzhaky et al. [98].

The tool VCDRYAD was able to prove *all* the above programs correct, automatically, without any user-provided proof tactics. With this automation, VCDRYAD presents a significant advance in automated reasoning in deductive verification for heap manipulation. While there are, of course, several other high-level properties (such as properties about graphs which are not recursively definable) for which effective natural proofs are not known, we believe that for most programs manipulating standard inductively defined data-structures, verification using recursion can be significantly automated.

3.2 DRYAD and Natural Proofs

In this section, we give a brief description of the DRYAD logic and the natural proof technique. We present only those details that are required for understanding the encoding to VCC annotations presented in this chapter. The interested reader may find a more complete and more formal specification of DRYAD in [147].

$i : Loc \to Int_L$	$sl : Loc \to \mathscr{S}(Loc)$	$si : Loc \to \mathscr{S}(Int)$	$msi : Loc \to \mathscr{MS}(Int)_L$	$p : Loc \to Bool$
$j \in Int_L$ Variables	$L \in \mathscr{S}(Loc)$ Variables	$S \in \mathscr{S}(Int)$ Variables	$MS \in \mathscr{MS}(Int)_L$ Variables	$q \in Bool$ Variables
$x \in Loc$ Variables	$c : Int_L$ Constant	$pf \in PF$	$df \in DF$	

$$
\begin{array}{rrcl}
Loc \text{ Term: } & lt & ::= & x \mid \texttt{nil} \\
Int_L \text{ Term: } & it & ::= & c \mid j \mid i(lt) \mid it + it \mid it - it \\
\mathscr{S}(Loc) \text{ Term: } & slt & ::= & \emptyset_L \mid L \mid \{lt\} \mid sl(lt) \mid slt \cup slt \mid slt \cap slt \mid slt \setminus slt \\
\mathscr{S}(Int) \text{ Term: } & sit & ::= & \emptyset_I \mid S \mid \{it\} \mid si(lt) \mid sit \cup sit \mid sit \cap sit \mid sit \setminus sit \\
\mathscr{MS}(Int)_L \text{ Term: } & msit & ::= & \emptyset_M \mid MS \mid \{it\}_m \mid msi(lt) \mid msit \cup msit \mid msit \cap msit \mid msit \setminus msit
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Formula: } & \varphi & ::= & \texttt{true} \mid \texttt{false} \mid q \mid p(lt) \mid \texttt{emp} \mid lt \overset{\overrightarrow{pf},\overrightarrow{df}}{\longmapsto} (\vec{lt},\vec{it}) \mid lt = lt \mid lt \neq lt \mid it \leq it \mid it < it \mid sit \leq sit \mid sit < sit \\
& & & \mid msit \leq msit \mid msit < msit \mid slt \subseteq slt \mid slt \not\subseteq slt \mid sit \subseteq sit \mid sit \not\subseteq sit \mid msit \sqsubseteq msit \mid msit \not\sqsubseteq msit \\
& & & \mid lt \in slt \mid lt \notin slt \mid it \in sit \mid it \notin sit \mid it \in msit \mid it \notin msit \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi * \varphi
\end{array}
$$

$$
\text{Recursive function : } f_{\overrightarrow{pf},\vec{v}}(x) \overset{def}{=} \text{ITE}\Big(\varphi_1^f(x,\vec{v},\vec{s}) : t_1^f(x,\vec{s}) \;;\; \ldots \;;\; \varphi_k^f(x,\vec{v},\vec{s}) : t_k^f(x,\vec{s}) \;;\; \text{default} : t_{k+1}^f(x,\vec{s})\Big)
$$

$$
\text{Recursive predicate : } p_{\overrightarrow{pf},\vec{v}}(x) \overset{def}{=} \varphi^p(x,\vec{v},\vec{s})
$$

Figure 3.2: Syntax of DRYAD

Syntax. DRYAD is a dialect of separation logic, and its syntax is given in Figure 4.1. It is a multi-sorted separation logic, supporting five sorts: location, integer, set/multiset of integers, and set of locations, and allows all standard operations over these sorts, as well as the separating conjunction from separation logic.

DRYAD disallows explicit quantification but allows user-provided recursive definitions, which in turn allow a form of guarded quantification. Inductively defined data structures can be defined naturally using such recursive definitions. Furthermore, DRYAD has a slightly different semantics from separation logic that ensures statically determined heaplets for recursive definitions.

There are several different kinds of terms in the logic— location terms, integer terms, "set of locations" terms, "set of integers" terms, and "multi-set of integers" terms; all except the first can use recursive definitions of the corresponding type. Formulas combine these terms in the usual ways: integers with respect to arithmetic relations, sets/multisets with respect to the $<$ (or $\leq$) relation, which checks whether all elements of the first set are less than (or equal to) all elements of the second set, membership in sets. Formulas are closed under conjunction and disjunction (negation needs to be pushed all the way in) and under the separation conjunction operator $*$.

We will consider as a running example binary search trees. Binary search trees can be defined recursively in DRYAD as:

$$\begin{aligned} bst_{\{l,r\}}(x) \quad \stackrel{def}{=} \quad & (x = \texttt{nil} \wedge \texttt{emp}) \ \vee \\ & \big(\ (x \stackrel{l,r,k}{\mapsto} \mathsf{left}, \mathsf{right}, \mathsf{m}) * \\ & (bst(\mathsf{left}) \ \wedge \ keys(\mathsf{left}) \leq m) * \\ & (bst(\mathsf{right}) \ \wedge \ \mathsf{m} \leq keys(\mathsf{right})) \ \big) \end{aligned}$$

In the definition above, $x \stackrel{l,r,k}{\mapsto} \mathsf{left}, \mathsf{right}, \mathsf{m}$ implicitly and guardedly quantifies $\mathsf{left}, \mathsf{right}, \mathsf{key}$ to denote the values of x's fields l, r and k; note that these are uniquely determined. The first use of $\leq$ in the above asserts that every integer in the left-hand set is at most the integer on the right; similar semantics holds for the second usage. The definition of *keys* is given by the following *if-then-else* term:

$$\begin{aligned} keys_{\{l,r\}}(x) \stackrel{def}{=} \mathrm{ITE}\big(& x = \texttt{nil} : \emptyset_I, \\ & (x \stackrel{l,r,k}{\mapsto} xl, xr, xk) : \ \ \{xk\} \cup keys(xl) \cup keys(xr)\big) \end{aligned}$$

Semantics. The semantics of DRYAD is consistent with standard separation logic for basic constants and connectives like emp, separating conjunction $*$, and other Boolean operations. However, DRYAD enforces an exact heaplet semantics for recursive definitions. Unlike standard separation logic, the heaplet for a recursive definition is uniquely determined— it always is the set of all locations reachable using certain field pointers, with possible *stopping* locations. These field pointers and stopping location terms are given syntactically in the definition.

For the definition of $bst_{\{l,r\}}(x)$, its heaplet is the set of all reachable locations from x via fields l and r, and can be

recursively defined. As a naming convention, let the name of the heaplet corresponding to a definition d be d_heaplet. Then the heaplet of bst can be defined using the following recursive definition (in classical logic, with standard least fixed-point semantics):

$$
\begin{aligned}
&bst\_\mathsf{heaplet}(x) \quad \stackrel{def}{=} \\
&\quad \mathrm{ITE}\big(x = \mathtt{nil} : \emptyset_L, \\
&\qquad (x \stackrel{l,r,k}{\mapsto} xl, xr, xk) : \\
&\qquad\quad \{x\} \cup bst\_\mathsf{heaplet}(xl) \cup bst\_\mathsf{heaplet}(xr)\big)
\end{aligned}
$$

Now to interpret the recursive definition $bst(x)$, on a heaplet with domain R, if R is exactly the set of locations described by the reach set ($bst\_\mathsf{heaplet}(x)$), then the semantics of bst is defined to be the *least fixed-point* that satisfies its definition. Otherwise $bst(x)$ is undefined.

The heaplet for *keys*, *keys*_heaplet, can be defined similarly. In general, for any recursive definition d in DRYAD, its corresponding heaplet definition $d\_\mathsf{heaplet}(x)$ *always* uniquely delineates the heap domain for $d(x)$. Note that the subscript $\{l,r\}$ indicates the fields via which the reachable heaplet is defined, but is usually omitted when clear from the context.

By deterministically delineating the heap domain for recursive definitions, DRYAD can syntactically determine the *domain-exactness* and the *scope* for any part of a DRYAD formula. Intuitively, the scope of a formula/term is the domain of the *minimum* heaplet required to interpret it; and a formula/term is domain-exact if it cannot be satisfied/evaluated when the heaplet domain is larger than the scope. Neither the scope nor the domain-exactness can be syntactically determined in standard separation logic. But in DRYAD, for each basic binary connective $t \sim t'$ (other than $*$, $\wedge$, $\vee$ and $\neg$),

$$
\begin{aligned}
\text{domain-exact}(t \sim t') &\stackrel{def}{=} \text{domain-exact}(t) \wedge \text{domain-exact}(t') \\
\text{scope}(t \sim t') &\stackrel{def}{=} \text{scope}(t) \cup \text{scope}(t')
\end{aligned}
$$

Then when interpreted on a heaplet with domain R, the connective $\sim$ has the normal semantics if:

- either t or t' is not domain-exact, and $\text{scope}(t \sim t') \subseteq R$; or
- both t and t' are domain-exact, and there exists R_1, R_2 such that $R = R_1 \cup R_2$, and t/t' has a well-defined semantics on R_1/R_2, respectively.

Otherwise, $t \sim t'$ has no well-defined semantics.

Translation to Classical Logic. The determinacy of the heap domain for recursive definitions is important for exploiting translation to quantifier-free classical logic using the theory of sets. We can translate DRYAD to classical logic that models heaplets as sets of locations constrained appropriately. As the heap domain for a recursive definition is a

```
BNode * bst_insert_rec(BNode * x, int k)
   requires bst(x) & (~ (k i-in keys(x))) ;
   ensures  bst(ret)
   ensures  keys(ret) s= (old(keys(x)) union (singleton k)))) ;
{
  if (x == NULL) {
    BNode * leaf = (BNode *) malloc(sizeof(BNode));
    _(assume leaf != NULL)
    leaf->key = k; leaf->l  = NULL; leaf->r = NULL;
    return leaf;
  } else  {
    BNode * xl = x->l; BNode * xr = x->r;
    if (k < x->key) {
      BNode * tmp = bst_insert_rec(xl, k); x->l = tmp;
      return x;
    } else {
      BNode * tmp = bst_insert_rec(xr, k); x->r = tmp;
      return x;
    } } }
```

Figure 3.3: Recursive Implementation of BST-Insertion

recursively-defined heaplet, we can translate each recursive definition from DRYAD to classical logic with recursion. For example, the DRYAD predicate *bst* defined above can be translated to the following definition:

$$
\begin{aligned}
\mathsf{bst}(x) \quad \stackrel{def}{=} \quad & (x = \mathtt{nil} \wedge \mathsf{bst\_heaplet}(\mathsf{x}) = \emptyset_L) \ \vee \\
& \big(x \neq \mathtt{nil} \\
& \wedge (x \notin \mathsf{bst\_heaplet}(x.l) \cup \mathsf{bst\_heaplet}(x.r)) \\
& \wedge (\mathsf{bst}(x.l) \ \wedge \ \mathsf{keys}(x.l) \leq x.k) \\
& \wedge (\mathsf{bst}(x.r) \ \wedge \ x.k \leq \mathsf{keys}(x.r)) \\
& \wedge \backslash\mathsf{disjoint}(\mathsf{bst\_heaplet}(x.l), \mathsf{bst\_heaplet}(x.r)) \\
& \wedge \mathsf{bst\_heaplet}(x) = \\
& \quad \{x\} \cup \mathsf{bst\_heaplet}(x.l) \cup \mathsf{bst\_heaplet}(x.r) \ \big)
\end{aligned}
$$

A similar translation can be done from *bst*_heaplet, *keys*, and *keys*_heaplet.

Natural Proofs. Natural proofs and DRYAD have been co-designed, where proofs exploit the purely recursive formulation provided in the logic, with no explicit quantification (DRYAD allows implicit quantification, but these are always *guarded*, and hence is uniquely determined in the context). Natural proofs are sound but incomplete proof tactics that work for many programs, and are derived from common tactics found in manual proofs. In particular, the work in [147] identifies two main tactics (see also previous work [125] and [159] where these tactics were studied earlier). The tactics are to (a) unfold recursive definitions across the footprint (the locations explicitly dereferenced in

the program) of the program segment being verified, and (b) to make the recursive functions uninterpreted. Intuitively, (b) causes too much loss of precision that (a) recovers by making sure that the semantics of recursive definitions are at least correctly described on the locations on the footprint. Natural proofs tend to find pure induction proofs— if a program segment calls a function f, then the natural proof would apply the pre/post condition of f to infer certain facts hold for recursive definitions when f returns, but will not unravel these definitions further, hence looking for a simple induction proof.

Consider the example of insertion into a binary search tree presented in Figure 3.3, with pre/post conditions written in DRYAD. After translating the pre/post conditions to classical logic (elaborated in Section 3.3), each Hoare-triple extracted from the program corresponds to a verification condition, which is a formula in classical logic with recursively defined bst, keys, etc. For example, the following formula is the verification condition for the case of inserting k into the left subtree of x (proving only the BST-ness is preserved):

$$\begin{array}{l}\Big[\ \mathsf{bst}(x) \wedge k \notin \mathsf{keys}(x) \wedge x \neq \mathsf{NULL} \wedge k < key(x) \wedge \\ \quad \Big(\mathsf{bst}(l(x)) \wedge k \notin \mathsf{keys}(l(x))\ ) \rightarrow \\ \qquad (\ \mathsf{bst}(tmp) \wedge \mathsf{keys}(tmp) = \mathsf{keys}(l(x)) \cup \{k\}\Big) \wedge \\ \quad l'(x) = tmp \wedge r'(x) = r(x)\ \Big] \ \longrightarrow\ \mathsf{bst}'(x)\end{array}$$

where bst' is the updated version of bst, defined using updated fields (l' and r'). Now we can prove it using the natural proof strategy. For this example, the footprint is simply x, the only dereferenced variable. Therefore, we unfold *all* recursive definitions, namely bst, keys, bst' and keys', and their corresponding heaplet definitions, on x. For instance, every occurrence of $\mathsf{bst}(x)$ is replaced with its unfolded definition presented before. Once we make those recursive definitions uninterpreted, the validity of the formula can be easily proved by a theorem prover supporting the theory of sets.

Qiu et al. [147] show that in many data-structure manipulating programs, the inductive invariant can often be used without further unfolding to prove the program correct, and hence natural proofs often exist. Moreover, the validity of the produced formulas is decidable by SMT solvers supporting the array property fragment [69]. This approach looks for natural proofs by encoding verification conditions appropriately, but we will be concerned about encoding natural proofs at the code-annotations-level, which is the main technical contribution describe in this chapter.

3.3 Translating DRYAD Specifications and Modeling Natural Proofs as Ghost Annotations

In this section, we describe in detail the way we leverage natural proofs for C programs against DRYAD specifications by encoding natural proof tactics automatically into carefully crafted ghost-code annotations. The ghost code is directly at the source code level, handled by VCC, and consists of first order annotations that fall into decidable theories handled by any standard SMT solver. These automatically generated annotations help VCC carry out an automatic natural proof of the C program, freeing the programmer from guiding proofs using proof tactics.

We describe the synthesis in three phases. The first is on translating recursive definitions to first-order VCC annotations capturing the definitions as both uninterpreted functions as well as defining unfoldings of them according to their recursive definitions. The second phase describes how DRYAD annotations in code (pre/post conditions, loop invariants, assertions, etc.) are translated to VCC specifications. Finally, in the third phase, which is the most complex, we encode natural proof tactics using ghost annotations, unfolding recursive definitions on footprints and preserving the heap that doesn't change across statements and functions that modify the heap.

3.3.1 Phase I: Translating Recursive Definitions

As VCC and BOOGIE specifications have to be written in classical logic, roughly first-order[1], we need to translate DRYAD separation logic specifications to classical logic.

The first step is to translate recursive definitions. As we mentioned in the previous section, each recursive definition *d* in DRYAD can be translated to two recursive definitions in classical logic: one is the classical-logic equivalent of d, the other one recursively defines the heap domain for d, namely d_heaplet, which is the reachable locations according to certain pointer fields. In this work, these recursive definitions in classical logic are not directly amenable for developing a BOOGIE/Z3-based verifier. Remember that our goal is to encode the natural proof tactics completely using VCC annotations and give up controlling the BOOGIE-level VC-generation. Therefore, to deploy the unfoldings and the formula abstraction at the VCC-level, we translate DRYAD definitions slightly differently. All the recursive definitions are now translated to *uninterpreted functions*. In addition we define predicates in VCC describing how to unfold the definitions at particular locations using their true recursive definitions.

As an example, for the DRYAD predicate *bst* which is recursively defined as in Section 3.2, we can translate it to bst and bst_heaplet, both uninterpreted, and define how they should be unfolded:

```
_(pure \bool bst(struct node * hd) _(reads \universe()) ;)
_(pure \oset bst_heaplet(struct node * hd) _(reads \universe()) ;)
```

[1]The logic is typically first-order logic but over a richer class including maps, sets, arrays, etc., and further allows pure recursive functions described in FOL as well.

```
_(pure \bool unfold_bst(struct node * hd) _(reads \universe())
  _(ensures \result == (bst(hd) ==
    ((hd == NULL && bst_heaplet(hd) == {})
    ||(hd != NULL
      && (bst_heaplet(hd) == {hd} \union bst_heaplet(hd->l)
                                  \union bst_heaplet(hd->r))
      && bst(hd->l) && bst(hd->r)
      && \intset_lt_set(keys(hd->r), hd->key)
      && \intset_lt_set(hd->key, keys(hd->r))
      && \disjoint(bst_heaplet(hd->l), bst_heaplet(hd->r))
      && !(x \in bst_heaplet(hd->l)
                          \union bst_heaplet(hd->r))))));)
_(pure \bool unfold_bst_heaplet(struct node * hd)
    _(reads \universe())
    _(ensures \result == (
        (hd == NULL && bst_heaplet(hd) == {}) ||
        (hd != NULL && bst_heaplet(hd) ==
            ({hd} \union bst_heaplet(hd->l)
                    \union bst_heaplet(hd->r))))) ;)
```

The predicates unfold_bst and unfold_bst_heaplet mimic the unfoldings of the recursive definitions bst and bst_heaplet, respectively. When they are asserted on a location n, they guarantee that bst(n) and bst_heaplet(n) can be constructed from the evaluations of these functions on its neighbors, namely bst(n.l), bst_heaplet(n.l), bst(n.r), and bst_heaplet(n.r).

3.3.2 Phase II: Translating Logical Specifications

Using the determined heaplet semantics of DRYAD, Qiu et al. [147] show that DRYAD logic formulas can be translated to classical logic, still quantifier-free, preserving the heaplet semantics with respect to a given heaplet H. We follow a similar recursive translation as in [147], but adapt it to VCC syntax and the translation of recursive definitions presented above. Figure 3.4 shows this translation. We omit the formal definition for the scope function, which intuitively is the *minimum* heap domain required to evaluate the formula/expression (see [147] for details). Given a pre-/post-condition or a loop invariant φ in DRYAD, we introduce a ghost set variable G in the translation, denoting the local heaplet manipulated by the current function, and replace φ with $T_{VCC}(\varphi, \mathsf{G})$. For each function/method m in the program, we assume its pre-/post-conditions are domain-exact, i.e., we can statically compute the required

$$
\begin{array}{rcl}
T_{VCC}(\mathit{var} \,/\, \mathit{const},\, \mathsf{G}) & \equiv & \mathit{var} \,/\, \mathit{const} \\
T_{VCC}(\{t\} \,/\, \{t\}_m,\, \mathsf{G}) & \equiv & \{t\} \,/\, \{t\}_m \\
T_{VCC}(t \;\mathrm{op}\; t',\, \mathsf{G}) & \equiv & T_{VCC}(t,\mathsf{G}) \;\mathrm{op}\; T_{VCC}(t',\mathsf{G}) \\
T_{VCC}(f(lt),\, \mathsf{G}) & \equiv & \mathrm{ITE}\left(\mathsf{f\_heaplet}(lt) = \mathsf{G},\, \mathsf{f}(lt),\, \texttt{undef}\right) \\
T_{VCC}(\texttt{true} \,/\, \texttt{false},\, \mathsf{G}) & \equiv & \texttt{true} \,/\, \texttt{false} \\
T_{VCC}(\texttt{emp},\, \mathsf{G}) & \equiv & \mathsf{G} = \{\} \\
T_{VCC}(lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt},\vec{it}),\, \mathsf{G}) & \equiv & \mathsf{G} = \{lt\} \\
 & & \wedge \bigwedge_{pf_i} T_{VCC}(lt,\mathsf{G}).pf_i = T_{VCC}(lt_i,\mathsf{G}) \\
 & & \wedge \bigwedge_{df_i} T_{VCC}(lt,\mathsf{G}).df_i = T_{VCC}(it_i,\mathsf{G}) \\
T_{VCC}(p(lt),\, \mathsf{G}) & \equiv & \mathsf{p}(lt) \wedge \mathsf{G} = \mathsf{p\_heaplet}(lt) \\
T_{VCC}(t \sim t',\, \mathsf{G}) & \equiv & \begin{cases} t \sim t' \\ \qquad \text{if } t \sim t' \text{ is not domain-exact} \\ t \sim t' \wedge \mathsf{G} = \mathrm{scope}(t \sim t') \\ \qquad \text{otherwise} \end{cases} \\
T_{VCC}(\varphi \wedge \varphi',\, \mathsf{G}) & \equiv & T_{VCC}(\varphi,\mathsf{G}) \wedge T_{VCC}(\varphi',\mathsf{G}) \\
T_{VCC}(\varphi \vee \varphi',\, \mathsf{G}) & \equiv & T_{VCC}(\varphi,\mathsf{G}) \vee T_{VCC}(\varphi',\mathsf{G}) \\
T_{VCC}(\varphi * \varphi',\, \mathsf{G}) & \equiv & \begin{cases} T_{VCC}\big(\varphi,\, \mathrm{scope}(\varphi)\big) \;\wedge\; T_{VCC}\big(\varphi',\, \mathrm{scope}(\varphi')\big) \\ \quad \wedge\; \mathrm{scope}(\varphi) \cup \mathrm{scope}(\varphi') = \mathsf{G} \\ \quad \wedge\; \mathrm{scope}(\varphi) \cap \mathrm{scope}(\varphi') = \emptyset \\ \qquad \text{if both } \varphi \text{ and } \varphi' \text{ are domain-exact} \\ \mathrm{scope}(\varphi) \subseteq \mathsf{G} \;\wedge\; T_{VCC}\big(\varphi,\, \mathrm{scope}(\varphi)\big) \;\wedge \\ T_{VCC}\big(\varphi',\, \mathsf{G} \setminus \mathrm{scope}(\varphi)\big) \\ \qquad \text{if only } \varphi \text{ is domain-exact} \\ \mathrm{scope}(\varphi') \subseteq \mathsf{G} \;\wedge\; T_{VCC}\big(\varphi',\, \mathrm{scope}(\varphi')\big) \;\wedge \\ T_{VCC}\big(\varphi,\, \mathsf{G} \setminus \mathrm{scope}(\varphi')\big) \\ \qquad \text{if only } \varphi' \text{ is domain-exact} \\ T_{VCC}\big(\varphi,\, \mathrm{scope}(\varphi)\big) \;\wedge\; T_{VCC}\big(\varphi',\, \mathrm{scope}(\varphi')\big) \\ \quad \wedge\; \mathrm{scope}(\varphi) \cup \mathrm{scope}(\varphi') \subseteq \mathsf{G} \\ \quad \wedge\; \mathrm{scope}(\varphi) \cap \mathrm{scope}(\varphi') = \setminus\texttt{emptyset} \\ \qquad \text{if neither } \varphi \text{ nor } \varphi' \text{ is domain-exact} \end{cases}
\end{array}
$$

Figure 3.4: Translating DRYAD specs to VCC specs (with respect to a heaplet G)

heaplet, namely $\mathsf{G\_pre\_m}(\vec{p})$ and $\mathsf{G\_post\_m}(\mathsf{ret}, \vec{p})$, where $\vec{p}$ is the location parameters of m. Let the precondition of m be $\varphi_{\mathsf{pre-m}}(\vec{p})$ and let the postcondition of m be $\varphi_{\mathsf{post-m}}(\mathsf{ret}, \vec{p})$; then in VCC, we translate them to the precondition $T_{VCC}(\varphi_{\mathsf{pre\text{-}m}}(\vec{p}), \mathsf{G\_pre\_m}(\vec{p}))$ and postcondition $T_{VCC}(\varphi_{\mathsf{post\text{-}m}}(\mathsf{ret}, \vec{p}), \mathsf{G\_post\_m}(\mathsf{ret}, \vec{p}))$.

The translation of post-conditions is a bit more tricky, as the postcondition can refer to expressions or subformulas that are evaluated on the state *before* the function call, by using the $\backslash\mathsf{old}()$ function (such old-state lookups are not allowed in [147]). In our translation, old expressions/formulas are translated in a *heapless* manner, i.e., the heaplet with formulas involving the pre-state are considered to be empty. This, in essence, is equivalent to having some properties of the pre-state recorded in auxiliary ghost variables that are assigned at the beginning of the function, and using them in the post-condition (where they will be heapless, since they are state variables).

The current heaplet G is properly maintained using ghost-variable updates during the program execution (see Section 3.3 for details on how this is done).

Consider again the bst insertion algorithm presented in Figure 3.3. Note that there are no proof tactics provided by the user (in contrast, the verification of such a routine in a tool like VeriFast would contain a large number of proof tactics and lemmas tying the implementation to the specification [2]).

Using the translation we just described, the precondition gets translated in VCDRYAD to the VCC precondition:

```
_(requires bst(x) && !\intset_in(k, keys(x)))
_(requires bst_heaplet(x) == keys_heaplet(x))
_(requires G == bst_heaplet(x))
```

and the postcondition gets translated to:

```
_(ensures bst(\result)
_(ensures keys(\result) == \intset_union(\old(keys(x)), k)))
_(ensures G == bst_heaplet(\result))
_(ensures bst_heaplet(\result) = keys_heaplet(\result))
```

3.3.3 Phase III: Natural Proofs for VCC

We now describe the core of the annotation synthesis for forcing VCC to find natural proofs. Our annotation synthesizer simply instruments ghost code before and after each statement of the program, with some help of some (local) static analysis to figure out the parameters for the instrumentation. Figure 3.5 gives the precise ghost code instrumented before and/or after each statement.

The ghost code for each statement does four main things to encode inferable facts from the program state resulting from the execution of the statement:

[2] http://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/sorted_bintree.c.html

Stmt	After Instrumentation	Comment
u = $\langle expr \rangle$	u = $\langle expr \rangle$;	
u = v.f (assume v is of type T)	assume $\bigwedge_{d \in defs(T)}$(_dryad_unfold_d(v) ∧ _dryad_unfold_d_heaplet(v)); _(ghost T * _dryad_fp_$\langle i \rangle$ = v;) _(ghost \oset _dryad_scope_$\langle i \rangle$ = allLocFields(v);) u = v.f;	(Unfold dereferenced loc) (State memoization) (State memoization)
u = malloc() (assume u is of type T)	u = malloc() _(ghost G = \oset_union(G, {u});)	(Current heaplet update)
free(u) (assume u is of type T)	free(u) _(ghost G = \oset_diff(G, {u});)	(Current heaplet update)

u.f = v
(assume u is of type T)
	assume $\bigwedge_{d \in defs(T)}$(
	_dryad_unfold_d(u) ∧ _dryad_unfold_d_heaplet(u));
	_(ghost T * _dryad_fp_$\langle i \rangle$ = u;)
	_(ghost \oset _dryad_scope_$\langle i \rangle$ = allLocFields(u);)
	_(ghost \state _dryad_S_$\langle i \rangle$ = \now();)
	
	u.f = v;
	
	assume $\bigwedge_{p \in FP(i)} \bigwedge_{d \in defs(p)}$(
	_dryad_unfold_d(p) ∧
	_dryad_unfold_d_heaplet(p));
	assume $\bigwedge_{p \in EFP(i)} \bigwedge_{d \in defs(p)}$(
	(! \oset_in(u, d_heaplet(p)))
	==>
	(d(p) == \at(_dryad_S_$\langle i \rangle$, d(p)) &&
	d_heaplet(p) == \at(_dryad_S_$\langle i \rangle$, d_heaplet(p))));
		(Unfold dereferenced loc)
		(State memoization)
		(State memoization)
		(State memoization)
		
		(Unfolding on entire footprint)
		
		(Preserving definitions)

u = m($\vec{v}$, $\vec{z}$)
(assume $\vec{v}$ is
location parameters
and $\vec{z}$ is int parameters)
	_(ghost \state _dryad_S_$\langle i \rangle$ = \now();)
	
	u = m($\vec{v}$, $\vec{z}$);
	
	assume $\bigwedge_{p \in FP(i)} \bigwedge_{d \in defs(T_p)}$(
	_dryad_unfold_d(p) ∧ _dryad_unfold_d_heaplet(p));
	assume $\bigwedge_{p \in EFP(i)} \bigwedge_{d \in defs(T_p)}$(
	\oset_disjoint(G_pre_m($\vec{v}$), d_heaplet(p))
	==>
	(d(p) == \at(_dryad_S_$\langle i \rangle$, d(p)) &&
	d_heaplet(p) == \at(_dryad_S_$\langle i \rangle$, d_heaplet(p))));
	assume $\bigwedge_{p \in FP(i)} \bigwedge_{f \in flds(T_p)}$(
	(! \oset_in(p, G_pre_m($\vec{v}$)))
	==>
	\at(_dryad_S_$\langle i \rangle$, p.f) == p.f);
	_(ghost G = \oset_union(
	\oset_diff(G, G_pre_m($\vec{v}$)),
	G_post_m(u));)
		(Current state memoization)
		
		(Unfold on entire footprint)
		
		(Preserving definitions)
		
		(Preserving fields)
		
		(Current heaplet update)

Figure 3.5: Ghost code instrumented for every statement (*assuming the current statement is at position i in the program*)

Unfolding: Assumptions that the unfolding of recursive definitions hold at certain footprint locations;

Preservation: Inferences that fields and recursive definitions on locations that are maintained across destructive updates and function calls;

Current heap update: Updating the current heap maintained by the ghost variable G, after memory allocations and function calls;

State memoization: Create ghost variables to remember the state of the program at various points so that annotations can refer back to these states later to update information on locations pointed to by program variables at this state.

The current heaplet G is maintained as the domain of the heap changes. Since the current heaplet changes only with a `malloc`, `free` or a function call, we instrument different statements for each case. We instrument $\mathsf{G} = \mathsf{G} \cup \{x\}$ after each statement of the form x = malloc(), and we instrument $\mathsf{G} = \mathsf{G} \setminus \{x\}$ after each statement of the form free(x). After a function call to m, G can be updated by excluding G_m_pre and then including G_m_post: $\mathsf{G} = (\mathsf{G} \setminus \mathsf{G\_m\_pre}) \cup \mathsf{G\_m\_post}$. Note that when a DRYAD loop invariant is translated to a classical-logic formula with respect to current heaplet variable G, the obtained formula is still a valid loop invariant, as the current heap G is maintained using appropriate ghost updates through each iteration.

In order to simplify presentation, we will assume that in the program, location dereferences only appear in the form of u = v.f or u.f = v; all other statements (including conditions) with dereferences can be split into simpler ones, for example, the statement u.f.g = v can be split into two sequential statements: tmp = u.f; tmp.g = v.

We show in Figure 3.5 the ghost code instrumented for each basic statement. More complex statements such as if..then..else.., while, etc., are not transformed, i.e., they remain as they are, with the leaf statements transformed using the table.

The instrumentation relies on two sets of location variables: $FP(i)$ and $EFP(i)$, assuming i is the position of the current program statement (program counter). Intuitively, $FP(i)$ is the *footprint*, the set of dereferenced location variables, possibly memoized in previous locations using ghost variables. $EFP(i)$ is the *extended footprint* which includes all location variables in scope (where variables at other locations are captured using ghost-variables) as well as the ghost-variables that capture the locations obtained by dereferencing using location fields from these variables in the program. Variables in $FP(i)$ are of the form _dryad_fp_$\langle j \rangle$. Whenever a dereference u.f appears in a statement at position j, there is a ghost variable _dryad_fp_$\langle j \rangle$ defined before the dereferencing, remembering where u points to; and there is a ghost set variable _dryad_scope_$\langle i \rangle$ that remembers all location fields of the dereferenced variable. Then _dryad_fp_$\langle j \rangle$ and _dryad_scope_$\langle j \rangle$ will be added to $FP(i)$ and $EFP(i)$, respectively, as long as it is visible at the statement i. Both $FP(i)$ and $EFP(i)$ are computed by a simple local static analysis of the program before the

instrumentation, and we omit the details of their computation in this chapter.

We now briefly describe what gets instrumented for each type of statement, following Figure 3.5. For statements that do not update the heap (u = ⟨*expr*⟩ where ⟨*expr*⟩ is an expression without dereferencing), no instrumentation is done. For dynamic memory allocation (malloc() and free()), we only update the current heap variable *G* by including or excluding the location allocated or freed, respectively.

For a field lookup (u = v.f), we unfold all pertinent recursive definitions d and unfold the recursive definition corresponding to the heaplet of d (d_heaplet) defined on u before the lookup. Moreover, the location pointed to by v is stored in a ghost pointer associated with the current program location *i* (dryad_fp_*i*), to remember that this location is part of the footprint. The locations pointed to by the different fields from the dereferenced location v are also stored in a ghost variable (of type *object set*) pertaining to the current location (_dryad_scope_*i*). These field pointers are remembered so that we can restore them later after a destructive update.

Instrumentation for destructive updates (u.f = v) is more complex. We first unfold all pertinent recursive definitions for the location pointed to by u (since the heap is updated by the statement). We also store the current location u and the locations pointed to by its fields in ghost variables, as in the previous case. Then, after the statement, we unfold all recursive definitions on all locations in the footprint. The set $FP(i)$ statically captures the set of variables of the form dryad_fp_*j*, for various program locations in scope. We unfold recursive definitions on all of these locations *in the current program state*. Finally, we also explicitly infer the fact that for any location p in scope and any recursive definition d, if the heaplet corresponding to the definition (d_heaplet(p)) does not contain u, both the heaplet and the definition itself remain unchanged after the update.

Function calls are handled similar to destructive updates, but with a more complex inference after the call. First, we explicitly infer the recursive definitions that are maintained across the function call when the heaplet of the definition and the heaplet modified by the function (G_pre_m($\vec{v}$)) are disjoint. Secondly, as VCC havocs the entire heap after a function call (since the modified set is unspecified), we need to explicitly infer that the field pointers from a location in the footprint p.f is unchanged whenever p is disjoint from the heaplet (G_pre_m($\vec{v}$)) the function call modifies. Finally, the current heap variable G also gets updated (removing the heaplet of the pre-condition and adding the heaplet of the post-condition).

Regarding the soundness of the assumption annotations that we introduce, note that *unfold* definitions simply declare the recursive definition for a location and hence are always sound, and our tool liberally strews them across all possible locations touched by the program. The *preservation* assumptions, on the other hand, crucially rely on the heaplet semantics of DRYAD, and have to be instrumented carefully.

3.4 Design and Implementation of VCDRYAD

We engineered our tool VCDRYAD [3] as an extension to the (open-source) deductive verification tool VCC, but restricted to sequential C programs. The tool essentially processes C programs with DRYAD specifications, translating the specifications to first-order ghost-code as described in the Section 3.3 and Figure 3.5. Currently VCDRYAD does not handle all complexities of the C language, in particular we forbid function pointers and pointer arithmetic. The effort in engineering the tool was about 1 person year, where most time was spent in building the precise ghost-code that we needed by inserting it both at the VCC level and to the BOOGIE level (the latter is done through VCC's macros that provide injection to the BOOGIE level). The tool is written in F#, extending the VCC transformers written in the same language.

3.4.1 Encoding Sets and Multisets of Integers and Locations

Lifting natural proof tactics from the verification-condition-level to the code-level calls for careful encoding of integer and location *sets*. Our first attempt was to encode integer and location sets (in particular, heaplets) using VCC's specification primitives and reusing existing object set definitions. In particular, the integer sets/multisets were encoded using VCC's map, with set operations described using lambda functions over the maps. VCC creates a custom axiomatization of these maps even though it translates VCs to BOOGIE language which itself provides support for maps. However, we were not able to prove even simple properties without adding additional axioms describing basic properties of sets.

After several verification attempts and by examining the output that VCC/BOOGIE provided to Z3, we decided to abandon the above approach, and instead proceeded to model integer and location sets differently. Integer sets and heaplets in DRYAD can be modeled in a decidable theory, as described in Qiu et al [147], using particular custom maps and the decidable array property fragment. Working at the VC level, the work in [147] had full control over the formulas passed to the underlying SMT solver, and hence could perform this encoding easily. We proceeded to model sets of locations using the type (`\objset`), which is used in VCC to handle object ownership. However, to remove the source of incompleteness mentioned above, we encoded sets of type T as Arrays from T to Boolean using Z3's extended theory of arrays [69]. By carefully examining the Z3 output, we removed all forms of universal quantification that emanated from our annotations by modeling in the above decidable theory and in the decidable array property fragment theory [41]. VCC/BOOGIE itself generates universally quantified formulas to capture C's memory model—in practice we found that the triggers provided by VCC/BOOGIE handled this part well enough, and so we let those be. We implemented the encoding of our sets at the BOOGIE level, using stubs provided in VCC's header prelude file.

[3] `http://web.engr.illinois.edu/~pek1/vcdryad`

3.4.2 DRYAD Type Invariants

Each DRYAD definition can be thought as a type (object) invariant. The notion of object invariants is typically associated with strongly typed object-oriented languages. Strong typing disallows distinct objects and fields to overlap. In C, however, the notion of object is weaker, it means that type definitions provide a way of interpreting a chunk of memory. This means that objects in C can overlap almost arbitrarily. Therefore, sound verification of C programs typically requires an untyped memory model, which unfortunately comes with high performance and annotation overhead. VCC provides a sound and efficient typed memory model for C by maintaining object validity through invariants and proof obligations that guarantee that objects do not alias. In VCC aliasing is achieved through ownership annotations referring to memory regions [58]. The handling of aliasing is the key difference between our approach and that of VCC. Instead of the ownership methodology, we use separation logic to describe object disjointness and employ natural proof tactics through ghost code to enable automated reasoning (as described in Section 3.3). However, we do leverage all the reasoning VCC performs to ensure object validity that does not rely on ownership specifications.

To integrate our approach within the VCC framework we derive DRYAD type invariants from translation of recursive DRYAD specifications to first-order logic VCC specifications (see Sec. 3.3.1). Given this translation, we need to extract suitable information to perform unfolding and preserving definitions across the destructive updates and function calls. From the translated DRYAD definitions we have to extract information describing data structure invariants (such as singly-linked list, binary search tree, etc.), and its footprint definitions. We use a light-weight static analysis to find which DRYAD predicates and functions are associated with data structure definitions (expressed using `struct`), and the associated DRYAD predicates and their heaplet definitions. Moreover, we perform static analysis to determine fields of a data structure on which DRYAD specifications depend and vice versa. We use that information when performing the unfolding and preserving DRYAD definitions and field pointers as described in Figure 3.5 in Section 3.3.

3.4.3 Axioms Relating Recursive Definitions

We follow the natural proof methodology [147] in adding several axioms that relate different recursive definitions for data-structures, including those that relate partial data-structures to complete ones (like list segments to lists) and those that unfold recursive definitions in the reverse direction (like unfolding linked list segments from the tail to the left). Note that these axioms are provided for each *class* of data-structures (like linked list segments, doubly linked lists, etc.) but are not specific to the program being verified. The success of automatic verification does crucially depend on these axioms, and automating these axioms would be interesting future work.

3.4.4 Debugging Unsuccessful Verification Attempts

The VCC/BOOGIE/Z3 pipeline being the underlying framework for our verifier provides additional help in the process of understanding unsuccessful verification attempts. The reasons for failed attempts are typically due to mistakes in the code or in the specifications (it is easy to write invariants that are correct but not inductive). When VCC reports that the property did not verify it means that Z3 could not discharge the VC and it produces a counterexample model. The counterexample model can be interpreted with VCC's BOOGIE Verification Debugger (BVD) [115], which can be useful in debugging the failed attempts. Z3 Inspector is also a useful tool that shows the precise properties the underlying solver is trying to prove at the code level. Our experience in writing and specifying 150 programs included several attempts where we wrote a wrong program or specification, but where these tools helped us find and debug them.

3.5 Evaluation

We evaluated our tool VCDRYAD on more than 150 data-structure manipulating routines [4], which in turn exercised the natural proof technique for C for thousands of verification conditions. The programs were written with user-defined data structure definitions and annotated with preconditions, postconditions, and loop invariants. No further proof tactics were provided.

VCDRYAD handled *all* our programs automatically. Table 3.1 shows the result of the experiments. We follow the naming convention that routines with suffix `rec`/`iter` denote recursive/iterative implementations. Our routines include standard manipulations of singly-linked, doubly-linked, circular lists, binary trees, AVL trees, etc. Some of these structures are hard to define recursively (e.g., doubly-linked and circular lists), and also difficult to verify inductively. Furthermore, these routines were verified for full functional correctness, including properties involving the precise set of keys modified, the balancing of trees, etc.

We used our tool to verify routines taken from various real world programs. In particular, we verified a large set of routines manipulating singly-linked and doubly-linked lists from a well-known Glib C library, and queue manipulations as implemented in the OpenBSD operating system. Furthermore, we verified custom data structure procedures developed in ExpressOS, an OS that uses formal verification to provide stronger security guarantees. Our set of benchmarks also includes some heap manipulation programs from a software verification competition (SV-COMP) [34]. Finally we verified the programs used to evaluate recent tools that handle weak but decidable fragments of separation logic, namely GRASShopper [145] and AFWP [98].

Figure 3.6 shows the number of manual and automatically generated annotations for the various routines, sorted

[4] `http://web.engr.illinois.edu/~pek1/vcdryad/examples/`

Benchmark and total LOC	Routine	Time (s) / Routine	Benchmark and total LOC	Routine	Time (s) / Routine
Singly-linked list 130 LOC	insert_front, copy_rec, insert_back_rec, append_rec, find_rec, reverse_iter, delete_all_rec	< 1	Sorted list 260 LOC	find_rec, find_last, insert_sort_rec, delete_all_rec, reverse_iter	< 1
Doubly-linked List 120 LOC	insert_front, insert_back_rec, append_rec, mid_insert, delete_all, mid_delete, meld	< 1		insert_iter	1
				concat_sorted	3
				merge_rec	8
Circular list 110 LOC	insert_front, insert_back_rec, delete_front, delete_back_rec	< 1		quick_sort_iter	6
				insert_sort_iter	20
BST 140 LOC	find_rec, find_iter, delete_rec	< 1	Treap 170 LOC	find_rec	< 1
				delete_rec	2
	insert_rec	1		insert_rec	10
	remove_root_rec	1		remove_root_rec	35
AVL-tree 320 LOC	leftmost_rec	< 1	Tree Traversals 100 LOC	preorder_rec, inorder_rec, postorder_rec	< 1
	avl_insert	4			
	avl_delete	20		inorder_tree_to_list_rec	3
	avl_balance	260			
glib/gslist.c Singly-Linked list 550 LOC	free, find, prepend, last, concat, append, insert_at_pos, insert_before, remove, remove_link, delete_link, reverse, nth, nth_data, position, index, length	< 1	glib/glist.c Doubly-Linked list 170 LOC	free, prepend, reverse, nth, nth_data, position, find, index, last, length	< 1
	remove_all	4	OpenBSD Queue 70 LOC	simpleq_init, simpleq_insert_head, simpleq_insert_tail, simpleq_remove_head	< 1
	copy	5			
	merge_sorted_lists	98			
	insert_sorted_list	42			
	merge_sort	20			
SV-COMP Heap Manipulation 150 LOC	alloc_or_die_slave, dll_insert_slave, dll_create_slave, dll_destroy_slave, list_head_init, list_head_add, list_del	< 1	ExpressOS Memory Region 80 LOC	memory_region_init, create_user_space_region, split_memory_region	< 1
GRASS-hopper [145] Singly-Linked List 160 LOC	sl_concat, sl_copy, sl_dispose, sl_insert, sl_reverse, sl_traverse1, sl_traverse2	< 1	GRASS-hopper [145] Singly-Linked List 130 LOC	rec_concat, rec_copy, rec_dispose, rec_filter, rec_insert, rec_remove, rec_reverse, rec_traverse	< 1
	sl_filter, sl_remove	3			
GRASS-hopper [145] Doubly-Linked List 170 LOC	dl_concat, dl_copy, dl_dispose, dl_insert, dl_remove, dl_reverse, dl_traverse	< 1	GRASS-hopper [145] Sorted List I 80 LOC	sls_concat, sls_dispose, sls_reverse, sls_traverse1, sls_traverse2, merge_sort_rec	< 1
	dl_filter	5			
GRASS-hopper [145] Sorted List II 270 LOC	merge_sort_split	3	AFWP [98] Singly-Linked and Doubly-Linked List 240 LOC	SLL_create, SLL_delete_all, SLL_delete, SLL_filter, SLL_find, SLL_last, SLL_merge, SLL_reverse, SLL_rotate, SLL_swap, DLL_fix, DLL_splice	< 1
	sls_pairwise_sum	3			
	sls_insert	3			
	sls_remove	1			
	sls_filter	2			
	insertion_sort	30			
	sls_merge	7			
	sls_double_all	39			
	sls_copy	55		SLL_insert	3

Table 3.1: Experimental results of verification of 152 routines

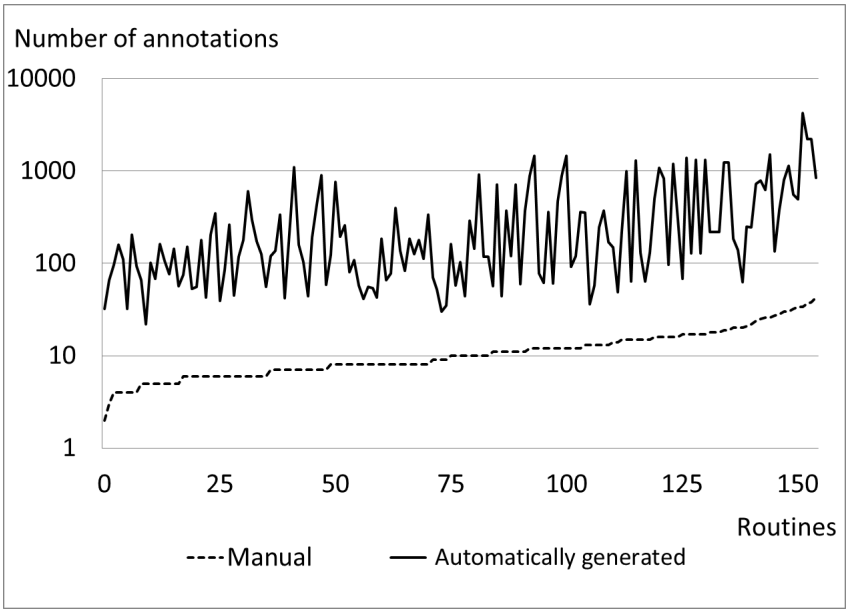


Figure 3.6: Comparison of manual annotation versus automatically generated annotations.

in ascending order of the manual annotations they had. Note that y-axis, depicting number of annotations, is in a logarithmic scale. The tool adds 3X to 150X new annotations over the number of manual annotations ($\sim 30X$ in average). It is interesting to see that though we add lots of annotations (up to 4000 annotations for a routine), these annotations are in a much simpler theory (uninterpreted functions, arithmetic, plus decidable quantified theories of arrays), and actually helps the prover. Also, note that the annotations that we add are either assumptions or ghost variable updates, and hence the *number* of verification conditions do not increase. Manually provided proof tactics commonly used span a wide range, and natural proofs seem to be a good *uniform* way to discover most proofs automatically.

3.6 Related Work

The literature on verification of data-structure manipulation in programs is rich. We focus mainly on related work on logics and deductive verification techniques for data-structures. Separation logic [149, 141] has emerged as a succinct heap logic that is amenable to local reasoning. Decidable fragments of separation logics were identified first in [29, 30], and the tool SMALLFOOT provides automated reasoning for a restricted class of linked lists and trees. Decidable fragments of separation logic have been investigated in several recent works (see [64, 137, 145, 98, 87]) and a recent result for a considerably more expressive logic for bounded tree width structures is known [97] (though its practical applicability is unclear as it reduces reasoning to MSO reasoning). Non-separation logics that are decidable

are also known [141, 125, 133].

Suter et al. [159, 160] explore proof tactics to prove properties of algebraic data-structures manipulated by functional programs. The work on natural proofs [125, 147] extend such tactics to imperative programs, where the proof tactics unfold recursive definitions across the footprint manipulated by imperative code and use formula abstraction to uninterpreted functions. Chin et al. [55] also exploit recursion, but reduce verification to standard logical theories that are not necessarily decidable.

Comparison with other deductive verification tools: There are, in general, three layers of annotations common in current deductive verification tools: (A) the specification, written as pre/post/assert annotations in code, (B) loop invariants and strengthening of pre/post conditions that encode *inductive* invariants, and (C) proof tactic advises from the user to the underlying prover to prove the resulting verification conditions.

Annotations of type (A) that encode specifications are always required. Most current tools, like VeriFast [144], Bedrock [56], VCC [58], and DAFNY [118], require all three levels (A, B, and C) of annotations for proving code correct.

The tool VCDRYAD presented in this chapter and our previous tool in [147] require A and B annotations only, relieving the programmer from writing annotations of type C, for data-structure verification.

The annotations encoding proof tactics (C) are clearly the hardest to write for a programmer, and typically require expertise in the workings of the underlying prover. We now describe what these proof-tactic annotations look like in the other tools, to give an idea of the relief of burden provided by our work to the programmer.

VCDRYAD vs VCC: We first compare our tool with VCC, over which we build. VCC compiles to the lower-level logical language BOOGIE, which generates verification conditions and passes them to an SMT solver. The SMT solvers are given formulas in undecidable theories but are aided by triggers using E-matching, model-quantifier instantiation, etc. [82], and these tactics can be specified at the level of the C program itself using VCC.

Consider an implementation of a function `list_find` that finds a key in singly-linked, and returns 1 iff the key is found [5]. Note that VCC does not support separation logic, and hence even the specification is complex and very hard to write (ghost variables are used to represent the set of keys in the list, and to map values to nodes). Invariants specify that abstract keys in the list correspond to data fields of nodes owned by the list and that pointers from the list nodes point to the nodes in the list structure. Moreover, acyclicity is encoded by assigning a strictly increasing ghost number to each node in the list and guarantee that each node can be reached by following the list.

Also, VCC requires hints using assertions (see lines 33 and 41), without which it would not be able to prove the loop invariants. In more complex examples, VCC would even need explicit triggers [6].

[5] `http://web.engr.illinois.edu/~pek1/vcdryad/cmp/VCDryad_vs_VCC.html`

[6] `http://vcc.codeplex.com/SourceControl/latest#vcc/Test/testsuite/vacid-0/RedBlackTrees.c`

VCDRYAD allows separation logic specifications which are considerably more succinct and does not need any additional guidance in terms of proof tactics.

VeriFast, Bedrock: The VeriFast tool for C/Java works with separation logic specifications, requires proof guidance at the code-level from the user. The system itself aims to do very little *search*, and hence gives predictably high performance. Consider an implementation of binary search tree insertion (`bst_insert`) in VeriFast [7] and VCDRYAD, with specifications in separation logic. Verification of `bst_insert` procedure in VeriFast (lines 105–146) relies on user's help in form of annotations at particular program locations to un-bundle (`open`) heaplets (lines 109, 127, 130) and re-bundle (`close`) heaplets (117, 119, 121, 128, 131, 136, 142), as well as three user-provided lemmas. Moreover, the user also needs to manually instantiate `tree_add_inorder` lemma at lines 135 and 141. The verification of the bst insert implementation in VCDRYAD (28–61) requires no annotations for helping proofs.

The tool Bedrock [56] provides Coq libraries and allows users to give proof tactics at the code-level, and automates significant parts of the reasoning. Consider an implementation of an in-place singly linked list reversal in Bedrock (from its tutorial)[8] and VCDRYAD. In this example, Bedrock and VCDRYAD versions do not prove exactly the same property because Bedrock uses Coq's list type family to abstractly represent memory location while VCDRYAD reasons about list structure and set of keys stored in the list. However, Bedrock does require hints to be provided by the user; user-provided administrative lemmas for each separation logic predicate (lines 9–13), to relate memory representation to abstract representation using four more lemmas that are need to be packaged together using a Bedrock tactic (lines 15–44). No such user intervention is needed to prove C version using VCDRYAD.

Finally, jStar [75] is a tool for Java against separation logic specifications that uses abstract shape analysis for inferring invariants and using a theorem prover similar to Smallfoot. In this context, we can see our work on VCDRYAD as essentially providing a deep embedding of separation logic into tools that use a classical logic pipeline, where arbitrary nested quantification is avoided using a restricted separated logic and verification is automated using natural proofs.

Comparison with our earlier work [147]: We are crucially building on the work reported in [147]. The difference between work reported here and that in [147] is that the latter is for a toy language while the current work is for the C language. The C language for has features such as (weakly enforced) types, base types (including `char`, `unsigned`, `long`), casting, aggregate data-types using `struct`, pointers to a structure inside `struct`, operators on bits, low-level access to memory, etc. The toy language does not contain these features. In particular, the toy language had only *one* type of structure (location) with a *fixed* set of field pointers for it. This means that we cannot model two different structures in this language or structures containing pointers to the other structures.

[7] `http://web.engr.illinois.edu/~pek1/vcdryad/cmp/VCDryad_vs_Verifast.html`
[8] `http://web.engr.illinois.edu/~pek1/vcdryad/cmp/VCDryad_vs_Bedrock.html`

For instance, in our benchmark suite, we verified a process address space manipulation in a secure OS (ExpressOS), using a data structure `MemReg`. This is a nested `struct`, and has a pointer to `Backingfile struct`. While we prove this C program correct, the work in [147] cannot handle it, and in fact the corresponding version in [147] simplifies the program so that it does not involve nested `struct` constructs, by removing the inner `struct BackingFile`.

Furthermore, our work here giving natural proofs for C over the tool framework VCC gives a general platform for verification, where programmers can get some proofs (involving data-structures) automated using natural proofs but are still allowed to use VCC for proving more complex specifications using the automatically proved properties.

Comparison with Liquid Types: At a very high level, we believe that natural proofs and *liquid types* [150, 103] are related and are in a sense *dual* to each other: while we take a logical deductive verification formalism and search intentionally for simple proofs, in liquid types, the type-checking mechanism is simple but types are enriched using general logic formulas. Despite the similarities in aiming for simple sound but incomplete proofs for properties expressed in logic, the approaches are mechanically radically different, and a formal comparison seems hard. The work on liquid types also has built-in invariant generation (using a form of the Houdini algorithm [79]) that we do not have yet for natural proofs. On the other hand, our work augments a verification framework where users can interact further to prove their programs correct using the powerful mechanisms VCC provides, while such an integration of liquid types into a deductive verifier seems harder.

3.7 Conclusions and Future Work

In this chapter we have described how we built a powerful technique and tool, by adapting the ideas of natural proofs for separation logic [147], to automate verification proofs of data-structure manipulation in a widely used systems programming language. The primary technical contribution of the work described in this chapter is to show how natural proofs, though defined only at the verification-condition level, can be encoded at the code-level, utilizing existing frameworks that handle the language semantics and memory model. The resulting tool VCDRYAD gives a powerful extension of VCC for sequential programs, building automaticity for the most difficult task of verifying the dynamic data-structures a program maintains.

Natural proofs, augmented with axioms for the data-structures that relate different recursive definitions, have been able to verify all data-structure examples we have tried. Typically, for a data-structure, once we have related the various recursive definitions using axioms, programs manipulating the data-structures seldom require more help.

However, natural proofs currently do not work for data-structures that cannot be defined recursively (such us DAGs, graphs, etc.). We have been able to prove some properties of Schorr-Waite algorithm only for trees, but not for general graphs.

Several future directions are interesting. First, it would be interesting to see how VCDRYAD can be used for verifying larger pieces of code, and how the programmer's manual interactions for proving more complex properties can be orchestrated with the automatically generated annotations provided by VCDRYAD. Second, while the current work has focused on automatic proof tactics, loop invariants (and strengthening pre/post conditions so that they become inductive) is a hard task for the programmer, and automating this, especially for DRYAD specifications, would advance the usability of deductive verification tools further.

Chapter 4

Abstraction-guided Runtime Checking of Assertions on Lists

The immediate motivation for this chapter is an extension of the work described in the previous chapter. In particular, one of the difficulties in using automated program verifier is understanding why the verifier fails to prove the specification. Run-time assertion checking can certainly alleviate that difficulty. In a more general sense the work described in this chapter investigates ways to specify and check, at runtime, assertions that express properties of dynamically manipulated linked-list data structures. Checking an assertion involving whether pointers point to a valid linked list and separation properties of these lists typically requires linear or even quadratic time on the size of the heap. The main technical contribution described in this chapter is a way to scale this checking by orders of magnitude, using a novel idea called abstraction-guided runtime checking, whereby we maintain an accurate abstraction of the dynamic heap by utilizing the evolving runtime state, and where the abstraction helps in checking the runtime assertions much faster. We develop this synergistic combination of abstractions and runtime checking for lists, list-segments, and their separation, implement it, and show the tremendous performance gains it yields. In particular, when lists are manipulated using library functions, maintenance of the abstraction is within the libraries and yields constant runtime checking of assertions in the client code. We show that, as the number of assertions get frequent and the data structures get large, abstraction-guided runtime checking, which includes maintenance of the abstraction and the runtime checks, gives close to constant-time per assertion overhead in practice.

4.1 Introduction

As we described in section 1.1, assertions are one of the most useful techniques for detecting errors and providing information about fault locations [57, 93], and are used widely in testing production code. The Eiffel system pioneered the systematic usage of assertions in terms of contracts and invariants for run-time checking, which led to wider adoption in mainstream programming languages. The JML notation [116] and the SPEC# language [27] and CODE CONTRACTS [26] systems at Microsoft are examples of specification languages that were influenced by Eiffel [131]; the latter were used for writing specifications mainly for testing code within Microsoft. Assertions are widely used; for instance, a study showed that there are more than a quarter million assertions in the Microsoft Office suite [93].

Since they are simple to write and popular because programmers use them for testing.

In this chapter, we study the problem of expressing and efficiently checking assertions over list-segments, lists, and the way they merge or remain separate, when dynamically allocated and manipulated by a program using pointers. Mainstream programming languages lack any standard assertion logic for the heap, and programmers often write assertions by writing *procedures* that check properties. For instance, in Java, programmers write so-called REPOK methods for checking representation class invariants during testing [121]. We develop efficient runtime checking for a *declarative and logical* specification language for expressing properties of list data-structures.

In this chapter, we focus on a *separation logic* over lists and list-segments. (Detailed account of separation logic can be found in the previous chapter or in section 1.3.1.) As mentioned before, separation logic is a succinct logic that can express complex properties of the heap, including structure (e.g., "x points to a list") and separation ("the lists pointed to by x and y are disjoint"). For instance, the succinct assertion:

```
assert lseg(x,y) * list(z) * true
```

expresses that there is a list segment from x to y (defined by a *next* pointer), and z points to a list, and these two are disjoint sets of locations of the heap.

The simplest runtime check on the heap for the above property would take quadratic time on the length of the lists involved; in general, checking `list` or `lseg` requires linear time, while checking separation properties that involve checking whether heaplets (subparts of the heap) are disjoint take quadratic time (though this can be made linear with techniques such as hashing or using additional linear space).

Note that when the sizes of the data structures get large (several thousand nodes), linear-time algorithms cause nontrivial overhead (and quadratic-time algorithms are just not acceptable), and hence the runtime checking of such data structures, *whether written by the programmer* or generated automatically from the specification, do not scale.

The key contribution of the work described in this chapter is a new idea, which we call *abstraction-guided runtime assertion checking*. The idea is to maintain an abstraction of the concrete heap dynamically, as the program executes, which will maintain certain structural properties of the heap symbolically. (Note that the abstraction is *not* part of a static analysis— the abstraction is maintained as the program executes.) This abstraction obviates the need to dynamically check crucial and expensive properties, like separation, on the concrete heap. However, note that in classical abstractions of heaps, such as static shape analysis [128, 151], the abstraction will always lose accuracy in reflecting the program's runtime state, and hence will quickly evolve into a "blob" of uncertainty that isn't very useful.

We propose an abstraction of lists and list segments that involves both a structural abstraction and a concrete mapping that relates the abstract vertex to concrete memory addresses on the runtime heap. Using this map, we show that we can keep the abstraction in check by using the runtime state as the program executes. Consequently, the structural abstraction always accurately reflects the runtime heap. Furthermore, when we reach a point in the program

with an assertion, we show that we can, instead of evaluating the formula on the concrete heap, check it much faster by evaluating it on its abstraction instead.

We emphasize that the curious synergy between the abstraction and runtime checking is also inexpensive. Maintaining the abstract state relies on the concrete state and imposes a low overhead on the runtime execution, but brings huge performance gains when assertions are checked using the abstraction. As far as we know, this is the first time abstractions have been shown to be useful in speeding up runtime checking.

Let us consider software that consists of *library code* implementing low-level manipulations of lists, etc., with abstract data-type interfaces for collections or sequences, and *client code* calling such libraries to implement a higher-level functionality. Both the library code and the client code could contain assertions about the lists in the heap, written by the programmer. In such a setting, our scheme would work as follows:

- We maintain an abstraction A of the dynamic heap pertaining to all list nodes globally.
- Assertions in the client code are transformed to check assertions on the abstraction A instead. The client code (which does not manipulate lists directly) is left untouched otherwise.
- The assertions in the library code are also transformed to check assertions on the abstraction A. In addition, all manipulations of the lists within the library methods are *instrumented* to update and maintain the abstract heap A. Furthermore, updating the abstraction correctly and accurately itself *requires the concrete heap* of the program.

The abstractions are typically of *constant* length (i.e., they are as long as the number of program pointer variables, but typically do not grow unboundedly with the length of the lists they represent) and hence checking of assertions is achieved usually in *constant time*, as opposed to linear/quadratic in the size of the heap. However, note that maintaining the abstraction does incur an overhead— typically, every update to the heap requires an instrumentation to maintain the abstract heap accurately, which generally incurs constant time execution overhead. Consequently, when the number of assertions is large and the lists are long (which is precisely when overheads matter), our technique gives close to constant time, amortized, for checking an assertion.

Evaluation: We implement both assertion checking techniques for specifications that express properties of lists, list-segments, and separation— the one that evaluates assertions on the concrete heap and the technique based on abstraction-based runtime checking. We evaluate both techniques on a suite of 25 programs, including both library code manipulating lists and client code calling these libraries. Our evaluation shows that cost of evaluating an assertion on the concrete heap grows with the size of the list, as expected, typically growing quadratically with the sizes of the lists. However, remarkably, checking an assertion on the abstraction performs orders of magnitude faster, and seems

to essentially take only constant time in checking an assertion, when there are many assertions and when the sizes of the data structures are large.

Our evaluation shows that our new idea of using the synergy of abstractions and runtime state, where the runtime state helps keep the abstraction in check and where the abstraction helps in performing runtime assertion checking fast, is a powerful idea, and holds promise for building truly scalable assertion checking for dynamically manipulated data structures.

4.2 Assertion Logic

In this section we present the separation logic on lists that we consider and the class of assertions that we allow programmers to write.

$j \in$ *Scalar Int Variables* $\quad q \in$ *Scalar Bool Variables*
$x, y \in$ *Loc* Variables $\quad c \in$ *Int* Constant

$$
\begin{array}{rlll}
\textit{Loc Terms:} & lt & ::= & x \mid \texttt{nil} \\
\textit{Scalar Int Terms:} & st & ::= & c \mid j \mid st + st \mid st - st \\
\textit{Scalar Formulas:} & sf & ::= & q \mid st = st \mid st \neq st \mid st \leq st \mid st < st \mid lt = lt \mid lt \neq lt \\
\\
\text{Formulas:} & \varphi & ::= & \texttt{true} \mid \texttt{false} \mid sf \\
 & & & \texttt{emp} \mid x \longmapsto lt \mid x \longmapsto ? \mid \texttt{list}(x) \mid \texttt{lseg}(x,y) \mid \varphi \wedge \varphi \mid \varphi * \varphi \\
\text{Assertions:} & \alpha & ::= & \varphi * \texttt{true} \mid \alpha \vee \alpha
\end{array}
$$

Figure 4.1: Syntax of a quantifier-free separation logic on lists and list segments

We will first define program configurations, consisting of a store and a heap. Let *Loc* be a countably infinite set of heap locations and let `nil` be the special location term for the null pointer. Let us assume that each heap location has a single pointer field *next* and let $next_c : Loc \backslash \{\texttt{nil}\} \rightarrow Loc$ be that function which maps non-nil heap locations to the adjacent location in the concrete heap. Let *PV* be the set of program variables pointing to locations, *IV* be the set of variables of the type *Int* and *BV* be the set of variables of type *Boolean*. Let S_c be the concrete store that maps program variables to constants of the appropriate type– that is, S_c maps *PV* to *Loc*, *BV* to *true* or *false* and *IV* to integer constants. Let *PC* be the set of concrete program configurations where a configuration is a tuple of the heap $next_c$ and the store S_c.

The syntax of our assertion separation logic is shown in Figure 4.1. The semantics is fairly standard separation logic [149], and we skip giving a formal semantics. The formulas are evaluated over program configurations, where the store gives the valuation of scalar and pointer variables and a heaplet containing a subdomain of the dynamic heap with the pointer field *next*. Scalar formulas depend only on the store and not on the heaplet. The formula `emp` evaluates to true iff the heaplet is empty. $x \longmapsto lt$ evaluates to true iff the next-pointer from x points to lt and the heaplet is a singleton containing the location x points to. The formula $x \longmapsto ?$ evaluates to true iff x is a non-nil location and the

heaplet is again a singleton set containing the location that variable x points to. `list(x)` evaluates to true iff x points to a list (wrt `next` pointer) ending with the `nil` location, and the heaplet is the set of all locations on this list. Similarly, `lseg(x,y)` evaluates to true iff the next-pointer from x reaches y eventually, and the heaplet is the locations on this segment that includes x and excludes y, unless $x = y$ when the heaplet is empty. Conjunction of two formulas hold on a heaplet iff both sub-formulas hold on that *same* heaplet. Finally, $\alpha * \beta$ holds iff the heaplet can be partitioned into two parts such that α holds in one and β holds in the other.

Formula	Truthhood	Domain-exact	Scope
$true$	$true$	$false$	$\emptyset$
$false$	$false$	$false$	$\emptyset$
sf	$eval(sf)$	$false$	$\emptyset$
emp	$true$	$true$	$\emptyset$
$x \longmapsto lt$	$S_c(x) \neq nil \wedge next_c(S_c(x)) = S_c(lt)$	$true$	$\{S_c(x)\}$
$x \longmapsto ?$	$S_c(x) \neq nil$	$true$	$\{S_c(x)\}$
$list(x)$	$IsList(S_c(x))$	$true$	$between(S_c(x), nil)$
$lseg(x,y)$	$IsLseg(S_c(x), S_c(y))$	$true$	$between(S_c(x), S_c(y))$
$\varphi \wedge \varphi'$	$th(\varphi) \wedge th(\varphi') \wedge comp(\varphi, \varphi')$	dom-ext(φ) $\vee$ dom-ext(φ')	scope(φ) $\cup$ scope(φ')
$\varphi * \varphi'$	$th(\varphi) \wedge th(\varphi') \wedge$ scope(φ) $\cap$ scope(φ') $= \emptyset$	dom-ext(φ) $\wedge$ dom-ext(φ')	scope(φ) $\cup$ scope(φ')

where $comp(\varphi, \varphi')$:= (scope(φ) = scope(φ'), if dom-ext(φ) and dom-ext(φ')
:= ($scope(\varphi) \subseteq$ scope(φ')), if $\neg$dom-ext(φ) and dom-ext(φ')
:= ($scope(\varphi') \subseteq$ scope(φ)), if dom-ext(φ) and $\neg$dom-ext(φ')
:= *true*, otherwise.

Figure 4.2: Truthhood, Domain-exactness and scope functions.

Assertions: While writing specifications for a program in separation logic, using assertions and pre/post-conditions, the pre-conditions implicitly delineate the part of the heap that the method should access. Conveying the footprint on which the program works implicitly through separation logic specifications has several advantages, the most important one being the *frame rule* [36] that one gets for free for static checking.

However, we do not want to insist on programmers to write pre-conditions to all methods in separation logic to delineate the fragment of the heap the method will change. It turns out that checking whether a method stays within the heaplet defined by its pre-condition is inherently expensive anyway to check at runtime; consequently, in the literature, there have been suggestions of delineating the heaplet using compiler and hardware-based isolation techniques [14]. In this chapter, we do *not* consider this problem of a method staying within the confines of a heaplet. Rather, we assume that all assertions are disjunctions of formulas of the form $\varphi * true$ (see Figure 4.1), and evaluated on the store for the variables currently in scope and the global heap of the program. When preconditions to methods are lacking, this is the only reasonable way to evaluate assertions.

4.2.1 Evaluating Assertions on the Concrete Heap

In general, when checking a formula of the kind $\alpha * \beta$ on a concrete heap, we need to find a way to divide the heaplet into two parts to evaluate α and β on. However, the separation logic we have has the property that heaplets for the atomic formulas (including `list` and `lseg`) are precisely determined. This allows us to evaluate any separation formula *bottom-up*, computing the relevant heaplets the formulas hold on, and hence is algorithmically efficient.

Following the work on DRYAD [147, 142], which is based on similar ideas, we evaluate a separation logic formula on a concrete heap bottom-up by computing a triple $\langle th(\beta), dom\text{-}exact(\beta), scope(\beta)\rangle$, for every subformula β of α. The boolean $th(\beta)$ captures the truth or falsehood of β. If $th(\beta)$ is true, then the boolean $dom\text{-}exact(\beta)$ captures whether the formula is true only on a particular heaplet. The set $scope(\beta)$ is a set of locations— and when $dom\text{-}exact(\beta)$ is true, it demands that the heaplet be precisely this scope, and when $dom\text{-}exact(\beta)$ is false, it demands that the heaplet is some *superset* of the scope. The functions $th(\beta)$, $dom\text{-}exact(\beta)$ and $scope(\beta)$ are defined in Figure 4.2. In the figure, the method *IsList(x)* checks whether location x points to a list along the *next* pointer field; similarly, *IsLseg(x,y)* checks if the traversal of the heap along *next* from x eventually reaches y. The method *between(x, y)* returns a set of locations depending on whether traversing *next* from x reaches y or not. If there exists a heap traversal from x to y, then *between(x, y)* returns the set of locations on the path including x and excluding y. When $x = y$ or when there is no traversal from x to y along the *next* field, *between(x, y)* is the empty set.

4.3 Runtime Guided Abstractions and Abstraction Guided Runtime Checking

In this section, we present our main contribution— an abstraction for lists that helps scale runtime checking of separation logic assertions. The base abstraction for lists that we define is itself straightforward, and is a mild adaptation of common abstraction of lists used in shape analysis [128]— the abstraction is a graph that tracks only the concrete nodes pointed to by program variables and the points where lists "merge" into one another, and also keeps track of precise memory addresses corresponding to these nodes. However, the salient aspect of our abstraction is that we keep the abstract graph *accurate* using knowledge acquired at runtime. This essentially means that for any assertion α in our logic, the evaluation of α on the abstract heap is the same as its evaluation on the concrete heap. The resulting scheme is an interesting *synergy* between the runtime state and the abstract state— the abstract state helps in runtime checking by providing it details that can be easily inferred using the abstraction (such as whether x points to a list, whether x to y forms a list segment, etc.), and the runtime state helps keep the abstract state in check, ensuring that it most accurately describes the current state of the system, which in turn is crucial in using it for runtime checks.

4.3.1 Abstracting Lists

The abstraction we use for lists is similar to those used in classical shape analysis literature [128, 151], except that it stores some amount of information in terms of concrete pointers. In particular, we track only the concrete nodes that are currently pointed to by some pointer variables in the program and the concrete nodes that are the first nodes of the merging lists (i.e., the first nodes where the lists start to overlap). However, instead of using summary nodes that stand for unbounded sections of the lists, we have two kinds of edges in the abstraction. The first kind is a concrete edge— a *concrete edge* denotes that the source node of the edge points to the destination node through the $next_c$ pointer on the concrete heap. The second kind of edge is a summary edge— a *summary edge* from node u to v represents a list segment with length larger than one, stretching from u to v in the concrete heap.

We must emphasize that though the abstraction itself is fairly standard (and is described next), the abstract transition relation is not standard— in our setting, we can use the runtime state to build the next abstract state, and we exploit this to keep the abstraction accurate. Note that the program can do a lot of things that the abstraction is not intended to track (see below for an example). We need to keep the abstraction accurate no matter what the program does. We also emphasize that we are *not* verifying that the program satisfies an assertion (and hence getting rid of the assertion). Our abstraction is woefully inadequate in proving any such property of programs; the abstraction does, however, help in runtime checking.

Let us fix a finite set of program pointer variables PV, and the set of all memory locations (concrete heap locations) Loc.

Definition 4.3.1. *An* abstract heap of lists *over the set of pointer variables PV is a graph $G_a = (V, next_a, S_a, addr)$, where*

- *V is a finite set of nodes that has two special nodes v_{nil} and v_{undef},*
- *$next_a : V \setminus \{v_{nil}, v_{undef}\} \longrightarrow V \times \{c, s\}$ is a function that encodes the next-pointer, where a node maps to another node either through a concrete edge (c), or through a summary edge (s),*
- *$S_a : PV \rightarrow V$ associates each program pointer variable to the node it points to, and*
- *$addr : V \setminus \{v_{undef}\} \longrightarrow Loc$ is a map that associates each node with a location in the concrete heap.*

We also require the graph to satisfy the following:

- *for every program pointer variable $p \in PV$, there is precisely one node $v \in V$ such that $S_a(p) = v$,*
- *the address labels of the nodes are all different,*

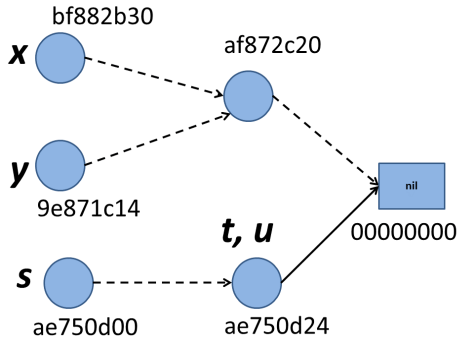


Figure 4.3: Abstraction representing heap where x and y point to two lists that merge, but they are different from the list pointed to by s, and where s to t forms a list segment, and where $t = u$. Dashed lines represent the summary edges, while solid lines represent the concrete edges.

- *for every* $v \in V \setminus \{v_{nil}, v_{undef}\}$*, either there exists* $p \in PV$ *such that* $S_a(p) = v$ *or there are at least two nodes* u, u' *such that* $u \neq u'$ *and* $next_a(u) = next_a(u') = (v, e_t)$*, where* e_t *can be either c or s.* □

In the above definition, v_{undef} is a special node in the abstract graph that represents all unallocated or uninitialized heap locations. We do *not* consider programs that result in undefined behavior on dereferencing an uninitialized heap location or manipulating ill-formed linked-list structures that can reach an uninitialized location.

Figure 4.3 shows an abstraction of a heap that contains lists (we omit the node v_{undef} in the pictures and any pointer variable not depicted is assumed to point to v_{undef}). In the figure, the S_a and *addr* are depicted using labels on vertices. This abstract heap represents all heaps where (a) x and y both point to lists that merge, and merge first at address `af872c20`, (b) s to t forms a list segment and `t` $\rightarrow$ `next` points to the `nil` location, and the list from s is disjoint from the list from x and the list from y, (c) $u = t$, and (d) x, y, s and t are the pointers `bf882b30`, `9e871c14`, `ae750d00` and `ae750d24`, respectively.

In general, any abstract graph $G_a = (V, next_a, S_a, addr)$ depicts the set of concrete program configurations $(next_c, S_c) \in PC$ where (a) the program pointer variables map to the addresses stored at the abstract nodes representing them, i.e., $addr(S_a(pv)) = S_c(pv)$ for every pointer variable $pv \in PV$, (b) for every concrete edge in the abstract graph such that $next_a(u) = (v, c)$, the next pointer from the address at u points to the address at v, (c) for every summary edge (u, v) in the abstract graph such that $next_a(u) = (v, s)$, the next pointer from the address at u eventually leads to the address of v through a non-empty set of intermediate addresses, and (d) the concrete list segments represented by every edge are disjoint in the concrete heap. Let $\gamma : G_a \rightarrow 2^{PC}$ be the concretization function that maps an abstract graph to the set of concrete heaplets corresponding to it, as described above.

It is easy to see that every concrete heap corresponds to precisely one abstract heap. Note that our abstraction does not allow expressing arbitrary sets of concrete heaps, and hence is not a standard abstraction used for static analysis. The set of concrete heaps that an abstract heap represents have the property that they all satisfy the *same* formulas in

our assertion logic.

Note that though an assertion is checked only when the program reaches the assertion, keeping track of the abstract graph accurately demands instrumenting *every* heap operation the program does. However, the amount of help required to keep the abstraction in check is very small (close to constant time per instrumented operation of the program) while the reductions given by the abstraction to runtime checking is significant (often reducing from linear or quadratic checks on lists to close to constant time when the assertion is checked frequently).

4.3.2 Evaluating Assertions on the Abstract Heap

We can evaluate our assertion logic on abstract heaps. Notice that our abstraction includes all nodes pointed to by program variables in scope, includes the reachability information between these nodes, and includes nodes where these list segments merge as well, which helps deciding separation properties. Consequently, we can evaluate an assertion bottom-up on an abstract heap (and the concrete store) similar to the evaluation of assertions on the concrete heap and store.

More formally, we define the scope of a formula in the abstract heap as a set of nodes and summary edges of the abstract graph. Intuitively, a summary edge (u, v) in the abstract graph represents all locations along the list segment from u to v. More specifically, the scope for *true*, *false* and the scalar formulas *sf* is the empty set; the scope for the formula $x \mapsto lt$ and $x \mapsto ?$ is the singleton set containing the node pointed to by variable x; the scope for $list(x)$ is the set of nodes and summary edges that occur along the path (if one exists) from the node pointed to by x to the node v_{nil}, otherwise the scope is an empty set if x does not point to a list; and similarly, the scope for $lseg(x, y)$ is the set of nodes and summary edges that occur along the path between the nodes pointed to by x and y. The scope for conjunction and separating conjunction of formulas is defined in the same way as in Figure 4.2. Provided the new definition of scope, an assertion formula α can be evaluated on the abstract heap and abstract store in the same way as its evaluation on the concrete heap (see Figure 4.2). Note that domain-exactness is a property of the formula and hence is independent of whether a formula is being evaluated in the abstract heap or in the concrete heap.

The salient property of our abstraction is that it is very precise for the assertion logic under consideration, and *yet* it can be maintained at this level of precision using the runtime concrete heap. More precisely, we can show the following:

Theorem 4.3.2. *Let A be an abstract heap of lists and let C be a concrete heaplet such that $C \in \gamma(A)$. Then, for any assertion φ, φ evaluates to true on the abstract heap iff φ evaluates to true on the concrete heap C.*

A proof gist is presented in the Section 4.7. The above theorem gives us the ability to evaluate the assertion on the abstract heap instead of the concrete heap. When the runtime reaches a configuration with a concrete heap c, we

can evaluate φ on it by evaluating φ on its abstraction a. For instance, if we come across an assertion of the form $list(x) * list(s) * true$ with the abstract graph of Figure 4.3, we can quickly answer that the assertion holds by examining the abstraction, while we can quickly also declare that the assertion $list(x) * list(y) * true$ is false.

Note that the above theorem does not hold for certain abstractions of the heap in the literature— for example, a naive shape analysis would maintain an abstraction that over-approximates any set of concrete configurations, and hence it may be the case that φ evaluates to false on an abstract state but true on a particular concrete configuration in the set of concrete configurations corresponding to the abstract state.

4.3.3 The Synergy between Abstraction and the Runtime State

The goal of our runtime procedure is to maintain the abstract graph above. As we will show below, we can always maintain the abstract graph accurately using runtime checks, aided by the concrete addresses in the abstract graph. The naive way would be to compute the abstract graph from the concrete heap each time; this is, of course, too expensive— we will show how to maintain the graph with much less cost.

Let us now illustrate the way the runtime state helps keeping a check on the abstraction. Consider the scenario depicted in Figure 4.3, where s points to a list and t points somewhere on this list. Now, assume that the program apriori knows that there is a key k stored somewhere in the list segment from s to t (a recursive search for a key in a sorted list, like in quicksort, may result in such a situation). Consequently, assume that the program executed the following code:

```
while (s.key != k) { s:= s->next };
assert (lseg(s,t) * list(t) * true);
```

Note that the assertion after the while-loop is true, as s would not have passed the pointer t, since the key would have been found before that.

Now, if we just kept track of the abstraction (as in static shape analysis), we would have no idea when or whether the pointer s would go past t (since the abstraction cannot track every detail of the program, and in particular would not know whether the key k would occur before t). However, in our runtime guided abstraction, we *will* precisely know when s passes t. Every time we execute the statement $\mathtt{s} := \mathtt{s} \rightarrow \mathtt{next}$, we will check whether the successive concrete address, namely `ae750d24`, has been reached by s; if not, the abstract graph would stay the same (except the address associated with s would be updated). If s does reach `ae750d24`, then the entire abstraction graph would change (with s, t, and u, all pointing to the same location).

Consequently, in the above setting where s stays in the list before t, the assertion will hold and will be validated to be true by just using the abstraction graph. In pure static verification, this shape graph would get divided into *two*

$$
\begin{array}{rl}
\mathtt{x},\mathtt{y} : \mathit{List}^* & x^{ctx}, y^{ctx} : \mathit{PV} \qquad \mathit{loc} : \mathtt{Loc} \qquad \mathit{var} : \mathtt{PV} \\
S_c : \mathtt{PV} \to \mathtt{Loc} & \mathit{next}_c : \mathtt{Loc} \setminus \{\mathtt{loc}_{\mathtt{NULL}}\} \to \mathtt{Loc} \\
S_a : \mathtt{PV} \to \mathtt{V} & \mathit{next}_a : \mathtt{V} \setminus \{\mathtt{v}_{\mathtt{undef}}, \mathtt{v}_{\mathtt{nil}}\} \to \mathtt{V} \times \{\mathtt{c},\mathtt{s}\} \\
[\![\_]\!]_{a\_\_} : \mathtt{Stmt} \times & \mathit{PC} \times G_a \to G_a \qquad\qquad [\![\_]\!]_{c\_} : \mathtt{Stmt} \times \mathit{PC} \to \mathit{PC} \\
\tau = ((\mathit{next}'_c, S'_c), (V, & \mathit{next}_a, S_a, \mathit{addr})) \;\; \mathit{where} \;\; (\mathit{next}'_c, S'_c) = [\![\mathtt{stmt}]\!]_c(\mathit{next}_c, S_c)
\end{array}
$$

$$
\begin{array}{rl}
[\![\mathtt{List}^*\ \mathtt{x}]\!]_a(\tau) & = (V, \mathit{next}_a, S'_a, \mathit{addr}) \;\mathit{where}\; S'_a = S_a[x^{ctx} \mapsto v_{undef}] \\
[\![\mathtt{x} := \mathtt{y}]\!]_a(\tau) & = (V, \mathit{next}_a, S'_a, \mathit{addr}) \;\mathit{where}\; S'_a = S_a[x^{ctx} \mapsto S_a(y^{ctx})] \\
[\![\mathtt{x} := \mathtt{y} \to \mathtt{next}]\!]_a(\tau) & = (V', \mathit{next}'_a, S'_a, \mathit{addr}') \\
\mathit{where} & \;\mathtt{if}\; \mathit{next}_a(S_a(y^{ctx})) = (v_{y\to next}, c) : \\
& \qquad S'_a = S_a[x^{ctx} \mapsto v_{y\to next}] \\
& \qquad \mathit{next}'_a = \mathit{next}_a,\; V' = V,\; \mathit{addr}' = \mathit{addr} \\
& \;\mathtt{else}\;\; \mathit{let}\; v = \mathit{fresh\_vertex}() \\
& \qquad V' = V \cup \{v\},\; \mathit{addr}' = \mathit{addr}[v \mapsto S'_c(x^{ctx})] \\
& \qquad \mathit{next}'_a = \mathit{next}_a[S_a(y^{ctx}) \mapsto (v,c)][v \mapsto (\mathit{next}_a(S_a(y^{ctx})), s)] \\
& \qquad S'_a = S_a[x^{ctx} \mapsto v] \\
[\![\mathtt{x} \to \mathtt{next} := \mathtt{y}]\!]_a(\tau) & = (V, \mathit{next}'_a, S_a, \mathit{addr}) \;\mathit{where}\; \mathit{next}'_a = \mathit{next}_a[S_a(x^{ctx}) \mapsto (S_a(y^{ctx}), c)] \\
[\![\mathtt{x} := \mathtt{malloc}()]\!]_a(\tau) & = (V', \mathit{next}'_a, S'_a, \mathit{addr}') \;\mathit{where}\; \mathit{let}\; v = \mathit{fresh\_vertex}() \\
& \qquad V' = V \cup \{v\},\; \mathit{addr}' = \mathit{addr}[v \mapsto S'_c(x^{ctx})] \\
& \qquad \mathit{next}'_a = \mathit{next}_a[v \mapsto (v_{undef}, c)],\; S'_a = S_a[x^{ctx} \mapsto v] \\
[\![\mathtt{free}(\mathtt{x})]\!]_a(\tau) & = (V', \mathit{next}'_a, S'_a, \mathit{addr}') \\
\mathit{where} & \;\forall\; pv.\; S_a(pv) = S_a(x^{ctx}) \Rightarrow S'_a = S_a[pv \mapsto v_{undef}] \\
& \qquad \forall\; v \in \mathit{pred}(v, V, \mathit{next}_a, S_a, \mathit{addr}).\; \mathit{next}'_a = \mathit{next}_a[v \mapsto (v_{undef}, s)] \\
& \qquad V' = V \setminus \{S_a(x)\},\; \mathit{addr}' = \mathit{addr}[S_a(x) \mapsto .]
\end{array}
$$

$$
\begin{array}{rl}
\mathit{clean}(V, \mathit{next}_a, S_a, \mathit{addr}) & = \forall v. |\{\mathit{var} \mid S_a(\mathit{var}) = v\}| = 0 \wedge |\mathit{pred}(v, V, \mathit{next}_a, S_a, \mathit{addr})| \leq 1 \\
& \qquad \Rightarrow \mathit{remove}(v, \mathit{next}_a)
\end{array}
$$

Figure 4.4: Abstract State Updates

graphs— one assuming s reached t and one that doesn't, and usually soon leads to very coarse abstractions of the heap. The runtime check using concrete addresses prevents this effectively and at low cost.

4.3.4 Maintaining the Abstract State

In this section, we describe the instrumentation for capturing the abstraction transformation for various basic heap-manipulating statements in an imperative sequential programming language. Our programming language includes standard control-flow statements like `if` and `while`, dynamic memory manipulation, and function calls.

Figure 4.4 describes the operations we perform for maintaining the list abstraction. We only need to perform transformations on our abstract state whenever list manipulating statements occur. We rely on the type system of our programming language to detect which statements are manipulating lists. More formally, we instrument each list-manipulating statement and perform our abstract-state update after the statement execution and the concrete-state

update. The abstract transformer, $[\![\_]\!]_a$, takes as input a statement, a concrete state produced by the statement execution, and an abstract state and it produces the updated abstract state. Note that $[\![\_]\!]_a$ inspects the updated concrete state only in the immediate neighborhood of the manipulated pointer (the `next` address of the manipulated pointer). The concrete transformer, $[\![\_]\!]_c$, just updates the concrete state based on the statement.

Next we describe our notation in Figure 4.4. We use $f' = f[x \mapsto y]$ to denote that f' returns the same value as f for all arguments except x; for x it returns y. Further we use $f' = f[x \mapsto .]$ to denote the fact that f' is not defined on x. We use *pred* to get a set of vertices representing the predecessors of a node in the abstract graph. The *remove* function removes a node from the abstract graph, linking its successor and predecessor through a summary edge. Note that *clean* is always called after each abstract update, on the updated abstract graph; we omit explicitly calling *clean* in each update for brevity. The `refine_edges` function ensures that edges collapsed to summary edges after an abstract graph update are refined to a concrete edge when the source node address maps to the destination node address through the `next` pointer. The `refine_edges` function is called also at every point after the abstract update.

When a pointer variable $x \in PV$ is declared in the program, we update the abstract store S_a to map x to v_{undef}. Similarly, on a pointer assignment `x := y`, the abstract store is updated to map variable x to the node pointed to by variable y. The instrumentation for statement `x := y→next` is more involved. We distinguish two cases. If the abstract node corresponding to the new value of x is already in the abstract graph we just update the S_a such that x now points to the corresponding abstract node. If not, a new node is created, the $addr$ map is updated to map this node to the concrete address of x. The newly created node is introduced after the abstract node corresponding to y by linking it through a concrete incoming edge and an outgoing summary edge. Note that $refine\_edges$ will check whether the summary edge needs to be changed to a concrete edge. The statement `x→next := y` does not change the store nor the address map in the abstraction, but it only modifies the shape of the abstract graph. In particular, the instrumentation code updates the edge function $next_a$ to now have a concrete edge from x to y. Abstraction for the *malloc* statement involves adding a fresh node v to the abstract graph, updating the abstract store to point x to v, $addr(v)$ is the location pointed to by x after the *malloc* and adding a concrete edge in the abstraction from v to v_{undef}. Abstraction for the $free(x)$ statement involves removing the node v in the abstract graph corresponding to variable x, updating the $next_a$ edges to join all predecessors of v to v_{undef}, and updating the store to map all variables pointing to v to point to v_{undef}.

The transformations in Fig. 4.4 extend to function calls. Each variable is scoped using ctx, which represents the dynamic context based on the call-stack. The scoping enables us to uniquely identify variables in our abstraction, across (potentially recursive) function calls. First, at each call-site, we introduce a mapping in our S_a from the formal function parameters to the actual arguments at the call-site. Next, at returns, we remove from S_a the variables in the current scope.

Theorem 4.3.3. *For any statement s, an abstract state A and concrete program configuration C such that $C \in \gamma(A)$,*

Node* *insert_before*(**Node*** `slist`, **Node*** `sibling`, **int** `data`)
`requires` *lseg*(`slist`, `sibling`) * *list*(`sibling`)
`invariant` *list*(`slist`) $\wedge$ $\big($*lseg*(`slist`, `last`) *
(`last` $\longmapsto$ `node`) * *lseg*(`node`, `sibling`) * *true*$\big)$
`ensures` *list*(`return`)

Figure 4.5: The contract and assertions for insert_before

if C' is the concrete program configuration obtained after executing s in configuration C, i.e., $C' = [\![\mathtt{s}]\!]_c(C)$, and A' is the abstract state obtained after executing the instrumented code in configuration C' and abstract state A, i.e., $A' = [\![\mathtt{s}]\!]_a(C', A)$, then $C' \in \gamma(A')$.

Using Theorem 4.3.2 and Theorem 4.3.3 we can show that for every assertion φ in our logic, φ evaluates to true on the updated concrete state c' if and only if φ evaluates to true on the updated abstract state a'.

4.4 Evaluation

In this section we validate the claim that using abstractions for runtime checking results in faster assertion checking than just standard checking on the concrete heap.

Our benchmarks consists of (a) programs manipulating singly-linked lists obtained from the C GLib library [84], such as concatenation of two lists, insertion into and deletion from a list, reversal of a list, etc., and (b) programs that use linked lists as a library, such as stack and queue implementations, LRU-cache implementations, etc. Our LRU implementation is based on the the Least Recently Used page replacement algorithm, used for memory page management. We based our implementation on the algorithm description in Bovet et al. [39]. The first column of Table 4.1 lists the names of the benchmark programs we use in our evaluation; the names are self-descriptive. The list programs are fairly concise, ranging in size from 15 lines to 50 lines. Our programs that use lists as library are moderately concise, with up to 200 lines of code. The programs vary in complexity, some executing in constant time, e.g., prepend, and others being linear time.

We annotated GLib subjects with assertions for preconditions, postconditions, and loop invariants; recall that we do not check that the function stays within the heaplet. Figure 4.5 shows an example of an annotated program in our framework. It shows the precondition, loop invariant, and postcondition, for a program that inserts a new node with key **data** before the node **sibling**. `last` is the pointer used to iterate over the list and it is the last node before sibling when the loop stops. Our assertions check structural correctness using the *list* and *lseg* (for list segments) recursive predicates.

Note that the assertions in the GLib programs occur very often, within the loops/recursive calls. Consequently, the assertions are checked a large number of times, typically linear in the length of the lists manipulated. Some of the

⇒ List Length		10		2048		4096		16384	
⇓ Program	#Asrt	Conc	Abs	Conc	Abs	Conc	Abs	Conc	Abs
Library									
append	$\mathcal{O}(n)$	6	6	10368	8.00	t.o.	8.78	t.o.	9.45
concat	$\mathcal{O}(n)$	5	5	10528	7.79	t.o.	8.51	t.o.	9.97
copy	$\mathcal{O}(n)$	5	4	11840	4.98	t.o.	5.55	t.o.	6.70
find	$\mathcal{O}(n)$	4	3	10344	2.67	t.o.	2.19	t.o.	2.68
free	$\mathcal{O}(n)$	0	1	3.10	1.12	6.33	1.06	27	1.07
insert	$\mathcal{O}(n)$	3	5	1562	7.57	t.o.	8.48	t.o.	9.79
last	$\mathcal{O}(n)$	5	6	10343	7.08	t.o.	7.64	t.o.	9.14
reverse	$\mathcal{O}(n)$	3	3	t.o.	3.42	t.o.	3.52	t.o.	4.01
remove-link	$\mathcal{O}(n)$	4	5	t.o.	7.28	t.o.	8.25	t.o.	9.61
remove-all	$\mathcal{O}(n)$	4	4	t.o.	7.63	t.o.	8.01	t.o.	9.59
position	$\mathcal{O}(n)$	6	4	t.o.	4.17	t.o.	4.07	t.o.	4.14
nth-data	$\mathcal{O}(n)$	3	3	t.o.	2.81	t.o.	2.73	t.o.	2.73
nth	$\mathcal{O}(n)$	3	3	t.o.	2.75	t.o.	2.69	t.o.	2.74
length	$\mathcal{O}(n)$	4	4	t.o.	3.24	t.o.	2.75	t.o.	3.16
insert-at-pos	$\mathcal{O}(n)$	3	3	t.o.	5.00	t.o.	5.47	t.o.	6.36
index	$\mathcal{O}(n)$	2	2	18	0.01	35	0.00	141	0.00
prepend	$\mathcal{O}(1)$	0	1	0.01	0.00	0.01	0.00	0.01	0.00
create	$\mathcal{O}(n)$	0	1	3.09	1.33	6.33	1.34	27.22	1.34
swap	$\mathcal{O}(1)$	0	0	0.01	0.00	0.01	0.00	0.01	0.00
Client									
split	$\mathcal{O}(n)$	2	2	t.o.	1.35	t.o.	1.33	t.o.	1.45
merge	$\mathcal{O}(n)$	7	6	t.o.	5.17	t.o.	5.10	t.o.	5.00
reverse-sublist	$\mathcal{O}(n)$	1	1	t.o.	1.13	t.o.	1.13	t.o.	1.12
insert-sorted	$\mathcal{O}(n)$	1	1	4.52	0.00	8.85	0.00	35	0.00
queue	$\mathcal{O}(1)$	0	0	87.95	3.39	171	3.48	695	5.08
stack-queue	$\mathcal{O}(1)$	0	0	48.82	4.63	82.27	5.37	286	11.37
LRU	$\mathcal{O}(n)$	9	2	94.97	2.11	162	2.29	574	2.24

Table 4.1: Comparison of average running time per assertion check using concrete checks versus leveraging abstraction. Times are shown in μs. Timeout (t.o.): 40min for checking all assertions

clients, such as the queue implementations do not have loop invariants, having only preconditions or postconditions, and hence the assertions occur less frequently in those subjects.

We performed all our experiments on an Intel Core i7 with 32 GB of RAM. We vary the workload in our programs by increasing the input list sizes (lists' sizes range from 10 to 16384 nodes). We report the average running time for each assertion check, obtained over 100 repeating runs, to account for noise in measurement, for the increasing list sizes.

We present the results of our evaluation in Table 4.1. The various columns denote the length of the list input to the program. The first column lists the names of the programs we used, and the second column indicates the number of assertion checks that the program performs for a list of length n. Each 2-column group shows, for lists of varying sizes ranging from 10 to 16384, the average time for checking a *single* assertion in the program with *runtime checking*

on the concrete state (**Conc**) and *runtime checking using the abstract heap* (**Abs**).

The average time taken to check a single assertion is calculated, both for the concrete checking and the abstraction-guided checking, by calculating the time taken by the program with assertions, subtracting the time taken by the program without assertions, and then dividing by the total number of assertions checked. When the program with assertions takes too long, exceeding 40 minutes, we denote a timeout (t.o.).

The runtime checking of the assertions on the concrete heaplet (Conc) grows with the size of the list, and takes typically linear to quadratic time to check the property. This is reasonable and acceptable for smaller lists (say a few hundred), but gets prohibitive for larger lists, timing out on larger lists. We emphasize that even with a *manual encoding by the programmer*, the check will typically take this time— checking properties of lists should, after all, take longer when the lists are longer.

However, the time taken per runtime assertion aided by the abstract heap is almost *constant*, varying very little with the length of the input! In lists ranging to 16K, this shows 20x to 3000x speedup in checking the assertions, in comparison with the checking on the concrete heap. Consequently, runtime checking using the abstract heap scales with acceptable overheads even for our largest input sizes.

In the `stack-queue` client the number of checks is smaller because we do not require the loop invariant for the properties we check. The smaller number of checks is reflected as an increase in the assertion checking time.

Intuitively, the size of the abstract heaplet stays constant, and hence checking an assertion on the abstract heaplet can be done in constant time. However, the abstract heaplet requires maintenance, but this is typically a constant amount of work for each heap manipulation. When the number of assertions are high and the lengths of the lists are large, the maintenance cost of the heaplet is not significant, and we obtain essentially a constant amount of cost for checking an assertion, independent of the length of the list.

Our experiments clearly show that the use of runtime-guided abstractions can make runtime checking of data-structure properties much faster, when the assertions are checked often and the sizes of the data structure are large.

We believe that using these in settings where assertions abound, such as in *class invariants*, where invariants are checked every time the data structure is accessed, can benefit greatly by our approach. However, when assertions are sparse, the cost of maintaining the abstract state may get expensive, and it may be more prudent to check the assertion on the concrete heap. An automatic hybrid approach that exercises these choices to instrument large programs is an interesting future direction.

4.5 Related Work

Runtime assertion checking has been successfully used in software engineering and programming language design (e.g., [57]). Debugging using assertions expressed as Boolean formulas is a routine software development practice [93]. However, there are relatively few approaches that can be used for checking properties of programs manipulating structurally complex data. A common technique of specification for that class of program are *representation invariants* (i.e., REPOK [121]). Implementing representation invariants can be hard to get right, also it imposes a significant burden on the developer [127, 40]. Jump et al. [101] introduce dynamic shape analysis and check structural properties of the heap. Crane and Dingel [67] present a declarative language for specifying object models using the *Alloy* language, and perform runtime checks to ensure that certain user specified locations conform to an object model. However, runtime checking of these properties incurs large overheads.

Some recent approaches (e.g., [140, 14]) propose using runtime checking assertions written in separation logic. Separation logic has been successfully used as a specification language in deductive verification of programs manipulating complex data structures, making it an attractive choice for runtime assertion checking. However, as noted by Nguyen et al. [140], runtime checks of separation logic assertions can be challenging due to implicit footprint and existential quantification. A main focus of the work described by Nguyen et al. [140] is alleviating potentially exponential blow-up of sets of locations that needs to be considered when splitting the heap in two parts when evaluating separation connective. They use a marking technique to limit the set of footprints that needs to be explored when evaluating the formula. Their approach works well when checking only preconditions and postconditions of data structures at the boundary between statically verified and unverified code, however performing multiple checks often incurs prohibitively large overheads. In contrast, we focus on choosing a separation logic for lists, suitable for runtime assertion checking, and developing a technique that will allow checking properties in near constant time using abstractions. Our technique works well both in cases where assertions are at the boundary and also when they are intensively checked throughout the program.

Agten et al. [14] devised another technique for run-time checks of separation logic annotations. This approach combines deductive verification with run-time checks of the unverified parts of the code to provide stronger run-time guarantees for the verified parts. Run-time checks are introduced at the boundary between verified and un-verified parts. A key difference comparing to our approach is that Agten et al. [14] the assertions are meant to be checked sparsely (only when crossing the verified-unverified boundary), while our approach excels even when assertions are checked frequently.

A technique proposed by Shankar and Bodik [152] reduces the run-time overhead through incremental assertion checking. The technique devises automatic memoization for a class of side-effect-free representation invariants. However, it is developer's responsibility to provide correct representation invariants. Koukoutos and Kuncak [109] also

focus on reducing run-time overhead induced by dynamic checks of complex program properties. They tackle the run-rime overhead using memoization techniques, restricting the domain to purely functional Scala programs.

Our abstraction is inspired by a common abstraction used in shape analysis [128]. However, comparing to static shape analysis approaches, we exploit runtime information to maintain accurate abstract graph, which we use to reduce overhead of runtime checks.

Vechev et al. [167] present PHALANX, a tool that uses parallelism to speed up assertion checking. PHALANX evaluates the assertions in a different thread, on a snapshot of the entire state at the assertion. Similarly, Atandilian et al. [13] introduce asynchronous assertions, which can be used during debugging. Our work shares the goal of speeding up assertion checking, but we use abstractions to perform the assertions in constant time; these techniques are complementary to our work.

4.6 Conclusions

In this chapter we describe an abstraction-guided runtime checking for linked lists, where we have shown how an abstraction maintained using the runtime state helps in efficient runtime assertion checking, often close to constant time per assertion.

We would like to, explore abstraction-guided runtime checking for more rich data structures (such as doubly-linked lists, trees, etc.) and for recursively defined functions (not just predicates) on these structures (such as the length of lists, heights of trees, set of keys stored in the structure, etc.). It is presently unclear how to build such abstraction guidance effectively. Furthermore, we believe that in larger program contexts, abstraction-guidance should be used for often checked assertions (like class invariants) while the concrete structure is checked for sparse assertions; building an effective hybrid scheme would be interesting. Finally, we would like to integrate work described in this chapter with the work described in the previous chapter to extend the applicability and improve the usability of VCDRYAD program verifier.

4.7 Proofs

Theorem 1 (Proof Sketch)

Lemma 4.7.1. *For a formula* α

if α *is domain exact then*

$$\forall H \subseteq G.\ H \models \alpha \quad \Rightarrow \quad H = scope(G, \alpha)$$

else if α *is not domain exact then*

$$\forall H \subseteq G.\ H \models \alpha \quad \Rightarrow \quad H \supseteq scope(G,\alpha) \wedge \forall H' \supseteq scope(G,\alpha).H' \models \alpha$$

Proof. The proof is straightforward and it follows by induction on the structure of α. □

Definition 4.7.2 ($rep_{A,C}$)**.** *Let $rep_{A,C} : V \cup next_a \rightarrow 2^{LOC} \setminus \{\emptyset\}$ be a function denoting a mapping from nodes and edges in the abstract graph to the corresponding concete heap locations.*

Let the map $rep_{A,C}$ be extended to sets of nodes and edges in the abstract graph in the natual way. For a set S of nodes and edges, $rep_{A,C}(S)$ is the union of $rep_{A,C}(s)$ for every $s \in S$.

Lemma 4.7.3. *Let G_a, G_c be an abstract and concrete heap (resp.), α a formula, and $scope(G_a,\alpha)$ the scope of the formula α in the abstract heap G_a, $scope(G_c,\alpha)$ the scope of the formula α in the concrete heap G_c. The following holds:*

$$rep_{A,C}(scope(G_a,\alpha)) = scope(G_c,\alpha)$$

Proof. The proof is a straightforward induction on the structure of α and follows from Definition 4.7.2. □

Lemma 4.7.4. *Let x and y be abstract heaplets, then the following holds:*

$$rep_{A,C}(x) \cap rep_{A,C}(y) \neq \emptyset \quad \textit{iff} \quad x \cap y \neq \emptyset$$

Proof. The proof follows from the definition of $rep_{A,C}$.

□

Corollary 4.7.5. *Given abstract heap G_a and concrete heap G_c s.t. $G_c \in \gamma(G_a)$, and formulas α, β:*

$$scope(G_a,\alpha) \subseteq scope(G_a,\beta) \quad \textit{iff} \quad scope(G_c,\alpha) \subseteq scope(G_c,\beta)$$

.

Proof. The proof simply follows from Defintion 1 and Lemma 4.7.3. □

Corollary 4.7.6. *Given abstract heap G_a and concrete heap G_c s.t. $G_c \in \gamma(G_a)$, and formulas α, β:*

$$scope(G_a,\alpha) \cap scope(G_a,\beta) = \emptyset \quad \textit{iff} \quad scope(G_c,\alpha) \cap scope(G_c,\beta) = \emptyset$$

Proof. The proof follows simply from Lemma 4.7.3 and Lemma 4.7.4.

□

Lemma 4.7.7. *For an abstract store S_a, abstract heap G_a, concrete store S_c, concrete heap G_c, and any formula α:*

$$S_c, scope(G_c, \alpha) \models \alpha \quad \textit{iff} \quad S_a, scope(G_a, \alpha) \models \alpha$$

Proof. The proof is by induction on the structure of α using Lemmas 4.7.1, 4.7.3, 4.7.4 and Corollaries 4.7.5, 4.7.6. □

Next we prove Theorem 1. Note that the following is a slight reformulation of the theorem presented in Section 3.2.

Theorem 4.7.8. *For a concrete heap G_c, abstract heap G_a, s.t. $G_c \in \gamma(G_a)$ and a formula α the following holds:*

$$G_c \models \alpha * true \quad \textit{iff} \quad G_a \models \alpha * true$$

Proof. Assume $G_c \models \alpha * true$, then there exists precisely one concrete heaplet (with the corresponding concrete store) on which it holds, namely its scope: $S_c, scope(G_c, \alpha) \models \alpha$. By Lemma 4.7.7 it follows that $S_a, scope(G_a, \alpha) \models \alpha$. Assume now that $G_a \not\models \alpha * true$, implying that there does not exist an abstract heaplet on which $\alpha * true$ holds, namely it does not hold on its scope with store S_a – a contradiction.

Assume $G_c \not\models \alpha * true$, then there does not exist a concrete heaplet (with the corresponding store) on which the formula holds: $S_c, scope(G_c, \alpha) \not\models \alpha$. Hence, by Lemma 4.7.7 it follows that $S_a, scope(G_a, \alpha) \not\models \alpha$, that is, there does not exist an abstract heaplet on which α holds. Assume $G_a \models \alpha * true$, which means there is precisely one abstract heaplet (and the corresponding abstract store) on which the formula holds: $S_a, scope(G_a, \alpha) \models \alpha$ – a contradiction. □

Chapter 5

Certified Programs as Model

In this chapter we present a new approach, *certified program models*, to establish correctness of distributed protocols. Unlike the previous chapter where we were focused on the system implementation, in this chapter we are focused on system modeling. In particular, we propose modeling protocols as programs in standard languages like C, where the program simulates the processes in the distributed system as well as the nondeterminism, the communication, the delays, the failures, and the concurrency in the system. The program model allows us to test the protocol as well as to verify it against correctness properties using program verification techniques. The highly automated testing and verification engines in software verification give us the tools needed to establish correctness. Furthermore, the model allows us to easily alter or make new design decisions, while testing and verifying them.

We carry out the above methodology for the distributed key-value store protocols underlying widely used frameworks such as Dynamo [70], Riak [2] and Cassandra [4]. We model the read-repair and hinted-handoff table based recovery protocols as concurrent C programs, test them for conformance with real systems, and then verify that they guarantee eventual consistency, modeling precisely the specification as well as the failure assumptions under which the results hold. To the best of our knowledge, this is the first verification technique that shows correctness of these distributed protocols using mostly-automated verification.

5.1 Introduction

Distributed systems are complex software systems that pose myriad challenges to formally verifying them. While many distributed protocols running in these systems stem from research papers that describe a core protocol (e.g., Paxos), their actual implementations are known to be much more complex (the "Paxos Made Live" paper [49] shows how wide this gap is).

The aim of this chapter is to strike a middle-ground in this spectrum by verifying models of actual protocols implemented in systems. We propose a new methodology, called *certified programs models*, where we advocate that the fairly complex protocols in distributed systems be modeled using *programs* (programs written in traditional systems languages, like C with concurrency), and certified to be correct against it's specifications.

The idea is to model the entire distributed system in software, akin to a software simulator of the system. The model captures the distributed processes, their memory state, the secondary storage state, the communication, the delays, and the failures, using non-determinism when necessary.

The salient aspects of this modeling are that it provides:

(a) a modeling language (a traditional programming language) to model the protocols precisely,

(b) an *executable* model that can be validated for accuracy with respect to the system using testing, where the programmer can write test harnesses that control inputs as well as physical events such as node and network failures, and test using mature systematic testing tools for concurrent software, like CHESS [134, 135].

(c) an accurate modeling of specifications of the protocol using *ghost state* in the program as well as powerful assertion logics, and

(d) a program model that lends itself to *program verification techniques*, especially using tools such as VCC [60] that automate large parts of the reasoning using logical constraint solvers.

In this chapter, we explore the certified model paradigm for modeling, testing, and formally proving properties of core distributed protocols that underlie eventually consistent distributed key-value/NoSQL stores. Eventually consistent key-value stores originated with the Dynamo system from Amazon [70] and are today implemented in systems such as Riak [2], Cassandra [4], and Voldemort [10]. We show how to build program models for them in concurrent C, test them for conformance to the intended properties of the systems by using automated testing tools like CHESS [134, 135], and formally verify the eventual consistency property for them using VCC [60], a verification tool for concurrent C.

5.1.1 Key-value/NoSQL Storage Systems and Eventual Consistency

Key-value/NoSQL stores are on the rise [8] and are used today to store Big Data in many companies, e.g., Netflix, IBM, HP, Facebook, Spotify, PBS Kids, etc. rely heavily on the Cassandra key-value store system while Riak is used by BestBuy, Comcast, the NHS UK, The Danish Health and Medicines Authority for patient information, and Rovio, the gaming company behind AngryBirds.

Key-value/NoSQL storage systems arose out of the CAP theorem/conjecture, which was postulated by Brewer [43, 42] (a proof under a particular model is given by Gilbert and Lynch [123, 83]). The conjecture states that a distributed storage system can choose at most two out of three important characteristics— strong data **c**onsistency (i.e., linearizability or sequential consistency), **a**vailability of data (to reads and writes), and **p**artition-tolerance. Hence achieving strong consistency while at the same time providing availability in a partitioned system with failures is impossible.

While traditional databases preferred consistency and availability, the new generation of key-value/NoSQL sys-

tems are designed to be partition-tolerant in order to handle highly distributed partitions that arise due to the need of distributed access, both within a datacenter as well as across multiple data-centers. As a result, a key-value/NoSQL system is forced to chose between one of either strong consistency or availability— the latter option providing low latencies for reads and writes.

Key-value/NoSQL systems that prefer availability include Cassandra [111], Riak [2], Dynamo [70], and Voldemort [10], and support weak models of consistency (e.g., eventual consistency). Other key-value/NoSQL systems instead prefer strong consistency, e.g., HBase [7], Bigtable [50], and Megastore [22], and may be unavailable under failure scenarios.

One popular weak consistency notion is eventual consistency, which roughly speaking, says that if no further updates are made to a given data item, all replicas will eventually hold the same value (and a read would then produce this value). Eventual consistency is a *liveness property*, not a safety property [21]. The precise notion of what eventual consistency means in these protocols (the precise assumptions under which they hold, the failure models, the assumptions on the environment, etc.) are not well understood, let alone proven. Programmers also do not understand the subtleties of eventually consistent stores; for instance, default modes in Riak and Cassandra can permanently lose writes— this is dangerous, and has been exploited in a recent attack involving BitCoins [9].

5.1.2 Contributions

The primary contribution of the work described in this chapter is an approach that enables us to precisely reason about the guarantees of eventual consistency that real implementations of key-value stores provide. We model two core protocols in key-value stores as programs, the *hinted-handoff protocol* and the *read-repair* protocol, which are anti-entropy mechanisms first proposed in the Amazon Dynamo system [70], and later implemented in systems such as Riak [2] and Cassandra [4].

We build certified program models— program models for these protocols written in concurrent C and that are verified for eventual consistency. The program uses *threads* to model concurrency, where each get/put operation as well as the asynchronous calls they make are modeled using concurrently running threads. The state of the processes, such as stores at replicas and the hinted-handoff tables, are modeled as shared arrays. Communication between processes is also modeled using data-structures: the network is simulated using a set that stores pending messages to replicas, with an independent thread sending them to their destinations. Failures and non-determinism of message arrivals, etc., are also captured programmatically using non-determinism (modeled using stubs during verification and using random coin-tosses during testing). In particular, system latency is captured by threads that run in the background and are free to execute anytime, modeling arbitrarily long delays.

In the case of the *hinted-handoff protocol*, we prove that this protocol working alone guarantees eventual consis-

tency provided there are only transient faults. In fact, we prove a stronger theorem by showing that for any operation based (commutative) conflict-free replicated data-type implementing a register, the protocol ensures *strong eventual consistency*— this covers a variety of schemes that systems use, including Riak and Cassandara, to resolve conflict when implementing a key-value store. Strong eventual consistency guarantees not only eventual consistency, but that the store always contains a value that is a function of the set of updates it has received, independent of the order in which it was received. We prove this by showing that the hinted-handoff protocol (under only transient failures) ensures *eventual delivery* of updates; this combined with an idempotent CmRDT [156, 154] implementing a register ensures strong eventual consistency. We model the eventual delivery property in the program model using a ghost *taint* that taints a particular write at a coordinator (unbeknownst to protocol), and asserts that the taint propagates eventually to every replica. Eventual delivery is a *liveness property*, and is established by finding a ranking function that models abstractly the time needed to reach a consistent state, and a slew of corresponding safety properties to prove this program correct.

For the *read-repair protocol*, we first believed the popularly-held opinion that a read-repair (issued during a read) would bring the nodes that are alive to a consistent state eventually, and tried to prove this property. However, while working on the proof, we realized that there is no invariant that can prove this property, and this made us realize that the property in fact does not hold. A single read is insufficient, and we hence prove a more complex property: at any point, if a set of nodes are alive and they all stay alive, and if all requests stop except for an unbounded sequence of reads to a key, then the live nodes that are responsible for the key will eventually converge.

Note that the certification that the program models satisfy their specification is for an *unbounded* number of threads, which model an unbounded number of replicas, keys, values, etc., model arbitrarily long input sequences of updates and reads to the keys, and model the concurrency prevalent in the system using parallelism in the program. The verification is hence a *complete* verification as opposed to several approaches in the literature which have used under-approximations in order to systematically test a bounded-resource system [122, 139, 138, 129]. In particular, Amazon has reported modeling of distributed protocols using TLA, a formal system, and used model-checking (systematic testing) on bounded instances of the TLA system to help understand the protocols, check their properties, and help make design decisions. Our results, in contrast, model protocols using C programs, which we believe are much simpler for systems engineers to use to model protocols, and being executable, is easy to test using test harnesses. Most importantly, we have proven the entire behavior of the protocol correct (as opposed to the work using TLA) using the state-of-the-art program verification framework VCC [60] that automates several stages of the reasoning.

We also give an account of our experience in building certified program models (Section 5.6). In addition to resulting in proven models, there were several other side benefits that resulted, including a vocabulary of reasoning that the model provided, a nuanced accurate formalization of the assumptions under which eventual consistency holds,

as well as helping us realize that certain specifications that we believed in did *not* hold.

The chapter is structured as follows. Section 5.2 describes key-value stores, eventual consistency, and the main anti-entropy protocols that are implemented in systems and that we study in this chapter (readers familiar with these topics can choose to skip this section). We describe our main results in Section 5.3, where we describe the precise property we prove for the protocol models as well as some properties that we expected to be initially true, but which we learned were not true through our experience. Section 5.4 describes our models of protocols using programs in detail, including the testing processes we used to check that our model was reasonable. The entire verification process, including background on program verification, the invariants and ranking functions required for proving the properties, etc., are given in Section 5.5. A gist of the effort we put in, the experience we had, and the lessons we learned are described in Section 5.6. Section 5.7 describes related work and Section 5.8 concludes with interesting directions for future work.

5.2 Background

In this section we describe in detail the read and write paths involved in a key-value store, and the anti-entropy mechanisms which are used to implement eventual consistency by reconciling divergent replicas. Readers familiar with key-value store system internals can skip this section without loss of continuity.

5.2.1 Key-value Stores

Key-value stores have a simple structure. They store pairs of keys and values, and they usually have two basic operations: get(key) for retrieving the value corresponding to the key, and put(key, value) for storing the value of a particular key[1]. Key-value stores typically use consistent hashing [102] to distribute keys to servers, and each key is replicated across multiple servers for fault-tolerance. When a client issues a put or get operation, it first interacts with a server (e.g., the server closest to the client). This server plays the role of a *coordinator*: it coordinates the client and replica servers to complete *put* and *get* operations. The CAP theorem [43] implies that under network partitions (the event where the set of servers splits into two groups with no communication across groups), a key-value store must choose either consistency (linearizability) [91] or availability. Even when the network is not partitioned, the system is sometimes configured to favor latency over consistency [11]. As a result, popular key-value stores like Apache Cassandra [111] and Riak [2] expose tunable *consistency levels*. These consistency levels control the number of processes the coordinator needs to hear from before returning and declaring success on reads and writes. For instance, a write threshold of one, would allow the system to return with success on a write when it has successfully written to

[1] We use both read/write and get/put terms to mean data fetch and data update operations.

just one replica. When the sum of read and write thresholds is greater than the number of replicas, the system will ensure strong consistency. Consistency levels weaker than quorum are the most popular since they achieve low latency (e.g., Amazon [1]).

5.2.2 Eventual Consistency

In general, a consistency model can be characterized by restrictions on operation ordering. The strongest models, e.g., linearizability [91] severely restrict the possible orderings of operations that can lead to correct behavior. Eventual consistency lies at the opposite end of the spectrum; it is the weakest possible consistency model. Informally, it guarantees that, if no further updates are made to a given data item, reads to that item will eventually return the same value [168]. Thus until some undefined time in the future when the system is supposed to converge, the user can never rule out the possibility of data inconsistency. Despite the lack of any strong guarantees, many applications have been successfully built on top of eventually consistent stores. Most stores use some variation of anti-entropy [71] protocols to implement eventual consistency mechanisms.

5.2.3 Anti-entropy Protocols

To achieve high availability and reliability, key value stores typically replicate data on multiple servers. For example, each key can be replicated on N servers, where N is a configurable parameter. In the weakest consistency setting (consistency level that has read and write thresholds of one), each get and put operation only touches one replica (e.g., the one closest to the coordinator). Thus in the worst case scenario, if all puts go to one server, and all gets are served by a different server, then the replicas will never converge to the same value. To ensure convergence to the same value, key-value stores like Dynamo [70], Apache Cassandra [4], and Riak [2] employ *anti-entropy* protocols. An anti-entropy protocol operates by comparing replicas and reconciling differences. The three main anti-entropy protocols are: (1) Read-Repair (RR), (2) Hinted-Handoff (HH), and (3) Node-Repair (NR). While the first two are *real-time* protocols involved in the read and write paths respectively, the third one is an off-line background maintenance protocol, which runs periodically (e.g., during non-peak load hours) to repair out-of-sync nodes (e.g., when a node rejoins after recovering from a crash). In this chapter we are only concerned with the real-time anti-entropy protocols. Node-repair is mostly an offline process whose correctness lies solely in the semantics of the merge, so we do not consider it in this chapter.

Read-Repair (RR)

Read-repair [70] is a real-time anti-entropy mechanism that ensures that all replicas have (eventually) the most recent version of a value for a given key (see Figure 5.1). In a typical read path, the coordinator forwards read requests to

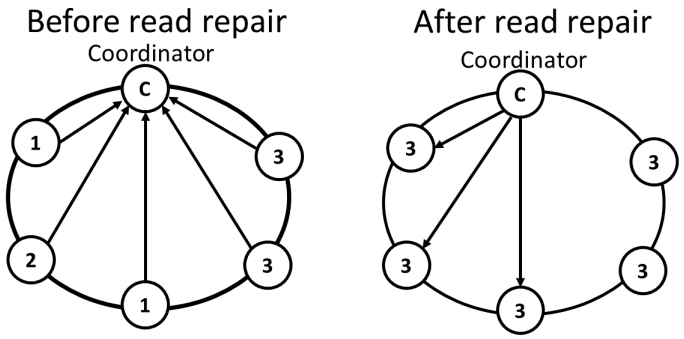


Figure 5.1: Read-repair propagation : before read-repair, replicas can have inconsistent values (C is the coordinator). After the propagation of the latest value to all out-of-date replicas, all replicas converge on a value.

all replicas, and waits for a consistency level (*CL* out of *N*) number of replicas to reply. If read-repair is enabled, the coordinator checks all the read responses (from the nodes currently alive), determines the most recent read value[2], and finally pushes the latest version to all out of date replicas.

Hinted-Handoff (HH)

Unlike read-repair, hinted-handoff [70] is part of the write path. It offers full write availability in case of failures, and can improve consistency after temporary network failures. When the coordinator finds that one of the replicas responsible for storing an update is temporarily down (e.g., based on failure detector predictions), it stores a hint meta-data for the down node for a configurable duration of time. Once the coordinator detects that the down node is up, it will attempt to send the stored hint to that recovered node. Thus hinted-handoff ensures that no writes are lost, even in the presence of temporary node failures. In other words this mechanism is used to ensure that eventually all writes are propagated to all the replicas responsible for the key.

5.3 Characterizing and Proving Eventual Consistency

The goal of this work describe in this chapter is to prove eventual consistency of the hinted-handoff and read-repair protocols that systems like Cassandra and Riak implement, delineating precisely the conditions under which they hold. Our effort spanned a period of 15 months, with about 6 person months of effort for modeling and verification. In order to accomplish this task, we *abstract* away from the particular instantiation of these protocols in these systems, and also abstract away from the various options they provide to users to modify the behavior of the system.

[2]Determining the most recent version of data to push to out of date replicas is implementation dependent. For Apache Cassandra, the replica value with highest client timestamp wins. Riak uses vector clocks to decide the winner, and can deduce multiple winners in case of concurrent writes.

For instance, in Riak, using one set of options, every write is tagged with a vector clock at the client, and every replica responsible for it maps it to a *set of values*, one for each last concurrent write that it has received. When a read is issued, Riak can return the set of *all* last concurrently written values to it (these values are called "siblings" in Riak). However, in Cassandra, vector clocks are not used; instead each client labels every write with a timestamp, and despite there being drift amongst the clocks of clients, each replica stores only the last write according to this timestamp. Further, these policies can be changed; for instance in Riak, a user can set options to mimic the Cassandra model.

We will capture these instantiations by generalizing the semantics of how the store is maintained. For the hinted-handoff protocol, we prove eventual consistency under the assumption that the stores are maintained using some *idempotent* operation-based commutative replicated data-type (CRDT) [154, 155] that implements a *register*, while for read-repair, we prove eventual consistency assuming an arbitrary form of conflict resolution.

Failure Models

Let us first discuss the failure models we consider, which are part of the assumptions needed to prove properties of protocols. We consider two failure modes:

- *Transient failure:* Nodes or network edges can fail, but when they come back, they preserve the state at which they crash and resume from there.
- *Permanent failure:* Nodes or network edges can fail, and when they come back, they have lost main memory and start with some default store.

5.3.1 Properties of the Hinted-Handoff Protocol

The hinted-handoff protocol is an opportunistic anti-entropy mechanism that happens during writes. When a write is issued, and the asynchronous call to write to certain replicas fail (either explicitly or due to a time-out), the coordinator knows that these replicas could be out of sync, and hence stores these update messages in a hinted-handoff table locally to send them later to the replicas when they come back alive. However, if there is a memory crash (or a permanent failure), the hinted-handoff table would be lost, and all replicas may not receive the messages. In practice, the read-repair (and node-repair) protocols protect against permanent failures.

Commutative Replicated Data Type for Registers: Our main abstraction of the key-value store is to view the underlying protocol as implementing a *register* using an operation-based conflict-free replicated datatype (CRDT)(also called a commutative replicated data-type CmRDT [156, 155]).

We also assume another property of these CmRDTs, namely idempotency— we assume that all messages are tagged with a unique id, and when a message is delivered multiple times, the effect on the store is the same as when exactly one message is delivered. Let us call these idempotent CRDTs[3].

When implementing a simple key-value store, the vector-clock based updates in Riak and the simpler time-stamp based update in Cassandra can in fact be both seen as idempotent CmRDTs, the former being a *multi-valued* (MV) register, and the latter being a *last write wins* (LWW) register (see [154]). (However, since a global wall-clock time is not available, in general, this strategy in Cassandra can *lose* updates [3]). The CmRDTs for both Last Write Wins (LWW) and Multi-valued (MV) registers are in fact idempotent— the systems tags each write with a timestamp, and the conflict-resolution will ignore the future deliveries of a message with same time-stamp (see [154], Section 3.2).

The main property we prove about the hinted-handoff protocol is a property called *eventual delivery*, which says that every successful write eventually gets delivered to every replica at least once (under assumptions of kinds of failure, assumptions on replicas being eventually alive, etc.). Hence, instead of eventual consistency, we argue eventual delivery, which in fact is the precise function of these protocols, as they are agnostic of the conflict resolution mechanism that is actually implemented in the system. Furthermore, assuming that each replica actually implements an idempotent operation-based CRDT register, and update procedures for these datatypes are terminating, eventual delivery ensures eventual consistency, and in fact *strong eventual consistency* [155]. Recall that strong eventual consistency guarantees not only eventual consistency, but that the store always contains a value that is a function of the set of updates it has received, independent of the order in which it was received.

Our first result is that a system running only hinted-handoff-based repair provides eventual delivery of updates to all replicas, provided there are only transient faults.

Result#1: *The hinted-handoff protocol ensures eventual delivery of updates to all replicas, provided there are only transient faults. More precisely, if there is any successful write, then assuming that all replicas recover at some point, and reads and write requests stop coming at some point, the write will get eventually propagated to every replica.*

We formally prove the above result (and Result#2 mentioned below) for arbitrary system configurations using program verification techniques on the program model (see Section 4 and Section 5 for details).

The following is an immediate corollary from the properties of eventual delivery and idempotent CRDTs:

Corollary#1: *A system following the hinted-handoff protocol, where each replica runs an operation-based idempotent CRDT mechanism that has terminating updates, is strongly eventually consistent, provided there are only transient faults.*

Aside: The above corollary may lead us to think that we can use any operation-based CmRDT for counters at stores

[3]Standard definitions of Operation-based CRDTs do not guarantee idempotency— instead they assume the environment delivers every message precisely once to each replica (see [155], text after Definition 5). Note that state-based CRDTs are defined usually to be idempotent.

to obtain strong eventually consistent counters in the presence of transient failures. However, CmRDTs for counters are in fact *not idempotent* (and the CmRDT counters in [155] assume that the system will deliver messages precisely once, which hinted handoff cannot guarantee).

5.3.2 Properties of the Read-repair Protocol

Our second result concerns the read-repair protocol. Read-repair is expected to be resilient to memory-crash failures, but only guarantees eventual consistency on a key provided future reads are issued at all to the key. Again, we abstract away from the conflict resolution mechanism, and we assume that the coordinator, when doing a read and getting different replies from replicas, propagates *some* consistent value back to all the replicas. This also allows our result to accommodate anti-entropy mechanisms [71] that are used instead of read-repair, in a reactive manner after a read. Note that this result holds irrespective of the hinted-handoff protocol being enabled or disabled.

It is commonly believed that when a read happens, the read repair will repair the live nodes at the time of the read (assuming they stay alive), bringing them to a common state. We modeled the read-repair protocol and tried to prove this property, but we failed to come up with appropriate invariants that would ensure this property. This led us to the hypothesis that the property may not be true.

To see why, consider the time-line in Figure 5.2. In this scenario, the client issues a *put* request with the value 2, which is routed by the coordinator to all three replicas– A, B, and C (via messages $w_A(2), w_B(2)$, and $w_C(2)$). The replica C successfully updates its local store with this value. Consider the case when the write consistency is one and the put operation succeeds (inspite of the message $w_B(2)$ being lost and the message $w_A(2)$ being delayed). Now assume that the replica C crashes, and the last write (with value 2) is in *none* of the alive replicas– A and B. If we consider the case where B has the latest write (with value 1) amongst these two live nodes, a subsequent read-repair would write the value 1 read from B to $A's$ store (via message $rrw_A(1)$ in Figure 5.2). But before this write reaches A, A could get a pending message from the network ($w_A(2)$) and update its value to a more recent value– 2. In this situation, after replica A has updated its value to 2, the two alive replicas (A and B) do not have consistent values. Due to the lack of hints or processes with hints having crashed B may never receive the later write (message $w_B(2)$).
We therefore prove a more involved property of read-repair:

Result#2: *After any sequence of reads and writes, if all operations stop except for an infinite sequence of reads of a key, then assuming the set R of replicas are alive at the time of the first such read and thereafter, the replicas in R will eventually converge to the same value.*

We prove the above result also using program verification on the program model. Intuitively, as long as an indefinite number of reads to the key happen, the system will ensure that the subset of live replicas responsible for the key converge to the same value, eventually. A read-repair may not bring the live replicas to sync if there are some pending

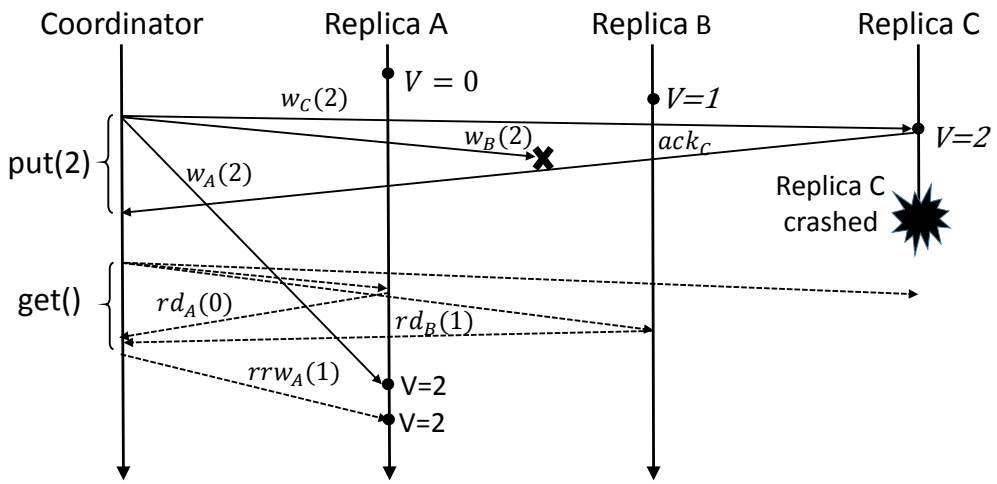


Figure 5.2: The time-line showing that a single read-repair operation does not guarantee convergence of the live replicas. In the figure w_r are write messages to replica r, rd_r are messages from replica r to the coordinator on the read path, and rrw_r is the read-repair message to replica r. Time in the figure advances from top to bottom. The messages along the read(-repair) path are shown as dotted lines and along the write path as solid lines.

messages in the system. However, since there is only a finite amount of *lag* in the system (pending messages, pending hints, etc.), and once the system is given enough time to finish its pending work, a read-repair will succeed in synching these replicas.

5.3.3 Read-repair and CRDT

It is tempting to think that one could implement any CRDT and reach eventual consistency of the CRDT store using solely read-repair, similar to the Corollary we obtained for Result#1. However, this is tricky when clients send operations to do on the CRDT and the conflict-resolution in read-repair happens using state-based merges.

For instance, assume that we implement a counter CRDT, where state-merges take the maximum of the counters, and operations increment the counter [155]. Then we could have the following scenario: there are 7 increments given by clients, and the counter at replica *A* has the value 5 and replica B has 7 (with two increments yet to reach *A*), and where a read-repair merges the values at these replicas to 7, after which the two pending increments arrive at *A* incrementing it to 9 (followed by another read-repair where B also gets updated to 9). Note that consistency is achieved (respecting our Result#2), but the counter stores the wrong value.

Systems such as Riak implement CRDTs [6] using these underlying protocols by *not* propagating operations (like increments) across replicas, but rather increment one replica, and pass the *state* to other replicas, and hence implement a purely state-based CRDT [5].

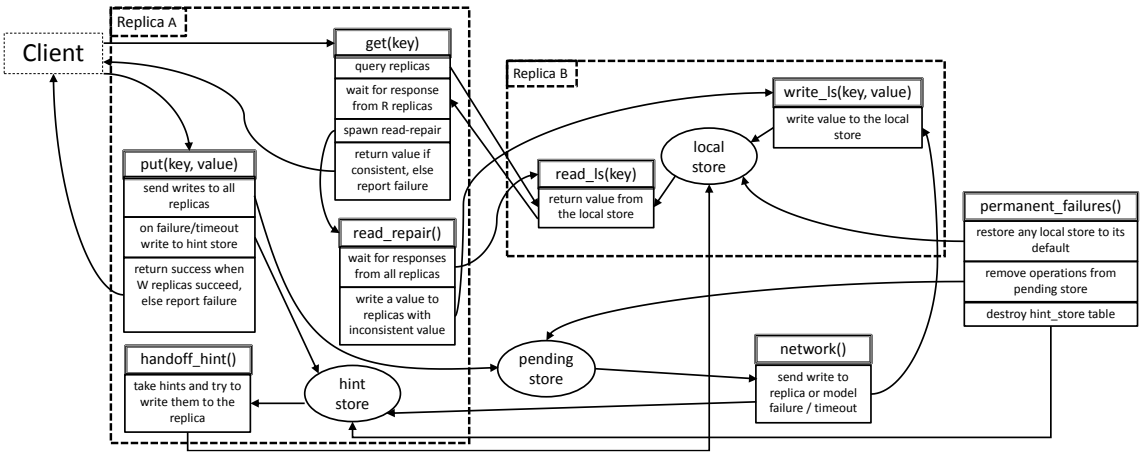


Figure 5.3: Architecture of the model: boxes indicate methods, ellipses show data structures, arrows show communications.

5.4 Program Models for Protocols

In this section we describe how we model the anti-entropy protocols used in eventually consistent key-value stores. The architecture of our model is depicted in Figure 5.3.

Our model consists of several methods (*get*, *put*, *write_ls*, *read_ls*, etc.) for each replica, that run concurrently as threads, make asynchronous calls to each other, and keep their state through shared data-structures (*local_store*, *hint_store*, etc.). Furthermore, in order to model asynchronous calls, we maintain a data-structure *pending_store*, that models messages in the network that haven't yet been delivered.

The methods in our model include:

- The *get* and *put* methods at coordinators that forms the interface to clients for reading and writing key-values.
- An internal method *handoff_hint* for each replica that runs all the time and removes hints from the hinted-handoff table and propagates them to the appropriate replicas (provided they are alive).
- An internal method *read_repair* which is part of the read path, waits for all the replicas to reply, and on detecting replicas with inconsistent values writes the consistent value to those replicas.
- Internal methods *read_ls* and *write_ls*, that read from and write to the local stores (provided they are alive).
- An internal method *network* that runs all the time and delivers messages in the pending store to replicas.
- An internal method *permanent_failures*, which when permanent failure is modeled, runs all the time, and can remove elements from the pending set (modeling loss of messages), restore any local store to its default value (modeling store crashes), and destroy hinted-handoff tables.

We use the following data-structures:

- An array *LocalStore[]* that stores the local store for each replica and each key that the replica maintains.
- An array *HintStore[]* that stores, for each replica, the set of hints stored at the replica.
- An array *PendingStore[]* that stores a set of pending messages on the network between replicas.

Note that the modeling of these methods using fine-grained concurrency ensures arbitrary interleaving of these processes as well as arbitrary delays in them. Also, transient failures, where nodes fail but resume later with the correct state, can be seen as delays in processes, and hence are captured in this concurrency model. The thread that delivers messages in the pending set models arbitrary delays in the network.

The *read_ls* and *write_ls* methods are modeled abstractly as idempotent CRDTs by defining them as stubs which maintain properties. When testing, these methods need to be instantiated to particular conflict-resolution strategies (like *MV* and *LWW*).

Modeling the *get* operation: When a client issues a *get* request for a key, in our model the request is routed to the coordinator that is determined for this key according to an abstract map (our verification hence works for all possible hashing schemes). Every key-value datum is replicated across multiple nodes, where the number of nodes that contain the key-value datum is determined by a replication factor. The coordinator maintains a preference list of replicas that contain data values for keys that are mapped to it. Along the read path, the coordinator asynchronously issues the read request to all replica threads (an asynchronous call to a replica is depicted in Figure 5.3 as an arrow from the *get* method to *read_ls*). As shown in Figure 5.3, the coordinator blocks for a non-deterministic amount of time or until it receives enough responses (the arrow directed from *read_ls* to *get*) as specified by the read consistency level R. On receiving responses from R replicas, it returns the read value(s) to the client. If read repair is enabled, the coordinator also spawns a background thread (depicted as a call to *read_repair* from *get* in Figure 5.3) which will wait for responses from the other replicas (it already knows about responses from the R replicas) for a non-deterministic amount of time. This thread determines the most recent data value of all the values stored in various replicas, and writes it to the replicas with stale values.

Modeling the *put* operation: When a client issues a *put* request to store a key-value pair, the request is routed to the appropriate coordinator, as explained before. The coordinator asynchronously issues write requests to all replica threads in its preference list. The coordinator then blocks for a non-deterministic amount of time or until it receives enough responses, as specified by the write consistency level W. To model arbitrary network delays or failures of the replicas, the write operations to these replicas are inserted by the coordinator into the pending store data structure (in

Figure 5.3 this is depicted as an arrow from *put* to the *pending_store*). If the coordinator receives responses from W replicas, it informs the client about the successful *put* operation.

Modeling the network: A background *network* thread models arbitrary network delays or failure scenarios as it removes a write operation from the pending store data structure and, non-deterministically, either updates the local store of the appropriate replica with the write or simply loses the operation. When the hinted-handoff protocol is enabled and read-repair is disabled, we assume that the write operations are not lost. In this scenario, when losing/removing the write operation from the pending store, the *network* thread inserts the operation as a hint in the hinted-handoff table of the appropriate coordinator. The *permanent_failures* thread does not execute in this case and data in the global data structures is not lost.

Testing Program Models: Once we devised a model of the anti-entropy protocols, we tested it to make sure that it corresponds to actual systems. In our testing model we provide implementations for the stubs that model failure and non-determinism in message arrivals. In particular, for testing we use random coin-tosses instead of non-deterministic choices present in the verification model. Besides this, we also provide concrete implementations for conflict-resolution strategies for operations on CRDTs based on the last write wins (LWW) and vector clocks (MV).

We wrote a test harness that arbitrarily issues put and get operations for various key-value pairs. We then checked if the results of these operations can be realized by the actual eventually consistent key-value stores. We also used CHESS [135], which is a systematic testing tool for concurrent programs, to systematically enumerate all possible thread schedules. Using CHESS we were able to ensure that our model realized strange but possible behaviors of the eventually-consistent stores.

We exhaustively tested a number of possible scenarios. Here, we discuss a configuration with three replicas, where the write consistency level is set to two, and the read consistency level is set to one. One interesting scenario is where the client successfully performs a write operation on a key with a value 0, followed by an unsuccessful write on the same key with a value 1. A subsequent read of the key returns the value 1. This is a nonintuitive scenario, but it can manifest in a real system because failures are not guaranteed to leave the stores unaffected and an unsuccessful write can still write to some of the replicas.

In another scenario, the client successfully performs two consecutive write operations to a key with values 0 and 1. Subsequently, one read returns the value 1, while a subsequent read returns the stale value 0. This behavior can happen in a real system where the client gets staler values over time. In particular, this scenario occurs when the two replicas store the value 1 after the second write operation (remember the write consistency level is two) and the third replica still stores the stale value 0.

Now consider a configuration with three replicas but where both read and write consistency levels are set to one.

In the third scenario we consider the case when the client issues three consecutive successful writes with values 0, 1, and 2. Subsequent reads first return a stale value 1 followed by returning an even more stale value 0. This unusual behavior manifests when the three replicas all store different values (the read requests may access any of them).

Finally, we consider a scenario where there are four consecutive successful writes to a key with values 0, 1, 2, and 3. If the subsequent two reads for the same key return values 2 followed by 1, then a following third read cannot return the value 0. This scenario cannot happen because the three replicas must have values 1, 2, and 3 at the time of the last read.

We used CHESS to confirm the realizability of the first three scenarios, and infeasibility of the last scenario. CHESS took from less than a second to up to 10 minutes to exhaustively explore all interleavings corresponding to these four test harnesses. We were also able to observe some of these scenarios in a real installation of Cassandra.

5.5 Verification of Anti-entropy Protocols

In this section we first describe our verification methodology, followed by our verification of the hinted-handoff and read-repair anti-entropy protocols.

5.5.1 Verification Methodology

Verification process: We use the *deductive verification* style for proving programs correct. For sequential programs, this style is close to Hoare logic style reasoning [92, 19]. It proceeds by the programmer annotating each method with pre/post conditions and annotating loops with loop invariants in order to prove assertions in the program. Furthermore, in order to prove that functions terminate, the user provides *ranking functions* for loops (and recursive calls) that are mappings from states to natural numbers that must strictly decrease with each iteration [166, 19]. Reasoning that these annotations are correct is done *mostly* automatically using calls to constraint solvers (SMT solvers), with very little help from the user.

There are several different approaches to verify concurrent programs, especially for modular verification. We use VCC [60] tool to verify our models. VCC is a verifier for concurrent C programs [4]. The basic approach we take to verify our models is to treat each concurrent thread as a sequential thread for verification purposes, but where every access to a shared variable is preceded and succeeded by a *havoc* that entirely destroys the structures shared with other threads. However, this havoc-ing is guarded by an *invariant* for the global structures that the user provides. Furthermore, we check that whenever a thread changes a global structure, it maintains this global invariant. This

[4] Even though our model and invariants apply to unbounded number of instances, verification of C programs, strictly speaking, assumes integer manipulations to `MAX_INT` (i.e., typically 2^{32} on 32-bit architectures).

approach to verification is similar to *rely-guarantee* reasoning [100], where all threads rely and guarantee to maintain the global invariant on the shared structures.

Specifications: Another key aspect of the verification process is writing the specification. Though the specification is written mainly as assertions and demanding that certain functions terminate, specifications are often described accurately and naturally using *ghost code* [19, 60]. Ghost code is code written purely for verification purposes (it does not get executed) and is written as instructions that manipulate ghost variables. It is syntactically constrained so that real code can never see the ghost state. Hence this ensures that the ghost code cannot affect the real code.

In our framework, we use ghost code to model the taint-based specification for eventual delivery (see Section 5.5.2). It is important that the protocol does not see the tainted write, because we do not want a flow of information between the executable program and the specification. We also use ghost code to maintain mathematical abstractions of concrete data-structures (like the set associated with an array, etc.).

Testing annotations: We extensively used testing, especially in early stages, to assert invariants that we believe held in the system at various points in the code. Prior to verification, which requires strong inductive invariants, testing allowed us to gain confidence in the proof we were building (as well as the model we were constructing). These invariants then were the foundation on which the final proof was built upon.

5.5.2 Verifying the Hinted-handoff Protocol

As explained in Section 5.3, verification that hinted-handoff protocol maintains strong eventual consistency under transient failures and for idempotent operation-based CRDT reduces to verification of eventual delivery (Result#1 in Section 3.1). Recall that, eventual delivery is the property that every successful write eventually gets delivered to every replica at least once.

Taint-based specification of eventual delivery: We model eventual delivery using a ghost field *taint*, that records a particular (exactly one) write operation issued to the coordinator. For a sequence of reads (r) and writes (w) operations, a write for an arbitrary key is designated as tainted:

$$history \xrightarrow{\cdots,\mathbf{r},\mathbf{r},\mathbf{w},\mathbf{w},\mathbf{r},\mathbf{r},\mathbf{r},\mathbf{w},\mathbf{r},\mathbf{r},\mathbf{w},\mathbf{r},\mathbf{w}^{\mathbf{taint}}\cdots}$$

We now assert the specification that this taint will eventually propagate to each replica's local store. Intuitively, the write that was chosen to be tainted will taint the value written, and this taint will persist as the value moves across the network, including when it is stored in the hint store and the pending store, before being written to the local store.

Taints are persistent and will not disappear once they reach the local store. Hence demanding that the local stores eventually get tainted captures the property that the chosen write is eventually delivered at least once to every local store.

Note that the tainted values are ghost fields which the system is agnostic to, and hence proving the above property for an arbitrary write in fact ensures that *all writes* are eventually delivered.

Proving the taint-based specification: To prove the specification we introduce several ghost fields:

(a) *ls_tainted_nodes*, the set of replicas that have updated their local store with the tainted write,

(b) *hs_tainted_nodes*, the set of replicas for which the coordinator has stored the tainted write operation as a hint in its hint store, and

(c) *ps_tainted_nodes*, the set of replicas for which the tainted write has been issued, but its delivery is pending on the network.

We add ghost-code to maintain the semantics of the taint in various methods, including *put*, *network* and *handoff_hint*. Every time any of these methods transfers values, we ensure that the taints also get propagated. At local stores, when a value is written, the value at the local store is tainted if it either already had a tainted value or the new value being written is tainted; otherwise, it remains untainted. (In fact, the taint-based store can itself be seen as an operation based CRDT which never loses the taints.) Furthermore, the ghost fields described above, which are set abstractions of the related stores, are also kept up to date using ghost updates.

For eventual delivery we want to prove that, when all replicas remain available and all the read/write operations have stopped, regardless of how all concurrent operations are scheduled, the tainted write operation is indeed eventually propagated to the local stores of all the replicas.

We model eventual taintedness of stores as a *termination property* by modeling a schedule harness that takes over the scheduler, and arbitrarily schedules *network* and *handoff_hint* threads until the taint has propagated to all replicas. The termination of this harness then proves eventual taintedness, which in turn proves eventual delivery. In the schedule harness, the *permanent_failures* thread is not scheduled since we assume only transient failures can occur.

In order to prove termination of the harness, we specify a (safety) invariant for the scheduler and specify a ranking function for arguing the termination of this method. The invariant for the scheduler loop states that for every replica responsible for the tainted key, either its local store is tainted or there is a tainted write pending in the network for it, or there is a hint in the corresponding coordinator which has a tainted write for it. More precisely, for each replica responsible for the tainted key, we demand that the replica is present in one of the ghost-sets, namely, *ps_tainted_nodes*, *hs_tainted_nodes*, and *ls_tainted_nodes*:

```
_(invariant (\forall int j;
  (j >= 0 && j < PREFLIST_SIZE) ==>
  (ls->tainted_nodes[pl->pref_list[coord+j]]
|| hs->tainted_nodes[pl->pref_list[coord+j]]
|| ps->tainted_nodes[pl->pref_list[coord+j]])))

_(decreases hs->size + 2 * ps->size)
```

Figure 5.4: The invariant and the decreases clause for the scheduler in the hinted-handoff protocol

$$\forall r.(\quad r \in ps\_tainted\_nodes \quad \vee \quad r \in hs\_tainted\_nodes$$
$$\vee \quad r \in ls\_tainted\_nodes)$$

where the quantification is over replicas r responsible for the tainted key. In VCC, this invariant is written as shown in Figure 5.4.

The ranking function for the scheduler is a function that quantifies, approximately, the *time* it would take for the system to reach a consistent state. In our case, the ranking function $|hint\_store| + 2 \cdot |pending\_store|$ suffices. Note that we prove that the rank decreases with the scheduling of any thread, thereby guaranteeing termination. In VCC, Figure 5.4 shows the *decreases* clause that represents this ranking function. Since the *network* thread can remove a write from the pending store and can insert it into the *hint_store*, the factor 2 in the ranking function is necessary.

5.5.3 Verifying the Read-repair Protocol

As explained in Section 5.3, we want to verify that the read-repair protocol maintains eventual consistency in the presence of permanent failures (as stated in Result#2 in Section 3.2). We prove this result both when hinted-handoff is turned on as well as when it is disabled (we capture whether hinted-handoff is enabled/disabled using a macro directive, and prove both versions correct). For simplicity of presentation we only explain here the case when the hinted handoff protocol is disabled.

Recall that permanent failures could: (a) modify the local store by setting them to default values, (b) remove an operation from the pending store, and (c) destroy the hint store.

For eventual consistency we want to prove that when all the write operations have successfully returned to the client, then after only a finite number of read operations on a key, the read-repair mechanism ensures that the set of R available replicas will converge. Note that we want to prove this property when the replicas in R remain available throughout these read operations, but regardless of how all the other operations are scheduled concurrently with read-repair.

When the writes stop and only the read of a particular key occurs (infinitely often), we write a schedule harness that takes over the scheduler at this point, in order to argue that consistency is eventually reached (similar to the verification of the hinted-handoff protocol).

The schedule harness arbitrarily schedules the reads and the repairs, the *network* threads and *permanent_failures*, but restricts it to not modify local stores of replicas in R (since replicas in R cannot fail any longer). The harness has an outer loop that continually issues reads of the key and executes the read-path and the read-repair with interference from other threads modeling the system (*network* and *permanent_failures*). This loop terminates only when convergence is reached, and hence our task is to prove that the loop terminates.

We verify the harness again by specifying safety invariants and a ranking function. The ranking function for the scheduler is that the size of the pending store, $|\ pending\_store\ |$, decreases with every loop.

Intuitively, an unbounded number of read-repairs get executed, and if the network thread does not interfere during the read-repair, then the replicas will reach a consistent state. However, if the network does interfere (delivering pending writes to replicas), then read-repair may not succeed in syncing the replicas in this round, but the size of the pending set must necessarily decrease.

5.5.4 Verification Statistics

We performed the verification on an Intel CORE-i7 laptop with 8 GB of RAM, running Windows 8 and using Visual Studio 2012 with VCC v2.3 as a plugin. Our verification model consists of about 1500 lines of code and annotations, where about 900 lines are executable C code and the rest are annotations (not seen by the C compiler). The annotations comprise ghost code (20%) and invariants (80%) [5]. The total time taken for the verification of the whole model is around a minute. Since the verification is modular, we focus on verifying one function at a time while modeling the protocol. Verification of each function takes around 5 seconds. The verification is hence highly interactive: we add small chunks of executable code and annotations, run the verifier, refine the code or annotations, re-run the verification, and iterate.

5.6 Discussion: Experience and Lessons

We now describe our experience and lessons learned while modeling and verifying the protocols.

Lesson 1: Building models is an iterative process of refinement, and it takes time. We did not build the model in one day. Our effort spanned a period of 15 months, with about 6 person months of effort for modeling and verification. Initially, we built a coarse-grained model that eschewed parallelism of reads/writes in favor of building complete and correct models for the store mechanisms and for failures. Even with this initial coarse-grained model, we came up with the taint-based specification and realized that this specification was actually capturing not eventual consistency but rather eventual delivery (Section 3.1).

[5] The web-site for the project and our code is here: `http://web.engr.illinois.edu/~pek1/cpm/`

We then retrofitted the model with concurrency, which led to a much more complex and lengthy proof that took into account interactions of threads. The concurrency retrofitting required the most effort, as it involved learning the intricacies of our verification platform, VCC, and its concurrency verification model. In fact, when retrofitting concurrency, we found errors in our read-repair specification (see below and see Section 3.2).

Lesson 2: The quest to prove results lead to surprising outcomes after building, writing, and verifying with models. The result that a single read-repair does ensure eventual consistency of the stable live nodes was a result we wrongly believed in before the verification. In fact, in the coarser-grained concurrency model we built for it, we were able to prove the result, but this proof fell through when we retrofitted it with fine-grained concurrency. The inability to establish a global ranking function that ensured consistency was reached in finite time, –this fact led us to disbelieving it and disproving it. This led us to believe that an unbounded number of reads would give eventual consistency, *provided the scheduler fairly scheduled the system*. However, when we proved the property, we realized that a fair scheduler isn't necessary, which was another surprise in itself(Result#2 in Section 3.2).

Lesson 3: Non-obvious and unknown results can arise out of building, writing, and verifying with models. The taint-based modeling of eventual consistency led us to realize that the hinted-handoff protocol actually was ensuring eventual delivery, and would hence work for any CRDT register. Proving this property led us to the transient failure model that is needed for this protocol to ensure eventual delivery. The result that, under transient failures, hinted-handoff ensures strong eventual consistency of *any* idempotent CRDT (Corollary#1, in Section 3.1) was a result that we did not know before, and resulted directly from the abstract proof.

We also realized that CRDTs for *counters* in the literature are not idempotent [155]— these CRDTs assume that messages are delivered *precisely once* (as opposed to at least once), and we realized that systems like Riak and Cassandra do not assure delivery precisely once, even when only transient failures are present. This explained to us the predominance of purely state-based implementation of CRDTs in systems such as Riak [5].

Lesson 4: Systems developers need to be building and writing models either concurrent with or prior to, building the actual system. As a direct consequence of the above Lessons 2 and 3, we conclude that building models can aid systems developers understand the targeted properties of the system being built, with great clarity. The verification experience helped us to understand protocols much better than we had previously.

The building of our model and testing this model was useful in both building a faithful model as well as in understanding it, in expressing assumptions on the failure model, and figuring out the correct specifications. The model gave us a concrete vocabulary (especially data-structures like the pending set, which captures the messages in the network, etc.) to reason more formally even in discussions (similar to the way a formal modeling on paper gives such

a vocabulary). We believe building models can inform the vocabulary that developers use in talking about their system and its internals at the development stage, and holds the potential to minimize bugs arising out of miscommunication.

There is corroborating evidence from Amazon researchers [138, 139] that building such models and verifying them gives additional benefits of understanding protocols, making design decisions, and adding features to protocols. Note that Amazon used TLA$^+$ to model protocols (in [138], they discuss why they chose TLA, and find VCC also met most of their requirements). However, in their work, they did not verify the protocols, but only model-checked them (i.e., systematically tested them) for all interleavings for small instantiations of the system.

Lesson 5: Modeling in a high-level language closer to implementation (like C) is programmer-friendly, testable, and supports verification. We believe that modeling in C offers many advantages (in particular, in comparison with languages such as TLA). First, systems engineers understand C, and the modeling entails building a *simulator* of the system, which is something engineers do commonly for distributed systems anyway. Being executable, the model can be subject to standard testing, where engineers can write test harnesses, tweaking the environment's behavior and fault models, simply by writing code. Third, for systematic testing (model-checking), we have powerful tools such as CHESS that can systematically explore the behavior of the system, exploring non-determinism that arises from interleavings. Fourth and finally, the ability to prove programs using pre/post conditions and invariants, using VCC, gives a *full fledged verification platform* to prove the entire protocol correct, where most reasoning is pushed down to automated logic-solvers.

We advocate certified program models as a sweet-spot for modeling, testing, and verification. It abstracts from the real system, but in doing so captures many instantiations and versions of these systems. And yet is written in code, allowing for easier model-building and testing. Finally, it affords full fledged verification using mostly-automated verification platforms.

5.7 Related Work

The CAP theorem [43, 123] indicates that a distributed system that can tolerate partition failures can either provide strong data consistency (eg., linearizability, sequential consistency) or high availability. Strong consistency guarantees like linearizability can be provided using a single commit point ensured by two-phase/three-phase commit protocols [158] or by distributed consensus (eg., Paxos [113]). As opposed to strong consistency, most existing distributed systems provide only weaker eventual consistency guarantees, which roughly mean that when the updates stop the entire system eventually converges to the same value. There are also known some other models of consistency such as consistent prefix, bounded staleness, monotonic reads, and read-my-writes [163, 12, 45].

Amazon's use of formal techniques [139, 138] for increasing confidence in the correctness of their production systems is in the same spirit as this work. The engineers at Amazon have successfully used TLA+ [112] to formally specify the design of various components of distributed systems. The formal TLA+ specifications are executable like our program models and these design specifications have been systematically explored using a model checker to uncover several subtle bugs in these systems. However, instead of modeling distributed algorithms in TLA+, we model them as C programs. Newcombe et al. [139, 138] acknowledge that modeling systems in a high-level language like C increases the productivity of the engineers. More importantly, in addition to checking the models using model checkers up to a certain trace length, a model written in C lends itself to mostly automated verification using tools like VCC that utilize automated constraint solvers, and that can verify unbounded instances of the system.

Previous attempts at formal modeling of distributed systems for design and verification include the Farsite project [35] and the modeling of Pastry protocol for distributed hash-tables [129] using TLA; the modeling of the Chord ring maintenance protocol using Alloy [178]; the verification of consensus algorithms using Isabelle [51]; and the verification of event-driven device drivers written in the P language [73, 72]. There have also been efforts towards formally modeling key-value stores like Cassandra using Maude [122]. In this work, consistency properties are expressed in Maude using linear temporal logic (LTL) formulae. This model checking approach either is not exhaustive or is exhaustive on bounded instances while ours is exhaustive on unbounded instances.

In "Paxos Made Live" [49], the authors show that directly building a system that implements an algorithm that is as well-studied as the Paxos consensus algorithm is a non-trivial task. In our experience, also as in Amazon's [138, 139], modeling algorithms as programs is a good intermediate step that allows one to quickly prototype the algorithm, understand issues concerning the implementation, explore the design space, and also test and verify the design.

Recently, there has been work on programming languages that ease development of distributed systems, in particular, with respect to consistency properties at the application level [16, 157, 46] and failfree idempotence [148]. Kuru et al. [110] verify properties of transactional programs running under a relaxed scheme that provides snapshot isolation.

In a recent work [171], the authors have used Coq to implement distributed algorithms that are verified. In [47, 45, 38], the authors explore logical mechanisms for specifying and verifying properties over replicated data types. Deductive verification using automatic tools, such as VCC [60] and Dafny [120] has been extensively used for verifying systems in domains other than distributed systems. Some of the examples are: verifying a hypervisor for the isolation property [117], verifying operating systems like Verve [175] and ExpressOS [126] for security, verifying the L4 microkernel for functional correctness [108, 106] and verifying high-level applications for end-to-end security [89].

5.8 Conclusions

In this chapter we have shown how the philosophy of certified program models can be used to verify and find fault-tolerance violations in distributed systems, with a specific focus on key-value/NoSQL storage systems. We have verified both eventual delivery of the hinted-handoff protocol under transient failures (which ensures strong eventual consistency for any store maintained as a CmRDT register) as well as eventual consistency of the read-repair protocol when arbitrary number of reads are issued. We also discovered several surprising counter-examples during the verification for related conjectures, and the experience helped us develop a firm understanding of when and how these protocols guarantee eventual consistency.

Based on our experience, we believe the notion of certified program models is applicable to a broad swathe of distributed systems properties beyond hinted-handoff and read repair. For instance, while we have assumed a CRDT abstraction, some artifacts of the way they are implemented in systems deserve verification— for instance, Riak's use of vector clocks (now called Causal Context) and the associated conflict resolution and pruning mechanisms are worth verifying for correctness, even in the case where there are no failures. The verification of how counters (and in general other CRDTs) work in today's distributed systems in tandem with eventual consistency protocols, and the idempotence guarantees on them, is another worthy target. Beyond distributed systems, properties of file systems are also a good match for certified program models, e.g., works like [54, 15]. In these systems the ordering and consistency of file system operations are decoupled, and verifying that the desired ordering and consistency properties hold, under failures or otherwise, is an interesting future direction.

Chapter 6

Conclusions

This thesis explored program verification for systems software in the setting of automated deductive verification. We showed that deductive verification techniques does not need to require prohibitive levels of effort and expertise. Deductive verification is perhaps the only scalable technique for automated software verification that can provide strong security and reliability guarantees. Moreover, show how deductive verification can be adapted for reasoning about systems software even when distribute nature of such software make reasoning at the code level infeasible. In summary, we have shown the following:

- A secure Android-based operating system (ExpressOS) can be designed and implemented with the help of automated deduction verifier. ExpressOS is close to Verve in terms of goals and effort invested. This point in a design space represents a middle ground between full functional verification (as done with seL4) and unsound but pragmatical approach that identifies violations of common systems code patterns (see e.g., [77]).

- Natural proof technique [147] can be employed for the verification of programs written in a real programming languages. We have developed a framework called VCDRYAD that extends VCC [58] to provide an automated deductive verifier against separation logic specifications for C programs based on natural proofs. We have successfully certified more than 150 programs ranging from standard data structures manipulations, widely used open source libraries (GLib, OpenBSD), Linux kernel routines, customized OS data structures, etc. The verification is automatic – requires only domain specific knowledge and no user-provided proof tactics.

- Runtime checking of assertions on list can be efficiently performed with the help of abstraction. Our approach – abstraction guided runtime checking – maintains accurate abstraction of dynamic heap using the evolving runtime state to significantly reduce the runtime of assertion checking. For frequent assertion checks on large data structures our approach yields only constant time overhead.

- *Certified program models* can be effectively used to show correctness of distributed systems protocols. We use C as a modeling language to simulate the nondeterminism and processes in the distributed system. The program model allows us not only to test the model but to *verify* against correctness properties using program verification

techniques. We employed our technique in verification of distributed key-value store anti-entropy protocols underlying widely used NOSQL databases such as Dynamo [70], Riak [2], and Cassandra [4].

Future directions

One interesting extension of the work describe in Chapter 2 would be to prove all the properties using only automated deduction methods. Recall that we used both abstract interpretation based static checker (via Code Contracts) and automated deduction (via Dafny) to prove the security properties in ExpressOS. A problem with using the abstract interpretation based static checker is that it is not guaranteed to be applicable to a more refined version of the code because if the checker is not able to infer the invariant there is no mechanism to ”help” with the proof. Moreover, Code Contracts static checking was designed pragmatically, so it sometimes sacrifices soundness to ensure lower false positive rate. Even though, we have tried to avoid the annotations that are known to possibly lead to an unsound proof, it makes the whole effort less robust. The tradeoff in applying pure deductive verifier is increase in the number of annotations. However, we believe that most of the annotations inferred by abstract interpretation based checker are not particularly difficult, and would not cause dramatic increase in the number of annotations.

Another important direction is establishing end-to-end security for ExpressOS, through information flow. HiStar tackled the problem of the explicit information flow leaks, however preventing implicit malicious information flow still poses quite a few challenges for the community. Implicit information flows can be formalized using the notion of non-interference [85]. How to effectively apply non-interference techniques together with declassification in the deductive verification setting is an interesting research question. A recent work by Hawblitzel et al. [90] is one of the first works that has shown feasibility of such an approach.

VCDRYAD framework allows for compelling extensions for wider range of data structures. In the systems software setting extending natural proof technique to reason about array properties would be an interesting direction. Specifically [1], we could build upon the work by Cousot et al. [65]. This approach uses array-based analogues of list segments and unrolling of inductive definitions. Thus, the basic idea would be to set-up a ghost code generation, as we have described in chapter 3 for heaps, to obviate the need for proof hints when reasoning about array specifications.

One particularly useful practical extension would extension of the approach described in chapter 4 on wider range of data structures so that it can be integrated with VCDRYAD. The integration of run-time checking with the program verifier would be particularly helpful in debugging code or specifications when the verifier does not succeed in a proof. From our experience, understanding why the verifier does not prove a separation logic specification can be difficult even for an experienced user. Thus, executable specification would increase likelihood of adopting richer specifications during development of systems software.

[1] We thank Peter O’Hearn for suggesting this extension.

Finally, we expect that *certified program models* can be applied to verification of a variety of distributed protocols. The modular nature of the approach would allow one to effectively create models of applications over the proven protocols. As we argued in chapter 5 this approach is scalable – it allows developers to focus on the newly developed ”client” code. Moreover, developers can quickly prototype new decisions, test them and finally certify the model before changing the implementation. We believe that certified program models would lead to a better understanding of often intricate distributed computation and yield more reliable distributed systems' implementations.

References

[1] Amazon: milliseconds means money. http://goo.gl/fs9pZb.

[2] Basho Riak. http://basho.com/riak/.

[3] Call me maybe: Cassandra. https://aphyr.com/posts/294-call-me-maybe-cassandra/.

[4] Cassandra. http://cassandra.apache.org/.

[5] Data Structures in Riak. https://vimeo.com/52414903.

[6] Data Types. http://docs.basho.com/riak/latest/theory/concepts/crdts/.

[7] HBase. http://hbase.apache.org/.

[8] Market research media, NoSQL market forecast 2013-2018. http://www.marketresearchmedia.com/?p=568.

[9] NoSQL Meets Bitcoin and Brings Down Two Exchanges: The Story of Flexcoin and Poloniex. http://hackingdistributed.com/2014/04/06/another-one-bites-the-dust-flexcoin/.

[10] Project Voldemort. http://goo.gl/9uhLoU.

[11] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[12] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, February 2012.

[13] Edward E. Aftandilian, Samuel Z. Guyer, Martin Vechev, and Eran Yahav. Asynchronous assertions. In *OOPSLA '11*, pages 275–288, 2011.

[14] Pieter Agten, Bart Jacobs, and Frank Piessens. Sound modular verification of C code executing in an unverified context. In *POPL'15*, 2015.

[15] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Beyond storage apis: Provable semantics for storage stacks. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[16] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

[17] James P. Anderson. Computer security technology planning study. Technical report, HQ Electronic Systems Division (AFSC), October 1972. ESD-TR-73-51.

[18] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of the 23rd ACM Symposium on Operating System Principles*, Cascais, Portugal, October 2011.

[19] Krzysztof R. Apt. Ten years of hoare's logic: A survey—part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, October 1981.

[20] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998.

[21] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, March 2013.

[22] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[23] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proc. EuroSys '06*, 2006.

[24] Thomas Ball, Brian Hackett, ShuvenduK. Lahiri, Shaz Qadeer, and Julien Vanegue. Towards scalable modular checking of user-defined properties. In GaryT. Leavens, Peter OHearn, and SriramK. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg, 2010.

[25] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05*, pages 364–387, 2006.

[26] Mike Barnett, Manuel Fahndrich, and Francesco Logozzo. Embedded contract languages. In *ACM SAC - OOPS*. Association for Computing Machinery, Inc., March 2010.

[27] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS'04*, pages 49–69, 2005.

[28] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, Santa Barbara, CA, August 1996.

[29] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, pages 97–109, 2004.

[30] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS'05*, pages 52–68, 2005.

[31] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO'05*, pages 115–137, 2006.

[32] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[33] W.R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15:1382–1396, 1989.

[34] Dirk Beyer. Competition on software verification (sv-comp), 2013.

[35] William J. Bolosky, John R. Douceur, and Jon Howell. The farsite project: A retrospective. *SIGOPS Oper. Syst. Rev.*, 41(2):17–26, April 2007.

[36] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *Software Engineering, IEEE Transactions on*, 21(10):785–798, Oct 1995.

[37] Richard Bornat. Proving pointer programs in hoare logic. In *Proceedings of the 5th International Conference on Mathematics of Program Construction*, MPC '00, pages 102–126, London, UK, UK, 2000. Springer-Verlag.

[38] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying eventual consistency of optimistic replication systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 285–296, New York, NY, USA, 2014. ACM.

[39] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[40] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *ISSTA'02*, pages 123–133, 2002.

[41] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI'06*, 2006.

[42] Eric Brewer. A certain freedom: Thoughts on the CAP theorem. In *Proc. ACM PODC*, pages 335–335, 2010.

[43] Eric A. Brewer. Towards robust distributed systems (Invited Talk). In *Proc. ACM PODC*, 2000.

[44] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Reversible debugging software quantify the time and cost saved using reversible debuggers, 2013.

[45] Sebastian Burckhardt. Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150, 2014.

[46] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[47] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. *SIGPLAN Not.*, 49(1):271–284, January 2014.

[48] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, pages 23–50, Edinburgh, 1972. Edinburgh University Press.

[49] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[50] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[51] Bernadette Charron-Bost and Stephan Merz. Formal verification of a Consensus algorithm in the Heard-Of model. *Intl. J. Software and Informatics*, 3(2-3):273–304, 2009.

[52] Brian X. Chen and Nick Bilton. Et tu, google? android apps can also secretly copy photos. `http://bits.blogs.nytimes.com/2012/03/01/android-photos`.

[53] Yoonsik Cheon and Gary T. Leavens. The larch/smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, 1994.

[54] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, page 9, 2012.

[55] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, pages 1006–1036, 2012.

[56] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI '11*, pages 234–245, 2011.

[57] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, pages 25–37, 2006.

[58] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, pages 23–42, 2009.

[59] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, Munich, Germany, August 2009.

[60] Ernie Cohen, MichałMoskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. *Electron. Notes Theor. Comput. Sci.*, pages 85–103, 2009.

[61] Ernie Cohen, Wolfgang J. Paul, and Sabine Schmaltz. Theory of multi core hypervisor verification. In *SOFSEM'13*, pages 1–27, 2013.

[62] Common Vulnerabilities and Exposures (CVE). `http://cve.mitre.org`.

[63] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *POPL'09*, pages 302–314, 2009.

[64] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR'11*, pages 235–249, 2011.

[65] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. *SIGPLAN Not.*, 46(1):105–118, January 2011.

[66] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A safety-oriented platform for web applications. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[67] Michelle L Crane and Juergen Dingel. Runtime conformance checking of objects using Alloy. In *Electronic Notes in Theoretical Computer Science*, volume 89, pages 2–21. Elsevier, 2003.

[68] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Budapest, Hungary, March 2008.

[69] Leonardo Mendonça de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *FMCAD'09*, pages 45–52, 2009.

[70] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP '07*, pages 205–220, 2007.

[71] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[72] Ankush Desai, Pranav Garg, and P. Madhusudan. Natural proofs for asynchronous programs using almost-synchronous reductions. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 709–725, New York, NY, USA, 2014. ACM.

[73] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 321–332, New York, NY, USA, 2013. ACM.

[74] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[75] Dino Distefano and Matthew J. Parkinson J. jStar: Towards practical verification for Java. In *OOPSLA '08*, pages 213–226, 2008.

[76] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, Canada, October 2010.

[77] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[78] Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *Proceedings of the 1979 National Computer Conference*, New York City, NY, June 1979.

[79] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME '01*, pages 500–517, 2001.

[80] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[81] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[82] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV'09*, pages 306–320, 2009.

[83] Seth Gilbert and Nancy A. Lynch. Perspectives on the CAP theorem. *IEEE Computer*, 45(2):30–36, 2012.

[84] `https://developer.gnome.org/glib/`, 2015.

[85] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20, 1982.

[86] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX operating system. In *Proceedings of the USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.

[87] Christoph Haase, Samin Ishtiaq, Joël Ouaknine, and Matthew J. Parkinson. SeLoger: A tool for graph-based reasoning in separation logic. In *CAV'13*, pages 790–795, 2013.

[88] John Hatcliff et al. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.

[89] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 165–181. USENIX Association, 2014.

[90] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, Broomfield, CO, October 2014. USENIX Association.

[91] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[92] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[93] C. A. R. Hoare. Assertions: A personal perspective. *IEEE Ann. Hist. Comput.*, pages 14–25, 2003.

[94] C.A.R. Hoare, Jayadev Misra, Gary T. Leavens, and Natarajan Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41(4):22:1–22:8, October 2009.

[95] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, Glasgow, UK, July 2005.

[96] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[97] Radu Iosif, Adam Rogalewicz, and Jirí Simácek. The tree width of separation logic with recursive definitions. In *CADE-24*, pages 21–38, 2013.

[98] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *CAV*, pages 756–772, 2013.

[99] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: a powerful, sound, predictable, fast verifier for C and Java. In *NFM'11*, pages 41–55, 2011.

[100] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[101] Maria Jump and Kathryn S. McKinley. Dynamic shape analysis via degree metrics. In *ISMM '09*, pages 119–128, 2009.

[102] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[103] Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. Type-based data structure verification. In *PLDI'09*, pages 304–315, 2009.

[104] Gregg Keizer. Google pulls 22 more malicious android apps from market. `http://www.computerworld.com/s/article/9222595/Google_pulls_22_more_malicious_Android_apps_from_Market`.

[105] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[106] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[107] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, Big Sky, MT, October 2009.

[108] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[109] Emmanouil Koukoutos and Viktor Kuncak. Checking data structure properties orders of magnitude faster. In *RV'14*, pages 263–268, 2014.

[110] Ismail Kuru, Burcu Kulahcioglu Ozkan, Suha Orhun Mutluergil, Serdar Tasiran, Tayfun Elmas, and Ernie Cohen. Verifying programs under snapshot isolation and similar relaxed consistency models. In *9th ACM SIGPLAN Workshop on Transactional Computing*, 2014.

[111] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS OSR*, 44(2):35–40, 2010.

[112] Leslie Lamport. The TLA Home Page. `http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html`.

[113] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[114] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4Android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, Chicago, IL, October 2011.

[115] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. The Boogie verification debugger. In *SEFM'11*, pages 407–414, 2011.

[116] GaryT. Leavens, AlbertL. Baker, and Clyde Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, volume 523, pages 175–188. Springer US, 1999.

[117] Dirk Leinenbach and Thomas Santen. Verifying the microsoft hyper-v hypervisor with vcc. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 806–809, Berlin, Heidelberg, 2009. Springer-Verlag.

[118] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, pages 348–370, 2010.

[119] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, Dakar, Senegal, April 2010.

[120] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[121] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

[122] Si Liu, MuntasirRaihan Rahman, Stephen Skeirik, Indranil Gupta, and Jos Meseguer. Formal modeling and analysis of cassandra in maude. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering*, volume 8829 of *Lecture Notes in Computer Science*, pages 332–347. Springer International Publishing, 2014.

[123] Nancy Lynch and Seth Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[124] Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive proofs for inductive tree data-structures. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Philadelphia, PA, January 2012.

[125] Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL'12*, pages 123–136, 2012.

[126] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in ExpressOS. In *ASPLOS '13*, pages 293–304, 2013.

[127] Muhammad Zubair Malik, Aman Pervaiz, Engin Uzuncaova, and Sarfraz Khurshid. Deryaft: A tool for generating representation invariants of structurally complex data. In *ICSE '08*, pages 859–862, 2008.

[128] Roman Manevich, Eran Yahav, Ganesan Ramalingam, and Shmuel Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI'05*, pages 181–198, 2005.

[129] Stephan Merz, Tianxiang Lu, and Christoph Weidenbach. Towards Verification of the Pastry Protocol using TLA+. In R. Bruni and J. Dingel, editors, *31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, volume 6722 of *FMOODS/FORTE 2011*, Reykjavik, Iceland, June 2011.

[130] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[131] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.

[132] Microsoft. Windows driver kit (wdk), 2015.

[133] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI'01*, June 2001.

[134] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, 2007.

[135] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, 2008.

[136] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, Saint Malo, France, October 1997.

[137] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI'11*, PLDI '11, pages 556–566, 2011.

[138] Chris Newcombe. Why amazon chose TLA +. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, pages 25–39, 2014.

[139] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015.

[140] Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. Runtime checking for separation logic. In *VMCAI'08*, pages 203–217, 2008.

[141] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL'01*, pages 1–19, 2001.

[142] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *PLDI '14*, pages 440–451, 2014.

[143] T. Perrine, J. Codd, and B. Hardy. An overview of the kernelized secure operating system (KSOS). In *Proceedings of the 7th DoD/NBS Computer Security Initiative Conference*, Gaithersburg, MD, September 1984.

[144] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. Software verification with VeriFast: Industrial case studies. *Sci. Comput. Program.*, 82:77–97, 2014.

[145] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using smt. In *CAV'13*, 2013.

[146] Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. Unpublished manuscript, 2012.

[147] Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. In *PLDI '13*, pages 231–242, 2013.

[148] Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 249–262, New York, NY, USA, 2013. ACM.

[149] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002.

[150] Patrick Maxim Rondon. *Liquid Types*. PhD thesis, UCSD, 2012.

[151] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[152] Ajeet Shankar and Rastislav Bodík. Ditto: Automatic incrementalization of data structure invariant checks (in Java). In *PLDI '07*, pages 310–319, 2007.

[153] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, Charleston, SC, December 1999.

[154] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, January 2011.

[155] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, 2011.

[156] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011.

[157] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *PLDI 2015, Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation*, Portland, OR, USA, June 15–17, 2015.

[158] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Eng.*, 9(3):219–228, May 1983.

[159] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL'10*, pages 199–210, 2010.

[160] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS'11*, pages 298–315, 2011.

[161] Jakub Szefer and Ruby B. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, London, England, UK, March 2012.

[162] Shuo Tang, Haohui Mai, and Samuel T. King. Trust and protection in the Illinois browser operating system. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, Canada, October 2010.

[163] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, December 2013.

[164] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of 38th International Symposium on Computer Architecture*, San Jose, CA, June 2011.

[165] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Nice, France, January 2007.

[166] A. Turing. The early british computer conferences. chapter Checking a Large Routine, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.

[167] Martin Vechev, Eran Yahav, and Greta Yorsh. Phalanx: Parallel checking of expressive heap assertions. In *ISMM '10*, pages 41–50, 2010.

[168] Werner Vogels. Eventually consistent. *ACM CACM*, pages 40–44, 2009.

[169] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM symposium on Operating Systems Principles*, Asheville, NC, December 1993.

[170] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, February 1980.

[171] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed system. In *PLDI 2015, Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation*, Portland, OR, USA, June 15–17, 2015.

[172] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–23, September 1990.

[173] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 2010.

[174] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *PLDI'10*, pages 99–110, 2010.

[175] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. *Commun. ACM*, 54(12):123–131, 2011.

[176] Dachuan Yu, Nadeem Abdul Hamid, and Zhong Shao. Building certified libraries for PCC: dynamic storage allocation. *Sci. Comput. Program.*, 50(1-3):101–127, 2004.

[177] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A stackable vnode level encryption file system. Technical report, Columbia University, June 1998. CUCS-021-98.

[178] Pamela Zave. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, March 2012.

[179] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.