Technical Report

UCAM-CL-TR-775
ISSN 1476-2986

Number 775

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Creating high-performance, statically type-safe network applications

Anil Madhavapeddy

March 2010

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 2006 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Robinson College.

ISSN 1476-2986

Abstract

A typical Internet server finds itself in the middle of a virtual battleground, under constant threat from worms, viruses and other malware seeking to subvert the original intentions of the programmer. In particular, critical Internet servers such as OpenSSH, BIND and Sendmail have had numerous security issues ranging from low-level buffer overflows to subtle protocol logic errors. These problems have cost billions of dollars as the growth of the Internet exposes increasing numbers of computers to electronic malware. Despite the decades of research on techniques such as model-checking, type-safety and other forms of formal analysis, the vast majority of server implementations continue to be written unsafely and informally in C/C++.

In this dissertation we propose an architecture for constructing new implementations of standard Internet protocols which integrates mature formal methods not currently used in deployed servers: (*i*) static type systems from the ML family of functional languages; (*ii*) model checking to verify safety properties exhaustively about aspects of the servers; and (*iii*) generative meta-programming to express high-level constraints for the domain-specific tasks of packet parsing and constructing non-deterministic state machines. Our architecture—dubbed MELANGE—is based on Objective Caml and contributes two domain-specific languages: (*i*) the Meta Packet Language (MPL), a data description language used to describe the wire format of a protocol and output statically type-safe code to handle network traffic using high-level functional data structures; and (*ii*) the Statecall Policy Language (SPL) for constructing non-deterministic finite state automata which are embedded into applications and dynamically enforced, or translated into PROMELA and statically model-checked.

Our research emphasises the importance of delivering efficient, portable code which is feasible to deploy across the Internet. We implemented two complex protocols—SSH and DNS—to verify our claims, and our evaluation shows that they perform faster than their standard counterparts OpenSSH and BIND, in addition to providing static guarantees against some classes of errors that are currently a major source of security problems.

Contents

CHAPTER 1

Introduction

"Oh wait, you're serious. Let me laugh even harder."
BENDER THE ROBOT (FUTURAMA)

The last half-century has seen the growth of the Internet: a global computer network connecting hundreds of millions of computers. Global culture has been transformed by e-mail and the Web, and the value of electronic commerce has grown to hundreds of billions of dollars [153]. However, this interconnectivity has brought its own share of problems with it. An application exposed to the Internet finds itself under constant threat from malware seeking to subvert it and take control. In particular, servers providing critical Internet infrastructure have been found to have numerous security vulnerabilities over recent years, costing millions of dollars in recovery costs [266] and denting global consumer confidence in the Internet [183].

In this dissertation, we argue the thesis that:

> Applications which communicate via standard Internet protocols must be rewritten to take advantage of developments in formal methods to increase their security and reliability, but still be high-performance, portable and practical in order to make continued deployment on the Internet feasible.

For the remainder of this introduction we justify the importance of this thesis, starting with the rapid rate of Internet growth (§1.1), the security and reliability concerns clouding the modern Internet (§1.1.1), how conventional network defences have been insufficient to allay the problems (§1.1.2), and the software monoculture that has developed around critical Internet infrastructure (§1.1.3). We then present the argument for software reconstruction (§1.2), and finally define our contributions and the structure of the remainder of this dissertation (§1.3).

1.1 Internet Growth

In his 2003 analysis of Internet growth [217], Andrew Odlyzko notes that "*Internet traffic continues to grow vigorously, approximately doubling each year, as it has done every year since 1997*". The types of hosts connected are also changing—from computers on fixed links to mobile personal devices such as laptops, PDAs or mobile phones. The trend towards mobility has led to a surge in "wireless hotspots" where high-bandwidth connectivity is available for laptop computers in metropolitan areas. GSM and third generation (3G) mobile networks offer roaming Internet connectivity almost anywhere in the world. Broadband uptake at home has increased, recently surpassing modem usage in the United Kingdom [26]. Consumers have taken advantage of this improved connectivity by spending an increasing amount of time and money on-line. Forrester Research notes that online retail sales[1] will grow from $172 billion in 2005 to $329 billion in 2010, with an expected compound annual growth rate of 14%.

Remarkably, the Internet has sustained this growth while still remaining a decentralised, globally-accessible body, consisting of many industrial, academic, domestic and national networks. Hosts and networks communicate with each other via openly specified protocols, freely available from the not-for-profit Internet Society (ISOC). Development of new protocols is typically a community process centred around working groups in the Internet Engineering Task Force (IETF).

The end-to-end principle [239] states that whenever possible, communication protocol state should occur at the end-points of a communications system. This is central to the design philosophy of the Internet, which places much of the complexity of higher-level protocols inside the software stacks running on operating systems, and requires a relatively simple core network which can route datagrams. This has permited rapid experimentation with new protocol designs (such as Jacobson's famous congestion control algorithm [149]) without requiring the replacement of established network infrastructure. A downside to this approach is the extra complexity imposed on host software, which has led to security and reliability problems described in the next section.

1.1.1 Security and Reliability Concerns

The rapid evolution of the Internet has led to some growing pains, particularly in the areas of security and reliability (§2.1). In the early days, networks and hosts were largely academic organisations which trusted each other. As commercial and domestic interest grew in the fledgling network, malicious attacks and electronic crime began to necessitate additional security measures. Rather than rewriting existing applications, additional layers of cryptography were introduced in 1994 to encapsulate the traffic such as SNP [286] and the now-ubiquitous Secure Sockets Layer (SSL) [93]. This approach typifies the evolutionary methodology followed by the IETF, which rarely radically changes protocols and encourages post-hoc, experience- and deployment-driven specifications of them (§2.1.1).

The phenomenon of viruses and worms has been one of the biggest causes for concern on the modern Internet (§2.1.3). Worms are self-propagating code—often malicious—which use the Internet as a transmission medium to look for hosts running vulnerable software with security holes due to poorly written software (§2.1.2) and "infect" them. The Sapphire Worm was one of the fastest in history; it infected 90% of vulnerable hosts on the Internet within *10 minutes* [210], and caused disruption in airlines, banking and even nuclear power plants [228].

1.1.2 Firewalls Prove Insufficient

As the levels of malware on the Internet grew, so-called *firewalls* were deployed to apply security polices to data traffic passing between networks. Firewalls operate on different levels, from the low-level inspection of packets to high-level application-level protection [245]. However, firewalls are increasingly easy to bypass; popular applications such as Skype [253] specifically

[1]Online retail sales are defined as business-to-consumer sales of goods including auctions and travel.

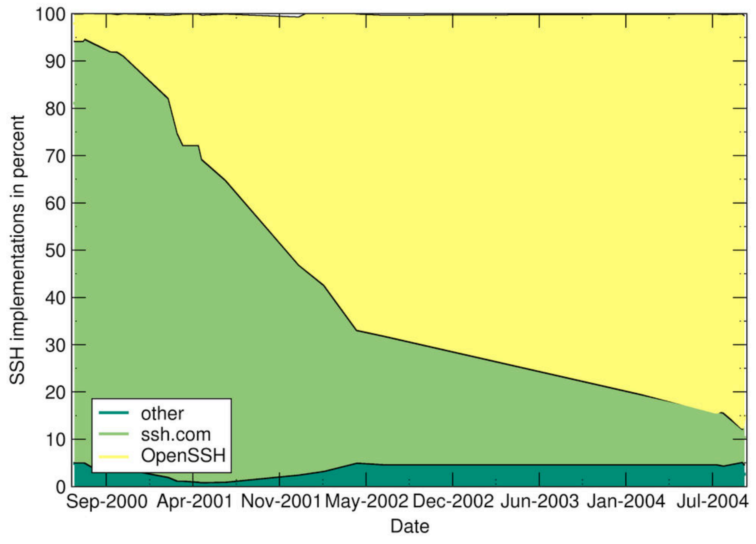


Figure 1.1: Breakdown of SSH servers on the Internet *(source: www.openssh.com)*

tunnel past them to provide a more consistent user experience. The increase of web services has led to a "port 80" culture of tunnelling traffic through the well-known HTTP port, negating much of the benefit of simple packet-level filters.

The insecurity of host software is also increasing, as the number of vulnerabilities and incidents reported continue to grow yearly (§2.1.3). The insecurity is generally not due to fundamental deficiencies in the network protocols used to communicate (although this does also happen), but rather due to errors in the *implementation* of the protocol. Traditionally, applications have been written using low-level systems languages such as C or C++ which can allow bugs to propagate with serious consequences—particularly in network applications where remote attackers can often end up taking over complete control of a host due to these software errors (§2.1.2).

1.1.3 The Internet Server Monoculture

A remarkable number of Internet services are based around a software monoculture; typically due to a reference implementation of the protocol which is integrated into widely-deployed operating system distributions. Common examples include:

HTTP: Apache [10] is deployed on over 70% of Internet web servers, and when combined with Microsoft Internet Information Server (ISS), consists of over 92% of the market.

DNS: BIND [4] serves over 70% of DNS second-level .com domains, according to Bernstein's 2002 survey [32] and later confirmed by Moore in 2004 [209].

SSH: OpenSSH [262] powers nearly 90% of SSH servers, as recorded by Provos and Honeymoon [231] and illustrated in Figure 1.1. These servers range from general-purpose computers to Cisco, Nokia and Alcatel routers [263].

SMTP: In Credentia's 2003 survey of 21258 random e-mail servers [84], over 90% of them are written using C or C++. The main contender is Sendmail [249] with a 38% share among the e-mail servers.

A lack of diversity is well known to be dangerous to a large network [137], and each of the implementations described above has had a steady history of serious security flaws which have allowed attackers to take complete control of hosts from across the Internet. Alternative software has emerged in response to these insecurities, notably Dan Bernstein's `qmail` (SMTP) and `djbdns` (DNS) [33]. These alternatives have been much more robust but are still written in C, and thus very hard for other parties to modify without the risk of creating security holes. There have been no large-scale deployments of infrastructure servers constructed in alternative languages to C/C++. This is largely due to the unique blend of flexibility, performance, and portability enjoyed by C due to its adoption as the *de facto* systems programming language, and the ready availability of free tool-chains to compile C code (e.g. `gcc`).

1.2 Motivation for Rewriting Internet Servers

In our thesis we state the importance of constructing *new* implementations of Internet applications, instead of simply improving existing software. In this section we explain the reasons for this argument.

Systems research has long been concerned with the preservation of compatibility with existing code, especially conformance with specifications such as POSIX. This has driven much of the research into *containment*, which seeks to protect the operating system from unsafe code [23, 164, 229, 80]. Another popular alternative is code evolution and re-factoring, such as Cyclone [152] or Ivy [48] which provides a migration path away from existing C code. However, we argue that this compatibility with existing code is not essential for Internet applications, due to the ready availability of RFCs which specify the precise communication mechanisms between hosts (§2.1.1). Hence the restriction of our reconstruction thesis to the domain of Internet applications and not the general domain of code found in the wider world where the only specifications are often the applications themselves.

Research into formal methods has made great advances in recent decades, with the development of functional languages which provide a means to write expressive, elegant and safe code (§2.2) and software model checkers which can exhaustively and efficiently verify safety properties about abstract models of complex systems (§2.4). In particular, we use the Objective Caml language (§2.3) which combines elements of imperative, functional and object-oriented styles in a statically type-safe language, while retaining the portability and high-performance code output so prized by C programmers. We argue that authoring applications which leverage these techniques is a better approach than laboriously mapping existing code—which was not designed with these high-level abstractions in mind—into them.

To date, functional languages have had little impact on the Internet, instead being popular in research circles to solve academic problems. The pioneering FoxNet [38] project was an attempt to break out of this state of affairs by demonstrating that a functional language could elegantly express network protocol abstractions. FoxNet succeeded in demonstrating this by constructing a modular TCP/IP stack, but was short of the performance required to make it a serious replacement for existing software stacks written in C. Other projects such as the Ensemble distributed communication toolkit [131, 271] have shifted to using functional languages with great success (discussed further in §3.3.3).

Many systems researchers have discarded the notion of using high-level languages to rewrite servers due to the perceived performance hit. For example, in their work on dynamic information flow tracking [258], Suh et al. claim that "*[..] safe languages are often less flexible and result in slower code compared to C*". Similarly Qin et al. note in their work on memory corruption detection [232] that "*[..] type-safe languages typically introduce significant overhead, and do not allowed fine-grained manipulation of data structures*". High-level languages undoubtedly do introduce additional overhead in return for safe execution of code, but it is far from clear that this overhead will result in significant observable performance loss for carefully constructed network applications. Our research focusses especially on static type safety which further places the burden of authoring correct code on the developer of the original application, and not on the operating system to enforce dynamically.

1.3 Contributions

The primary observation this dissertation makes is to highlight the importance of a strong distinction between a "data plane" and "control plane" when constructing network applications in a high-level language. This abstraction has been used in the construction of high-performance network routers for many decades, normally for low-level protocols such as IPv4 [125]. Our work demonstrates that the distinction holds even for complex software network applications such as SSH or DNS servers, and that with suitable tool-chain support there is no necessary *intrinsic* performance cost to using statically type-safe languages such as OCaml. We claim the following specific contributions:

Meta Packet Language (MPL): A data-description language and compiler for Internet Protocols, analogous to `yacc` for language grammars, that outputs code to transmit and receive network packets in a type-safe fashion with minimal overhead and data copying. MPL is the first data description language to output high-performance, statically type-safe ML code and the associated interfaces to parse packets. The language also has custom parsing actions which permits a greatly simpler core grammar for typical Internet protocols than alternative packet parsing languages.

Statecall Policy Language (SPL): A language and compiler which describes program-defined state-machines using an intuitive, imperative syntax, and outputs both PROMELA code for model-checking and ML code which is linked with the application code to dynamically enforce the state-machine. Most current uses of model-checking rely on extracting models from existing source code, which makes the maintenance of high-level constraints against changing source code a complex task. SPL is novel in that it permits developers to author both complex ML source code and simpler non-deterministic state machines (e.g. by reading RFCs) which can be efficiently dynamically enforced against the main ML server. A normal testing cycle reveals common bugs, and rarer errors not caught during testing result in dynamic termination of the server rather than potential security violations.

The MELANGE Architecture: We combine MPL and SPL into a practical architecture for constructing complete, statically type-safe network applications in OCaml, and demonstrate its feasibility by detailing our implementations of the SSH and DNS protocols which have equal or better performance and latency characteristics than their standard alternatives written in C. The implementations are available under a BSD-style open source code license at `http://melange.recoil.org/` to ensure that the ideas described in this dissertation can continue to be developed.

We continue this dissertation in Chapter 2 with the necessary technical background in formal methods such as functional languages (in particular OCaml) and model checking, as well as justifying our statement that "*Applications which communicate via standard Internet protocols must be rewritten* . . ." by examining the past and current Internet security situation. In Chapter 3 we examine the large body of related work in the area of constructing network application software. Chapter 4 qualifies the next portion of our thesis statement that "*... must be rewritten to take advantage of developments in formal methods ...*" by deciding on a set of design goals, a concrete system architecture and threat model we protect against. Chapters 5 and 6 define our two domain-specific languages MPL and SPL which enable our architecture to be "*... high-performance, portable and practical ...*". Finally we evaluate two complex network applications (SSH and DNS servers) in Chapter 7 to confirm our assertions about the high performance and stable latency characteristics of applications written in our architecture.

CHAPTER 2

Background

Well-typed programs never go wrong.
ROBIN MILNER

This chapter provides the technical background on the concepts used in the rest of this dissertation. We being by discussing the current state of Internet security (§2.1), the area of functional languages which promotes a safer and higher-level programming style than the currently dominant C (§2.2), the Objective Caml language which we use extensively through this dissertation (§2.3), and finally the technique of model checking to exhaustively verify properties about an abstract model of reactive systems (§2.4).

2.1 Internet Security

The Internet has had a poor security record in recent years and the exploitation of software errors on hosts has resulted in millions of dollars of damage to individuals and businesses. In this section we first describe the history of the network (§2.1.1), the prevalence of applications written in unsafe languages (§2.1.2), the proliferation of networked malware (§2.1.3) and finally the current situation with defending against these attacks (§2.1.4).

2.1.1 History

In the 1960s the US Department of Defence research agency DARPA funded ARPANET, a pioneering effort that resulted in the world's first operational *packet switching* network. Previously data communications was based upon the idea of *circuit switching*, which required the network to dedicate resources (or a "circuit") for each call, and only allowed point-to-point communication between parties. After the success of ARPANET, Robert Kahn and Vint Cerf developed the first two protocols for the fledgling Internet: IP, an unreliable, best-effort, datagram protocol [224] and TCP, a stream-based protocol which makes reliability and in-order guarantees to the receiver [225].

Packet switching (nowadays the dominant basis for data communications) takes advantage of the memory and computation resources available to networked host machines. A packet switched network is a best-effort relay for simple data packets which are routed independently and multiplexed over a single communication channel. The end hosts re-assemble the data

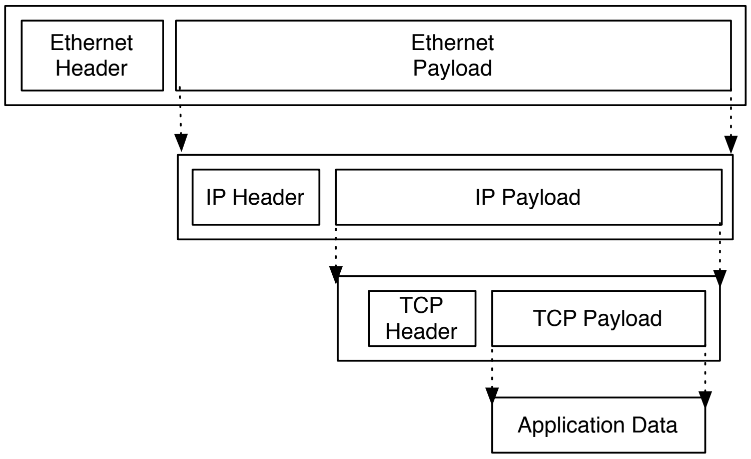


Figure 2.1: Illustrating how packet payloads are embedded in TCP/IP

packets into the original message—this requires more complexity than the circuit switching model, but permits far more flexibility since many protocol changes are possible by simply modifying the software running on end-hosts.

The Internet protocols can be seen as analogous with the OSI model[1]; although the complete OSI specification is widely considered too complicated to be practically implemented, the concept of protocol layers is useful to describe various portions of the Internet protocol suite. Figure 2.1 shows how a payload is embedded in a typical TCP/IP packet running over an Ethernet link layer. Each of the protocol stacks typically consists of a packet header and a variable length payload determined from the header. The host parsing the network traffic must inspect the packet header, classify the payload according to some header fields, and repeat until the application data has all been retrieved.

Request for Comments

Since the development of IP and TCP, the Internet community has developed a number of other protocols to solve various network- and application-level problems. In 1986, the regular meetings of the government-funded researchers were formed into the Internet Engineering Task Force (IETF). The IETF is a mostly-volunteer organisation responsible for the development and promotion of Internet standards, and there are no formal membership requirements. As it grew rapidly, there was a pressing need for a more formal corporate structure to manage financial and legal issues. In 1992, the Internet Society (or ISOC) was formed as a non-profit educational organisation dedicated to the promotion of Internet use and access. At the same time, a committee was appointed to oversee the technical and engineering development of the Internet, known as the Internet Architecture Board (IAB).

Internet development occurs primarily through the Request For Comments (RFCs) process; a series of numbered documents copyrighted and published by the ISOC, and freely available for use by anyone. All of the basic protocols such as IP and TCP are specified in RFCs, as well as more experimental protocols, informational notes, or best current practices. RFCs are never depublished; rather they are superseded by a new publication which marks the previous as obsolete or acts as errata. The "Internet Standard" (or STD) series of documents are reg-

[1]The OSI model is a 7-level representation of network stacks documented in "The Basic Reference Model for Open Systems Interconnection", published as a standard in ISO 7498 and CCITT X.200

ularly re-published with the latest RFCs for their respective protocols. The RFC process is notably different from more established standards organisations such as ANSI or the IEEE. The IETF encourages a more pragmatic, experience-driven and *post-hoc* standardisation of protocols, in recognition of the value that wide-scale Internet deployment brings to building robust systems [44]. However, this approach poses some challenges to protocol implementors.

Firstly, RFCs are written in English, with special keywords such as “MUST” or “SHOULD” indicating the level of importance of statements [45]. The IETF places guidelines on the use of formal languages to specify protocols [147], requiring that (*i*) the formal language itself be specified according to IETF standards [44]; (*ii*) the language must be used as specified (i.e. not pseudo-code); (*iii*) the specification must be complete and verifiable using automated tools; and (*iv*) the specification must be reasonably architecture independent (e.g. not depending on the size of integers or character set in use). Because of these restrictions, and the general lack of a widely-accepted language for protocol formalisation, the majority of RFCs only specify protocols informally in English. This makes it difficult to verify that a protocol implementation is compliant without extensive and ongoing interoperability testing, either in controlled conditions or “in the wild” on the Internet.

Secondly, although the RFCs conform to a general structure, the actual method of specification is entirely up to the individuals in the working group that authored the RFC. Thus a developer implementing, for example, a protocol pretty-printer across a wide variety of protocols must cope with numerous styles and a large bodies of text to extract the information that they need.

Finally, the observation that RFCs are never de-published is matched by the fact that old implementations also do not simply disappear. As protocols evolve, the older implementations will still attempt to communicate, and it is generally important to support as many of these older versions as is practical. Jon Postel, the first RFC Editor, stated in RFC793 [225] (the TCP specification): “*Be conservative in what you do, be liberal in what you accept from others*.” The problem with being liberal in accepting network traffic is, of course, ensuring that supporting the extra complexity does not introduce new security flaws in applications.

2.1.2 Language Issues

The Internet was not originally designed to be a highly secure network; instead, the first links were between trusted institutions and hosts. As it rapidly grew in scale, security issues began to emerge, ranging from protocol-level problems such as lack of encryption or strong authentication, to application-level issues in the implementations of network software running on the hosts. In this section, we focus on the programming language issues and the growth of viruses and worms as a result.

Most of the software running on hosts and routers connected to the Internet is currently written in the C language. C is an imperative programming language originally developed in the 1970s by Ken Thompson and Dennis Ritchie for use on the fledgling UNIX operating system [166]. Since then C has been ported to almost every general-purpose processor architecture in existence and is one of the most widely used programming languages in the world. Due to this it is often referred to as “portable assembly language” since it allows programmers to compile programs on different processor architectures without knowing the underlying assembly language.

Although this flexibility and efficiency has made C popular, it also exposes programs to serious security and stability programs if they are not carefully designed. For example, consider this simple example of a C program which accepts a single command-line argument, and echoes it back out to the standard terminal output.

```
1  int main (int argc, char **argv)
2  {
3    char buf[64];
4    if (argc < 2) {
5      fprintf(stderr, "Usage:␣%s␣<string>\n", argv[0]);
6      return 1;
7    }
8    strcpy (buf, argv[1]);
9    printf("%s\n", buf);
10   return 0;
11 }
```

In line 3, we allocate a 64-byte buffer called `buf` on the stack. After checking that an argument has been supplied, in line 8 we copy the contents of the argument into `buf` and then print it out again. This program will appear to work fine as long as the user only supplies a command-line argument which is less than 64 bytes long. If a longer argument is supplied, then the *strcpy(3)* function on line 8 will cause a *buffer overflow* as it copies the additional data past the buffer, overwriting internal program information on the stack and causing corruption. Problems can be more subtle than just buffer overflows:

```
1  char *bufread (int len)
2  {
3    char *buf;
4    int i;
5    if (len > 1024)
6      errx(1, "length␣too␣large");
7    buf = malloc(len);
8    read (fd, buf, len);
9    return buf;
10 }
```

At first glance, this code appears to be a safe way of reading in a network buffer by allocating up to 1024 bytes of memory, reading into that memory, and then returning a pointer to the new memory. However, the function accepts a signed integer as its input (i.e. it can be negative), while *malloc(3)* accepts an unsigned integer (i.e. only a positive number) as input. This means that if a negative number is passed to this function, the check on line 5 will pass as it is indeed less than 1024, but the subsequent conversion of the number to an unsigned integer on line 7 results in it becoming a very large value, potentially causing the machine to allocate gigabytes of memory and attempting to read into it. These are called *integer overflows* and can result in security vulnerabilities (§2.1.4).

2.1.3 The Rise of the Worm

Given the immense and ongoing popularity of languages such as C in the present day, it is easy to imagine that the programming errors described above are merely academic, and do not pose a threat to the safety of hosts connected to the Internet. Unfortunately, this is not the case; consider a buffer overflow present in a web server listening to traffic from the Internet. By crafting appropriate traffic, a malicious attacker could overwrite data on the server that would allow them to execute arbitrary code on the machine, resulting in a security breach. Although the mechanisms for doing this are highly architecture and operating system dependent [5], numerous so-called "root-kits" are now available that make the process much more automated [221].

Takanen et al. have summarised the literature on buffer overflows in their survey paper [261].

In 1988, Robert Morris wrote a simple program which was intended to gauge the size of the Internet at the time. It exploited a buffer overflow in servers (or "daemons") present in the BSD UNIX distribution—in widespread use at the time—to execute code on a remote host. The program was intended to be self-replicating; when it invaded a host, it would use that host as a base to infect further targets. The mistake that the author made was to insert a degree of randomisation into the program which would cause it to attack a machine even if the target reported that it was already infected (in order to prevent system administrators from "innoculating" their systems by claiming they were already infected). Unfortunately, this aggressive approach turned the worm from a potentially harmless exercise into one of the first *denial of service* attacks on the Internet, as network bandwidth was overloaded by hosts attempting to send the program's traffic to each other [255].

This class of self-replicating programs was termed a "computer worm" [281], and has since become one of the major security threats on the Internet. In recent years, hundreds of millions of hosts have connected to the Internet, and adoption of high-speed broadband access has been on the increase. Many of these hosts are home or office users running Microsoft Windows [199] which has been been demonstrated to have buffer or integer overflows in every version released to date. Staniford, Paxson and Weaver analyzed the danger posed by so-called *flash worms* [256] to take over millions of hosts on the Internet, and concluded that they could potentially infect all vulnerable sites with high-bandwidth links to the Internet in less than thirty seconds!

After the attack by the Morris worm in 1988 revealed just how susceptible the Internet was to being attacked through exploiting software errors, DARPA established CERT/CC [60], an organisation dedicated to tracking security emergencies and co-ordinating responses among vendors and network providers to deal with the problems. As part of its duties, CERT maintains detailed statistics of serious vulnerabilities reported on the Internet which have resulted in exploitation "in the wild". Since its inception, CERT/CC has received over 19,600 vulnerability reports, and dealt with over 315,000 incidents, some involving thousands of hosts. These vulnerabilities are summarised in the CERT Knowledge Base [61], and assigned various metrics such as their severity, impact, references, and any solutions or workarounds known. Every vulnerability is assigned a "vulnerability id" (VU#) which uniquely tracks that particular security issue.

Figure 2.2 shows how rapidly attacks against Internet connected systems have increased over recent years. The Distributed Intrusion Detection System (DShield) [96] tracks virus activity over the Internet, and reports on the "most probed ports" across its sensor system. The results for one day in December 2005, summarised in Table 2.1, show that 9 out the 10 top probes were attempts to exploit buffer overflows in the services involved (the exception at number 8 were HTTP probes, which are a broader class of attacks not covered in this thesis [245]).

2.1.4 Defences Against Internet Attacks

The increasing insecurity of the Internet (illustrated in Figure 2.2) has led to a lot of research focusing on finding effective solutions. A common assumption is that host operating systems and software will always have bugs and thus network-level approaches are required to contain worms. This approach is proving difficult as any containment procedure must be faster than the propagation rate of the network worms[2]. Vigilante [78] allows untrusted hosts to broadcast Self Certifying Alerts (SCAs) to each other when they detect a worm; the SCAs are automatically-

[2]Recall that flash worms can propagate across a majority of Internet hosts in 30 seconds [256]

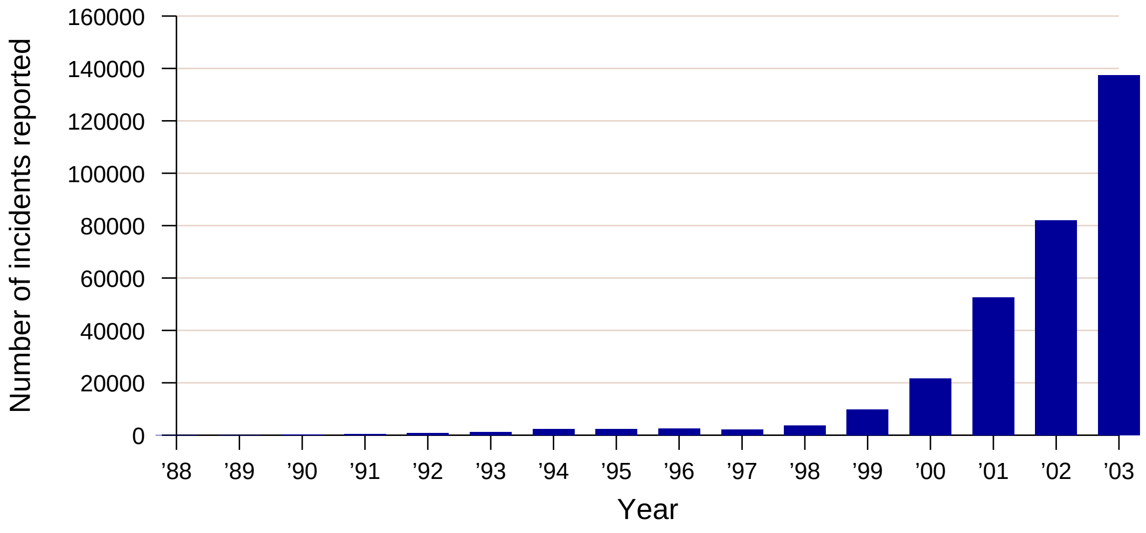


Figure 2.2: Number of CERT incidents since 1988. CERT stopped reporting incident statistics after 2003 since "attacks against Internet-connected systems have become so commonplace." *(source: cert.org)*

Table 2.1: Most Probed Ports on the Internet *(source: DShield.org, 5/12/2005 1645 UTC)*

	Service Name	**Port**	**Description**
1	win-rpc	1026	Windows RPC
2	microsoft-ds	445	Windows 2000 Server Message Block
3	netbios-ssn	139	NETBIOS Session Service
4	epmap	135	DCE Endpoint Resolution
5	auth	113	ident tap Authentication Service
6	gnutella-svc	6346	gnutella-svc
7	win-rpc	1025	Windows RPC
8	www	80	World Wide Web HTTP
9	netbios-ns	137	NETBIOS Name Service
10	AnalogX	6588	AnalogX Proxy Server

generated machine-verifiable proofs of vulnerability which can be independently and inexpensively verified by any other hosts. Weaver et al. take a different approach by attempting to throttle the scanning rate of worms [282] to give other defences more time to react.

Modern operating systems attempt to protect the integrity of binaries which were written in unsafe language such as C. Examples of protections include C compiler modifications to instrument binaries with "canary" values to detect buffer overflows [105, 81], virtual address space protection via the "non-executable bit" flag present in modern processors [9], hardened memory allocation functions [182], and system call monitoring to ensure that only valid system calls are permitted to be executed by an application [80, 164, 229, 184]. Despite their sophistication, none of these mechanisms guarantee protection against malicious attackers executing code on a host running vulnerable software. Kuperman et al. recently summarised these efforts [172] and observed that "*no silver bullet is available for solving the problem of attacks against stored return addresses, and attackers have a long history of learning how to circumvent detection and prevention mechanisms*". Wagner and Soto also noted attackers can easily bypass system call monitoring by executing an observationally equivalent program which still performs malicious activities [279].

This "Internet arms race" continues to the modern day as we now show with a recent case study. Skype [253] is a popular application used by millions of users for making peer-to-peer voice calls over the Internet. It uses a custom protocol [24] and a variety of firewall-punching techniques to ensure that users can connect to each other despite the presence of packet filtering or Network Address Translation routers. The protocol uses a number of sub-formats, each of which has a custom parser in the Skype client. In October 2005, security researchers at EDAS/CRC observed [98] that specially crafted packets sent to a Skype client could cause it to crash with a memory exception. Analysis of the suspect packets narrowed down the bug to the "Variable Length Data" (VLD) portion of the packet parsing code. The VLD packet (see Figure 2.3) consists of an initial counter which indicates the length of the remaining objects.

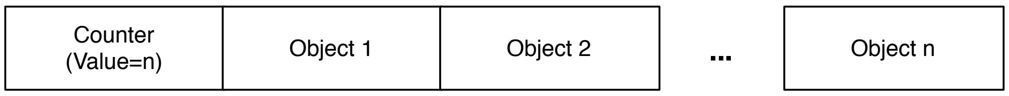


Figure 2.3: A sample Variable Length Data packet from Skype

The Skype client parses the counter, reads its value $\mathcal{V}$, and allocates $4\mathcal{V}$ bytes to hold the rest of the objects. However, the parsing code fails to verify that the maximum value of $\mathcal{V}$ is less than `0x4000000`, since any greater value than that will result in the integer overflowing and wrapping around to a small number when multiplied by 4. For example, an attacker could send a value $\mathcal{V}$ = `0x40000010`, which will result in $4\mathcal{V}$ = `0x40` bytes being allocated, but the full `0x40000010` objects being read into this small buffer.

Since the attacker is free to craft any value $\mathcal{V}$ and the subsequent object contents they can overwrite the heap with chosen values and modify the control flow of the program. This heap overflow is normally caught by address-space randomisation protections built into modern operating systems [250]. Unfortunately another quirk of Skype's design (its use of function pointers on the heap) means that attackers can bypass this protection and execute arbitrary code on the host. This error is particularly dangerous in view of the fact that Skype is specifically designed to bypass firewalls by using application-level tunnelling mechanisms (e.g. HTTP proxies). A worm written to propagate over the Skype network is therefore extremely difficult to stop by conventional network defences which block ranges of TCP and UDP ports.

This recent security hole is a perfect illustration of the difficulties encountered by efforts to *contain* vulnerabilities at the host or network level. Despite the sophisticated OS-level protection provided by (for example) Windows XP SP2, it is not a perfect protection and the nature of the coding style used by Skype meant that arbitrary code execution was still possible (although certainly more difficult than in earlier versions of Windows). Similarly, the Skype application is *specifically designed* to circumvent firewalls in an effort to make the application easier to use for end-users, and this means that a single security hole allows attackers to use this application-level tunnelling as an easy attack vector to attack other hosts also running Skype—without ever triggering an intrusion detection system since all the data is encrypted by Skype!

2.2 Functional Programming

Broadly speaking, functional programming is "*a style of programming that emphasizes the evaluation of expressions, rather than execution of commands*" [145]. Functional programming is often considered more analagous to evaluating mathematical equations than to the conventional sequences of commands found in an imperative programming language. The treatment of a program as mathematics has great significance when formally reasoning and analysing programs—for example, multiple calls to a function known to be idempotent can be safely evaluated a single time and the result re-used.

2.2.1 History

In 1932, Church conceived the λ-calculus to describe the behaviour of *functions* mathematically; it was not originally intended to be a programming language. It also turned out to be a remarkable basis for expressing computation, as Kleene [168] and Turing [268] later proved. Most modern functional languages are considered as non-trivial extensions to the original λ-calculus. Henk Barendregt summarises the relation neatly in his book [22] which describes the λ-calculus in more technical detail:

> Lambda calculus is a formal language capable of expressing arbitrary computable functions. In combination with types it forms a compact way to denote on the one hand functional programs and on the other hand mathematical proofs.

In the 1950s, John McCarthy developed the Lisp programming language [191, 192], which featured Church's λ-notation for expressing anonymous functions. Lisp developed a number of important contributions which influenced functional languages: (*i*) the conditional expression as a mechanism to express generation recursion; (*ii*) lists and higher-order operations on them such as `map`; and (*iii*) the introduction of automatic garbage collection and `cons` cells as an atom of allocation. In addition, Lisp was a very pragmatic language and featured imperative features such as sequencing, assignment and other side-effecting statements.

By his own account [193], McCarthy was not greatly influenced by the λ-calculus beyond the adoption of the nomenclature for anonymous functions in LISP. In the 1960s, Peter Landin introduced the Iswim[3] language which attempted to move away from LISP towards a smaller language core which could form the basis for "the next 700 programming languages" [173]. Iswim developed syntactic innovations such as the use of infix operators, simultaneous and mutually recursive definitions, and indentation-based parsing (recently popularised by Python [272]). Semantically, Iswim emphasised generality and equational reasoning, which resulted in a very small language core on which more complex programs could be built.

[3] short for If You See What I Mean

Landin was the first to make the argument that the denotational (or "declarative") style of programming permitted by Iswim was superior to the prevalent imperative style.

In 1978, John Backus delivered a powerful encomium for functional programming in his Turing Award lecture [15]. Backus described imperative programming as "word-at-a-time programming", and argued that this was insufficient to meet the demands of large, complex software engineering projects. Ironically, Backus was given the Turing Award for his pioneering work in developing FORTRAN (the major imperative language in use at the time), and as a result his argument for the functional style of programming was highly influential. Backus also noted that basing languages on the λ-calculus would lead to problems due to the difficulty of handling complex data structures; the realm of efficient and purely functional data structures is understood better today [218]. In his language FP, Backus introduced higher order functions as a useful abstraction for programming but the language itself was not popular.

During the 1970s, researchers at Edinburgh were developing the LCF theorem prover for analysing recursive functions. The command language developed for LCF—dubbed Meta Language (ML)—proved to be extremely popular and was developed as a stand-alone functional language [122]. ML deviated from the pure equational reasoning advocated by Backus and Landin and introduced the notion of *references* and side-effects, all encapsulated in a type system based on work by Hindley [135] and Milner [203]. Although this eliminated referential transparency and thus "pure" functional programming, the language still encouraged programming in a functional style. ML featured an advanced module system, I/O facilities, exceptions, and a novel type system characterised by: (*i*) type checking performed statically at compilation time; (*ii*) types automatically inferred from the program source (including a limited form of polymorphism)[4]; and (*iii*) user-definable algebraic data structures (added after the initial specification). As the popularity of the language grew into the 1980s, it integrate ideas such as pattern matching from other languages such as Hope [54], and was standardised as Standard ML [206].

At the same time as ML was being developed, David Turner was a powerful proponent for the purely functional approach to languages [269]. The most notable language was Miranda [270], which is still popular today as a teaching language. Miranda used the Hindley-Milner type inference algorithm and algebraic data-types, but was one of the first languages to adopt *lazy evaluation* semantics. Lazy evaluation delays the computation of expression until the results are actually required, which enables constructions such as infinite data structures and the minimisation of redundant calculations. However, it does make reasoning about the space and performance properties of a program much more difficult, as Ennals notes [103].

Throughout the 1980s, functional programming was extremely popular as a research topic, and a number of alternative implementations and languages emerged [39]. Since the semantic and formal underpinnings of these languages was remarkably similar, a committee of researchers proposed a more unified approach, and thus a new purely functional programming language named Haskell was born [143]. Haskell, much like ML, is a very complete programming language, and combines many of the concepts discussed earlier such as higher order functions, static typing, lazy evaluation and pattern matching. It also includes a module system, I/O and a large standard library of functions to make it easier to program with. The language continues to be developed and standardised (e.g. Haskell 98 [157]) and is generally regarded as the most mainstream lazy functional language available today.

[4]A good practical discussion of the ML type-checking algorithm is available on-line [159] in the now out-of-print book by Simon Peyton Jones.

Until the late 1980s, functional programming had failed to make an impact on "real" systems, until the Swedish telecommunications company Ericsson began to investigate better mechanisms to program telephone exchanges. Until then, development of these systems primarily used low-level imperative languages such as C, and Ericsson sought a language with primitives for concurrency and error recovery, but without the back-tracking execution model used in Lisp and Parlog [70]. In 1986, Erlang emerged as a functional language featuring strict evaluation, assignment and dynamic type checking [11]. It focussed on concurrent programming to help construct distributed, fault-tolerant, soft-real-time applications which needed high levels of up-time (e.g. portions of a running application can be upgraded in-place without restarting it). Erlang has many functional features, such as higher order functions and list comprehensions, and is notable for its fast message passing and fast task switching between thousands of parallel processes. The language rapidly gained popularity within Ericsson and by 1993 was being used by several other telecommunications companies such as Bellcore in a variety of real products [284]. It is still actively developed and is available as open-source software.

This history is not intended to be exhaustive; the reader interested in a more details is referred to our sources [145, 142, 79, 11], especially Hudak's excellent ACM survey [142]. We seek to convey to the reader a sense of the rich theoretical foundations that have led to modern functional programming.

2.2.2 Type Systems

Functional languages are characterised by their well-defined *type systems*. In programming languages, a *type* is a name for a set of values, and operations over that set of values. Types are either implicitly or explicitly supported by languages and may be statically verified at compilation time or dynamically enforced at run-time. In his book "Types and Programming Languages" [223], Pierce defines:

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

Type systems impose extra restrictions on the language to ensure that programs cannot be written which violate some properties—these restrictions must be carefully chosen to balance the conflicting requirements for language flexibility and safety. All modern useful type systems provide a basic *type safety* guarantee that no valid program can ever assign a value to a variable of type τ if that value is not a valid member of τ. This guarantee is not provided by C or C++ since type casts between incompatible values (e.g. integers and memory pointers) are permitted which can corrupt the underlying memory representations. Most functional languages provide automatic memory management to ensure that all memory accesses are guaranteed to be safe, and require run-time garbage collection [156, 234] to prevent space leaks.

Static type systems have proven to be very effective for specifying basic interface specifications; a well-known example is the Hindley-Milner algorithm used by ML [203, 206]. The creation of more powerful static type systems has been encumbered by the requirement that they must be statically decidable. Areas of active research into more expressive static type systems include type-and-effect systems [216] which describe the side-effects of a program (e.g. I/O) by capturing aspects of the language semantics, linear types to guarantee the single ownership of variables [46, 170, 104], and dependent types which are types that depend on a variable value [7, 287].

Dynamic type systems remove the static decidability constraints and add run-time checks enforced during the execution of the program. The guarantees offered by these systems are weaker than static type systems; they offer *partial correctness* proofs, ensuring programs either comply with the type system or experience a type exception at run-time. In some languages, these dynamic type errors can be caught and action taken to resolve the problem, but the most common result is to terminate the program.

Despite this weaker type discipline, dynamic contracts can be much more precise and easier to specify than their static equivalents. Some languages, such as the object-oriented Eiffel [196] are designed specifically with this model in mind [197], in order to encourage the "top-down programming" style[5]. Dynamic contracts can also be applied to higher-order languages; for example, Findler and Felleisen proposed extending ML with contracts [108]. This allowed the parameters and results of arguments to be restricted in certain domains such as the range of integer arguments. A "blaming" mechanism also exists, similar to exceptions but reserved for contract violations. Some other examples of dynamic contracts include ordering (e.g. sorting an integer array in ascending order), size (e.g. two lists passed as function arguments are the same size), or range (e.g. an integer argument must be non-zero).

Most programming languages adopt a varied combination of static and dynamic checking. Statically typed functional languages such as ML still require run-time bounds checking, unless extended with dependent types which can eliminate some of them. Java maintains run-time type information to allow type casting but also statically verifies them where possible. The dividing line between the static and dynamic checking is an arbitrary one; Cormac Flanaghan recently proposed hybrid type checking [111] where specifications are checked statically where possible and dynamically enforced otherwise, which he is argues is more practical than requiring the rigour of proof-carrying code [214].

2.2.3 Features

The majority of functional languages treats functions as *first class values*, meaning that functions can act as arguments as to other functions, and the return value of a function can be another function. This allows a language to define abstract *higher order functions* which accept other functions as their arguments. An example is the `map` used in ML and Haskell, which applies a supplied function to every element of a list and returns the results of that function as a new list. Higher order functions are a powerful notion originating from the λ-calculus, and are now a central feature in most functional languages.

Functional languages also allow these functions to either be created anonymously (a so-called λ-function), or bound to a variable name. Generally, the scoping rules for names are statically determined from the program source—a system known as *lexical scoping*. Lexical scoping greatly simplifies the problem of reasoning about the values of variables while writing code (by making it into a simple substitution). The alternative—dynamic scoping—is used in some languages such as Common LISP, but significantly increases the complexity and run-time overhead of a program.

Functions can also be partially applied by not supplying all of the function's arguments. This returns a *curried function* which has the provided arguments fixed as constants, and the unknown arguments remaining as parameters which must be passed to the new function. As Hughes points out [144], the combination of higher-order functions and currying enables a safe modular style, since it allows programs to be constructed by the composition of higher-order

[5]We discuss the top-down programming style further in §3.3.1.

functions in an unrestricted way (within the limits of the type system in use). For example, consider the following fragment of OCaml:

```
let rec fold fn acc = function                                    OCaml
     | [] → acc
     | x::xs → fold fn (fn acc x) xs ;;
# val fold : (α → β → α) → α → β list → α = <fun>
let sum = fold add 0 ;;
# - : int list → int = <fun>
let prod = fold mul 1 ;;
# - : int list → int = <fun>
sum [1;2;3;4];;
# - : int = 10
prod [1;2;3;4];;
# - : int = 24
```

In this fragment, we first define `fold`, which is an abstraction for applying a function over a list and returning the result. We can then specialise this abstraction (via currying) into the functions `sum` and `prod` which return the sum and product of integer lists. Observe that the type of `fold` contains two polymorphic type variables α (representing the type of the argument list) and β (representing the type of the return value). These type variables are resolved into concrete types in `sum` and `fold` during type inference.

Lazy Evaluation

Although support for higher-order functions is ubiquitous among functional languages, the question of the evaluation order of arguments has been controversial. The λ-calculus encourages the use of *normal order* reduction rules which simultaneously reduce arguments and permit recursion to be performed via the Y-Combinator. Unfortunately, when normal order reduction is applied to computing expressions, it can result in a lot of redundant computation as expressions are re-evaluated multiple times. For this reason, languages such as Lisp, ML and Hope adopted applicative order semantics which evaluate expressions in a particular order, and which can exploit the call-by-value conventions used by imperative language compilers.

In 1971, Wadsworth proposed a mechanism to efficiently implement normal order reduction semantics much more efficiently via *graph reduction* [278]. In graph reduction, the results of computed expressions are shared via pointers, guaranteeing that arguments to functions are at most evaluated only once, and only when they are needed. This strategy is dubbed *lazy evaluation* or call-by-need evaluation, and is the approach adopted by some modern functional languages such as Haskell. In addition to being closer to mathematical reasoning, lazy evaluation enables a number of novel data structures such as infinite lists (a comprehensive survey is available by Okasaki [218]). However, as Simon Peyton Jones points out in his Haskell retrospective "Wearing the Hair Shirt"[6], lazy evaluation comes with a significant implementation cost as it make reasoning about program performance and space usage very difficult [158, 103].

Polymorphism and Pattern Matching

Programming languages which provide support for strong abstraction of data improve the quality of programs for several reasons: (*i*) modularity is improved as the representation of a program is separate from its implementation; (*ii*) reliability is increased since cross-interface vi-

[6]This slightly bizarre title presumably refers to the old Catholic practise of wearing uncomfortable garments made from goat's hair as a form of penance.

olations are prohibited; and (*iii*) clarity is improved since the conflicting concerns of different implementations can be hidden from a programmer analysing a program. Over the years, functional languages have steadily improved their support for expressive abstractions. Starting with Hope [54] and subsequently integrated into ML and Haskell, user-defined algebraic data types and pattern matching are effective mechanisms for performing symbolic manipulation and manipulating data structures. These data types work particularly well when combined with strong static type systems which exhaustively verify at compilation time that the algebraic data types are used consistently and correctly in all cases. Once the type correctness has been verified, the compiler can discard all type information, which means that no run-time overhead is incurred by using these abstractions.

```
OCAML
type α tree =
    | Leaf of α
    | Node of (α tree × α tree) ;;
# type α tree = Leaf of α | Node of (α tree × α tree)
let rec map fn = function
    | Leaf a → Leaf (fn a)
    | Node (a,b) → Node(map fn a, map fn b) ;;
# val map : (α → β) → α tree → α tree = <fun>
let x = Node ( Leaf 1, Node ( Leaf 2, Leaf 3)) ;;
# val x : int tree = Node (Leaf 1, Node (Leaf 2, Leaf 3))
map (fun x → x + 10) x ;;
# - : int tree = Node (Leaf 11, Node (Leaf 12, Leaf 13))
```

In the above fragment, we define a binary tree which can contain either a single `Leaf` value or `Node` branch, and a `map` which applies a function across every `Leaf` and returns the result. The definition of this data structure is remarkably succint; notice in particular the use of *pattern matching* in the `map` to recursively iterate over the data structure. The main definition of `tree` is *polymorphic* since the actual data contained in the tree can be of any valid type (represented by α). A specialised `int tree` variable is declared and run through the mapping function, which adds 10 to the contents of every leaf and returns the new tree. An attempt to (for example) concatenate a string to a value in an `int tree` would result in a type error, since the type is no longer polymorphic.

```
OCAML
type α tree =
    | Leaf of α
    | BigLeaf of (α × α)
    | Node of (α tree × α tree) ;;
# type α tree = Leaf of α | BigLeaf of α | Node of (α tree × α tree)
let rec map fn = function
    | Leaf a → Leaf (fn a)
    | Node (a,b) → Node(map fn a, map fn b) ;;
# Warning P: this pattern-matching is not exhaustive.
# Here is an example of a value that is not matched:
# BigLeaf _
# val map : (α → β) → α tree → α tree = <fun>
```

We extend our binary tree with a `BigLeaf` which can hold two values instead of just one. If we then re-use the `map` function declared previously, the compiler can statically determine that we have not pattern-matched all cases of the data structure, and will emit a warning with a counter-example (in this case `BigLeaf`). This is immensely useful when developing and refactoring large applications written using static typing, as the compiler helps the programmer flag regions of the code which may need to be modified once the data structure definition has been updated.

The example above illustrates the ML approach to data abstraction. Haskell adopts different ways of declaring algebraic data types, but has the same pattern matching facility with exhaustiveness checks. Erlang [11] extends pattern matching with a bit-matching syntax designed for implementing low-level protocols [128]. Of course, these facilities for static typing can also be available in an imperative language; functional languages just choose to offer greater flexibility via polymorphism and automatic type inference, just as they do with higher-order functions.

Formal Semantics

Earlier, we described the history of functional programming (§2.2.1), and the mathematical foundations from which it emerged. This tradition has continued to recent times, and the development of type systems and extensions to functional languages are based on rigorous proofs. In contrast, the development of imperative languages is often rather ad-hoc, with the formal foundations following after the language has been developed (e.g. the "Theory of Objects" [1] was published well after C++ and Java popularised the concept). Standard ML is one of the few languages which has been rigourously defined using operational semantics, both in terms of its static type checking rules and its dynamic execution rules [206]. This definition provided a solid foundation for future implementation and extensions to the language [273].

2.2.4 Evolution

Functional languages have never been popular in industrial circles, being primarily used by academics to solve research-oriented problems. Recently however, things have begun to change somewhat, as platforms such as Java and .NET have begun to integrate features such as generics [165]. So-called *scripting* languages such as Python [272] and Ruby [188] have also gained popularity as rapid prototyping tools to "glue" disparate components together, particularly on complex web sites. Both Python and Ruby have adopted functional features such as anonymous lambda functions and statements which return values. The vague definition of the term "functional languages" extends even to languages conventionally considered to be "functional", such as ML and Haskell, which have very different semantics regarding purity and evaluation order.

When we refer to "functional programming" in this thesis, we do not restrict the term to mean pure, lazily evaluated languages such as Haskell. Rather, we focus on languages which **encourage programming in a functional style**, and seek to relate this style to constructing practical, high-performance network applications. The current trend of integration of functional features into more conventional platforms such as Java or .NET is a strong validation of our research, as it means that the techniques we describe in later chapters will apply beyond the narrow set of functional languages used in academic circles today.

2.3 Objective Caml

Our earlier description of functional languages (§2.2) revealed a large variety of implementations with varying levels of flexibility, maturity and safety. We chose Objective Caml (OCaml) [181] as our language for constructing network applications since:

Static Typing: OCaml is based on ML and so provides a mature static type system with strong support for abstraction, but with the pragmatic facility for side-effects.

Flexibility: OCaml provides many useful extensions to ML (e.g. an object system), and has been described as an "object-oriented, imperative, functional programming language"—instead of imposing one programming style, it allows them to be mixed in as required by the problem at hand. This is important to network programming, which requires both low-level octet and high-level data structure manipulation.

Native Compilation and Portability: OCaml offers support for native compilation directly to multiple host architectures such as Alpha, Sparc, x86, MIPS, HPPA, PowerPC, ARM, Itanium and AMD64. For architectures which are not supported, OCaml can also compile into a machine-independent byte-code.

Fast Garbage Collection: The OCaml garbage collector is a fast modern hybrid generational and incremental collector. It allows extremely fast memory allocation (a simple pointer increment), and separates data into two separate heaps: a *minor heap* for small, short-lived objects and the *major heap* for longer-lived data structures. Data structures in OCaml have less overhead than in languages such as Java (due to the lack of run-time type information and use of 31-bit integers), leading to a level of memory usage comparable to C/C++.

System Integration: Due to the simplicity of the OCaml heap layout, it is relatively simple to safely bind foreign functions into OCaml. This is essential to provide full access to the range of kernel system calls often required by network applications[7], and integration with external system libraries.

A full description of OCaml is beyond the scope of this thesis and can be found in the literature [79, 181, 63]. However, we briefly discuss below some key features and extensions which we refer back to later in the thesis.

2.3.1 Strong Abstraction

C programs are generally split up into multiple object files, with shared header files which declare the types of common functions between the object files. Although the compiler performs some consistency checking, the object files themselves do not contain type information (only a list of symbolic names), and thus linking an object file with an incorrect prototype will result in memory corruption. This problem is accentuated when multiple object files are linked into shared libraries used between multiple applications, as there is no way to ensure binary compatibility between the applications linking to this library beyond careful versioning and code management.

In contrast, an OCaml program with type inference results in shorter source code with less redundancy than the C equivalents (since there is no need to declare external prototypes which duplicate the function definitions themselves). An external interface file can still be specified and auto-generated from the source code itself if desired. Including this interface file allows the exported types of a library to be *opaque types*—types which are manipulated by functions within the library, but are exposed in the external interface as an abstract name. The utility of this can be demonstrated by an implementation of Church numerals (from the λ-calculus), where integers are represented by repeated calls to a successor function.

[7]System calls such as *sendfile(2)* or *kqueue(2)* are not part of POSIX standards, but often used by applications to increase scalability and throughput.

```
type num =                                OCAML
    | Zero
    | Succ of num
let zero = Zero
let succ x = Succ x
let rec to_int = function
    | Zero → 0
    | Succ x → to_int x + 1
```

```
type num                                  OCAML
val zero : num
val succ : num → num
val to_int : num → int
```

The example above shows the implementation *(left)* and the associated interface *(right)*. Church numerals are represented by a variant type `num`, and the functions `zero`, `succ` and `to_int` manipulate the numerals. However, notice that in the interface, the exact type of `num` is left abstract. This means that the only way to create values of type `num` is to use the `zero` function from that library.

```
type num = int                            OCAML
let zero = 0
let succ x = x + 1
let to_int x = x
```

```
type num                                  OCAML
val zero : num
val succ : num → num
val to_int : num → int
```

In the fragment above, we have replaced the implementation of Church numerals with one which takes advantage of native OCaml integers[8]. However the external interface is precisely the same, and programs which used the old library will not require modification. Notice that the `to_int` function in this representation is simply the identity function $x = x$. The identity function is optimised away at compilation time, and the entire opaque typing procedure imposes no run-time overhead in the application as type information is discarded early. Opaque types also provide a simple mechanism to enforce API sequencing to statically ensure that functions are applied in the correct order.

```
type one = string                         OCAML
type two = string
type three = string
let first () = "one"
let second x = x ^ "two"
let third x = x ^ "three"
let final x = x
```

```
type one                                  OCAML
type two
type three
val first : unit → one
val second : one → two
val third : two → three
val final : three → string
```

In the above example, the implementation *(left)* defines the functions `first`, `second`, `third` and `final`, which perform simple operations upon strings. `first` returns a new string, and `second` and `third` concatenate a value to it. Now we wish to enforce that the functions can only be called in the order they are defined. The interface *(right)*, simply replaces the function arguments with the opaque types `one`, `two`, and `three`. This ensures that, for example, `second` can only be called with the return value of `first` as an argument. As we noted before, all of these opaque types only exist during the type-checking phase, and are optimised away in the final binary. In Chapter 7, we show how useful this technique is to enforce the correct ordering of cryptographic operations in our SSH implementation.

[8]This version is not precisely a Church numeral representation since native integers are modulo the architecture word-size, and will thus wrap around.

2.3.2 Polymorphic Variants

One of the most useful features of ML-style functional languages are their user-defined algebraic data-types (also known as *variants*). These data types are generally defined once, and then re-used through the program. A classic example is the list construct:

OCAML
```
type α list = | Nil | Cons of α × α list ;;
# type α list = Nil | Cons of α × α list
let x = Cons (1, Cons (2, Cons (3, Nil))) ;;
# val x : int list
let rec length = function
    | Nil → 0
    | Cons (hd, tl) → 1 + length tl ;;
# val length : α list → int
```

This defines two constructors—`Nil` to represent the empty list, and `Cons` to hold a value and the rest of the list. The type of the list is parameterised over the type of the value stored in the list. We can then define a list of integers `x`, which is of the specialised type int list. To illustrate how functions can be written to manipulate this custom data type, we have the `length` function to calculate the number of elements.

Variant types are used extensively in most functional languages as a safe alternative to the pre-processed `#define` constructs used in C. Pattern matches using variant data types are checked for exhaustiveness and a warning output if a case has not been checked. However, these data types can also be a drawback when creating large applications—to re-use the `length` function defined above in a different application, the associated type definition must also be duplicated (normally via textually copying the source code).

Jacques Garrigue introduced polymorphic variants [118] into OCaml to support more flexible use of variant types [119]. Syntactically they are distinguished from normal variants by placing a back-tick in front of the name, and type definitions enclosed in square brackets. Unlike conventional variants they do not need to be explicitly declared before being used. The example below defines an int list `x` and `length` function, but this time uses polymorphic variants and does not depend on a commonly defined data type.

OCAML
```
let x = ‘Cons (1, ‘Cons (2, ‘Nil)) ;;
# val x : [> ‘Cons of int × [> ‘Cons of int × [> ‘Nil ] ] ]
let rec length = function
    | ‘Nil → 0
    | ‘Cons (hd, tl) → 1 + length tl ;;
# val length : ([< ‘Cons of β × α | ‘Nil ] as α) → int = <fun>
length x ;;
# - : int = 2
type (α,β) t = [ ‘Cons of α × β | ‘Nil ] ;;
# type (α, β) t = [ ‘Cons of α × β | ‘Nil ]
```

In the example above, `x` is defined as list using polymorphic variants. The returned type is of the form [> ‘Foo] which can be refined by *adding* more variants to the type (but not removed, so any pattern match against this type must contain at least every variant in the type). Conversely, the definition of the `length` function has a type of the form [< ‘Foo] which may be refined by *removing* variants from it (since the `length` function uses the variable in a pattern match, this is safe). Our example then demonstrates how the variable `x` can be applied to the `length` function despite the lack of a common type definition between them. Finally, we define an explicit polymorphic variant which lacks the refinement symbols of the previous definitions; by explicitly annotating the type all ambiguity regarding its use can be eliminated.

OCaml, like other ML variants, features a sophisticated module system which is useful for creating distinct name-spaces in ML programs. A drawback to using a lot of nested modules is that the module name must be prefixed to any variant types used elsewhere:

```
OCaml
module Foo = struct
    type t =
    | One
    | Two
    | Three
end
let a = Foo.One ;;
# val a : Foo.t = Foo.One
let b = ‘One;;
# val b : [> ‘One ] = ‘One
```

The module `Foo` defines a variant type `t`. To use any of the variants outside the module, `Foo` must be prefixed to the name of the variant. In practise, this can lead to some very verbose code if a lot of modules are used (as we show later in Chapter 5, our generated code does exactly this). Polymorphic variants provide an effective work-around, as they can be used as-is without a module prefix, and the type checker will ensure their correct and consistent use.

Polymorphic variants do also come with some drawbacks. The compiler has fewer chances to perform static optimisation, leading to a larger representation (although this is only apparent on large data structures). More importantly, the polymorphic variants result in a weaker type discipline since the safety net of a pre-declared data type is no longer present. A simple typographic error can result in an incorrect inference which silently slips through the type checker (it would still be type safe, but have different semantics from the programmer's original intention). This can be avoided by adding explicit type annotations which precisely define the valid set of polymorphic types when using them in a function. In practice, we find these annotations are extremely useful, since without them the OCaml type checker can output long and hard to deceipher type errors when programs fail type checking.

Polymorphic variants proved to be particularly useful when used in *automatically* generated OCaml code as we do extensively in both our SPL language (§6) and MPL packet specifications (§5). The translator generating the OCaml code can reliably insert explicit type annotations, and ensure that all uses of polymorphic variants are "closed" (i.e. cannot be further refined). This is normally the most error-prone aspect of using polymorphic variants if defining them by hand, as typographic errors can have serious consequences. Once the code has been output, the polymorphic variants can be used from other components, such as the main network application code, extremely easily. Polymorphic variants are used extensively in the interfaces output by the MPL packet description language described in Chapter 5.

2.3.3 Mutable Data and References

OCaml supports destructive data update through: (*i*) *mutable* data which can be modified in-place; and (*ii*) *references* which are mutable pointers to other data structures. The use of both mutable data and references is type-safe[9], so for example a reference to an integer could not be assigned to point at a string without causing a type error at compile-time.

[9]ML imposes a so-called *value restriction* to make references compatible with its polymorphic type inference. More details are available from Smith and Volpano's paper [254].

In OCaml, string types are mutable arrays of characters, meaning that the underlying buffer can be modified in-place by library functions such as `String.put`. They can be considered analogous to (`char` ∗) pointers in C/C++, with the contents of the pointer being safely bounds-checked and managed by the garbage collector. Any field in an OCaml record can be marked as *mutable* which allows it to be changed, or as a *reference* to a data structure instead of directly storing it in the record. The difference between mutable record types and references is best illustrated with a simple example.

```
OCAML
type t = { mutable a: int; b: int ref; c: int } ;;
let x = { a=1; b=(ref 2); c=3} ;;
# val x : t = {a = 1; b = {contents = 2}; c = 3}
let y = { x with c=30 } ;;
# val y : t = {a = 1; b = {contents = 2}; c = 30}
```

We have defined a record type `t`, a record value `x` of this type, and created a copy of it using the `with` operator. This copy is called `y`, and we set the field `c` to a different value while performing this copy.

```
OCAML
y.a ← 10 ;;
# val x : t = {a = 1; b = {contents = 2}; c = 3}
# val y : t = {a = 10; b = {contents = 2}; c = 30}
x.b := 500 ;;
# val x : t = {a = 1; b = {contents = 500}; c = 3}
# val y : t = {a = 10; b = {contents = 500}; c = 30}
```

In the code fragment above, we perform two different operations. Firstly, we changed the value of the mutable field `y.a`, but the value of the field in `x.a` does not change. Secondly, we modified the contents of the reference field `x.b`, which had the effect of also changing the contents of `y.b`. Thus, we can create record data types which have both shared and independent variables, which are all updated in a strongly type-safe fashion. As an example of the type safety, any attempt to change the value of the immutable field `c` will result in a type error, and similarly the actual reference in field `b` cannot be pointed elsewhere since it is immutable (only the location to which it points can be altered).

```
OCAML
y.c ← 100 ;;
# The record field label c is not mutable
y.b ← (ref 10) ;;
# The record field label b is not mutable
```

It is important to clearly understand the different semantics between mutable and reference variables, as we take advantage of it to efficiently implement packet handling routines in our MPL data description language (§5.2.1).

2.3.4 Bounds Checking

OCaml dynamically bounds checks all array and string accesses to guarantee that program execution will never result in memory corruption. This can be a drawback if extra bounds checks are introduced which unnecessarily slow the program down.

```
OCAML
let x = "hello" in
for i = 0 to String.length x - 1 do
    output (String.get x i)
done
```

```
OCAML
let x = "hello" in
for i = 0 to String.length x - 1 do
output (String.unsafe_get x i)
done
```

The first example above *(left)* iterates over a string and outputs it, performing a bounds check for every access. The second version *(right)* is faster since it uses the `unsafe_get` function to retrieve each character. The unsafe version disables the type safety guarantees of ML and can potentially return garbage data (or even worse, `unsafe_put` can cause memory corruption). However, the above code can be restructured somewhat:

```ocaml
OCaml
let iter fn x =
    for i = 0 to String.length x - 1 do
        fn (String.unsafe_get x i)
    done ;;
# val iter : (char → α) → string → unit = <fun>
iter output "hello";;
# hello
# - : unit = ()
```

In this version, we have defined `iter`, which accepts a function and a string, and applies that function to every character in the string. By providing a theorem that `iter` is safe under all circumstances (either axiomatically or by using a proof assistant), we can write OCaml code which iterates over a string as fast as the C equivalent, but with the "danger" encapsulated into a single function. As we will see later (§5.4), this ability to turn off bounds checking is particularly useful in the automatically generated OCaml code we use for packet parsing.

2.4 Model Checking

A well-constructed system can *provably* meet its design specification. Actually delivering rigorous proofs of real systems is extremely difficult; attempting manual proofs will only work for small-scale problems, and in 1936 Turing demonstrated the impossibility of a general proof procedure for arbitrary software (the *halting problem* [267]). One solution is to adopt more modest requirements, and construct a simplified model of the underlying software which accurately represents the properties we wish to test, while avoiding the complexity of the software itself. This model can be exhaustively *model checked* by automated tools to find errors, which can be fixed in the original program, and the model updated accordingly.

Model checking works especially well for concurrent applications, which have a large number of possible execution traces arising from interleaving the execution of parallel components. The model checker can efficiently cover *all* possible interactions between these components and apply safety constraints over this state space. This allows it to locate unreachable code or identify common protocol bugs such as deadlocks which only occur rarely in practice and are hard to track down by more conventional debugging.

We use the popular SPIN [138] model checker, originally developed at Bell Labs in the 1980s. SPIN is freely available and is one of the most widely used model checkers in the world in both academia and industry[10]. SPIN models are specified using a special language—the Process Metalanguage (or PROMELA)—which we describe in §2.4.1. Next we describe how to verify safety properties about these abstract models (§2.4.2). Finally we will discuss the issues surrounding the creation of PROMELA models, either manually or via extraction from existing source code (§2.4.3).

[10]SPIN received the prestigious 2001 ACM Software System Award in recognition of its contributions to software reliability.

2.4.1 SPIN and PROMELA

SPIN models capture the co-ordination and synchronisation aspects of complex software and replace the computation portion with non-determinism. This enables the size of the model to be decreased and made suitable for exhaustive state-space checking. Models are a labelled, directed graph of a Kripke structure [72] represented by the tuple (S, S_i, R, L) where S is a finite set of states with initial states $(S_i \subseteq S)$, the transition relation $(R \subseteq S \times S)$ such that $(\forall \alpha \in S(\exists \beta \in S((\alpha, \beta) \in R)))$ and a labelling function $(L : S \rightarrow 2^{AP})$ where AP is a set of atomic propositions (i.e. boolean expressions over variables and constants). Infinite paths can be constructed through the Kripke structure (essential for modelling practical reactive systems). Model checkers optimise this representation through different techniques such as partial order state space reduction [139] or symbolic representation of Kripke structures [51, 50], with varying degrees of success in different problem areas [74]. In this dissertation we focus on the use of the SPIN model checker, which uses partial order reduction and explicit model checking of the state space.

The PROMELA language includes primitives for process creation, interprocess communication, and non-deterministic choice, but lacks other familiar programming language constructs such as function calls (it only provides procedures with no return value) or side-effecting statements. A PROMELA model consists of a finite set of *processes* which communicate across shared message channels using *data objects*. Processes are defined using a `proctype` declaration and consist of data declarations and a sequence of statements. Multiple statements can be grouped into atomic blocks via the `atomic` keyword. SPIN can switch between processes and arbitrarily interleave non-atomic statement blocks when performing model checking.

Data Objects

Variables in PROMELA are scoped either locally to a process or globally across all processes, and must be declared before use. Table 2.2 summarises the type declarations available; the `unsigned` type is notable for allowing a variable range of bits ($1 \leq n \leq 32$). The `mtype` declaration permits the limited use of enumerations, and the `typedef` keyword can construct structured records, as shown below:

Table 2.2: Basic Data Types in PROMELA *(source: The* SPIN *Model Checker [138])*

Type	Range
bool	false,true
byte	$0 \ldots 255$
chan	$1 \ldots 255$
short	$-2^{15} \ldots 2^{15} - 1$
int	$-2^{31} \ldots 2^{31} - 1$
unsigned	$0 \ldots 2^n - 1$

```
mtype = { Alpha, Beta, Gamma };                                   PROMELA
mtype x = Beta;
typedef Foo {
    unsigned m : 5;
    chan p;
    byte flags[3];
};
```

In the above example, we have defined the enumeration Alpha/Beta/Gamma and declared a global variable x using the enumeration. We also define a record Foo which consists of a 5-bit value m, a message channel p, and an array declaration of a 3-byte value flags. PROMELA strictly enforces the range of values of variables, and so m can have range $2^5 = 0 \ldots 31$. Unlike algebraic data types in ML, multiple mtype declarations are not treated distinctly; instead all enumerations are combined into a single type, and only 255 distinct names can be used in a PROMELA model (we show later in §6.3.2 how we bypass this limitation).

Message Channels

Processes can transmit messages to each other via channels (normally represented by global variables). Channels are declared as a list of fields and a message queue size:

```
chan c = [4] of { short, byte, bool }                             PROMELA
c ! expr1,expr2,expr3
c ? var1,var2,var3
```

The channel c accepts three variables for each message, and can buffer up to 4 messages before the channel blocks. The variable types can include records or other channels, providing a mechanism for passing channels between processes. A succinct syntax is provided for transmitting and receiving messages on a channel— the ! operator transmits the three expression arguments across the channel c, and the ? operator receives messages into the provided variables. Message receipt is non-executable[11] if the channel buffer is empty, and attempts to write to a full channel will either block or drop the message (depending on how SPIN is configured). Unbuffered channels are useful to permit synchronous (or rendezvous) communication between processes. By defining a channel of size 0, a single sender and receiver can communicate with tight synchronism.

Control Flow

PROMELA provides support for labels to mark statements, and jumping to them via the goto statement. Unlike most normal programming languages, it also supports a non-deterministic choice construct (in the spirit of Occam's ALT [155]):

```
chan ch = [1] of { bit }                                          PROMELA
int count = 0;
if
:: ch!1 → count–
:: ch!0 → count++
:: count++
fi
```

[11] PROMELA defines precise semantics for when a statement can be executed for a process to make progress. The model checker exhaustively interleaves all possible executable statements, and can detect situations where no further progress is possible due to all processes being non-executable.

We define a channel `ch` and an integer counter. The non-deterministic choices then allow either a `1` or `0` message to be transmitted down the channel, with the counter being decremented or incremented respectively. As a final alternative, a message might never be transmitted, but the counter incremented anyway. SPIN explores all possible execution paths non-deterministically until no further execution is possible (e.g. if the channel `c` is full then the choices to transmit a message would not be selected).

Processes and Executability

Processes in PROMELA consist of a `proctype` declaration consisting of a set of variables and the statements which make up the process. There are two ways of instantiating processes: (*i*) adding the `active` keyword to a declaration with an optional numeric argument indicating the number of processes to start; and (*ii*) the `run` statement to spawn a new process. Every process is assigned a unique process identifier which is a non-zero integer. Since a PROMELA model can only express finite state machines[12], an infinite number of processes cannot be spawned; by default SPIN supports up to 255 processes.

Depending on the state of the system, every statement in PROMELA is either *executable* or *blocked*. Some statements, such as assignments are unconditionally executable, while others such as channel communications can block until the channel becomes free. If a process has a choice of valid executable statements, SPIN can non-deterministically choose one (or all) of the options to execute. If all of the processes are blocked, then the system is said to be in *deadlock*, and an error is issued with a message trace which triggers the deadlock.

2.4.2 System Verification using SPIN

The manual verification of PROMELA models is impractical for non-trivial models of real systems. Instead, SPIN can mechanically verify safety properties such as liveness (a system does not deadlock), progress (a system does not remain in an infinite loop) and the validity of assertions (a proposition is never violated). If a verification reveals an error, SPIN produces a guided backtrace which simulates the shortest sequence of events required to trigger the error in the model. Figure 2.4 shows a simulation of a simple DHCP client/server running over a delay-free network, and Figure 2.5 shows the more complex executions possible when delays are introduced. These examples were produced by SPIN running in a simulation mode which randomly selects non-deterministic choices.

SPIN also supports higher-level safety checks through Linear Temporal Logic (LTL) constraints, by translating them into so-called *never claims* which specify system behaviour that should never happen. Never claims can also be constructed manually (since they are strictly more expressive than LTL) but care must be taken to ensure that they are stutter-invariant [90] or the partial order reduction algorithms used by SPIN cannot be safely applied. Table 2.3 lists some commonly uses LTL formulae and their informal meanings.

The variables in the LTL formulae represent atomic propositions over system state. This means boolean expressions over global variables and constants and SPIN also supports "remote references" which permits the inspection of process-local variables. The use of remote references, although convenient when manually creating PROMELA models, is not compatible with partial order reduction and can thus significantly increase the resource requirements for model verification.

[12]There do exist infinite state model checking algorithms which lazily evaluate an unbounded model on demand, but SPIN does not support this, nor do we require it.

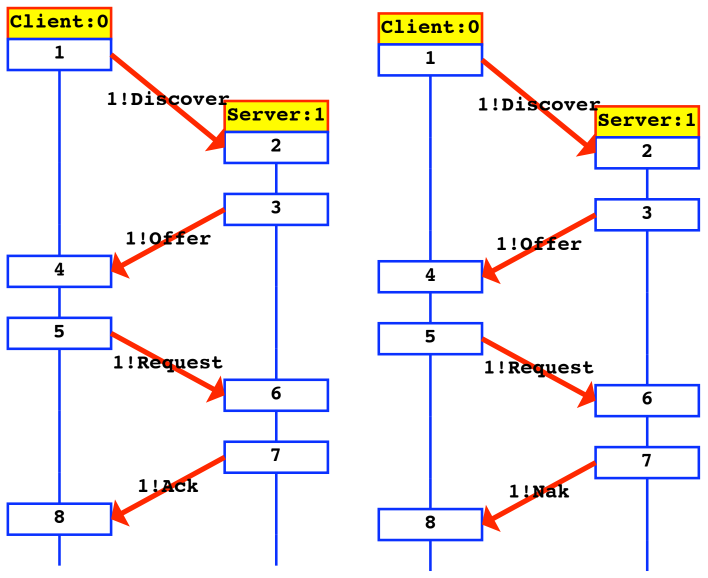


Figure 2.4: Guided trace from SPIN showing two alternative execution paths for a simple DHCP [95, 177] client and server model

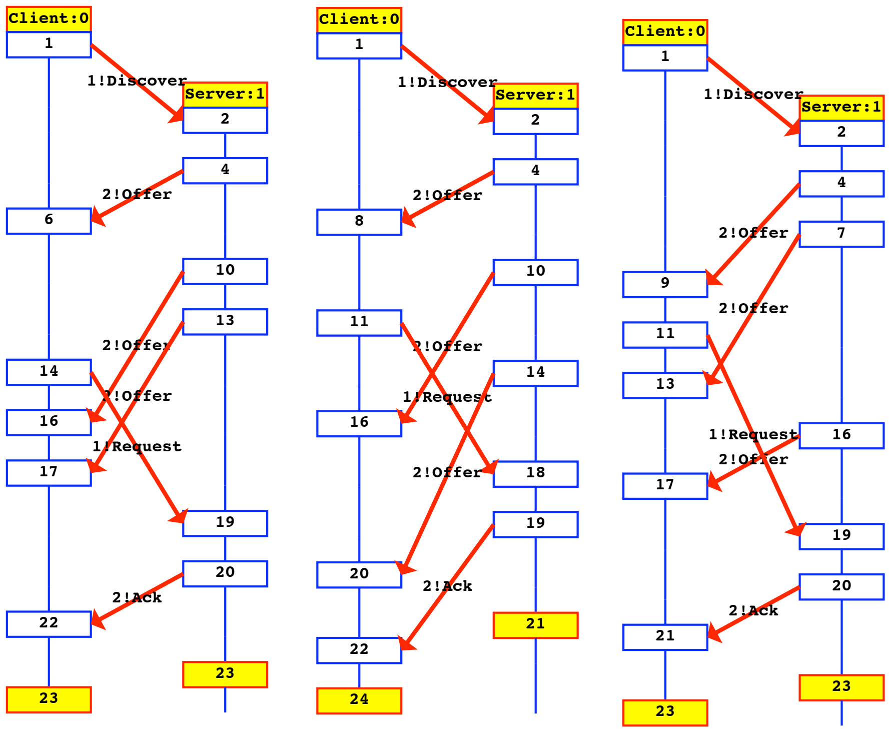


Figure 2.5: Possible execution paths for a more complicated DHCP example with network buffering delays.

Table 2.3: Frequently used LTL formulae *(source: The* SPIN *Model Checker [138])*

Formula	**Template**	**Meaning**
$\Box$p	invariance	always p
$\Diamond$p	guarantee	eventually p
p $\rightarrow \Diamond$q	response	p implies eventually q
p $\rightarrow$ q $\cup$ r	precendence	p implies q until r
$\Box\Diamond$p	recurrence	always eventually p
$\Diamond\Box$p	stability	eventually always p
$\Diamond$p $\rightarrow \Diamond$q	correlation	eventually p implies eventually q

2.4.3 Model Creation and Extraction

PROMELA is a difficult language to manually construct large, complex models with since it lacks the high-level type system which makes the functional languages described earlier (§2.2) so easy to program with. Although it offers limited symbolic data types via `mtype`, this does not scale beyond 255 labels and enumerations cannot be held distinct from each other. Even if such a model were manually constructed, the problem of proving its equivalence to the underlying source code it represents is difficult. In this case, a proof of the model's correctness is essential to prevent the executing program diverging from the model. One solution is to automatically extract models from a program's source code, a process known as *model extraction*.

Model extraction is a popular method for verifying safety properties of large systems written in C. Holzmann developed MODEX to extract PROMELA models from ANSI-C code [140]. Chen and Wagner created MOPS [65] to test an entire Linux distribution for security property violations such as race conditions [244, 64]. Engler has applied model checking techniques to test operating system kernel code for concurrency [100] and file-system [289] bugs, finding some serious errors in Linux and OpenBSD. BLAST [133], SLAM [20], Java PathFinder [130, 275] and Bandera [75] all rely on model-extraction techniques to prove temporal properties about their respective systems.

SPIN also plays a role in the verification of specifications, by compiling elements of the specifications automatically into PROMELA models. Kaloxylos developed a compiler which translates Estelle [52] specifications directly into PROMELA [162]. Chen and Cui convert the Unified Modelling Language (UML) into PROMELA to check the validity of programs communicating using CORBA-based middleware [66]. As we explain later in Chapter 6, we adopt a similar approach by using a simpler language to specify non-deterministic state machines and automatically output PROMELA.

Model extraction normally involves iterative predicate abstraction [19, 124] where portions of an application are replaced with simpler (often boolean) abstractions. This process, while effective for creating an equivalent and tractable model from source code, is difficult to maintain over a long period of time as the code-base evolves. A seemingly small change to the main application can result in a large variance in the extracted abstract model, and safety properties over the model may require rewriting. Conversely, the approach of generating source code stubs automatically from an abstract specification (and resolving the non-determinism through those stubs) works well initially, but can require re-writing source code if the specification changes. As we discuss later (4.1), one of the key goals of our research is to investigate a practical middle-ground between model extraction and specification while ensuring that the equivalence between model and application is always preserved.

2.5 Summary

This chapter has explained the technical background of the concepts used in the rest of this dissertation. We began by motivating our thesis statement that "*Applications which communicate via standard Internet protocols must be rewritten* ..." by examining the past and current Internet security situation (§2.1). Next we introduced functional programming as a mature programming style with well-specified type systems which help write reliable network applications (§2.2), and in particular Objective Caml (OCaml) as a mature language which encourages programming in a functional, imperative or object-oriented style and produces high-performance native code (§2.3). We use OCaml throughout this disseration and demonstrate through our case studies (§7) that secure applications can be constructed with equal or better performance and latency than their C counterparts by using it. Finally we introduced the SPIN model checker which can exhaustively verify safety properties of abstract models of complex, concurrent software (§2.4). We use SPIN as part of our SPL state machine language (§6) to apply temporal logic assertions against protocols, for example in our OCaml SSH server (§7.1.5).

CHAPTER 3

Related Work

Blank... BLANK?! You're not looking at the bigger picture!
80S GUY (FUTURAMA)

The construction of reliable, secure and high-performance distributed applications has been a staple research topic for many decades. In order to analyse the considerable body of related work, we first split network application architectures along two lines: (*i*) the "data plane" which processes a high volume of low-level information (e.g. bytes from a network connection); and (*ii*) the "control plane" which determines the behaviour of the data plane. The control plane is normally more abstract and complex than the data plane which, due to performance requirements, can only perform simple processing on data passing through it. The interaction between the two planes is rather arbitrary—e.g. they can run asynchronously in separate hardware configurations, as often happens in network routing architectures [125], the control plane can be "reactive" to events coming from the data plane [186], or they can be completely integrated as in most general-purpose programming languages. In this dissertation we are primarily concerned with software applications, and do not cover the literature of hardware generation languages unless relevant to software architectures.

Figure 3.1 illustrates the design space graphically by plotting techniques on two axes: (*i*) their operation on the control or data plane (or both, in the middle); and (*ii*) the degree of mathematical rigour applied to the technique. At the top of the graph are techniques ranging from the formal Petri nets or process calculi which deal with highly abstract models of a system, to software model checking systems such as SLAM or BLAST which verify simplified models of a complex system, and finally the more ad-hoc but very practical dynamic instrumentation techniques such as `systrace` that enforce control flow at run-time (§3.1). At the bottom of the graph are the "data manipulation languages" which specify the nature of data being transmitted across a network, either formally in terms of ASN.1 or CORBA IDL, as grammars of varying power (e.g. `yacc`) or very ad-hoc but easy-to-use text processing languages such as Awk (§3.2). In the middle lie the general-purpose programming languages (e.g. ML, Java or C) which deal with both control and data planes with varying degrees of formal rigour (§3.3).

This classification is rather fuzzy, for example some languages such as Python or Perl over-

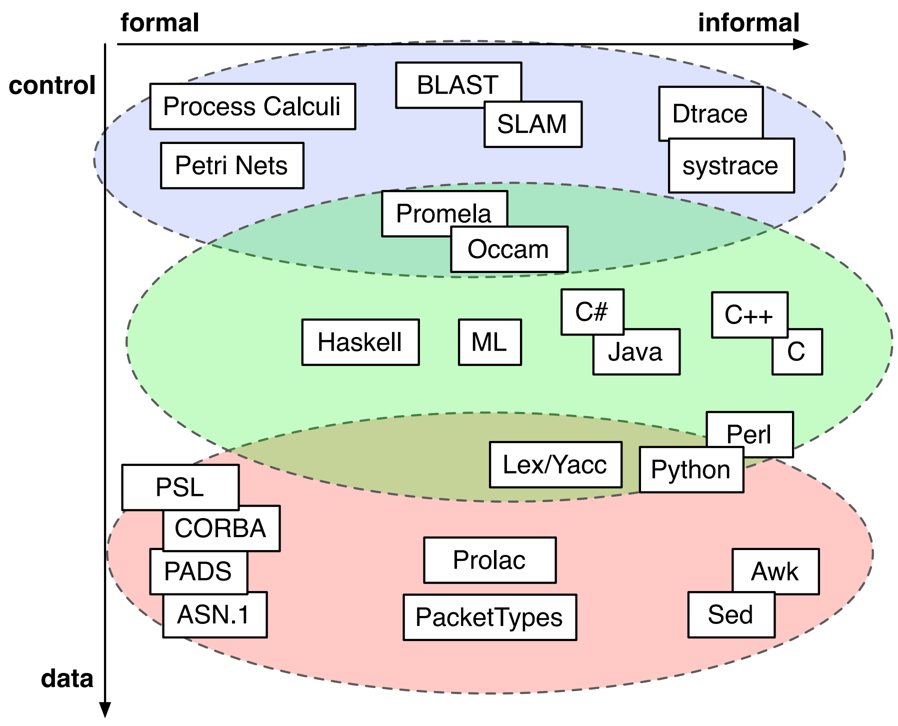


Figure 3.1: Broad categorisation of the design space for constructing network applications, in terms of formality and the level of abstraction

lap between data processing and general-purpose languages, and Occam and PROMELA are general-purpose languages oriented towards control systems. Nevertheless, it is a useful guide to categorise the literature on constructing network applications.

3.1 Control Plane

In this section we survey the literature on control plane manipulation, ranging from very formal systems such as process calculi and Petri nets (§3.1.1) to the recent advances in software model checking and extraction (§3.1.2), and finally the very flexible low-level instrumentation techniques used to enforce policies against untrusted binaries (§3.1.3).

3.1.1 Formal Models of Concurrency

Process Calculi are a family of related algebras to formally model concurrent systems, as opposed to the sequential computation modelled by (for example) the Turing Machine [267]. They provide a high-level view of communication and synchronisation between independent processes, and laws to allow mathematical reasoning about equivalences between processes. Two influental early examples include Communicating Sequential Processes (CSP) first described in 1978 by Hoare [136], and the Calculus of Communicating Systems (CCS) developed in 1982 by Milner [204]. Both are labelled transition systems and use the notion of *bisimilarity* to define an equivalence relation between systems (intuitively systems are bisimilar if they match each other's transitions).

The development of process calculi is still active, with major improvements including the π-calculus which extends CCS to model distributed processes [205] and the ambient calculus which models physical mobility [59]. The ideas behind process calculi have been heavily influential in the development of concurrent programming languages such as Occam [155]. The reader of our previous introduction to PROMELA (§2.4.1) will recognise many of its features are derived from Occam's syntax which includes `SEQ` blocks for sequential execution, `PAR` blocks for concurrent execution and `ALT` blocks for non-deterministic guarded execution. As we will see in Chapter 6 Occam was also influential in the development of our SPL state machine language.

Petri nets (also knows as place/transition or P/T nets [235, 213]) are a formal representation of a distributed system first invented in 1962 by Carl Adam Petri [222]. A Petri net consists of a set of places and transitions, and directed arcs which run between them. The input and output places of a transition are the places from and to which an arc runs respectively. Places contain a finite number of tokens which are either consumed or created by transitions "firing". Transitions fire when they are enabled by having a token in every input place. Petri nets execute non-deterministically which means that: (*i*) multiple transitions can be simultaneously fired; and (*ii*) transitions *need not* fire if they are enabled; the interleaving of firing is arbitrary and so verification tools must test all possible combinations to verify safety properties of a Petri net.

Petri nets have been extended in many directions, such as coloured Petri nets which assign values to tokens with optional type restrictions [151], hierarchical Petri nets with support for refinement, abstraction and object models [2], and even an XML-based markup language to support generic data exchange between different Petri net models [67]. The CPN/Tools project [233] is a mature tool-chain for editing, simulating and analysing coloured Petri nets. It uses OpenGL to provide advanced interaction techniques such as marking menus and toolglasses rather than the more conventional WIMP approach [27]. The tool has a number of commercial licensees[1] and is primarily used for modelling existing systems rather than constructing them from scratch. Rather than inventing a new language to capture the computation during transitions firing, CPN/Tools uses a subset of Standard ML, although care must be taken to restrict the complexity of such code or a state explosion makes the analysis of the resulting model difficult.

3.1.2 Model Extraction

Petri nets and process calculi are elegant and precise ways of specifying concurrency, but are rarely used directly in real systems. A more common approach to formal verification is to extract abstract models directly from more complex application source code, and verify properties about the simpler model. The Bandera tool-chain [75] is a collection of components designed to ease the model-checking of Java source code. The components include components for program analysis and slicing, transformation, and visualisation. Bandera accepts Java source code as input and a set of requirements written in the Bandera Specification Language (BSL) [77]. A key design goal of BSL is to hide the intricacies of temporal logic by placing emphasis on common specification coding patterns (e.g. pre- and post-conditions to functions). BSL is also strongly tied to the source program code via references to variables and methods names, which takes advantage of the `javadoc` comment extraction system to explain the BSL specifications as well. Much of Bandera's utility arises from its mechanisms for model construction which provide tools to eliminate redundant components from a model [76], simplifying the eventual output to a model checking language such as PROMELA.

The BLAST [133] project introduced the *lazy abstraction* paradigm for verifying safety properties about systems code. Lazy abstractions follows the following steps: (*i*) an abstraction is extracted from the source code; (*ii*) the abstraction is model-checked; and (*iii*) the model is

[1]A list is available at `http://www.daimi.au.dk/CPnets/intro/example_indu.html`.

then refined using counter-example analysis. The process is repeated until the model is sufficiently refined, and the resulting proof certificates are based on Proof Carrying Code [214]. This mechanism helps make the model extraction process more scalable by reducing the amount of time and effort required to create abstractions of systems code. In contrast to the conventional abtract-verify-refine loop, lazy abstraction builds abstract models on demand from the original source code. This results in a non-uniformedly detailed model which contains just enough detail to show a counter-example to the developer. When combined with a temporal safety automata, the tool either generates a small, quickly verifiable proof certificate, or an error trace indicating the location of the error otherwise. BLAST was applied to low-level systems code, such as Windows and Linux device drivers, discovered several errors and generated automated and small proofs of correctness for these drivers.

In order to specify *observer automata*, BLAST adopts a set of patterns which, when matched against the execution point of the program, trigger a state change in the observer automata. BLAST specifies events which are tied to C function calls in the original source code. CCured [215] is a tool which instruments C code with run-time checks to make it memory safe. BLAST was used to remove as many of these run-time checks as possible in order to narrow the performance gap between the safe and unsafe versions, and also generate execution traces for code that could potentially fail [36].

SLAM[2] is a large project aiming to integrate software model extraction and checking directly into the Windows development kits. SLAM seeks to check whether or not a C application obeys "API usage rules" which specify temporal constraints on sequences of API calls. The toolkit does not require annotation of the source code and infers many invariants automatically. Like BLAST, it simplifies model extraction and slicing through a process dubbed "counterexample-driven refinement" [71, 238]. The SLAM process uses predicate abstraction [19, 124] to create boolean programs (which have all the control-flow constructs of normal C code but with only boolean variables). These abstractions are iteratively analysed and refined until the system is satisfied that no further refinement is necessary. The main practical use of SLAM within Microsoft has been the static verification of hardware device drivers in Windows, and in practice the refinement process has terminated within 20 iterations [18]. The authors of SLAM report that the technique works best for programs whose behaviour is governed by an underlying finite state machine—device drivers clearly fall into this category, as do most network applications which are implementing Internet protocols.

The temporal interface language used in SLAM—dubbed SLIC [21]—is similar to the Bandera Specification Language described earlier, and indeed has much in common with other automaton specification languages in the literature which are concrete versions of Schneider's formalisation of security automata for software [242]. The Property Specification Language (PSL) is a language designed for expressing constraints about hardware designs constructed in Verilog or VHDL. The properties specified create assertions which are passed to hardware verification tools and either dynamically monitored by simulation tools or statically proven by model checkers. PSL, although geared towards hardware model checking, is a concise language for expressing temporal properties in a friendlier manner than LTL, and could also be used to construct never claims for software model checkers such as SPIN (§2.4.2). PSL is currently being standardised by the IEEE P1850 Working Group, and is summarised along with other hardware verification languages in the Bunker et al. survey on the topic [53].

Alur and Wang have tackled the problem of model checking real-world protocols by ex-

[2]See `http://research.microsoft.com/slam/`.

tracting a specification from RFCs and using symbolic refinement checking to verify the model against protocol implementations written in C [8]. They evaluate their technique by creating and verifying models of DHCP and PPP, and conclude that "*[manual model extraction] is unavoidable for extracting specification models since RFC documents typically describe the protocols in a tabular, but informal, format*".

Researchers at the University of Cambridge have completed a 9 man-year project to rigourously specify the semantics of TCP/IP stacks [40]. They recognise the difficulty of the *post-hoc* specification style, and use a combination of manual extraction from RFCs and books (such as Stevens [257]) as well as extensive testing of the *specification* against concrete implementations such as FreeBSD and Linux (a reversal of the usual testing of implementations against specifications). The work required significant advances in mechanised theorem proving, ranging from instrumenting operating systems to authoring an appropriate specification language for HOL and managing the demands of distributed theorem proving. However, this is still an area of active research and much work remains before it is practical for the myriad of real-world Internet protocols.

3.1.3 Dynamic Enforcement and Instrumentation

The model extraction techniques described above require access to application source code; we now discuss systems which can monitor an application from the binary itself.

Sun Microsystems developed DTrace [57] as part of their Solaris operating system as a facility for dynamically instrumenting kernel- and user-level components of production systems without modifying their behaviour. DTrace has thousands of "hooks" into the operating system to allow developers to decide where to instrument their program, and features a C-like automaton language to control when the instrumentation is active. DTrace does not require modification to the source code and can operate on binaries.

Another dynamic enforcement system which operates on binaries is `systrace` [229] which monitors the sequences of system calls to the kernel and can accept, deny or pass the request to a userland policy daemon for verification. The `systrace` policy language is not stateful[3] and it can be difficult to keep applications binaries in synchronisation with the low-level system call policies required by `systrace`.

The Model-Carrying Code (MCC) project led by Sekar combines the model-extraction techniques described earlier with system call interception to provide a platform for the safe execution of untrusted binaries [248]. Untrusted code is bundled with a model of its security-relevant behaviour which can be formally verified against a local security policy by a model checker. The execution of the binary is dynamically verified by system call interception to fit the model and the application terminated if a violation occurs.

As Wagner and Soto point out [279], the low-level nature of system call interception does make it easy for attackers to launch an observationally equivalent attack by crafting a valid sequence of syscalls, and so this technique is only really useful as a last-resort if more formal and reliable verification techniques against the source code cannot be applied. We have drawn inspiration from the work described above, in particular the MCC approach of providing static models and dynamic enforcement, but our work operates at a higher level with explicit support from the application source code.

[3]We proposed using the `systrace` kernel interface with stateful policies in previous work [184].

3.2 Data Plane

The data plane is primarily concerned with processing the bulk of network data with low-overhead and resource requirements. In this section, we describe formal data description languages which map raw network data to higher level structures (§3.2.1), active networks which execute mobile code to process data (§3.2.2), and the view-update problem which relates to converting to and from concrete and abstract representations of data (§3.2.3).

3.2.1 Data Description Languages

One of the early innovations in language research was `Yacc` (or "Yet Another Compiler-Compiler") tool [154]. Stephen Johnson recognised that the inputs to computer programs generally have some structure, and created `Yacc` as a way of simplifying the tedious and error-prone task of checking all input tokens for validity. `Yacc` accepts a specification of a grammar and outputs a table-driven automaton which is driven by inputs of lexical tokens and outputs an abstract syntax tree of the language. Although `Yacc` cannot handle the complete set of context-free grammars [252] as it uses a Look Ahead LR (LALR) parser, it strikes a good middle ground between the set of grammars it can handle and the size of the resulting automata. It has been used to specify the grammars of many languages since, including ANSI C [175], and has been ported to other languages such as Scheme and OCaml.

Analogous specification languages for transmitting network data also exist. Abstract Syntax Notation One (ASN.1) is a formal notation used for describing data structures transmitted by communications protocols, independently of the underlying transmission medium or implementation language. An ASN.1 specification can express low-level fields such as integers, booleans or strings, as well as higher level constructs such as sequences, lists, choices, etc. Sub-typing constraints and versioning information may also be added, and specific *encoding rules* applied which define transmission format—popular rules include the Basic Encoding Rules (BER), and the Packed Encoding Rules (PER) optimised for low-bandwidth channels. ASN.1 was first standardised in 1984 by the CCITT, and subsequently chosen by the ISO as standard notation[4]. However despite the standardisation and a high degree of acceptance from other industry sections, ASN.1 was not adopted as the encoding format for many of the original Internet protocols such as IP, UDP and TCP. This is a historical choice, as it was difficult of fit a complete ASN.1 parser on the embedded IP routers at the time the protocols were being developed. The requirements for integration with external data sources has meant that ASN.1 is often mentioned in RFCs, notably in directory protocols [121, 290, 296], Voice-over-IP [55, 41], and security mechanisms such as SSL/TLS [93, 56].

CORBA is another method for safely exchanging data structures over the network, developed in the early 1990s as the Object-Oriented (OO) programming model was gaining popularity (§3.3.1). CORBA uses an Interface Description Language (IDL) to precisely specify the external interfaces exposed by object components. A mapping is also defined between this IDL and the target language (e.g. C/C++/Java) which defines how CORBA data types translate into structures in the implementation language. The first version of CORBA was released in 1991, and was relatively simple (it defined the core set of APIs for remote invocation and included a single language mapping for C). However, as CORBA evolved to provide a language and platform-neutral specification for performing Remote Procedure Calls (RPCs), it grew to include aspects of security, transactions and real-time guarantees. Today CORBA is an extremely

[4]In 1987, ISO published ASN.1 and BER as documents 8824 and 8825.

feature-rich (albeit complex) mechanism to create applications which require distribution component communication.

CORBA and ASN.1 are not suitable for the task of Internet protocol processing, since although they can specify data structures to match those used by Internet data structures, it is more difficult to match the wire formats used by those protocols without defining a new encoding format per protocol. Data Description Languages (DDLs) are often used to perform the opposite function of converting a *physical* representation into a *logical* structure. DDLs tend to be specialised for a particular task; e.g. PACKETTYPES [189] and DATASCRIPT [12] are specialised to parsing protocols and Java jar-files respectively.

PADS [109] aims to cope with truly ad-hoc data by providing mechanisms for error recovery and support for non-binary data. Typically, consumers and producers of ad-hoc data formats create bespoke tools to manipulate these formats, and a lot of time is spent parsing the data instead of concentrating on the information it contains. PADS provides a declarative data description language language which is error-aware by maintaining an association with a data stream and a description of the errors in that stream. More recently in 2006, Fisher et al. have defined a formal semantics for DDLs [110].

An early stub compiler was USC [219], which provided an extended form of ANSI C to succinctly describe low-level header formats and generate near-optimal C code to parse them. McCann and Chandra proposed PACKETTYPES as a language for specifying network protocol messages [189]. A PACKETTYPES specification consists of a series of named fields and types, and packet constraints are declaratively specified separately. The specification is translated into C code which can be used as a library to receive and transmit well-formed packets with respect to the PACKETTYPES specification for that protocol. USC, PACKETTYPES and most data description languages output C code, unlike our MPL compiler which outputs efficient, high-level code directly in statically type-safe ML.

Prolac is a statically-typed, object-oriented language used to create readable, extensible and practical protocol implementations [171]. Kohler et al. also report on a modular TCP implementation derived from the 4.4BSD stack and split up into minimally interacting components. The Prolac compiler performs whole program optimisation to eliminate cross-module dynamic dispatch, and the TCP implementation was found to have similar end-to-end performance to a Linux 2.2 stack. However the code is no longer maintained and no other protocols aside from TCP appear to have been implemented using Prolac.

Dabbous et al. implemented a compiler which converts a specification written in Esterel[5] into highly optimised C code [88]. The compiler optimises the common code paths for performance, and uncommon code paths for smaller size. Remarkably, their evaluation of a subset of TCP compiled using their system versus a similar subset of the standard BSD stack led to code which was 25% faster and only 25% larger in code size. They conclude that "*there is no intrinsic performance penalty incurred when compiling from a high level protocol description*". However as the Ensemble authors also note [131], this approach does not scale to more complex higher-level protocol stacks due to the difficulty of creating and maintaining large specifications.

3.2.2 Active Networks

The Packet Language for Active Networks (PLAN) [134] is a language intended for programs running over a programmable network. Rather than parsing existing protocols, PLAN seeks to *replace* packet headers with programs with limited functionality. PLAN also has a

[5]Esterel is language designed for reactive systems, with excellent support for hardware synthesis [34].

foreign-function interface (known as *service routines*) which can be invoked when a PLAN program needs greater expressivity and power. PLAN is based on the simply-typed λ-calculus, is strongly statically type checked, and can guarantee that the execution of a PLAN program uses a bounded amount of network resource. The drawback is that a PLAN system requires network level support and thus cannot interoperate with existing protocols over the Internet.

SafetyNet [280] is a language designed to safely implement a best-effort distributed network, and uses a strong type system to enforce network policy such as resource usage, webs of trust and security. It uses advanced static type systems such as linear types [170] to move beyond the memory safety guarantees provided by traditional static type systems.

Menage developed the Resource Controlled Active Node Environment (RCANE) [195] to facilitate executing untrusted code in a network. RCANE supports scheduling, resource (I/O and CPU) accounting and garbage collection on a network node, and was implemented using an early version of OCaml. Menage notes OCaml had a number of deficiencies when used as an active networking node. Many of these have since been corrected in more recent versions, such as dynamic byte-code loading, just-in-time compilation of byte-code and a well-documented foreign-function interface.

In their analysis of the cost of security across multiple active networking implementations [6], Alexander et al. describe several low-level applications such as the "active ping". It is interesting to note that despite their use of advanced programming languages and type systems, active networking research tries to remain compatible with low-level packet formats such as IPv4, but does not seek to recreate more complex application-level protocols such as DNS or SSH.

3.2.3 The View-Update Problem

The *view-update problem* expresses the difficulty of representing a data structure in an abstract form in such a way that any changes made to the view will be reflected back to the original structure. Although this is a classical topic in the database literature, programming language research is only recently beginning to tackle the problem. A popular area is the two-way manipulation of tree structured data (e.g. XML) using strong static typing to eliminate run-time type errors, and ensuring that output is always valid with respect to an XML schema.

Two examples of this work are XDuce [141] and CDuce [30]. XDuce (pronounced "transduce") features (*i*) regular-expression types, similar to XML Document Type Definitions (DTDs); (*ii*) powerful subtyping over regular-expressions; and (*iii*) regular expression pattern matching over the tree structure, combining conditional guards, tag checks and extraction and iteration over sub-nodes. CDuce, inspired by XDuce, added several features familiar to functional languages such as (*i*) a richer semantic definition of subtyping, allowing the integration of first-class functions, boolean connectives, and open or closed record types; (*ii*) a backtrack-free sub-typing algorithm; and (*iii*) ML-style pattern matching for XML [31] with efficient code output driven by type information [114].

Although similar, XDuce and CDuce have since developed in different directions. XDuce has added mixed attribute-element types and powerful filtering operations. CDuce was integrated directly into OCaml, allowing normal OCaml programs to be augmented with "x-types" representing XML types, pattern matching across them, and converting back and forth between OCaml types and x-types.

Foster et al. developed the notion of bi-directional tree-structured synchronization as part of the Harmony [112] project. Harmony is a framework for reconciling disconnected updates to heterogeneous and replicated XML data. Examples include the synchronization of the bookmark files for several different web browsers, allowing bookmarks to be edited and reorganized by different users running different browser applications on disconnected machines while maintaining their organisational structures.

A theme of Harmony is to develop the foundations of *bi-directional programming languages*, in which every program denotes a pair of functions; one to create a *view* of a data structure, and another to update that view in a consistent manner. They term these two functions as the *get* and *putback* functions [113] respectively, and note that the difficulty lies in balancing the complexity of the *get* functions such that the complementary *putback* functions also exist. They discuss a combinatorial approach to ensure the consistency and ease of specifying these function pairs.

3.3 General Purpose Languages

General-purpose languages are not specialised to either a control or data plane, but rather act as the glue between more formal mechanisms and the operating system. In this section we look at some related work relevant to our thesis, starting with approaches from software engineering to construct reliable systems (§3.3.1), the emerging field of generative meta-programming (§3.3.2) and finally the inspirational work for much of our research by examining previous uses of functional languages for constructing networking applications (§3.3.3).

3.3.1 Software Engineering

There are two broad approaches to constructing software using a general-purpose language: (*i*) the "top down" approach which emphasises planning and a complete understanding of the system; and (*ii*) the "bottom up" method which creates low-level components and assembles them into a complete system. The top down approach requires exact specifications, and the system can only be tested at a very late stage. Conversely, the bottom up approach encourages low-level functional units to be created, tested and shared between multiple components. Both of these methodologies have benefits and drawbacks, and most pragmatic solutions are somewhere in between them.

Top down programming was a popular technique in the 1960s and 1970s. Harlan Mills developed an influential mathematical foundation for structured programming [201, 202], and Niklaus Wirth (the designer of the Pascal programming language) described the stepwise refinement technique [285] as a "sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures". Dahl, Dijkstra and Hoare discuss both structured and object-oriented programming in their 1972 book "Structured Programming" [89]. However, as software engineering became more commonplace and complex, and time-scales for development shortened, top-down programming began to lose favour. Reasons included the intolerance of top-down programming to changes in the specification (which often require a complete re-design), and also the difficulty of re-using code developed by a top-down project in other areas.

Object-oriented (OO) programming began to gain popularity in the late 1990s with the rise of C++ and Java. The OO style encouraged the bottom-up style, where low-level components are divided into objects which are composed together into complete systems. This gave developers more flexibility with respect to the final specification of the program, and also to share object libraries between applications. The associated formalisms to support the programming style were also rapidly developed [1, 116]. Unfortunately, bottom-up design can lead to unreliable large-scale systems, as unless the object interfaces are completely specified and understood (rarely the case), their composition in different applications can lead to subtle, hard-to-find bugs.

Both methods are still active research areas; for example Dijkstra's weakest precondition calculus [94] has been extended into the Refinement Calculus by Ralph-Johan Back [13, 14] and Carroll Morgan [211]. The Refinement Calculus formalises a step-wise refinement by a series of correctness-preserving transformations into an executable program. Formal proof assistants such as Coq [146] can transform proofs directly into executable programs such as OCaml (including programs which are normally untypable in ML, but are safe since they have been verified by the theorem checker). Xavier Leroy has recently documented his experiences with constructing a certified compiler using this technique [179].

Techniques to verify the soundness of interactions between components written in unsafe languages such as C are a more active area since many of the critical Internet attacks described earlier are due to bugs of this nature. For example, Engler has adopted statistical model checking techniques [102] and compiler extensions [101] to find bugs in millions of lines of C code. PSL[6] is a framework for specifying the dynamic and architectural properties of component-based systems [174]. It works in a co-inductive fashion by capturing the *incompleteness* of a system and specifying rules to eliminate behaviours instead of allowing them. This means it lacks familiar constructs found in closed-world models[7] such as step operators or frame axioms (which assert that unmentioned properties remain unmodified between state transitions). PSL provides support for refinement and generalisation to strengthen or weaken specifications, also a specialised version can be used to inter-operate with CORBA-based middleware.

3.3.2 Meta-Programming

Meta-programming is the approach of using programs which create further programs specialised to a particular task. Meta-programming is most commonly found in compilers, which accept a program specification (e.g. C code) and output an equivalent version in a lower-level language (e.g. assembly language). This is also known as *generative* meta-programming, but some very dynamic languages such as LISP, Python or Ruby permit the modification of a program at run-time and thus eliminate the generation step.

Meta-programming allows code to be parameterised over various design choices at compilation time and output code which is specialised to the task at hand, but also guaranteed to type check for all possible generated programs. The Fox project first investigated extending ML with run-time code specialisation, via a subset of ML which dynamically recompiled itself with a minimal performance cost [283, 176]. The DDLs described earlier (§3.2.1) can be considered as an instance of meta-programming with very specialised type systems dedicated to packet parsing or ad-hoc data formats. Walid Taha and his team are developing a more general type-safe multi-stage programming solution with MetaOCaml [260], which modifies OCaml to permit dynamic code generation via a set of syntax extensions.

3.3.3 Functional Languages for Networking

The inspiration for much of the work in this thesis stems from the FoxNet project [38, 37], which implemented the TCP/IP stack using Standard ML. The implementation made good use of SML/NJ features such as parameterised modules (known as *functors*) to separate out protocol elements into a series of signatures. A combination of these protocol signatures resulted in a TCP/IP implementation. Other protocols combinations were also possible such as TCP over

[6]Not to be confused with the Property Specification Language described earlier.

[7]Closed world models include most of the formalisms described so far such as process calculi or the Kripke models used by SPIN.

Ethernet; a useful exercise which revealed layering violations in the design of TCP/IP (e.g. the "pseudo-header" used to calculate the TCP checksum depends on the source and destination address from the IP layer [225]). FoxNet was one of the first attempts to apply a functional language to a low-level "systems" problem such as network protocol implementation.

It is interesting to examine some of the issues they faced [38], and design decisions they took versus our own. FoxNet used a modified version of the SML/NJ compiler which supported 1-, 2- and 4-byte unsigned integers and in-lined byte arrays and the associated logical shift operators. This modification is not necessary in modern OCaml as it natively supports integers of 2-, 4- and 8-bytes and strings can be used to represent byte arrays. However, the OCaml support for these types is syntactically much more difficult to use when compared to native integers. The FoxNet code is difficult to compile on a modern computer due to their compiler modifications and although they supply the compiler source code, it is dated and only supports code output on Alpha and MIPS processors (increasingly rare in the modern x86-dominated world). We discuss how we avoid these problems later in our design goals (§4.1).

FoxNet also use the SML/NJ extension for first-class continuations, enabling a co-operative threading model by regularly yielding to a `Scheduler` functor which dealt with multiplexing connections. OCaml is very different from SML/NJ with regards to its internal implementation; it is based on the ZINC machine and uses currying instead of tuples to represent function calls [178]. As a consequence, continuations are difficult to support efficiently in OCaml (although Xavier Leroy has created a naïve bytecode-only version which copies the entire stack [180]). In practice, we have found that fine-grained threading or continuations are not an essential component of networked servers; we rely instead on a high-level continuation style (by capturing connection state using a variant data type) and asynchronous I/O to ensure that the server does not block while waiting or transmitting data.

FoxNet primarily implemented a TCP/IP stack and a simple HTTP server to serve web pages. Web servers are a fairly common target for implementation in functional languages; Marlow developed one in Concurrent Haskell [187], and the SMLserver is a AOLserver plug-in which serves dynamic web-pages written using MLKit [99]. To our knowledge, there have been no attempts to create servers for more complex protocols such as SSH, DNS or BGP using functional languages.

Another large networking project which used OCaml is the Ensemble network protocol architecture [271, 131, 88]. Ensemble is designed to simplify the construction of group membership and communications protocol stacks. A system is constructed by composing simple *micro-protocol* modules which can be re-arranged in different ways depending on the exact needs of the underlying application. Some examples of micro-protocols include sliding windows, fragmentation and re-assembly, flow control, and message ordering. Ensemble is particularly interesting for us due to its use of OCaml, which it switched to from using C in its previous version (known as "Horus").

In Chapter 4 of his PhD thesis [131], Hayden discusses the impact of using OCaml and notes that reducing memory allocation is a key concern. He also reports that using the C interface led to hard-to-track bugs, confirming our approach of attempting to attain high performance without resorting to using foreign-function bindings. Ensemble compared favourably to Horus in terms of latency, lines of code, and performance, and the use of OCaml eased the integration with the Nuprl proof system to optimise micro-protocols automatically, formally and correctly [88].

Unlike our work, Ensemble does not seek to implement existing protocols, instead serving as an effective testbed for research into new distributed communications protocols. It uses the

OCaml `Marshal` module to transmit data structures over the network, which is not type-safe and can lead to program crashes if data is corrupted or tampered with[8]. In contrast, we use MPL to precisely specify the wire format of traffic sent or received to conform to existing Internet standards.

FoxNet and Ensemble are the two most major projects which use functional languages to deliver elegant, secure protocol implementations. Other systems research is concentrating on eliminating the user/kernel divide by exporting functionality in general-purpose operating systems into user-land. Gunawi et al. implemented icTCP which exposes key elements of internal TCP state to user-land applications with minimal changes to kernel code [127]. Although they acknowledge the benefits of structured protocol stacks such as FoxNet, they do not (yet) use functional languages as a regular part of their development.

3.4 Summary

This chapter has surveyed the body of related work that is relevant to the construction of reliable and secure network applications using both formal and informal methods. We categorised the body of work into those techniques dealing with an abstract control plane (§3.1), a high-bandwidth data plane (§3.2) and general-purpose programming languages which glue them together in software architectures (§3.3).

[8]Shinwell et al. are working on an extension to OCaml featuring type-safe network marshalling [251].

CHAPTER 4

Architecture

Everything is vague to a degree you do not realize till you have tried to make it precise.
BERTRAND RUSSELL

In this chapter we describe the basic design goals behind our research (§4.1) and the concrete architecture and research contributions which resulted from them (§4.2). The design goals are motivated by lessons learnt from the related work described earlier (§3) in our background survey about threats facing the modern Internet (§2.1), and our desire to solve this problem by deploying safe yet practical applications written in high-level languages (§2.2).

4.1 Goals

We noted earlier that the vast majority of critical Internet infrastructure hosts are running applications written in C despite decades of research into safer programming languages (§1.1.3) and have been vulnerable to numerous security vulnerabilities as a result (§2.1). In this section we define the goals which our architecture must meet, firstly by discussing the nature of data abstractions inside modern operating systems (§4.1.1) and secondly by discussing the language support needed to construct applications in a modern high-level language such as OCaml (§4.1.2).

4.1.1 Data Abstractions

Modern operating systems place great emphasis on the efficient handling of network traffic in order to enable applications to transfer data with high throughput and low latency overhead. The OS can also provide other services such as process isolation and protection, reliable storage and resource reservation guarantees. When the application interfaces were first developed in the 1960s for early operating systems, CPU time was a scarce resource compared to available memory. This situation has reversed in recent years as memory bandwidth is relatively low compared to the fast CPU speeds [198].

In order to minimise the high cost of memory access most operating systems provide abstractions which discourage the copying of data, instead modifying it in place. For example, the BSD network stack keeps track of network packets as `mbufs` which represent the payload

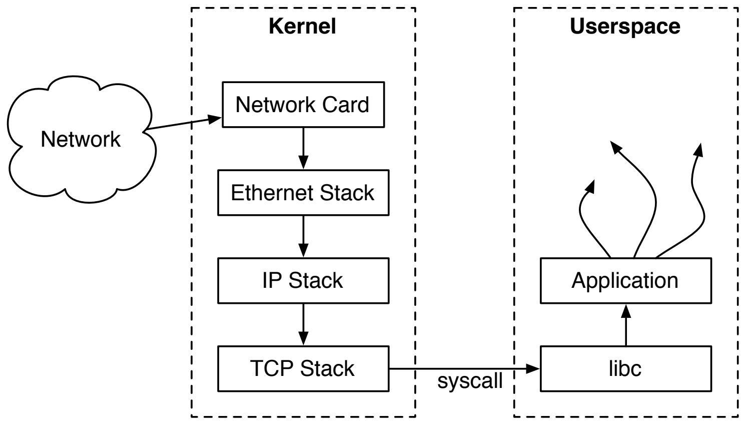


Figure 4.1: The data flow of traffic from the physical network through the kernel and finally into user-space where the buffers are sent to the application.

and headers of a network packet (represented internally as a single buffer or a chain of multiple buffers) [194]. Linux also uses a similar mechanism known as `skbufs`.

Figure 4.1 illustrates the flow of data from the physical network through the code paths in a kernel and into user-space. On the receive path, data is most often copied once from the network interface hardware into an appropriate kernel structure (e.g. `mbuf` or `skbuf`) and passed by reference into the various levels of kernel protocol stacks. When processing is complete the data is transferred to user-land to make it accessible to the application (in reality, stream protocols such as TCP place the data into a buffer from where it is later retrieved by the application).

A lot of systems research has focussed on making the data flow within the kernel to the application a "zero-copy" process to avoid making memory bandwidth a bottleneck, for example in TCP/IP stacks [68], virtual memory management [82] or I/O sub-systems [167]. The Berkeley Packet Filter [190] was developed to permit user-level processes to request complex filtering policies from the kernel while avoiding the overhead of copying every packet into user-land to inspect its contents. The filtering language is designed around a register-based machine and recent improvements in implementation have improved the performance of the system considerably [28].

In order to maintain performance the application must also minimise the copying of data once it has been retrieved from the kernel. In a language such as C this is normally accomplished by passing around pointers to structures containing the data. Unfortunately, higher-level languages featuring automatic garbage collection tend to abstract data structures away from the programmer, and it can be difficult to know when a value has been copied or merely referenced. Data copying is particularly pronounced in purely functional languages and the effort to eliminate redundant intermediate structures is known as "deforestation" [277]. OCaml does permit a finer control over when data is copied or referenced, but this requires a more verbose and imperative style of programming as opposed to the slower but more elegant functional approach it also supports (§4.1.2).

For some classes of applications which are short-lived and perform a lot of symbolic computation (e.g. compilers) this extra copying is not important. However it is a critical distinction for network applications which process a large number of data structures over a long period of time. A primary goal of the research in this dissertation is to make it easier to construct a "control and data" distinction inside network applications written in a high-level and safe language,

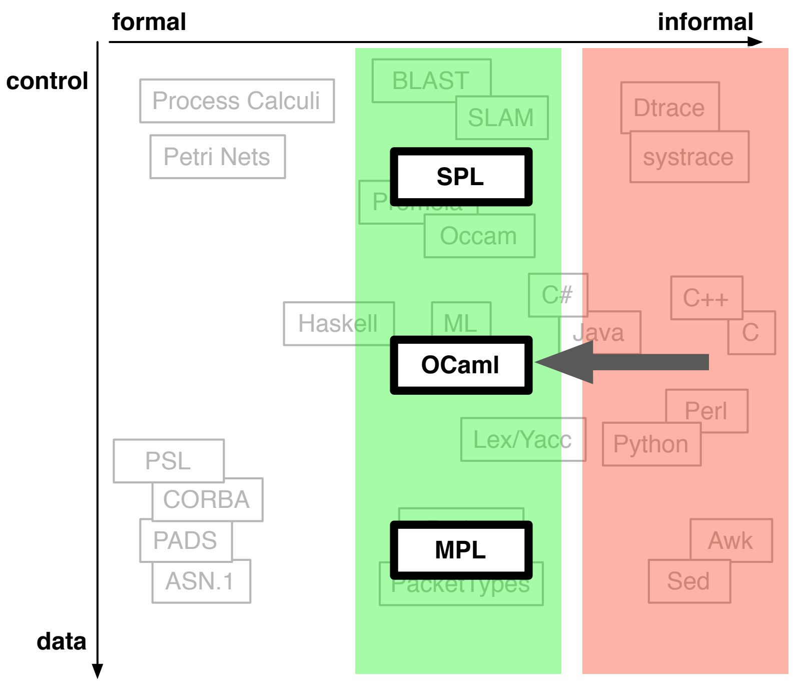


Figure 4.2: Diagram showing where our research fits into the design space. The red block indicates how most existing Internet applications are constructed and the green shows the space we are moving into.

as we shall see later in this chapter (§4.2.1).

4.1.2 Language Support

During our survey of related work, we created a diagram of the design space in Figure 3.1 which classified techniques by their formal rigour and level of abstraction. Formal methods are a valuable tool for verifying correctness properties of complex systems, especially with respect to security-related aspects, but are currently very under-used in real systems.

The overall goal of our research, shown in Figure 4.2, is to construct a software architecture which integrates the more practical formal methods into a *complete system* which does not sacrifice the performance and portability aspects of currently deployed servers. In particular, there is an incorrect but widely-held belief that the use of high-level languages with automatic garbage collection results in network applications running more slowly and unpredictably when compared to their equivalents constructed in C. As we will show later in our evaluation (§7) by carefully constructing applications with a control and data abstraction in mind, the opposite can also hold true!

Another element of our thesis is practicality—we wish to create a solution that does not depend on a modified tool-chain which will rapidly become deprecated once the research has been completed. We noted earlier that the code from the FoxNet project, although freely available, is difficult to compile up and use due to its dependence on a modified SML/NJ compiler. Thus, another design goal is that our architecture must work with a mature and well-tested language and tool-chain. Our choice of language is OCaml, for the reasons described in our background chapter (§2.3).

OCaml, as with most general purpose programming languages, is not ideal for expressing strict data and control plane abstractions. On the data plane, although it supports manipulation functions for variable-sized integers (essential for handling binary protocols), it does not provide polymorphic function operators across them. OCaml distinguishes between native integers (type `int`, e.g. `53`), 32-bit integers (type `int32`, e.g. `53l`) and 64-bit integers (type `int64`, e.g. `53L`). Since these types are distinct, they must be explicitly converted via library functions to the other representations. This means that the infix $(+)$ operator has type ($\mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int}$), which will not type-check against 32- or 64-bit integers. Separate modules are provided which implement the equivalent functions for other integer types; e.g. `Int32.add` with type ($\mathtt{int32} \rightarrow \mathtt{int32} \rightarrow \mathtt{int32}$).

```
Int32.sub (Int32.add 1l 2l) 1l ;;                                    OCAML
# val z : int32 = 2l
let (++) = Int32.add ;;
# val ( ++ ) : int32 → int32 → int32 = <fun>
let (−) = Int32.sub ;;
# val ( − ) : int32 → int32 → int32 = <fun>
1l ++ 2l − 1l ;;
# val z : int32 = 2l
```

The result is that manually constructed code to deal with network protocols is rather verbose and cumbersome. In addition, performance requirements dictate that protocol handling code minimises dynamic memory allocation in order to reduce the load on the garbage collector, meaning that the code for parsing network protocols is by nature highly imperative and vulnerable to human error.

For the control plane , we wish to deploy more formal model checking tools which can verify safety properties about the reactive state machines which form network protocol servers. Extracting abstract models from a functional language, although easier than from C due to the memory safety guarantees, is still not an automatic process. As we noted in our related work on model extraction (§3.1.2), changing the source code as an application evolves can have drastic effects on a generated model and require re-writing safety properties against it. This inevitably leads to the safety properties "bit-rotting" as they are kept in synchronisation with the source code in a rapidly developing project.

Secondly, OCaml does not provide an easy way to specify complex state machines—its native pattern matching is extremely powerful for iterating over data structures, but this rapidly becomes confusing when dealing with complex, interconnected graphs which are unrolled into long sequences of pattern matches. Writing these state machines in a more concise, non-deterministic form would be preferable for readability and maintenance.

To summarise our goals, we wish to create high-performance network servers which use OCaml to benefit from its flexibility and safety via static typing, but: (*i*) not require any compiler or language modifications; (*ii*) avoid writing verbose, error-prone, low-level packet parsing code; (*iii*) express high-level state-machines which can be fed into a model checker; and (*iv*) create a mechanism for keeping the state-machines and abstract models in synchronisation as the source code evolves.

4.2 The MELANGE Architecture

The MELANGE network application infrastructure is illustrated in Figure 4.3, consisting of two domain-specific languages built around OCaml. The main application consists of several discrete components which are output by the MELANGE tool-chain. Firstly, the MPL compiler accepts a protocol specification and outputs OCaml protocol parsing code (§4.2.1). Secondly, the

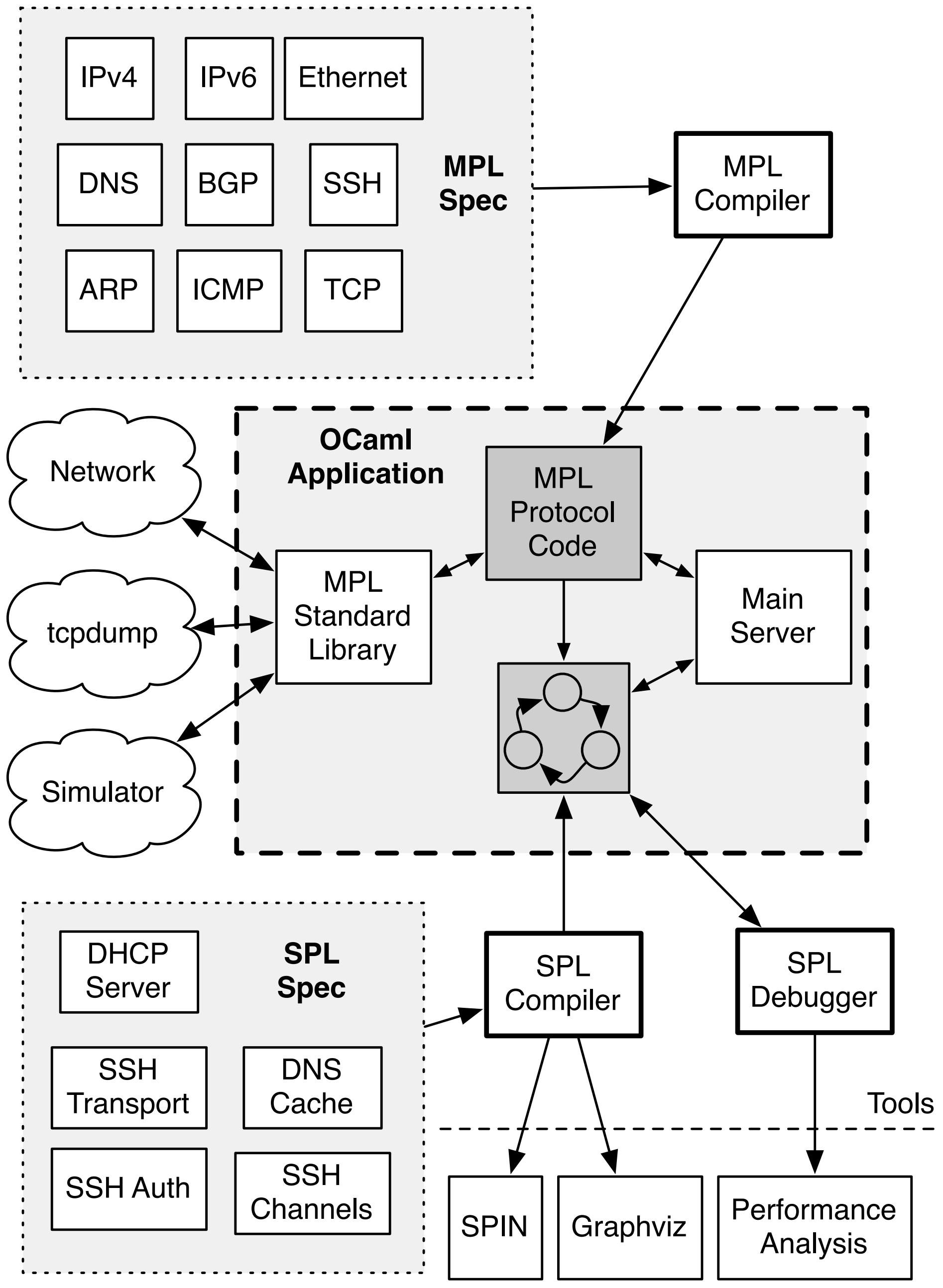


Figure 4.3: The MELANGE architecture for OCaml servers. The shaded boxes in the OCaml application represent auto-generated code.

SPL compiler translates the protocol state machines into an OCaml *inline automaton* (§4.2.2). Finally the main application, written in any style most convenient to the programmer, is linked in with these components and the MPL standard library to result in an executable.

Both the MPL and SPL compilers are instances of generative meta-programming compilers (§3.3.2) and are both written in OCaml and output further OCaml code, thus meeting our first goal that no compiler modifications are required to enable practical wide deployment. By using MPL, the main application does not have to deal with details of packet wire-formats directly, instead manipulating high-level data structures which abstract the details into functional objects. The SPL compiler provides a succinct front-end language to express non-deterministic finite state automata which can be dynamically enforced in the main application and statically verified via a model checker.

The MELANGE architecture makes the following specific contributions: (*i*) the use of a data description language which outputs type-safe, highly structured code instead of C for the purposes of efficient and safe packet parsing in a high-level language; (*ii*) the notion of specifying abstract models which are both dynamically enforced efficiently in the main application and statically verified by formal tools[1]; and (*iii*) the development and evaluation of complete servers for complex protocols such as SSH and DNS around this architecture which demonstrate that these languages can have equal or higher performance than their equivalents written in C, while also maintaining static type-safety and the ability to model check aspects of the application.

4.2.1 Meta Packet Language (MPL)

MPL is a *data description* language—analogous to `yacc` for language grammars—which accepts a specification for the wire format of a network protocol and outputs OCaml code which can efficiently parse and create those protocol packets in a type-safe fashion. The high-level OCaml data structures output by MPL are efficient and maintain only a single copy of the packet data. At the same time, they take full advantage of OCaml language features such as the object system and polymorphic variants to support an elegant functional programming style. For example, received network traffic can be classified using ML pattern matching, and packets can be partially created as curried functions. In order to take full advantage of the OCaml type system, MPL generates a unique OCaml type for each packet type in the specification, and automatically inserts values which can be inferred from other packet fields (e.g. length fields are calculated from their associated buffers and need not be specified when creating a packet). Developing the equivalent OCaml code by hand would be tedious and error-prone, as the MPL specification is significantly more succinct than its associated OCaml code.

MPL specifications can parse (or create) a packet by delimiting it into a series of *fields*, and then optionally classifying further parsing behaviour based on the contents of that field. Field definitions consist of several built-in types such as bit-fields, bytes, 16- 32- and 64-bit integers. MPL differs from many other data description languages by permitting custom fields to be defined which allow arbitrary parsing code written in OCaml. This enables real-world protocols with complex fields to be parsed using MPL, such as SSH (featuring multiple-precision integers for cryptography) or DNS (with its pointer-based parsing of strings). We cover these in our case studies in Chapter 7.

By exclusively using MPL to handle network traffic, an application guarantees that it only

[1]As we noted earlier (§3.1.2), tools exist which either do this at the system-call level (e.g. Model Carrying Code) or via model extraction (e.g. BLAST or SLAM) which make it difficult to specify a constant abstract model. Our contribution integrates this technique directly into an application.

sends and receives well-formed packets with respect to the packet specification; any violations will be caught at compile-time by the ML type system. Also, the OCaml code output by MPL does not directly communicate with the network; instead it interacts with an MPL standard library which abstracts away the specifics of the network communication. The MPL standard library supports communication via the network, a `tcpdump` format file, or directly linking in with a network simulator.

Although we focus on OCaml in this thesis, it is important to note that MPL can be easily modified to output code in other languages. An MPL backend could be structured as a portable C packet parsing core, with foreign function bindings to languages such as Haskell. However, we seek to implement as much of our application in OCaml as possible to ensure a consistent base-line of type safety. MPL is further defined in Chapter 5.

4.2.2 Statecall Specification Language (SPL)

There is no clear choice of programming styles when deciding how to express a protocol state machine in OCaml. Programmers familiar with C may elect to implement an imperative `if/then/else`-style machine. Functional programmers may use a continuation passing-style, and those who prefer an object-oriented approach might elect to use object design patterns [116]. Conversely, programmers more familiar with theorem proving could convert formal specifications into executable code using a proof assistant [179]. Each of these mechanisms has benefits and drawbacks; the ideal solution varies between applications and how critical the correctness of the final program is.

Regardless of the mechanism chosen, programming in OCaml does not automatically result in an application for which formal reasoning is easier beyond the type system guarantees; indeed, tracking down protocol bugs can be *more* difficult due to the presence of higher order functions into which control flow can "escape". There are two broad approaches to formal verification of the application: (*i*) begin with a formal specification and convert it into executable form using a proof assistant such as Coq [146]; and (*ii*) write the application as normal, and perform model or theorem extraction from the source code. Both of these options are currently active research topics and certainly not ready for the casual programmer who is not familiar with the usage of theorem provers.

However, for a large class of network applications, we do not wish to formalise the *entire* application; rather, there are certain key aspects which, if verified, are sufficient for the purposes of eliminating a large class of bugs. To allow this, we define a state machine specification language—dubbed SPL—which allows developers to specify models in terms of allowable program events (e.g. sequences of network packets). A compiler translates SPL into a non-deterministic model checking language (e.g. PROMELA), and executable code (e.g. OCaml). The generated PROMELA can be used with a model checker such as SPIN to verify safety properties of the automaton. The OCaml code provides a safety monitor which, when linked in with a program, ensures that the application behaviour does not deviate from the specified model.

Our approach offers a number of benefits: (*i*) the entire application does not need to be formally specified beyond being written in OCaml, as the critical portions can be abstracted out into separate automata; (*ii*) the models being verified are guaranteed to be dynamically enforced in the application[2]; and (*iii*) the executable models embedded in the application permit high-level debugging at run-time. Conventionally, safety monitors for applications written in unsafe languages must execute in a separate process to guarantee isolation from the main ap-

[2]Thus overcoming the model equivalence problem described in §2.4.3

plication [230]. This introduces a performance penalty due to the overhead of inter-process communication, as well as additional complexity. In contrast, the OCaml safety monitors output by SPL use the static type system to guarantee that the main application cannot interfere with the internal state of the safety monitor. This means that it can execute in the same process as the main application, reducing the task of monitoring program events to simple function calls. Thus, we dub the OCaml safety monitor an *inline automaton* which enforces the SPL specification with very little additional overhead.

The use of SPL is not without drawbacks however. Firstly, the safety monitor terminates the application by raising a software exception if it enters a bad state. Although this clearly not appropriate for some applications—nuclear control plants or aircraft systems—we judge it suitable for the network applications we are creating, since the protocols they communicate with are generally very tolerant to failure (e.g. due to link failure). It is certainly better to terminate the application rather than let it transition into an undefined state which possibly leads to a security compromise or incorrect data being transmitted. Since the error raised is a normal OCaml exception, it can also be caught by the application and dealt with appropriately (e.g. terminate a particular connection instead of all sessions).

Secondly, the main application needs to drive the safety monitor with messages (dubbed *statecalls*) to allow it to progress. If the application does not reliably transmit these messages, then the inline automata will be dormant. The MELANGE architecture in Figure 4.3 provides integration between MPL and SPL for this reason. As packets are transmitted and received via the auto-generated MPL interface, they can *automatically* trigger the appropriate statecall into the SPL automaton. All the programmer needs to do is to provide a simple higher-order function which performs the "statecall routing" into the application. As we show later in our evaluation of an SSH server (§7.1), this integration is sufficient to capture a number of important security properties of an application.

Finally, although the inline automata will enforce the SPL specification, the programmer needs to ensure that the SPL and the actual application code express the same state machine, or violations will occur. The use of SPL guarantees that the application will follow the specification or terminate, but does not make any assertions regarding the *quality* of the application (i.e. it could always just terminate). Xavier Leroy makes a similar distinction between implementation quality and correctness in his work on constructing a certified compiler [179]. This is largely solved by following standard testing methodologies; the main value of SPL automata are in catching rare edge cases (e.g. resulting from network timeouts) not detected through testing.

Although not immediately interesting from a research angle, the SPL compiler also offers useful visualisation functionality by: (*i*) outputting DOT code which can be graphically visualised by tools such as Graphviz [161]; (*ii*) an SPL debugger can attach to a running MELANGE program and obtain a variety of statistics and correlate its current state to the SPL source specification (a screen-shot is shown in §7.1.4). This is a much higher level of debugging than the usual function-trace information obtained from conventional debuggers such as `gdb`, and is particularly important when using OCaml which supports anonymous lambda functions which can make the use of normal debuggers harder than when using C.

The SPL language syntax is intended to be more imperative than other model-checking languages, although this is primarily a matter of style. For example, below is a locking automaton specified in the BLAST query language [35] (§3.1.2) and in SPL.

```
GLOBAL int locked;
EVENT {
  PATTERN { $? = init(); }
  ACTION { locked = 0; }
}
EVENT {
  PATTERN { $? = lock(); }
  ASSERT { locked == 0 }
  ACTION { locked = 1; }
}
EVENT {
  PATTERN { $? = unlock(); }
  ASSERT { locked == 1 }
  ACTION { locked = 0; }
}
```

```
automaton lock (bool locked)
{
  multiple {
    either {
      Init;
      locked = false;
    } or (locked) {
      Unlock;
      locked = false;
    } or (!locked) {
      Lock;
      locked = true;
    }
  }
}
```

For full details on SPL, please refer to Chapter 6.

4.3 Threat Model

It is essential to define a threat model to understand the security risks which our new architecture protects hosts from. For example, writing applications in OCaml will not prevent a malicious attacker from physically assaulting a computer to shut it down[3]. In this section we classify several attacks from the literature with respect to the MELANGE architecture.

Buffer Overflows: These result from a lack of dynamic bounds checking over blocks of memory and can result in arbitrary code execution (§2.1). All pure OCaml code is guaranteed to be safe from buffer overflows as long as certain unsafe features are not used, specifically the `−unsafe` compiler option, the `unsafe_put` function for strings and the `Obj.magic` function which bypasses the type system and is intended for use by code generated by theorem provers. We have only used unsafe functions in very bounded areas of the MPL standard library (§5.2.2) which can easily shown to be safe by inspection.

Bindings to foreign libraries written in C can also result in buffer overflows in that code—the most major library in all OCaml programs is the standard library which has been carefully inspected for problems (both by automated tools [106] and manual code auditing). Binding interfaces can also be statically checked for safety by Saffire [115]. In our experience, it is often easier to rewrite libraries than to bind to their C versions except for trivial system calls such as *kqueue(2)* which are not present in the standard library. From the servers in our case study (§7), the SSH server uses one external OCaml library for cryptography and the DNS server uses no external libraries.

Integer Overflows: These result from the silent overflow of integers due to the modulo nature of their machine representation. These problems are not normally directly exploitable to run arbitrary code in the style of buffer overflows, but are used as a step towards code execution by causing an application to under-allocate memory and thus result in a buffer overflow later in the control flow (§2.1.4). OCaml integers can silently overflow (and in fact are more likely to do so since they are one bit smaller than C integers) but this can

[3]It has been argued that there is a *higher* chance of being physically assaulted as an OCaml programmer due to the smug attitude shown towards their less fortunate colleagues still coding in C.

only lead to control flow errors and never arbitrary code execution due to dynamic bounds checking. Our use of MPL to abstract packet parsing (which is the most common source of integer overflows) ensures that all integer operations which can possibly overflow (e.g. addition) are dynamically checked for correctness.

Memory Exhaustion: Classically defined by the TCP SYN flooding attack [243], these attacks keep increasing the amount of state stored by a server until it runs out of memory. MELANGE applications are as vulnerable to this as normal servers if constructed in such a way that they store per-connection state (which in some protocols is unavoidable). However, the presence of a garbage collector which deals with low-memory situations by compacting and aggressively freeing memory such as weak references (see our DNS server in §7.2.2) may make this attack harder to exploit than in conventional servers which manually manage their memory space.

Complexity Attacks: Algorithmic complexity attacks [86] convert normally efficient data structures into much more expensive versions by inserting specially crafted data into them—e.g. by forcing hash collisions in an associative array. MELANGE applications are written in OCaml which makes it significantly easier to use an appropriate data structure instead of the "catch-all" hash tables typically found in scripting languages such as Perl or Python. Badly constructed OCaml applications are equally vulnerable to this attack however.

Dynamic Termination: Applications which use external safety monitors such as system call monitors are vulnerable to forced termination of the entire application by a single malicious connection which has knowledge of a flaw in the security policy. For example a `systrace` [229] policy applies to the entire process and a threaded server may receive a malicious request which triggers a rare code path which violates the policy and aborts the entire server. MELANGE safety monitors are integrated with the application code and raise software exceptions which can be caught and dealt with more cleanly. For example, our SSH server SPL policies (§7.1.3) are split up into per-session and per-connection safety monitors, and a violation will only terminate one session or one multiplexed connection respectively, and never the entire server.

Another source of dynamic termination is from dynamic typing errors; e.g. class cast exceptions in Java. OCaml is statically typed and a large class of these dynamic type errors cannot happen; however dynamic bounds checking errors can. Currently these errors are caught as normal software exceptions, but recent research into dependent types in ML [287, 288] promises to statically eliminate many run-time bounds checks and provides a path to future immunity against this attack.

Protocol Vulnerabilities: Many problems are the result of higher-level vulnerabilities than implementation issues; e.g. e-mail spam through open relays [83], WWW Cross-Site Scripting or SQL injection attacks [245] or phishing attacks [92]. MELANGE does not deal with these problems, beyond providing a more solid implementation of protocols which can be used to reason about higher-level issues in the future.

Covert Channels: Covert channels are mechanisms for sending and receiving data between two agents over existing communications channels without alerting observers [274]. Most Internet protocols are acknowledged to have many covert channels [25] and the current

MELANGE architecture does not seek to address them. However, experimental extensions to OCaml which perform information flow analysis point to type-based solutions for reducing their bandwidth in future work [227].

Memory Errors: A very novel recent attack which shows that soft memory errors (e.g. from cosmic rays) can lead to serious security vulnerabilities in virtual machines (e.g. Java or .NET) which depend on dynamic sand-boxing of untrusted code [123]. MELANGE applications do not execute untrusted code and so this attack is not relevant.

Source Code Trojans: First famously proposed by Ken Thompson in his 1984 ACM Turing Award speech on "*Reflections on trusting trust*" [265], these attacks are recently becoming more popular due to attackers inserting trojan horses inside the *source code* of popular open-source applications such as Sendmail[4] which execute malicious code when the application is compiled. MELANGE applications assume the source code accurately expresses the intentions of the programmer (i.e. it has not been tampered with by third parties) and that the compiler tool-chain and operating system are operating correctly.

4.4 Summary

We began this section by explaining the design goals for our proposed new application architecture which will solve some of pressing security concerns on the Internet (§4.1). These goals were translated into the MELANGE architecture (§4.2) and a threat model defined to clarify the level of protection which the architecture grants from the myriad of possible attacks (§4.3).

[4]See CERT CA-2002-28 at `http://www.cert.org/advisories/CA-2002-28.html`.

CHAPTER 5

Meta Packet Language

Be conservative in what you do, be liberal in what you accept from others.
JON POSTEL, RFC 793

In Chapter 4, we introduced the MELANGE architecture for constructing statically type-safe and high-performance network applications. A key component of MELANGE is the Meta Packet Language (MPL)—a domain-specific language used to specify how to transmit, receive and pretty-print network packets for most Internet protocols. Unlike other interface description languages such as CORBA IDL (§3.2.1) it specifies the wire format directly and generates a compatible interface for the programmer to use (and backend code to implement that interface). MPL ensures a separation between the concerns of statefully manipulating packets and of the low-level parsing required to convert to and from a low-level stream of network byte traffic.

MPL offers: (*i*) a modular system for reading and writing low-level types (such as bytes, booleans, integers, and bit-fields) according to the protocol's requirements, (*ii*) arrays for fixed-length and variable-length fields; (*iii*) attributes to specify constant values, variant types, or alignment requirements; and (*iv*) higher-level constructs such as packet classification via pattern matching, arrays of fields, and extensible custom types. Rather than outputting machine code, the MPL compiler acts as a *meta-compiler* and outputs code in a variety of different languages. The back-ends are optimised with the capabilities of the target language. For example, the OCaml interface takes advantage of the strong static type system to guarantee that a server linked with code output by MPL only sends and receives well-formed network packets for that protocol. Similarly, although the C backend cannot make such strong guarantees, it can ensure that all network traffic is parsed safely with respect to buffer and integer overflows. All of our examples use the OCaml backend as it is the implementation language of choice for our MELANGE architecture. However, the overall structure of the code generated can be easily translated to other languages which have constructs such as namespaces (or objects or modules) and first-class functional closures (e.g. Haskell, Python or Ruby).

MPL is primarily concerned with ensuring that a server receives and sends well-formed packets; it does not attempt to enforce that the *contents* of those packets are meaningful with respect to the protocol's semantics, but it does ensure that (for example) length fields are con-

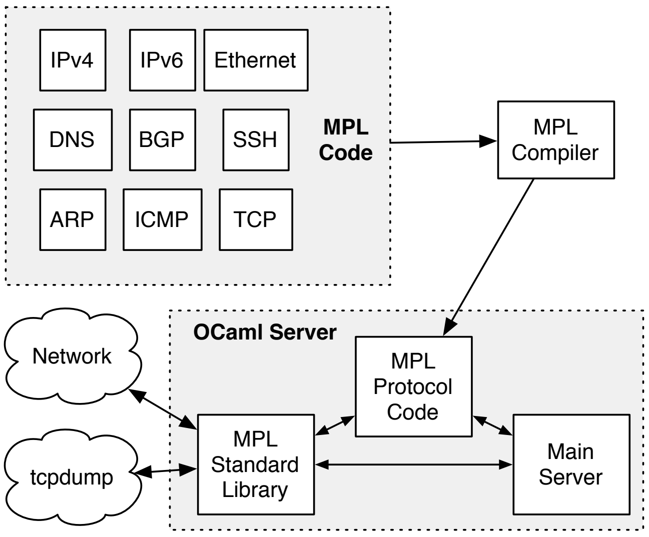


Figure 5.1: Architecture of an MPL-driven OCaml server

sistent with the amount of data that follows. Figure 5.1 illustrates how an OCaml network application which uses MPL would be structured. Firstly an MPL compiler accepts an input specification, type checks and compiles it, and outputs an OCaml module for that protocol. This code module depends on the presence of a basis library of code which deals with transmitting and receiving individual fields efficiently. Thus, the main body of the OCaml server never directly interacts with the network, instead going via the MPL code to ensure the well-formedness of any traffic.

This chapter offers two key research contributions: (*i*) showing that parsing network traffic in a high-level language (e.g. OCaml) can be as efficient as C code, but with significantly better static safety properties; (*ii*) demonstrating that for Internet protocols complex grammars are not necessary to parse the majority of protocols, with the compromise that field parsing is performed in a general-purpose language. In the remainder of this chapter we first describe the MPL language (§5.1), the basis library of standard functions (§5.2), the output structure of the OCaml interface (§5.3), and the performance evaluation of the system against an equivalent server written in C (§5.4).

5.1 Language

We now introduce the MPL language by an example (§5.1.1), discuss the theoretical space where it is based (§5.1.2), and finally define the syntax (§5.1.3) and semantics (§5.1.4).

5.1.1 Parsing IPv4: An Example

At its simplest level, an MPL specification is a list of named, typed fields with an optional list of attributes. In this section, we illustrate MPL by example by defining a specification to parse IPv4 packets. The following unmarshals IPv4 packets from a byte stream.

```
packet ipv4 {                                                    MPL
    version: bit[4];
    ihl: bit[4];
    flags: byte;
    length: uint16;
    id: uint16;
    frag_info: uint16;
    ttl: byte;
    protocol: byte;
    checksum: uint16;
    src: uint32;
    dest: uint32;
    options: byte[(ihl × 4) - offset(dest)];
    data: byte[length - (ihl × 4)];
}
```

Our specification begins with a *bitfield*; two variables `version` and `ihl` which are encoded into the first byte of the IPv4 packet. The MPL compiler converts these bit-fields into integers with the appropriate shifts and masks (§5.1.4). Once the first byte has been decoded, we progress by binding variables such as `ttl` or `length` by using built-in MPL types to represent bytes, 16- and 32-bit integers. When the `options` field is reached, we move onto the next feature of MPL: variable-length byte arrays. In MPL, variable-length fields can use any previously bound variables (which are of a numeric type) to calculate their length at run-time. In the case of the `data` field, it uses the values from the `length` and `ihl` fields. The function *offset(label)* used in the calculation of the `options` field returns the total length of all the variables until (and including) *label*.

However, MPL specifications must also include sufficient information to allow the packet to be created from scratch (i.e. marshalled). We perform this function in MPL by adding **value** attributes to fields which provide their values to a newly created packet:

```
packet ipv4 {                                                    MPL
    version: bit[4];
    ihl: bit[4] value(offset(options) / 4);
    flags: byte;
    length: uint16 value(offset(data));
    id: uint16;
    frag_info: uint16;
    ttl: byte;
    protocol: byte;
    checksum: uint16;
    src: uint32;
    dest: uint32;
    options: byte[(ihl × 4) - offset(dest)];
    data: byte[length - (ihl × 4)];
}
```

The above IPv4 specification now contains sufficient information to both send and receive packets, with all of the length fields (`ihl` and `length`) automatically calculated. MPL converts this specification into an OCaml implementation and interface:

```
module Ipv4 : sig                                                    OCaml
    class o : object
        method src: int32
        [...]
        method flags : int
        method data : string
        method options : string
    end
    val t : version:int → flags:int → id:int → frag_info:int →
        ttl:int → protocol:int → checksum:int → src:int32 →
        dest:int32 → options:blob → data:blob → env → o
    val unmarshal : env → o
end
```

The interface above (simplified for this example) contains an object definition `Ipv4.o` which has accessor methods for each field of the packet. The object is never instantiated directly; instead the function `Ipv4.unmarshal` accepts a parsing environment and returns the object after parsing the raw bytes from the network (an exception is raised if the traffic is malformed). To create packets, the function `Ipv4.t` is invoked with labelled arguments corresponding to the field bindings in the MPL specification. The OCaml types of these arguments are matched to the precision of the MPL types; e.g. the `src` and `dest` fields are 32-bit integers and are thus best represented by an OCaml *int32*. To manage payloads with minimal data copying, byte arrays are represented by an abstract *blob* type (§5.2). `Ipv4.t` does not require `ihl` and `length` arguments, since they have **value** attributes in the MPL specification to automatically calculate their values.

Attributes and Variants

The specification for IPv4 includes a number of other invariants which should be enforced; for example, the minimum value of `ihl` is 5, and the `options` field must be padded to 32-bit alignment. Another common idiom in network protocols is to map the values a field can contain to textual labels. MPL allows these labels to be represented symbolically in the specification and maps them to variant types in the OCaml interface, and the conversion code to and from the variant type in the generated implementation.

In addition to the **value** attribute described earlier, MPL offers attributes to restrict the range of a field, mark it as a variant type, give it a default value, or specify alignment restrictions. Below is a more complete IPv4 specification with the `flags` field expanded into its component bits and attributes added:

```
packet ipv4 {                                                          MPL
    version: bit[4] const(4);
    ihl: bit[4] min(5) value(offset(options) / 4);
    tos_precedence: bit[3] variant {
        |0 ⇒ Routine |1 → Priority |2 → Immediate
        |3 → Flash |4 → Override |5 → ECP
        |6 → Internetwork_control |7 → Network_control };
    tos_delay: bit[1] variant {|0 ⇒ Normal |1 → Low};
    tos_throughput: bit[1] variant {|0 ⇒ Normal |1 → Low};
    tos_reliability: bit[1] variant {|0 ⇒ Normal |1 → Low};
    tos_reserved: bit[2] const(0);
    length: uint16 value(offset(data));
    id: uint16;
    reserved: bit[1] const(0);
    dont_fragment: bit[1] default(0);
    can_fragment: bit[1] default(0);
    frag_offset: bit[13] default(0);
    ttl: byte;
    protocol: byte variant { |1→ICMP |6→TCP |17→UDP };
    checksum: uint16 default(0);
    src: uint32;
    dest: uint32;
    options: byte[(ihl × 4) - offset(dest)] align(32);
    header_end: label;
    data: byte[length-(ihl×4)];
}
```

The `version` field is defined as a constant value 4 in an IPv4 packet; the **const** attribute also removes this field from the OCaml creation function and adds to code to automatically insert it in the implementation code. The **default** attribute is a specification hint which allows the compiler to propagate the default value through to the creation function, or generate optimised code for the default "fast path" if the target language supports this. The `options` field has an **align** attribute added to it to indicate that the field must always be aligned to 32-bit boundaries and padding added if this is not the case. The `ihl` field also has a **min** attribute to indicate that the header must be at least 5 words long (there is also a **max** attribute, unused in this example). The `protocol` field defines integer values to indicate the nature of the payload contained in the `data` field. The **variant** attribute maps the `protocol` field as a variant type in OCaml:

```
type protocol_t = [                                                    OCAML
     | ‘ICMP
     | ‘TCP
     | ‘UDP
]
let protocol_t_unmarshal = function
     |1 → ‘ICMP
     |6 → ‘TCP
     |17 → ‘UDP
     |_ → raise Bad_packet
let protocol_t_marshal = function
     |‘ICMP → 1
     |‘TCP → 6
     |‘UDP → 17
let protocol_t_to_string = function
     |‘ICMP → “ICMP”
     |‘TCP → “TCP”
     |‘UDP → “UDP”
```

Since the full IPv4 `protocol` field defines over 150 labels, it is clearly safer and easier to mechanically generate these accessor functions. In addition, the compiler takes care of tracking the underlying data type of the field (e.g. *int32*) and labelling the pattern matches on the integers with the correct suffix (“l” or “L” for *int32* and *int64* types). Variant attributes also support a default value by the convenient syntax of using ⇒.

Classification

The keen reader will notice that the contents of the `protocol` field ought to mandate the type of the `data` payload. If TCP data were stored into `data`, a well-formed packet must store 6 in the `protocol` field. MPL allows packets to be classified into sub-types by using a pattern-matching style on fields that have previously been unmarshalled:

```
packet ipv4 {                                                          MPL
     version: bit[4];
     ihl: bit[4] value(offset(options) / 4);
     flags: byte;
     length: uint16 value(offset(data));
     id_frag_ttl: byte[5];
     protocol: byte;
     check_src_dest: byte[10];
     options: byte[(ihl × 4) - offset(check_src_dest)];
     classify (protocol) {
     |1:“ICMP” → data: packet icmp();
     |6:“TCP” → data: packet tcp();
     |17:“UDP” → data: packet udp();
     };
```

This example introduces the **packet** keyword, used to include external MPL specifications; for example, `packet icmp()` would reference an external “`icmp.mpl`” and have type `Icmp.o` in the OCaml interface. We use the **classify** keyword to distinguish packets based on the contents of `protocol`. The pattern-matching style is similar to ML, and the first match succeeds (in the example, an unknown IPv4 type will be represented by a byte array). The output OCaml is represented by a series of nested modules and objects:

```
module Ipv4 : sig                                                    OCAML
    module ICMP : sig
        class o
        val t : version:int → (...etc) → packet
    end
    module TCP : sig
        class o
        val t : version:int → (...etc) → packet
    end
    module UDP : sig
        class o
        val t : version:int → (...etc) → packet
    end
    module Unknown : sig
        class o
        val t : version:int → protocol:int → (...etc) → packet
    end
    type o =
    |‘ICMP of ICMP.o      |‘TCP of TCP.o
    |‘UDP of UDP.o       |‘Unknown of Unknown.o
    val unmarshal : env → o
end
```

The top-level type `Ipv4.o` is no longer an object type; instead it is a parametric polymorphic variant which represents the different classification options. The sub-modules for ICMP/TCP/UDP contain functions to create packets of their respective types, but with the `protocol` field considered constant according to its value from the classification pattern match. However the `Ipv4.Unknown.t` creation function still has the `protocol` argument, since the default pattern match does not contain a constant value.

This style of OCaml output permits the use of pattern-matching over packets, as shown below. The polymorphic variant definitions output by MPL are fully refined (§2.3.2), allowing the compiler to check for exhaustiveness. The code below also shows the types of the checksum functions; an attempt to pass a UDP packet to the function `tcp_checksum` would not type-check since the type `Udp.o` is distinct from `Tcp.o`[1].

```
# val icmp_checksum : Icmp.o → bool                                  OCAML
# val tcp_checksum : Tcp.o → bool
# val udp_checksum : Udp.o → bool
let ipv4 = IPv4.unmarshal env in
let checked = match ipv4 with
|‘ICMP icmp → icmp_checksum icmp#data
|‘TCP tcp → tcp_checksum tcp#data
|‘UDP udp → udp_checksum udp#data
|‘Unknown data → false in
output (if checked then “passed” else “failed”)
```

5.1.2 Theoretical Space

Some data description languages in the literature (e.g. PADS [109]) can express complex grammars in order to fully describe the wire format of a packet. Others only allow simpler and

[1]This is not precisely true since object definitions in OCaml are structural, and so an object with identical fields to `Udp.o` would in fact type-check. A solution is discussed later (§5.5).

less expressive classes of grammars (e.g. PACKETTYPES [189]) but are unable to fully describe more complex Internet protocols. As Pierce notes in his work on bi-directional programming (§3.2.3), it is desirable to simplify the grammar as much as possible in order to make it more practical to write bijective specifications (essential for packet parsing as we must both transmit and receive packets). Some protocols will always require special parsing for certain aspects (e.g. the DNS host-names described in §7.2.1) and even advanced data description languages such as PADS may not suffice since they do not permit arbitrary computation. For constructing practical network applications, it is not sufficient to be able to parse "almost" all of an RFC protocol specification of course!

MPL explores the middle-ground by permitting individual field parsing routines to be written directly in the target language, while their high-level composition and constraints are expressed using a more abstract specification language. This has several advantages: (*i*) the specification language need only describe a significantly simpler grammar; (*ii*) low-level field parsing routines can be highly optimised for the target language; and (*iii*) the complex parsing corner cases present in most Internet protocols can be written in a general-purpose language while keeping the core specification relatively simple.

To justify this approach we must consider the history of Internet protocols. The end-to-end principle provides a guideline that complexity in protocols should be present in end hosts and not in the core network [239]. This meant that, unlike conventional telecommunications systems which were primarily implemented directly in hardware, Internet protocols were always designed to be processed by general purpose CPUs in software. The original IP RFC [224] states that a module is "*an implementation, usually in software, of a protocol or other procedure*". Other early RFCs on efficient checksum implementation [43, 185] confirm this by actually providing C source code and discussing its performance on various prevalent hardware architectures of the time. Internet protocols have also tended to evolve over the decades rather than be reconstructed from scratch (e.g. TCP/IP itself was based on the ARPANET NCP protocol [237]). Due to their software nature changes were sometimes made to incur the minimum of disruption to existing code to avoid introducing errors.

Text-based protocols such as HTTP [107], SMTP [169] or FTP [226] are documented as context-free BNF grammars [85] and easily parsed using existing tools such as `yacc`. However binary protocols such as SSH [293], DNS [208] or BGP [236] are often simple regular grammars parsed using finite state machines, but are still complex to manually implement. This is evidenced by the number of packet parsing related security problems in, for example, OpenSSH (§7.1). It is these binary protocols that MPL is designed to parse as efficiently and succinctly as possible using a high-level language; the "quirks" introduced by gradual evolution can be cleanly hidden behind a general-purpose programming language interface, the overall composition can be expressed using MPL specifications, and efficiency is not sacrificed by excessive abstraction.

MPL utilises a non-lookahead decision-tree parsing algorithm which is simple enough to capture many binary Internet protocols while retaining a simple set of rules to ensure that specifications remain bijective (§B.1). It cannot express context-free grammars by design (since it has no stack), but many real-world binary Internet protocols are, due to their roots in early resource-constrained software stacks, fundamentally simple grammars which have a number of quirks due to the evolutionary nature of Internet protocol design. As we show later (§5.2) much of the general-purpose language code for field types can be factored out across common protocols into a basis library and re-used.

5.1.3 Syntax

We describe the LALR(1) grammar [252] for MPL below using an an extended BNF notation [16]. In the extended syntax, we represent terminals as *term*, tokens as **`token`**, alternation with {*one* | *two*}, optional elements as [*optional*], elements which must repeat once or more as (*term*)+ and elements which may appear never or many times as (*term*)*.

```
           main  →  (packet-decl)+ eof
    packet-decl  →  packet identifier [ ( packet-args ) ] packet-body
    packet-args  →  { int | bool } identifier [ , packet-args ]
    packet-body  →  { (statement)+ }
      statement  →  identifier : identifier [var-size] (var-attr)* ;
                 |  classify ( identifier ) { (classify-match)+ } ;
                 |  identifier : array ( expr ) { (statement)+ } ;
                 |  ( ) ;
 classify-match  →  | expr : expr [when ( expr ) ] -> (statement)+
       var-attr  →  variant { (| expr {→ | ⇒} cap-identifier)+ }
                 |  { min | max | align | value | const | default } ( expr )
       var-size  →  [ expr ]
           expr  →  integer | string | identifier | ( expr )
                 |  expr { + | - | * | / | and | or } expr
                 |  { - | + | not } expr
                 |  true | false
                 |  expr { > | >= | < | <= | = | .. } expr
                 |  { sizeof | array_length | offset } ( expr-arg )
                 |  remaining ( )
```

5.1.4 Semantics

The full user manual for MPL is available in Appendix B and we summarise the important points in this section. An MPL specification must contain enough information to unambiguously create and receive packets, and so the compiler performs well-formedness checks to ensure that this is the case (§B.1). MPL uses three different notions of types for a field: (*i*) *wire types* for the network representation of a field; (*ii*) *MPL types* which are used within MPL specifications only; and (*iii*) *language types* which are the native types of the field in the output programming language.

Internet protocols often use common mechanisms for representing values such as fixed-precision integers and bit-fields; e.g. *network byte order* is defined as "big endian" (the most significant byte is stored first). Wire types help capture this redundancy by defining the wire formats for common formats. The built-in types can be found in Table B.2 and custom wire types may also be defined on a per-protocol basis (§5.2.3).

Every wire type must be mapped onto a corresponding MPL type so that the contents of the field may be manipulated within the MPL specification (e.g. for classifying the packet). The supported MPL types are integers (of varying precision), strings and booleans. If a field is not intended to be manipulated as one of these MPL types it is mapped to a special "opaque" type which ensures it is treated as an abstract type and simply passed through to the application. Similarly, every wire type also has a corresponding language type for every language back-end. For example an unsigned 32-bit integer is mapped into the OCaml int32 type, and a DNS label with a more complex wire format (§7.2.1) becomes a native OCaml string.

Classification Tree

In addition to converting individual fields to their corresponding language equivalents, the overall MPL specification must be converted into a high-level data structure in the target language. The fields are structured into a series of nested modules that are used to separate differently classified packets into unique namespaces. This is accomplished by recursively iterating over the abstract syntax tree and forking a new list of namespaces for every **classify** keyword which is encountered.

The algorithm for calculating a classification tree is described below using ML-like pattern matching, where $\vec{\mathcal{N}}$ represents a list of elements, ϕ the empty list, :: and @ are the list cons and concatenation operators, and $\textbf{iter}\vec{\mathcal{L}} \xrightarrow{\mathcal{L}} f$ applies the function $f(\mathcal{L})$ to every element of $\vec{\mathcal{L}}$. $\vec{\mathcal{N}}$ and $\vec{\mathcal{L}}$ represents a list of labels, $\vec{\mathcal{R}}$ and $\vec{\mathcal{S}}$ a list of MPL statements.

```
$\textbf{let}\ walk\ \vec{\mathcal{N}} = \textbf{function}$
$\mid \textsc{classify}(\vec{\mathcal{L}} \times \vec{\mathcal{S}}) :: \vec{\mathcal{R}} \Rightarrow \textbf{iter}\ \vec{\mathcal{L}} \xrightarrow{\mathcal{L}} \{walk\ (\mathcal{L} :: \vec{\mathcal{N}})\ (\vec{\mathcal{S}}\ @\ \vec{\mathcal{R}})\}$
$\mid x :: \vec{\mathcal{R}} \Rightarrow walk\ \vec{\mathcal{N}}\ \vec{\mathcal{R}}$
$\mid \phi \Rightarrow \vec{\mathcal{N}}$
```

Observe that a classification propagates the name bound in its match through all statements subsequently after the classification, resulting in a multiplication effect of names if many classifications are used in series:

```
classify (a) {                                  MPL
    |1:"One" → ();
    |2:"Two" → ();
    |3:"Three" → ();
};
classify (b) {
    |4:"Four" → ();
    |5:"Five" → ();
    |6:"Six" → ();
};
```

```
module One = struct                             OCAML
    module Four = struct [...] end
    module Five = struct [...] end
    module Six = struct [...] end
end
module Two = struct
    module Four = struct [...] end
    module Five = struct [...] end
    module Six = struct [...] end
end
module Three = struct
    module Four = struct [...] end
    module Five = struct [...] end
    module Six = struct [...] end
end
```

On the left is an MPL fragment with two **classify** clauses in series. The right shows the output OCaml interface, which duplicates the contents of the second **classify** in every branch of the first. This multiplication effect can result in very large interfaces if **classify** statements are applied in series. However, we have encountered no Internet protocols where such a structure of classification is necessary. In general, classifications are nested and not placed in series (see §7 and Appendix C).

Bound Variables

The compiler must determine which variables need to be marked as *bound variables* that are not exposed in the external code interface. This allows fields to be automatically and reliably calculated from other fields of the packet (e.g. length fields). A variable is considered bound if: (*i*) it is marked with a **const** or **value** attribute; (*ii*) a variable is an argument in a **classify** statement (see below for some caveats); or (*iii*) a byte array or an **array** construct use a variable in their size specifier. Variables are only considered bound in the context of the current classification tree; a variable might later be bound in one classification branch but not another. Classification pattern-matches matches do *not* bind their variable if: (*i*) the pattern match represents an integer range (e.g. `1..5`) in which case the variable is left free[2]; or (*ii*) the default clause of the pattern match, if present, does not bind the classification variable as its value cannot be statically determined. Classification binding is considered to have a higher precedence than a **value** attribute, and thus the attribute is ignored if a classification later binds it.

Since every classification branch can have a different list of bound variables, the compiler walks across the AST and obtains a list of variables for every combination of **classify** statements. The MPL specification below is an artificial example which uses default classification, **const** attributes and ranges to show examples of the different ways variables can be bound. The classified modules are shown on the right (underlined values represent bound variables).

```
alpha: byte;                                  MPL
classify (alpha) {
|1:"One"  → beta: byte;
|2:"Two"  → beta: byte const(0);
|3:"Three"  →
    gamma: byte;
    classify (gamma) {
    |1..3:"India"  → delta: byte;
    |4:"Foxtrot"  → epsilon: byte;
    };
};
omega: byte;
```

$$\begin{aligned}
\textbf{One} &\leftarrow \underline{alpha}, beta, omega\\
\textbf{Three.India} &\leftarrow \underline{alpha}, gamma, delta, omega\\
\textbf{Two} &\leftarrow \underline{alpha}, \underline{beta}, omega\\
\textbf{Three.Foxtrot} &\leftarrow \underline{alpha}, \underline{gamma}, epsilon, omega
\end{aligned}$$

Notice in particular that *gamma* is not bound in `Three.India` since that pattern match uses a range variable, but is bound in `Three.Foxtrot`. Support for this assymmetry in binding variables across some classification branches is useful in real-world protocols; for example, consider the shortened MPL specification for an Ethernet frame:

```
packet ethernet {                                                        MPL
    dest_mac: byte[6];
    src_mac: byte[6];
    length: uint16 value(offset(eop)-offset(length));
    classify (length) {
        |46..1500:"E802_2"  →
            raw: byte[length];
        |0x800:"IPv4"  →
            raw: byte[remaining()];
        |0x806:"Arp"  →
            raw: byte[remaining()];
    };
    eop: label;
}
```

In an Ethernet frame, if the `length` field contains a value less than 1500, it represents a "raw" E802.2 frame, otherwise the value of the `length` field determines the specific type of the frame (e.g. IPv4 or ARP). It is clearly desirable for the `length` field to be automatically calculated, so if an `IPv4` or `Arp` packet is created, the constant pattern match values determine

[2]Strictly speaking, the variable ought to be constrained to the same range as the pattern match, but we do not do this in the current implementation.

the value of `length`. If an `E802.2` packet is created, the presence of a range of integers in the pattern match means `length` is instead bound by the **value** attribute (which uses offset calculations to assign a value to the field automatically). The **value** attribute could have been left out of the specification, in which case `length` would be a free variable for the interface to `E802.2` packets.

The variable binding rules are generally only needed in protocols which overload a field for multiple purposes, such as the Ethernet length field described above. New protocol designs should not use such techniques in the interests of simpler parsing.

Bit Conversion

The MPL compiler statically converts bit-fields into a sequence of bytes and the shifts and masks required to extract the relevant portion of the bit-field from those bytes. This is required since network interfaces can only manipulate data at the byte granularity. A well-formed MPL specification (§B.1) requires that bit-alignment is tracked and matched across **classify** statements such that all branches of a classification result in the same bit-alignment. This restriction permits the compiler to statically create "dummy" bytes for every 8 bits in the bit-field. During marshalling and unmarshalling, the dummy bytes are used for sending and receiving data, while the bit-field variables are exposed to the external code interface.

Bit shifting is a very useful feature of MPL, as it is an easily automated transformation performed by the compiler that hides the complexity of manually keeping track of bit-fields. A number of Internet protocol require this support, ranging from low-level formats such as IPv4 up to higher-level protocols such as DNS (§7.2).

5.2 Basis Library

The code output by the MPL compiler does not communicate with the network directly, instead operating via function calls to a basis library. This library deals with obtaining data for the application and provides support for *packet environments* which offer restricted views of a packet (§5.2.1). The library supports the basic MPL wire types (§ 5.2.2) and allows importing new custom types for more complex packet formats (§5.2.3).

In this section we describe the OCaml basis library, which (along with the OCaml code output by the MPL compiler) is designed to be linked directly with the main network application (see Figure 5.1). By structuring the application to only send or receive packets via the MPL interface, applications can guarantee that they only ever transmit or receive valid network packets with respect to the MPL specification.

5.2.1 Packet Environments

The basis library performs operations on an abstract *packet environment* which represents a single, mutable version of the underlying packet data. The packet environment consists of a static buffer and the total length of valid data within that buffer. Multiple *views* may be constructed from this environment which allow data to be manipulated and read in different portions of the packet. For example, Figure 5.2 illustrates an example environment of an ICMP Echo Request packet, most commonly used by the UNIX `ping` utility. The left box shows the complete buffer, of which the gray portion is unused. The packet can be broken down into different sections of the protocol stack, such as the IPv4 header, the ICMP header, and the ICMP payload. On the right, a view is constructed of the ICMP header, which represents offsets into the header (starting from 0). Code to manipulate the ICMP header can operate on this view irrespective of its actual underlying position in the complete packet. The OCaml

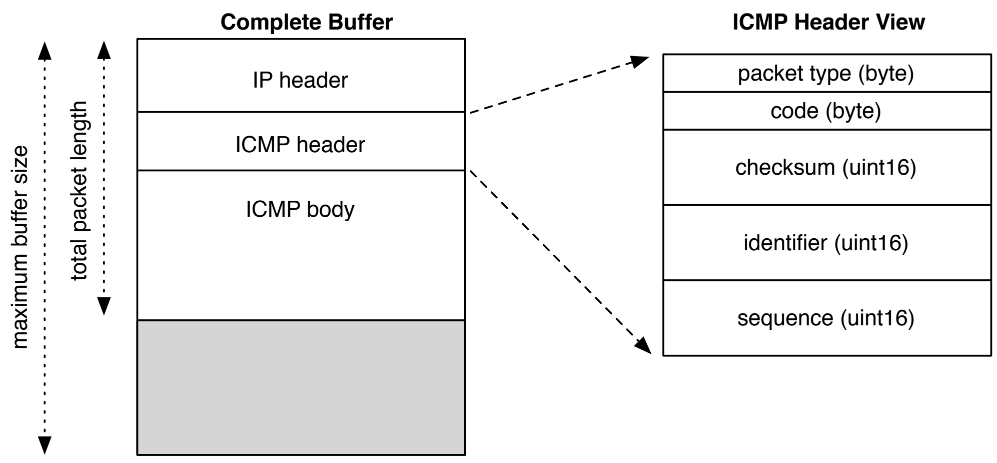


Figure 5.2: The MPL environment of the complete packet *(left)* and a view on the ICMP header alone *(right)*

packet environment is represented by:

```
type env = {                                                        OCAML
     buf: string;       (* data *)
     len: int ref;      (* total length of valid data *)
     base: int;      (* start position in buffer *)
     mutable sz: int;      (* valid length of data, relative to base *)
     mutable pos: int;      (* position in data, relative to base *)
}
```

The buf and len fields represent the global properties of the packet (the contents and total length), and are shared across all views. We use a large, fixed-size string and track packet length separately to eliminate the overhead of allocating and resizing strings dynamically. The rest of the fields (base, sz and pos) are used to provide views into the underlying contents. The base field acts as a base offset from which all positions are calculated. sz allows a view to be constrained in size (e.g. the length of the ICMP packet body in Figure 5.2), and pos tracks the current position while processing a packet.

A view is created by using the basis library function env_at (env → int → int → env) which specialises an input environment with a new base and size, and returns the new structure. A view maintains its own position and length (the mutable entries in the record), while sharing the global fields such as total packet size (this is essential to see for some protocols such as DNS where field parsing is determined by the global position in the packet). Since a view is just a normal environment, further sub-views are easily created by repeated invocations of env_at on the new view. The creation of a new view copies only a few integers and is relatively inexpensive compared to copying the buffer.

The basis library provides several ways to fill a packet environment: (*i*) directly from a string when instantiating a new environment; (*ii*) reading from a file descriptor (which could be a network connection or a file); or (*iii*) by defining a "fill function" which is a closure that is triggered when more data is required in the environment. The fill function is useful to embed packet environments in the middle of a more complex parsing structure; for example, if the network data is encrypted, a fill function could apply a closure which writes decrypted bytes into the environment provided (§7.1.2). Similar functions are provided to transmit the contents

Table 5.1: Mapping of MPL wire types to OCaml native types

`byte`	octet	`char`
`byte[x]`	array of `x` bytes	`string`
`uint16`	unsigned 16-bit integer	`int`
`uint32`	unsigned 32-bit integer	`int32`
`uint64`	unsigned 64-bit integer	`int64`

of a packet environment.

5.2.2 Basic Types

MPL has a set of built-in wire types to manipulate common, low-level fields used in network protocols. The most basic types are `byte` (representing a single octet), and `byte[x]` (representing a byte array of length `x`). However, since integers are used so often and their efficient representation is important, types are provided for 16-, 32- and 64-bit integers respectively. The built-in types map directly into OCaml native types (see Table 5.1). Another reason for distinguishing between integer types is that the native integers provided by many functional languages (e.g. SML and OCaml) are 1 bit short of the architecture word size (e.g. 31 bits on i386 and 63 bits on Alpha), to store unboxed integers on the heap alongside pointers. As we noted earlier (§4.1), the lack of polymorphic integer operators in OCaml results in very verbose network code when compared to the equivalent in languages such as Haskell or C. Even worse, if the programmer fails to realize that native integers in OCaml are smaller in size and carelessly converts between them, network values can be silently truncated. Although this truncation cannot result in buffer overflows in the manner of C programs, it will corrupt data and can result in application-level security issues. The OCaml code output by MPL automates this conversion process.

The basis library provides functions to manipulate the traffic being sent or received: (*i*) `unmarshal` to convert from the current position in the packet environment and advance the position of the current view; (*ii*) `at` to unmarshal from a specified offset and not modify the current environment; and (*iii*) `marshal` to write the field at the current position and modify the environment. Below is the interface for the `Mpl_byte` module:

```ocaml
module Mpl_byte : sig                                    OCAML
    type t
    val unmarshal : env → t
    val marshal : env → t → unit
    val at : env → int → t
    val to_char : t → char
    val of_char : char → t
    val to_int : t → int
    val of_int : int → t
end
```

A byte is represented by an abstract type `Mpl_byte.t`. The `unmarshal` function accepts a packet environment, reads the byte at the current position, advances the position by one, and returns the abstract type. Similarly, the `Mpl_byte.at` function accepts an environment and an offset into it, and returns the byte at that position without modifying the environment. The `marshal` function writes a byte, advances the position, and also updates the total length of the environment if it has written beyond the previously recorded limit. This total length is a reference and reflected across all views.

The $\{\texttt{to}, \texttt{from}\}\_\{\texttt{int}, \texttt{char}\}$ functions are provided to convert between the abstract MPL types and native OCaml values. Recall that abstract types may be used to enforce interface abstraction without run-time overhead (§2.3.1). In the basis library the internal representation of a `Mpl_byte` is the $char$ type (represented as a native integer by OCaml). Since the internal representation matches the external OCaml type, the conversion functions are actually the identity function $let\ x =\ x$ and are optimised away. However, a debug version of the basis library can track the offset of each byte, giving `Mpl_byte.t` the concrete type $(int \times char)$ and allowing debugging without modifying the interface.

The 16-, 32- and 64- bit integer modules are similarly composed by invocations of the `Mpl_byte` module functions and combined with appropriate shift operations. A flag in the environment is used to decide the byte-order (little-endian or big-endian) of the incoming traffic. This is normally big-endian (also known as network byte-order [257]) in the case of network traffic, but certain file formats (e.g. the popular `pcap` [150] library) can record traffic in little-endian format, potentially requiring a conversion while marshalling and unmarshalling. Also, protocols such as the Plan9 Remote Resource Protocol [132] mandate the use of little-endian byte order even for network traffic.

Each of the integer modules also defines a *dissection* function, which allows an environment to be iterated over as if it were a list of that integer type. The functions have the type $(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow env \rightarrow \alpha$, which is a similar type signature to the built-in OCaml `fold_left` function. The function repeatedly unmarshals integers (of some type β) from the current environment, and applies those integer values to the first argument (a function which accepts an integer and an accumulator, and returns the accumulator). These dissection functions are very important when using Internet protocols such as IPv4, UDP, TCP and ICMP; e.g. the checksumming routines require fast 16-bit unsigned integer iterators across the packet headers and bodies (see §5.4).

The final built-in data type is `Mpl_raw`, used for byte arrays. Since this type often holds large packet data, it provides special support for manipulating abstract fragments.

```
type env                                                              OCAML
type frag
module Mpl_raw : sig
    val marshal : env → string → unit
    val frag : env → int → int → frag
    val at : env → int → int → string
    val blit : env → frag → unit
    val prettyprint : string → string
end
```

In the above interface, the `marshal` and `at` functions operate similarly to the functions described earlier. The `frag` and `blit` functions manipulate fragments of data without actually copying them; `frag` returns a value which represents the data as a tuple of its environment, the offset and length. This fragment can be passed around until it needs to be copied into another environment (e.g. another packet) by using `blit`, which accepts an environment and copies a fragment into it.

5.2.3 Custom Types

The built-in wire types are sufficient to encode many Internet protocols such as IPv4, BGP, ICMP, ARP, Ethernet, UDP and TCP. However, higher-level protocols often require more complex parsing and so MPL provides support for custom wire types by: (*i*) defining the MPL type to which the custom type maps (e.g. string, boolean, or an integer); and (*ii*) providing an external library of functions to handle the wire format. An example custom type is a DNS string (§7.2) which consists of a single header byte and a number of bytes equal to the value of the header. We dub this custom type a `string8`, and register it as type `string` with the MPL compiler. The OCaml code output expects a module called `Mpl_string8` with the following interface:

```
module Mpl_string8 : sig                                    OCAML
    type t
    val size : t → int
    val unmarshal : env → t
    val to_string : t → string
    val of_string : string → t
    val marshal : env → t → unit
end
```

Since `string8` has the MPL type `string`, the accessor functions {`to`, `of`}`_string` are used to convert to and from OCaml types. Notice that the type of `Mpl_string8.t` is left abstract, allowing its internal representation to be flexible. In this example the internal representation is also a string, but we show more complex cases later (§7.2.1).

5.3 OCaml Interface

The MPL basis library provides concrete methods to manipulate low-level fields such as bytes, integers, blobs of data or other custom types. The output from the MPL compiler consists of efficient OCaml code which uses this library to combine sequences of fields into complete protocol packets. Conceptually, the OCaml interface must support: (*i*) packet *sources* which create new packets and transmit them; (*ii*) packet *sinks* which accept raw bytes and translate them into OCaml data structures; and (*iii*) packet *proxies* which read raw bytes into an OCaml structure, safely modify values via the OCaml interface, and transmit the resulting packet. Proxies are not simply a combination of a source and a sink—rather, they allow for the in-place modification of data in a packet environment without any additional data copying[3].

OCaml provides first-class support for representing functional objects (i.e. a collection of data and functions) natively in its type system. We therefore use the notion of *packet objects*—objects which wrap an environment (representing the packet data) with the accessor functions to retrieve and modify fields within that particular packet. The packet objects are not allowed to be instantiated directly; instead, they are returned by accessor functions: (*i*) an unmarshalling function which classifies a packet environment and returns the correct object; and (*ii*) a marshalling function which accepts arguments corresponding to the packet fields, writes the wire format of the packet into an environment, and returns a packet object containing that environment. By restricting object creation in this way, we ensure that the environment encapsulated by the packet object always holds a consistent wire-format version of the packet.

The MPL classification tree (§5.1.4) is converted into a series of nested OCaml modules. Each module contains an object definition with accessor functions for each field in the packet,

[3]An example use of a packet proxy would be an IPv4 router, which merely updates the time-to-live, destination and checksum fields and retransmits the rest unmodified.

```
packet example {                                              MPL
    ptype: byte;
    classify (ptype) {
    |1:"Var" →
        plen: uint16 value(sizeof(data));
        data: byte[plen];
    |2:"Fixed" →
        data: byte[512];
    |3:"Sig" →
        subtype: byte;
        classify (subtype) {
        |4:"Restart" → ();
        |5:"Exit" → code: byte;
        };
    };
}
```

```
module Example : sig                                          OCaml
    module Sig : sig
        module Exit : sig
            class o : env → object [...] end
            val t : code:int → env → o
         end
        module Restart : sig
            class o : env → object [...] end
            val t : env → o
         end
        type o = [ ‘Exit of Exit.o | ‘Restart of Restart.o ]
        type x = [ ‘Exit of env → Exit.o | ‘Restart of env → Restart.o ]
        val m : x → env → o
     end
    module Fixed : sig
        class o : env → object [...] end
        val t : data:data → env → o
     end
    module Var : sig
        class o : data_length:int → env → object [...] end
        val t : data:data → env → o
     end
    type o =
        [ ‘Fixed of Fixed.o | ‘Sig of Sig.o | ‘Var of Var.o ]
    type x =
        [ ‘Fixed of env → Fixed.o | ‘Sig of env → Sig.o
        | ‘Var of env → Var.o ]
    val m : x → env → o
    val unmarshal : env → o
end
```

Figure 5.3: An example MPL packet (manually written) and the corresponding signature of the auto-generated OCaml code output

and also a *creation* function used to create that packet object. Figure 5.3 illustrates an example protocol as an MPL specification and the associated OCaml structure. Each classified variable has two variant types defined; `o` represents a packet object and `x` is a packet suspension which captures all the arguments necessary to create the packet but without actually applying it to a packet environment. The `t` functions (e.g. `Example.Fixed.t`) are used to create a packet object directly and the `m` functions (e.g. `Example.m`) combine packet suspensions and an environment to result in a packet object. Next we explain this module structure by considering the use cases described earlier.

5.3.1 Packet Sinks

A packet sink receives raw bytes and classifies them into OCaml data structures. In Figure 5.3, the `Example.unmarshal` function accepts a packet environment and returns a polymorphic variant type `Example.o`. This can be pattern-matched to retrieve the exact packet object; we illustrate the definitions for two packets below:

```
OCaml
class Example.Sig.Exit.o : env → object
    method code : int
    method env : env
end
class Example.Var.o : data_length:int → env → object
    method data : string
    method data_env : env
    method data_frag : frag
    method data_length : int
    method env : env
end
```

The object for a `Sig.Exit` packet is straight-forward; the only unbound variable is the `code` field, exposed as a method which returns an integer—internally, the OCaml implementation invokes the MPL basis library code to unmarshal a single byte at an offset of 2 from the packet start (since the previous bytes in the packet were of fixed size this is statically calculated). The `Example.unmarshal` function does not parse fields which are not necessary to parsing the packet and instantiate a packet object, instead skipping directly past them. Thus the first time that the `code` field will be parsed is when the method is invoked on the packet object.

The `Example.Var` packet holds a variable-length payload `data`. Since its length must be known in order to calculate the offsets of any packets beyond it, the unmarshal code calculates it (from the value of the `plen` field), and passes `data_length` as an argument to the object constructor. The `Example.Var.o` object has several methods for accessing the contents of `data`: (*i*) `o#data` copies the contents as a string; (*ii*) `o#data_env` returns a view positioned at the start of the array, with a size equal to its length; (*iii*) `o#data_frag` returns an abstract fragment of the contents of the array; and (*iv*) `o#data_length` returns the length of the array. Each of these methods are necessary in different situations; creating a new view is used to further classify the contents of `data` as another MPL packet, the fragment can be copied into a reply packet, or the string can be used as a last resort to perform a data copy.

5.3.2 Packet Sources

A packet source transforms OCaml data into raw bytes in the format specified by the MPL specification. In Figure 5.3, packets are created by calling the appropriate creation function; e.g. `Example.Sig.Exit.t` creates a "Exit Signal" packet in our protocol. Two parameters need to be specified: (*i*) the labelled integer argument `code` (dynamically checked to be of range 0–255 or an exception is raised); and (*ii*) a packet environment to write the packet contents into. Specifying large byte-arrays is more problematic, since simply specifying a string will result in excessive data copying if using layered protocol stacks (e.g. TCP/IP). We use function currying to create a *packet suspension* which has the type (env → o) and can be created by omitting the environment argument when invoking the packet creation function; until an environment is given to this suspension, it is not evaluated. From Figure 5.3, `Example.x` is the variant type used to store packet suspensions for different packet types. Due to the support for suspensions and fragments, byte arrays use the following type in packet creation functions:

```
type data = [                                              OCAML
      | 'Str of string
      | 'Sub of env → unit
      | 'Frag of frag
      | 'None ]
```

This permits a string, suspension, fragment or null value to be used as the contents of a byte array. If a suspension is specified, the function will always be evaluated with a view beginning from 0. This permits, for example, ICMP packets to be created as packet suspensions, passed into an IPv4 packet suspension, which is finally passed into an Ethernet packet creation function. The Ethernet function will write its headers and evaluate the IPv4 packet suspension in the correct place in the packet, which in turn evaluates the ICMP packet suspension. After the IPv4 variable-length body has been written, the Ethernet layer automatically fills in the `length` field with the correct value (calculated via the **value** attribute from the Ethernet MPL spec in Appendix C.1). As we will see later (§5.4), combinator functions can be defined as a succinct and type-safe way to create packets in this fashion without unnecessary data copying.

5.3.3 Packet Proxies

The final category of packet parsing our interface must deal with is packet proxies. These are a combination of packet sinks and sources, which read in a packet, modify it, and output the result without having to re-create it from scratch. A common example of this idiom is an IP router, which inspects packet headers, adjusts a few fields such as checksum and time-to-live, and transmits the packet towards its next hop.

Support for packet proxies is possible due to our requirement that an instantiated packet object always maintains a consistent wire-format representation of the packet in its encapsulated packet environment. Thus, irrespective of how the packet object was created (i.e. from a packet sink or packet source), we can invoke methods on the packet object to change the value of a field, and changes will be immediately reflected in the underlying environment. The `Example.Sig.Exit` packet object from Figure 5.3 now has the following, complete interface:

```
class Example.Sig.Exit.o : env → object                    OCAML
      method code : int
      method set_code : int → unit
      method env : env
end
```

The `set_code` method marshals an integer at the correct offset in the packet environment, and leaves the rest of it unmodified. Currently, the output OCaml code does not generate `set_` methods for variable-length fields such as byte arrays. Although initially implemented, the functionality was not useful since changing the length of the structure would result in effects

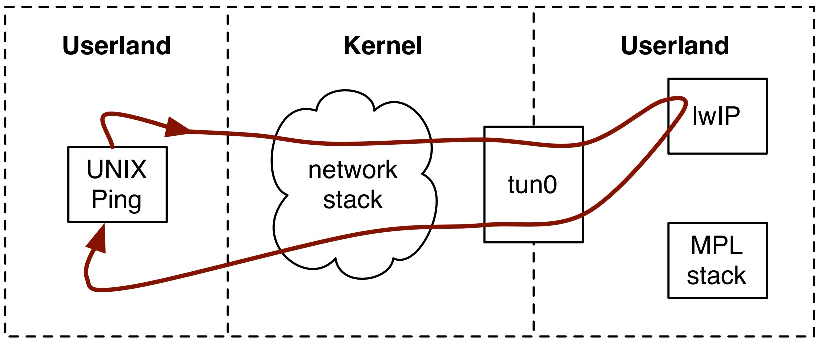


Figure 5.4: Architecture of our evaluation setup; the red line shows the data path of a single ICMP echo packet through the kernel into the user-level stack being evaluated.

through the entire packet (e.g. recalculation of length fields, and the moving of data following the field). In practise, proxies for many protocols (e.g. ICMP, IP) are designed to minimise the disruption caused to the packet payload [163] and so this is a reasonable restriction.

5.4 Evaluation

We have described the MPL specification language and the structure of the OCaml code output from the MPL compiler. We now evaluate the effectiveness of the MPL output in terms of its performance and latency. Since at this stage we wish to isolate the packet parsing aspect of MPL from the rest of the MELANGE architecture, we choose a simple network application: an ICMP echo server[4]. The ICMP protocol allows hosts to send "ping" packets to each other, which are returned unmodified to the originator. The transmitting host generally encodes a timestamp in the packet being sent, which can be read back in the response to determine the time-of-flight of the ping over the network. This protocol is a good choice for our experiments since it allows us to vary the size of ICMP pings, and gauge how well the MPL code scales over a variety of packet sizes.

5.4.1 Experimental Setup

Our benchmarks are performed on the OpenBSD 3.8 operating system, running on a 3.00GHz Pentium IV with 1GB RAM. The applications use the `tuntap` interface provided by OpenBSD which allows userland applications to send and receive network traffic by opening a `/dev/tun` interface and sending and receiving raw Ethernet (in the "tap" mode) or IPv4 packets (in the "tun" mode). As a reference implementation, we benchmark against the popular lwIP user-level networking stack [97]. lwIP is written in C and thus does not offer automatic garbage collection or dynamic bounds checking; nonetheless, it is a good way to measure the throughput of our OCaml ICMP echo server with another user-land implementation.

The experimental architecture is illustrated in Figure 5.4. A `tun0` interface is established, and configured with an IPv4 address and netmask. Either the lwIP or the MPL server (depending on which stack is being used) opens the file `/dev/tun0`, which binds that user process to the `tun0` interface. The `ping` command is executed on the same machine with the destination address equal to the IPv4 address of the `tun0` interface. The ICMP Echo Request packets from

[4]It is worth noting that even relatively simple utilities such as the SunOS `ping` have suffered from buffer overflows [73, 259]. These have led to root-level exploits due to the use of the setuid bit which permits `ping` to open raw network sockets.

this `ping` are routed to the user-level networking stack, which receives the raw IP packet as the result of the *read(2)* system call on the `/dev/tun0` file descriptor. The user-level stack processes the packet, and transmits a response via the *write(2)* system call on the same file descriptor. This packet is then injected into the kernel routing tables, and (assuming it is a valid ICMP echo response packet), is sent to the `ping` program which processes and prints out the time taken for the ping response to arrive.

During the experimental runs, only essential processes are running on the OpenBSD machine (e.g. `cron` and `syslogd` are killed to ensure they do not interfere with the timings), and logging is performed onto a memory file system to reduce jitters caused by physical disk accesses. We test over a variety of ICMP payload sizes; each size is repeated with 150 ICMP echo packets, and we increase our payload size by 32 until we reach the maximum MTU for the experiment. There is a delay of 0.5 seconds between each transmission. We plot the mean[5] of every set of 150 pings and use least-squares regression fitting to obtain a best-fit line against the mean values of each payload size.

lwIP Setup

The lwIP [97] TCP/IP stack is a mature user-level networking stack which we use as the reference implementation to measure our code against. It is a much more accurate benchmark than comparing against the kernel TCP/IP stack since it accounts for the extra overhead imposed by the `tuntap` interface. lwIP was compiled with the recommended optimisation level for the OpenBSD `gcc` compiler (-O2), and all debugging code was disabled to ensure maximum throughput. Unfortunately, lwIP only supports a maximum interface MTU of 1500, and so we have no results for the stack past that point.

The only aspect of lwIP we measure is its ICMP echo response handling. Examination of the code reveals that lwIP performs an extremely efficient response to ICMP echo requests by performing the following steps: (*i*) the raw IPv4 packet is read and classified as an ICMP echo request; (*ii*) the IP source and destination fields are swapped (which does not require a recalculation of the IPv4 header); (*iii*) the ICMP "packet type" byte (the first byte in the ICMP header) is modified to the value of an ICMP Echo Reply; (*iv*) the ICMP checksum is adjusted by performing a one's complement addition of the difference in the header constants modified in the previous step. The modified buffer is then directly transmitted back to the client.

This method of handling ICMP echo requests is efficient even for very large ICMP payload sizes since the packet payload is only iterated over once (to verify the incoming packet's checksum). The outgoing checksum is simply adjusted by a constant amount, and the payload does not need to be copied out of the original receive buffer.

OCaml ICMP Echo Server

We use the MPL specification for IPv4 (Appendix C.2) and ICMP (Appendix C.3) to provide the packet-parsing code for the OCaml ICMP echo server. We first define utility functions to perform IP and ICMP packet checksumming in OCaml, shown below:

[5]The 95% confidence interval for each set of pings is less than 0.01 s and thus not drawn for clarity.

```
let ones_checksum sum =                                                OCAML
    0xffff - ((sum lsr 16 + (sum land 0xffff)) mod 0xffff)
let icmp_checksum env =
    let header_sum = Mpl_uint16.unmarshal env in
    Mpl_stdlib.skip env 2;
    let body_sum = Mpl_uint16.dissect (+) 0 env in
    ones_checksum (header_sum + body_sum), body_sum
```

The `icmp_checksum` function accepts an environment which is a view of the ICMP header and body. The header byte is unmarshalled as a `uint16` and the checksum bytes are skipped (since they must be considered to be 0 during a checksumming). The subsequent body of the packet (which composes the bulk of the data) is dissected into chunks of `uint16` values and summed. Finally, a tuple is returned consisting of the ICMP checksum and the sum of the body bytes (useful to recalculate the ICMP checksum later). We now have all the functions required to write an ICMP echo server which can verify the checksum of incoming ICMP echo requests, and generate valid responses.

5.4.2 Experiments and Results

We chose the ICMP ping test because it is such a simple protocol, allowing us to isolate overhead of the MPL API for transferring data to and from the network. By varying the size of the payload contained inside the ping payload, we break down the API overhead into: (*i*) a fixed overhead in processing the packet and its constant-size headers; and (*ii*) the overhead in handling the larger variable-size payload (shown by the gradient of the lines in our graphs). The overhead of handling the payload is interesting since it highlights the cost of data copying and bounds checking, which MPL was minimises.

We create three different implementations of an ICMP echo server: (*i*) the "copying" version creates a new ICMP payload as a string, and copies that string into the IPv4 response packet; (*ii*) a "normal" version which creates a new IPv4 response packet and directly copies the incoming payload into the response via the MPL fragment support; and (*iii*) a "reflecting" server which directly modifies the incoming packet to convert it into a response and re-transmits it (matching the mechanism used by lwIP). For each version, we also ran exactly the same tests but with automatic bounds checking turned off[6]. The difference between the safe and unsafe versions highlights any bounds checking overhead versus the cost of data copying.

Figure 5.5 shows the results of the copying and normal echo server, with the lwIP results also included as a reference. The copying server clearly does more work per byte of payload than the lwIP stack as latencies increase as packet size grows. Interestingly, there is no significant difference between the safe and unsafe versions of the copying server, which we attribute to the string copying priming the CPU cache before ICMP checksumming (a phenomenon also noted by Miller and De Raadt with their safe C string API [200]).

Figure 5.5 also plots the performance of the normal server which uses the MPL fragment API to avoid an extra data copy. The unsafe version is now of equal speed to the lwIP server, but with the bounds checking turned off is as vulnerable to buffer overflows. The safe version is slightly slower; code inspection revealed that this is because the ICMP checksum is actually calculated *twice*; once when unmarshalling the payload, and secondly when creating the response packet. lwIP does not recalculate the checksum, instead simply adjusting it in-place

[6]In OCaml, this is the `-unsafe` flag to the compiler, and all calls in the MPL standard library to `String.{get, put}` were replaced with `String.unsafe_{get, put}`. The exception was the `dissect` function, discussed later in this section.

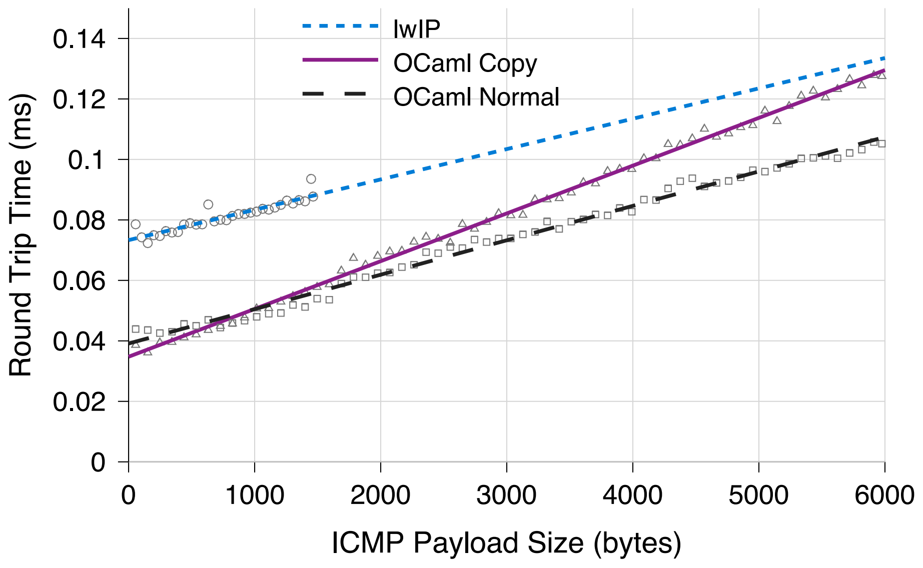


Figure 5.5: Performance of the normal and copying OCaml ICMP echo servers

(which is possible due to the weak nature of the ones-complement sum used by IP and ICMP).

In order to address this extra checksumming and see if a safe MPL implementation could match the lwIP stack, we implemented the reflecting echo server which adjusts the checksum in-place as lwIP does. The reflecting implementation of the echo server is less elegant than the normal version, since it violates layering and directly uses the MPL API to adjust the "ICMP echo type" before adjusting the check-sum. However, it is still statically type-safe when compared to lwIP (i.e. not vulnerable to buffer overflows through poorly written OCaml code), and we argue that this flexibility to optimise is an important feature. For code where elegance is more important, our normal echo server (from Figure 5.5) still shows a close parity with only a small speed difference.

At the start of this section we showed the code used for ICMP checksumming, which uses the `Mpl_uint16.dissect` function to represent an environment as a list of unsigned 16-bit integers. The implementation of `dissect` eliminates redundant bounds checks by performing a single bounds check at the start of the call. To test how much of a difference this actually makes, we tested the reflection server with two versions of the MPL basis library: (*i*) with the redundant bounds checks still present; and (*ii*) with the optimised `dissect` implementation which coalesces them. The results in Figure 5.6 show a dramatic difference in performance, with the optimised version performing on parity with lwIP and the overly safe version being as slow as our copying echo server (Figure 5.5).

We conclude from these performance tests that the reduced data copying approach by MPL is effective, and that the separation of auto-generated MPL code from the MPL basis library gives developers the flexibility to "break the abstraction" and perform low-level optimisations when matching the performance of C code is required. The MPL basis library provides packet iterators which coalesce redundant bounds checks which results in a large performance improvement over the standard OCaml bounds checks.

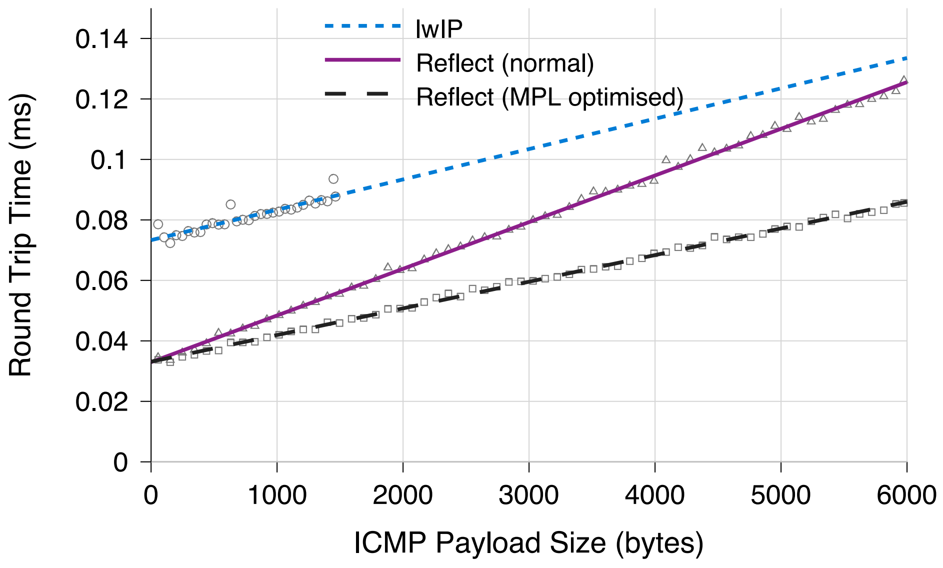


Figure 5.6: Performance of the reflecting OCaml ICMP echo server with both slow checksumming code and the MPL-optimised version

5.5 Discussion

In our introduction to MPL (§5.1.1), we noted that objects in OCaml are *structurally* typed instead of by name. In the context of MPL, this means that if a function is constrained to accept a particular packet object through a type annotation, another packet object which happens to have the exact same method signature would type-check successfully. Although this is sound in the sense of the OCaml type system, it is almost certainly not what the programmer intended and should be rejected at compile time. Another restriction which is difficult to statically enforce is the rule that packet objects should not be instantiated directly, instead going via accessor functions. If a packet object is instantiated by mistake, the environment it encapsulates will be in an inconsistent state.

These problems would not exist using the normal module system; OCaml provides support for "private" variant types which can be pattern-matched externally to a library, but only created via accessor functions in the library. However, Jacques Garrigue recently added support for "private row types" [120] into OCaml which extend these private variants to object signatures hidden inside modules. Private row types permit object interfaces to be exported from a module which can have method calls invoked on them as normal, but can only be *created* from within the module itself. This solves both of the problems with the existing MPL interface, at the cost of a more complex type specification in the auto-generated MPL output (the external interface remains the same).

Private row types have just been added to the most recent version of OCaml (3.09), and came too late to be evaluated as part of this thesis; however their addition does solve two outstanding problems with our interface that we felt were important to highlight. We plan to introduce support for private row types at a later date.

5.6 Summary

In this chapter we have described the Meta Packet Language (MPL) (§5.1), the OCaml basis library it uses (§5.2) and the generated OCaml interface (§5.3). We isolated packet parsing performance by implementing an OCaml ICMP echo server using MPL and measuring its performance against the standard lwIP user-level networking stack (§5.4). We concluded that the OCaml ICMP server matched lwIP in terms of latency and the amount of per-byte overhead for varying packet sizes.

This validates our approach of using OCaml as the target language for auto-generated packet parsing code rather than C since the static safety properties guaranteed by the OCaml type system are much stronger than those provided by C, and our performance results find no intrinsic overhead to the use of OCaml for parsing low-level protocols such as ICMP and IP.

CHAPTER 6

Statecall Policy Language

A computer lets you make more mistakes faster than any invention in human history—with the possible exceptions of handguns and tequila.
MITCH RATLIFFE

The end-to-end principle that most Internet protocols are based upon [239] requires the host software to encapsulate a significant amount of complex state and deal with a variety of incoming packet types, complex configurations and versioning inconsistencies. Network applications are also expected to be liberal in interpreting received data packets and must reliably deal with timing and ordering issues arising from the "best-effort" nature of Internet data traffic. Constructing software to deal with this complexity is difficult due to the large state machines which result, and mechanical verification techniques are very useful to guarantee safety, security and reliability properties.

One mature formal method used to verify properties about systems is *model checking*. Conventional software model-checking involves (*i*) creating an abstract model of a complex application; (*ii*) validating this model against the application; and (*iii*) checking safety properties against the abstract model. To non-experts, steps (*i*) and (*ii*) are often the most daunting. Firstly how does one decide which aspects of the application to include in the abstract model? Secondly, how does one determine whether the abstraction inadvertently "hides" critical bugs? Similarly, if a counter-example is found, how does one determine whether this is a genuine bug or just a modelling artifact?

In this chapter, we present the Statecall Policy Language (SPL) which simplifies the model specification and validation tasks with a view to making model checking more accessible to regular programmers. SPL is a high-level modelling language, SPL which enables developers to specify models in terms of allowable program events (e.g. valid sequences of received network packets). We have implemented a compiler that translates SPL into both PROMELA and a general-purpose programming language (e.g. OCaml). The generated PROMELA can be used with SPIN [138] in order to check static properties of the model. The OCaml code provides an executable model in the form of a *safety monitor*. A developer can link this safety monitor against their application in order to *dynamically* ensure that the application's behaviour does

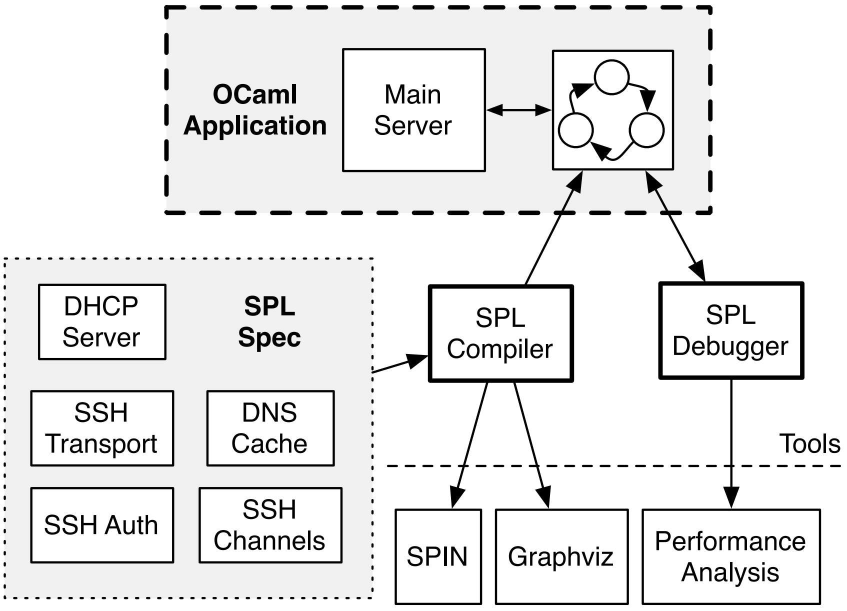


Figure 6.1: The SPL tool-chain architecture

not deviate from the model. If the safety monitor detects that the application has violated the model then it logs this event and terminates the application. This architecture is illustrated in Figure 6.1.

Although this technique simplifies model specification and validation it is, of course, not appropriate for all systems. For example, dynamically shutting down a fly-by-wire control system when a model violation is detected is not an option. However, we observe that there *is* a large class of applications where dynamic termination, while not desirable, is preferable to (say) a security breach. In particular, this thesis focusses on implementing Internet applications securely and correctly, and SPL delivers real benefits in this area. None of the major implementations of protocols such as HTTP (Apache), SMTP (Sendmail/Postfix), or DNS (BIND) are regularly model-checked by their development teams. All of them regularly suffer from serious security flaws ranging from low-level buffer overflows to subtle high-level protocol errors (§2.1.3), some of which could have been caught by using model checking.

There is no "perfect" way of specifying complex state machines, and the literature contains many different languages for this purpose (e.g. SDL [247], Estelle [148], Statemate [129], or Esterel [34]). In recognition of this, the SPL language is very specialised to expressing valid sequences of packets for Internet protocols and is translated into a more general intermediate "Control Flow Automaton" representation first proposed by Henzinger et al. [133]. All of the output code targets are generated from this intermediate graph, allowing for other state machine languages to be used in the future without requiring the backend code generators to be rewritten.

This chapter first describes the syntax, type rules and semantics of the SPL language (§6.1). Next we define the intermediate representation that SPL specifications are translated into (§6.2), and finally the output languages from the SPL compiler (in particular OCaml, PROMELA and HTML) (§6.3).

6.1 Statecall Policy Language

We now define the SPL language, firstly with a simple case study (§6.1.1), and then with its syntax (§6.1.2) and the type checking rules for a valid specification (§6.1.3).

6.1.1 A Case Study using `ping`

SPL is used to specify sequences of events which represent non-deterministic finite state automata. The automaton inputs are referred to as *statecalls*—these can represent any program events such as the transmission of receipt of network packets or the completion of some computation. The syntax of the language is written using a familiar 'C'-like syntax, with built-in support for non-deterministic choice operators in the style of Occam's ALT [155]. Statecalls are represented by capitalized identifiers, and SPL functions use lower-case identifiers. Semicolons are used to specify sequencing (e.g. `S1; S2` specifies that the statecall `S1` must occur before the statecall `S2`).

Before specifying SPL more formally, we explain it via a simple case study. Earlier in Chapter 5 we described how to use the MPL interface language to send and receive ICMP frames. We now consider the state machine behind the UNIX `ping` utility which transmits and receives ICMP Echo requests and measures their latencies. An extremely simple `ping` automaton with just 3 statecalls could be written as:

```
automaton ping() {                                                    SPL
    Initialize;
    Transmit_Ping;
    Receive_Ping;
}
```

This simple automaton guarantees that the statecalls must operate in the following order: `Initialize`, `Transmit_Ping`, and `Receive_Ping`. A more realistic implementation of ping transmits and receives packets continuously. To represent this, we provide the `multiple` keyword in our SPL specification; the example below specifies that one or more iterations must occur after initialisation.

```
automaton ping() {                                                    SPL
    Initialize;
    multiple (1..) {
        Transmit_Ping;
        Receive_Ping;
    }
}
```

Using this automaton, the `ping` process can perform initialisation once, and then transmit and receive ping packets forever; an attempt to initialise more than once is not permitted. In a realistic network a ping response might never be received, and the non-deterministic `either/or` operator allows programmers to represent this scenario.

```
automaton ping() {                                                    SPL
    Initialize;
    multiple (1..) {
        Transmit_Ping;
        either {
            Receive_Ping;
        } or {
            Timeout_Ping;
        };
    }
}
```

ping provides a number of command-line options that can modify the program behaviour. For example, `ping -c 10` requests that only 10 ICMP packets be sent in total, and `ping -w` specifies that we must never timeout, but wait forever for a ping reply. We represent these constraints by introducing *state variables* into SPL as follows:

```
automaton ping(int max_count, int count, bool can_timeout) {          SPL
    Initialize;
    count = 0;
    do {
        Transmit_Ping;
        either {
            Receive_Ping;
        } or (can_timeout) {
            Timeout_Ping;
        };
        count = count + 1;
    } until (count > max_count);
}
```

Observe that the `either`/`or` constructs can be conditionally guarded in the style of Occam's ALT, and state variables can be assigned to in an imperative style. Finally, a long-running `ping` process would need to receive UNIX signals at any point in its execution, take some action, and return to its original execution. Signal handlers are often a source of bugs due to their extremely asynchronous nature [65]—SPL provides a `during`/`handle` construct which models them by permitting a state transition into alternative statement blocks during normal execution of an SPL specification.

```
SPL
automaton ping(int max_count, int count, bool can_timeout) {
    Initialize;
    during {
        count = 0;
        do {
            Transmit_Ping;
            either {
                Receive_Ping;
            } or (can_timeout) {
                Timeout_Ping;
            };
            count = count + 1;
        } until (count > max_count);
    } handle {
        Sig_INFO;
        Print_Summary;
    };
}
```

Once we are satisfied that our SPL specification is of suitable granularity, the SPL compiler is run over it. The compiler outputs several targets: (*i*) a graphical visualisation using the Graphviz tool [117] as seen in Figure 6.2 for the example above; (*ii*) a non-deterministic model in the PROMELA language; and (*iii*) an executable model designed to be linked in with an application. The OCaml interface for the executable model is shown below:

```
OCAML
exception Bad_statecall
type t = [ `Initialize | `Print_summary | `Receive_ping
| `Sig_info | `Timeout_ping | `Transmit_ping ]
type s
val init : max_count:int → count:int → can_timeout:bool → unit → s
val tick : s → t → s
```

This code is linked in with the main `ping` application, and appropriate calls to initialize the automaton and invoke statecalls are inserted in the code. Crucially, we do not mandate a single style of invoking statecalls; instead the programmer can choose between automatic mechanisms (e.g. MPL code can automatically invoke statecalls when transmitting or receiving packets), language-assisted means (e.g. functional combinators, object inheritance, or pre-processors such as `cpp`), or even careful manual insertion in places where other methods are inconvenient.

6.1.2 Syntax

The SPL syntax is presented in Figure 6.3 using an extended Backus-Naur Form [16]. We represent terminals as *term*, tokens as **`token`**, alternation with {*one* | *two*}, optional elements as [*optional*], elements which must repeat once or more as (*term*)+ and elements which may appear never or many times as (*term*)*.

SPL source files are parsed using the `yacc` [154] implementation in OCaml, and represented in the abstract syntax tree shown in Figure 6.4.

6.1.3 Typing Rules

SPL is a first order imperative language, extended from Cardelli's simple imperative language [58]. We distinguish between *commands* (without a return value) and *expressions* which do have a

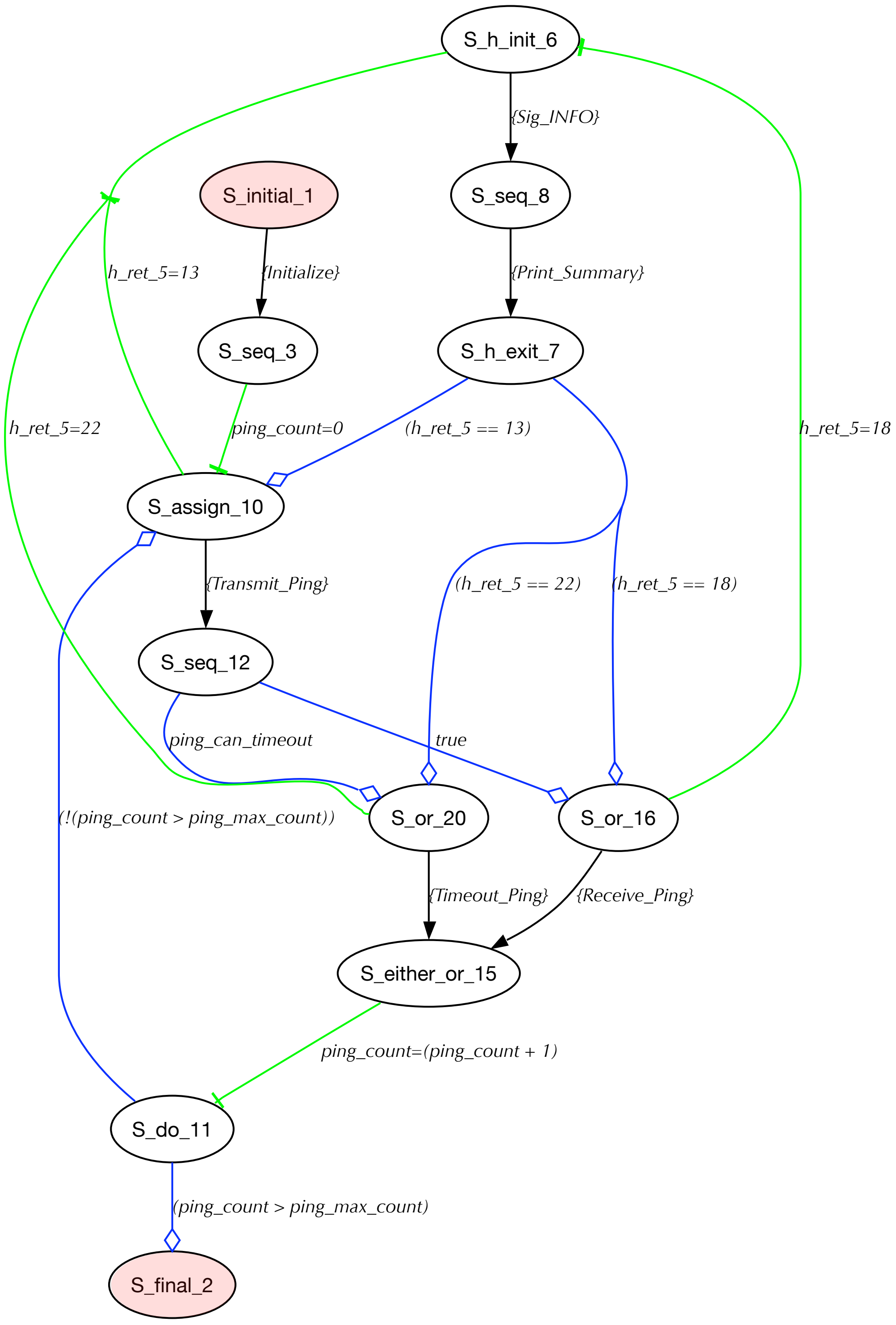


Figure 6.2: Graph output of the example `ping` state machine. Red nodes indicate the start and final states, black edges are statecalls, blue edges are conditional, and green edges are state variable assignments

$$
\begin{array}{rcl}
\mathit{main} & \rightarrow & (\mathit{fdecl})\text{+}\ \mathit{eof} \\
\mathit{fdecl} & \rightarrow & \{\texttt{automaton} \mid \texttt{function}\}\ \mathit{id}\ \mathit{fargs}\ \mathit{fbody} \\
\mathit{fargs} & \rightarrow & \texttt{(}\ \{\texttt{int}\ \mathit{id} \mid \texttt{bool}\ \mathit{id}\}\ [\texttt{,}\ \mathit{fargs}]\ \texttt{)} \\
\mathit{fcall\text{-}args} & \rightarrow & \mathit{id}\ [\texttt{,}\ \mathit{fcall\text{-}args}] \\
\mathit{statecall\text{-}args} & \rightarrow & \mathit{statecall}\ [\texttt{,}\ \mathit{statecall\text{-}args}] \\
\mathit{fbody} & \rightarrow & \{\ (\mathit{statement})^{*}\ \}\ [\texttt{;}] \\
\mathit{int\text{-}range} & \rightarrow & \texttt{(}\ [\mathit{int}]\ \texttt{..}\ [\mathit{int}]\ \texttt{)} \quad \mid \quad \texttt{(}\ \mathit{int}\ \texttt{)} \\
\mathit{statement} & \rightarrow & \mathit{statecall}\ \texttt{;} \quad \mid \quad \mathit{id}\ \texttt{(}\ \mathit{fcall\text{-}args}\ \texttt{)}\ \texttt{;} \\
 & \mid & \texttt{always-allow}\ \texttt{(}\ \mathit{statecall\text{-}args}\ \texttt{)}\ \mathit{fbody} \\
 & \mid & \texttt{multiple}\ \mathit{int\text{-}range}\ \mathit{fbody} \quad \mid \quad \texttt{optional}\ \mathit{fbody} \\
 & \mid & \texttt{either}\ \mathit{guard}\ \mathit{fbody}\ (\texttt{or}\ \mathit{guard}\ \mathit{fbody})\text{+} \\
 & \mid & \texttt{do}\ \mathit{fbody}\ \texttt{until}\ \mathit{guard}\ \texttt{;} \\
 & \mid & \texttt{while}\ \mathit{guard}\ \mathit{fbody} \\
 & \mid & \mathit{id}\ \texttt{=}\ \mathit{expr}\ \texttt{;} \\
 & \mid & \texttt{during}\ \mathit{fbody}\ (\texttt{handle}\ \mathit{fbody})\text{+} \\
 & \mid & \texttt{exit ;} \quad \mid \quad \texttt{abort ;} \\
\mathit{guard} & \rightarrow & \texttt{(}\ \mathit{expr}\ \texttt{)} \\
\mathit{expr} & \rightarrow & \mathit{int} \quad \mid \quad \mathit{id} \quad \mid \quad \texttt{(}\ \mathit{expr}\ \texttt{)} \\
 & \mid & \mathit{expr}\ \texttt{+}\ \mathit{expr} \quad \mid \quad \mathit{expr}\ \texttt{-}\ \mathit{expr} \\
 & \mid & \mathit{expr}\ \texttt{*}\ \mathit{expr} \quad \mid \quad \mathit{expr}\ \texttt{/}\ \mathit{expr} \\
 & \mid & \texttt{-}\ \mathit{expr} \quad \mid \quad \texttt{true} \quad \mid \quad \texttt{false} \\
 & \mid & \mathit{expr}\ \texttt{\&\&}\ \mathit{expr} \quad \mid \quad \mathit{expr}\ \|\ \mathit{expr} \quad \mid \quad \texttt{not}\ \mathit{expr} \\
 & \mid & \mathit{expr} > \mathit{expr} \quad \mid \quad \mathit{expr} >= \mathit{expr} \\
 & \mid & \mathit{expr} < \mathit{expr} \quad \mid \quad \mathit{expr} <= \mathit{expr} \\
 & \mid & \mathit{expr} = \mathit{expr}
\end{array}
$$

Figure 6.3: EBNF grammar for SPL specifications

A	$\leftarrow$		*state variable types*
	$\mid$	$Bool$	*boolean*
	$\mid$	Int	*unsigned integer*
	$\mid$	$Unit$	*unit*
S	$\leftarrow$	$<statecall>$	*statecall*
D	$\leftarrow$		*declarations*
	$\mid$	$\textbf{fun}\ I\ (V_1 : A_1 \ldots V_n : A_n) = C$	*function declaration*
	$\mid$	$\textbf{auto}\ I\ (V_1 : A_1 \ldots V_n : A_n) = C$	*automaton declaration*
C	$\leftarrow$		*commands*
	$\mid$	S	*statecall*
	$\mid$	$S_1 ; S_2$	*sequencing*
	$\mid$	$\textbf{allow}\ (S_1 \ldots S_i)\ = C$	*always allow statecalls*
	$\mid$	$\textbf{either}\ (C_1 \times E_1)(C_2 \times E_2) \ldots (C_j \times E_j)$	*guarded alternation*
	$\mid$	$\textbf{multiple}\ E_1\ E_2 = C$	*multiple*
	$\mid$	$\textbf{until}\ E_1 = C$	*do until*
	$\mid$	$\textbf{while}\ E_1 = C$	*while*
	$\mid$	$\textbf{handle}\ C_1(C_2 \ldots C_n)$	*during handle*
	$\mid$	$\textbf{exit}$	*normal exit*
	$\mid$	$\textbf{abort}$	*error exit*
	$\mid$	$\textbf{call}\ I\ (E_1 \ldots E_n)$	

Figure 6.4: Abstract Syntax Tree for SPL

Table 6.1: Type Judgments for SPL

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash C$	C is a well-formed command in Γ
$\Gamma \vdash E : A$	E is a well-formed expression of type A in Γ

return value. Function and automaton names are distinct, and are considered commands. Function types are written $\rho_1 \times \ldots \times \rho_i$, or abbreviated to $\vec{\rho}$. Γ_α represents a global environment with type signatures for functions and Γ a per-function environment containing state variable bindings. SPL does not have any built-in functions, so all type signatures are obtained from the SPL specifications.

Table 6.1 lists the imperative type judgements and Table 6.2 establishes the basic typing rules. Note that procedure environments contain only the variables passed in as arguments to the function declaration, and no global variables are permitted. Table 6.3 and Table 6.4 list the type rules for expressions and statements respectively.

6.2 Intermediate Representation

This section first defines the Control Flow Automaton graph used as an intermediate representation of SPL specifications (§6.2.1), the semantics of multiple automata in the same SPL specification (§6.2.2), and finally optimisations applied to the CFA to reduce the number of states

Table 6.2: Basic environment and typing rules

(ENV ϕ)
$$\frac{}{\phi \vdash \diamond}$$

(ENV X)
$$\frac{\Gamma \vdash A \qquad I \notin dom(\Gamma)}{\Gamma, I : A \vdash \diamond}$$

(TYPE INT)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Int}$$

(TYPE BOOL)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Bool}$$

(DECL PROC)
$$\frac{\phi, \vec{x} : \vec{\rho} \vdash C \qquad \Gamma_\alpha, I : \vec{\rho} \vdash \diamond}{\Gamma_\alpha \vdash (\mathbf{fun}\ I\ (\vec{x} \times \vec{\rho}) = C)}$$

Table 6.3: Expression typing rules

(EXPR BOOL)
$$\frac{\Gamma \vdash \diamond \qquad x \in \{true, false\}}{\Gamma \vdash x : Bool}$$

(EXPR INT)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash N : Int}$$

(EXPR VAL)
$$\frac{\Gamma_1, I : A, \Gamma_2 \vdash \diamond}{\Gamma_1, I : A, \Gamma_2 \vdash I : A}$$

(EXPR NOT)
$$\frac{\Gamma \vdash E_1 : Bool}{\Gamma \vdash \mathbf{not}\ E_1 : Bool}$$

(EXPR BOOLOP)
$$\frac{\Gamma \vdash E_1 : Bool \qquad \Gamma \vdash E_2 : Bool \qquad O_1 \in \{\mathbf{and}, \mathbf{or}\}}{\Gamma \vdash O_1(E_1, E_2) : Bool}$$

(EXPR INTOP)
$$\frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int \qquad O_1 \in \{+, -, \times, \div\}}{\Gamma \vdash O_1(E_1, E_2) : Int}$$

(EXPR COMPOP)
$$\frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int \qquad O_1 \in \{=, >, \geq, <, \leq\}}{\Gamma \vdash O_1(E_1, E_2) : Bool}$$

Table 6.4: Command typing rules

(CMD ASSIGN)
$$\frac{\Gamma \vdash I : A \qquad \Gamma \vdash E : A}{\Gamma \vdash I \leftarrow E}$$

(CMD SEQUENCE)
$$\frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1 ; C_2}$$

(CMD ALLOW)
$$\frac{\Gamma \vdash C}{\Gamma \vdash \textbf{allow}\ C}$$

(CMD EITHER OR)
$$\frac{\Gamma \vdash C_{1..n} \qquad \Gamma \vdash E_{1..n} : \mathit{Bool}}{\Gamma \vdash \textbf{either}\ (C_1 \ldots C_n)}$$

(CMD DO UNTIL)
$$\frac{\Gamma \vdash E : \mathit{Bool} \qquad \Gamma \vdash C}{\Gamma \vdash (\textbf{until}\ E = C)}$$

(CMD WHILE)
$$\frac{\Gamma \vdash E : \mathit{Bool} \qquad \Gamma \vdash C}{\Gamma \vdash (\textbf{while}\ E = C)}$$

(CMD MULTIPLE)
$$\frac{\Gamma \vdash E_1 : \mathit{Int} \qquad \Gamma \vdash E_2 : \mathit{Int} \qquad \Gamma \vdash C}{\Gamma \vdash (\textbf{multiple}\ E_1\ E_2 = C)}$$

(CMD EXIT)
$$\frac{}{\Gamma \vdash \textbf{exit}}$$

(CMD ABORT)
$$\frac{}{\Gamma \vdash \textbf{abort}}$$

(CMD FUNCTION CALL)
$$\frac{\Gamma^1_\alpha, I : \vec{\rho}, \Gamma^2_\alpha \vdash \diamond \qquad \Gamma \vdash \vec{x} : \vec{\rho}}{\Gamma^1_\alpha, I : \vec{\rho}, \Gamma^2_\alpha \vdash \textbf{call}\ I\ \vec{x}}$$

(§6.2.3). The CFA is a good abstraction for a software-based non-deterministic model and it is often used by model extraction tools (e.g. BLAST [133]) as the representation into which C source code is converted. Since there are a myriad of state-machine languages similar to SPL which share the properties formalised by Schneider's software automata [242], our adoption of the CFA representation ensures that the back-ends of the SPL tool-chain (e.g. the PROMELA output) remain useful even if the front-end language is changed into something specialised for another task.

6.2.1 Control Flow Automaton

The SPL compiler transforms specifications into an extended Control Flow Automaton (CFA) [133] graph. A CFA represents program states and a finite set of state variables in blocks, with the edges containing conditionals, assignments, statecalls or termination operations. The CFA is non-deterministic and multiple states can be active simultaneously.

More formally, our *extended control flow automaton* **C** is a tuple $(Q, q_0, X, S, Op, \rightarrow)$ where Q is a finite set of control locations, q_0 is the initial control location, X a finite set of typed variables, S a finite set of statecalls, Op a set of operations, and $\rightarrow \subseteq (Q \times Op \times Q)$ a finite set of edges labeled with operations. An edge (q, op, q') can be denoted $q \xrightarrow{op} q'$. The set Op of operations contains: (*i*) *basic blocks* of instructions, which consist of finite sequences of assignments **svar = exp** where **svar** is a state variable from X and **exp** is an equivalently typed expression over X; (*ii*) *conditional predicates* **if(p)**, where **p** is a boolean expression over X that must be true for the edge to be taken; (*iii*) *statecall predicates* **msg(s)**, where **s** is a statecall (s $\in S$) received by the automaton; and (*iv*) *abort traps*, which immediately signal the termination of the automaton. From the perspective of a Mealy machine, the input alphabet Σ consists of statecall predicates and the output alphabet $\wedge$ is the remaining operations. Thus a CFA graph is driven purely by statecall inputs, and the other types of operations serve to hide the state space explosion of a typical software model.

The CFA graph is constructed from SPL statements by recursively applying transformation rules to an initial state I and a final state O. Figure 6.5 illustrates the transformations for the basic SPL statements diagrammatically with the circles and lines representing CFA nodes and edges. The diamonds indicate a recursive application of the transformation rules with the initial and final states mapped to the input and outputs of the diamond node. Nodes within the dashed ellipses (named α, β, γ and so on) are newly created by the transformation rule. The **abort** and **exit** keywords signal the end of the automaton and thus do not connect to their output states. Each transformation rule has an environment ($\Gamma \times \Delta$) where Γ is the list of always allowed statecalls as seen in **allow** blocks and Δ represents statecalls which result in a transition to a handle clause (generated by the **during/handle** statement). A **during/handle** statement first creates all the handler nodes and transforms the main block with the handlers registered in the Δ environment. A **statecall** node creates a statecall edge and inserts appropriate edges to deal with **allow** and **during** handlers.

Some statements require the creation of new internal variables. The **multiple** call can optionally specify upper and lower bounds to the number of iterations; extra variables are automatically created to track these bounds in the CFA. **during/handle** statements create a new internal variable to track the state to which a handler must return. Function calls are either macro-expanded (if only called once) or temporary variables used to push and pop arguments in a single copy of the function graph (if called multiple times). An example of these internal variables can be seen in Figure 6.2 in our earlier `ping` sample.

6.2.2 Multiple Automata

It is often more convenient and readable to break down a complex protocol into smaller blocks which express the same protocol but with certain aspects factored out into simpler state machines. Accordingly, SPL specifications can define multiple automata, but the external interface hides this abstraction and only exposes a single, flat set of statecalls. The scope of automata names are global and flat; this is a deliberate design decision since the language is designed for light-weight abstractions that are embedded into portions of the main application code. Even a complex protocol such as SSH can be broken down into smaller, more manageable automata (§D.1). In this section, we explain how statecalls are routed to the individual automata contained in an SPL specification.

Each automaton executes in parallel and sees every statecall. If an automaton receives a statecall it was not expecting it reports an error. If *any* of the parallel automata report an error then the SPL model has been violated. When a statecall is received, it is dispatched *only* to automata which can potentially use that statecall at some stage in their execution.

More formally, let $\mathcal{A}$ represent an automaton or function definition in an SPL specification. Let $\mathcal{V}(\mathcal{A})$ represent the union of all the statecalls referenced in $\mathcal{A}$, and $\mathcal{F}(\mathcal{A})$ be the list of all functions called from $\mathcal{A}$. The *potentially visible statecalls* $\mathcal{P}(\mathcal{A})$ are the set of statecalls which the automaton $\mathcal{A}$ will use at some stage in its execution where $\mathcal{P}(\mathcal{A}) = \mathcal{V}(\mathcal{A}) \cup \{\mathcal{P}(\mathcal{F}_0) \ldots \mathcal{P}(\mathcal{F}_n)\}$. A statecall is only dispatched to an automaton $\mathcal{A}$ if it is present in its potentially visible set $\mathcal{P}(\mathcal{A})$. Since the set of externally exposed statecalls $\mathcal{P}_{all} = \{\mathcal{P}(\mathcal{A}_0) \ldots \mathcal{P}(\mathcal{A}_n)\}$ is calculated by the union of all the potentially visible sets of the automata contained in an SPL specification, it trivially follows that every statecall will be dispatched to at least one automaton.

Figure 6.6 gives a sample specification of 3 automata α, β and γ to demonstrate these semantics more visually. The CFA graph and potentially visible statecalls for each automaton is

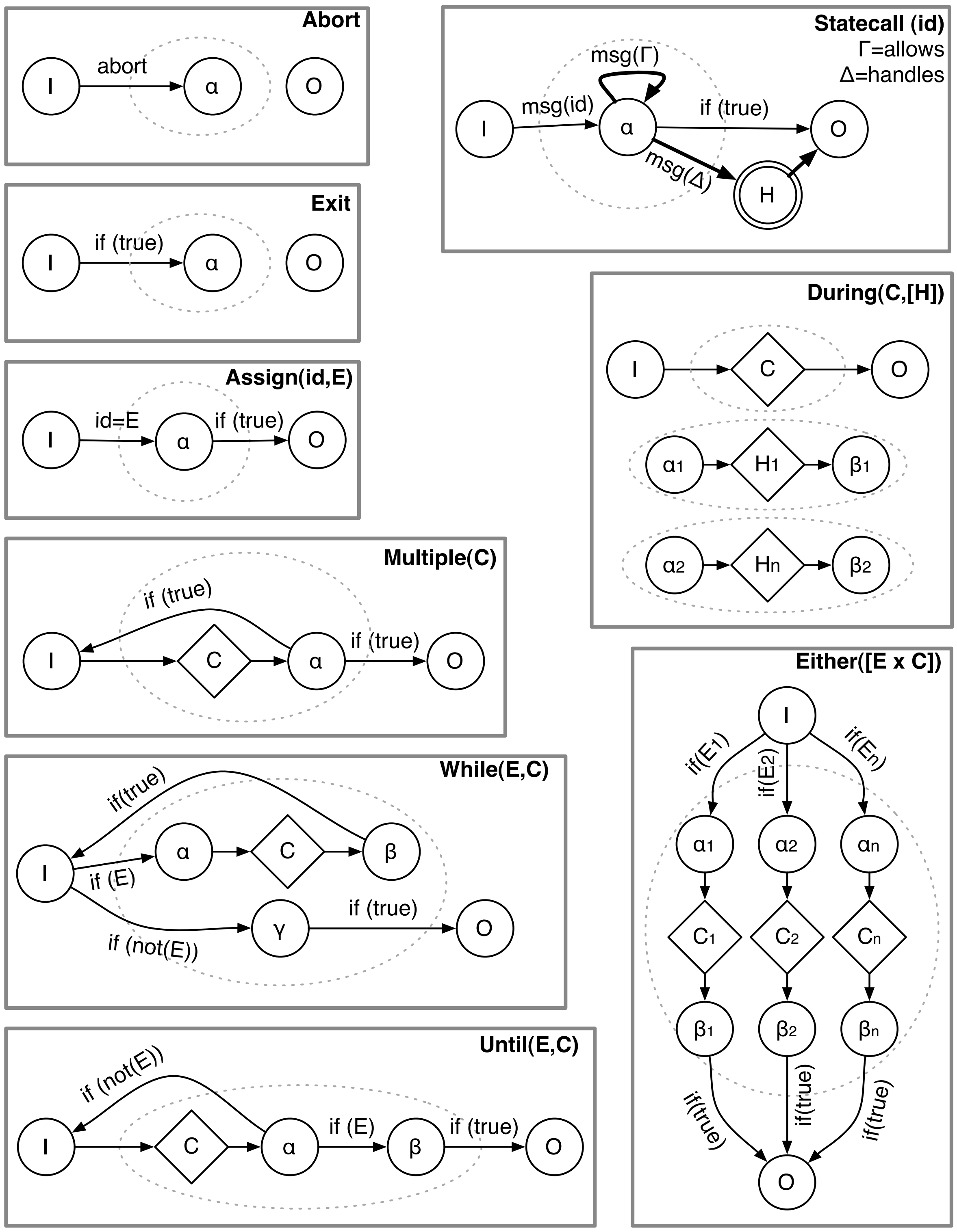
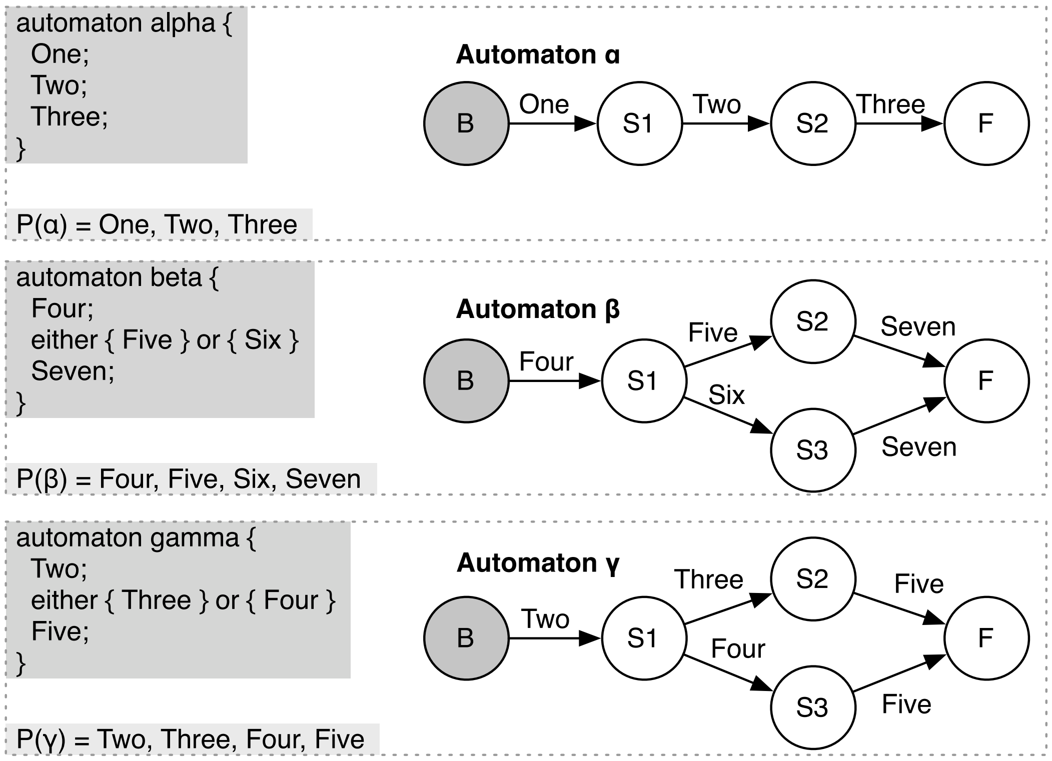


Figure 6.5: Transformations of SPL statements into the corresponding CFA nodes



```
automaton alpha {
  One;
  Two;
  Three;
}
```

P(α) = One, Two, Three

```
automaton beta {
  Four;
  either { Five } or { Six }
  Seven;
}
```

P(β) = Four, Five, Six, Seven

```
automaton gamma {
  Two;
  either { Three } or { Four }
  Five;
}
```

P(γ) = Two, Three, Four, Five

Figure 6.6: Initial automata states of the sample SPL specification

also shown; the circles represents nodes in the graph, all of the edges represent **msg** transitions and the gray circles indicate the initial active states.

Figure 6.7 illustrates the events that occur when the statecalls [One, Two, Three, Four] are sent to the automata. Firstly, One is only present in the potentially visible set of α and ignored by β and γ. A valid transition exists for One in α and the automaton performs the transition. Since none of the automata register an error, the statecall is successful. When Two and Three are sent, only β ignores them and α and γ successfully transition into new states. When Four is sent, α ignores it but β and γ have it in their potentially visible set and attempt to transition. β has a successful transition but γ does not and raises an error. Since one of the automata flagged an error, the safety monitor raises an exception.

As we will see later (§7.1) this mechanism allows complex protocols such as SSH to be broken down into simpler automata which are still connected together by common messages. The SPL compiler can output the list of statecalls which are shared between automata as a development aid; in practise while specifying Internet protocols we have observed that most automata share only one or two statecalls between them (normally global messages to indicate protocol termination or authentication status).

6.2.3 Optimisation

The transformation rules defined earlier (§6.2.1) result in a CFA which has a number of redundant edges (e.g. *if(true)* conditionals). In the interests of creating a correct compiler, these edges are optimised away in a separate phase once the initial CFA has been created. At this stage, the optimisation focusses on reducing the number of states in the CFA without modifying the semantics of the graph. We first iterate over all nodes and edges in the graph and perform constant folding [3] to simplify any conditional expressions. Since SPL only has expressions

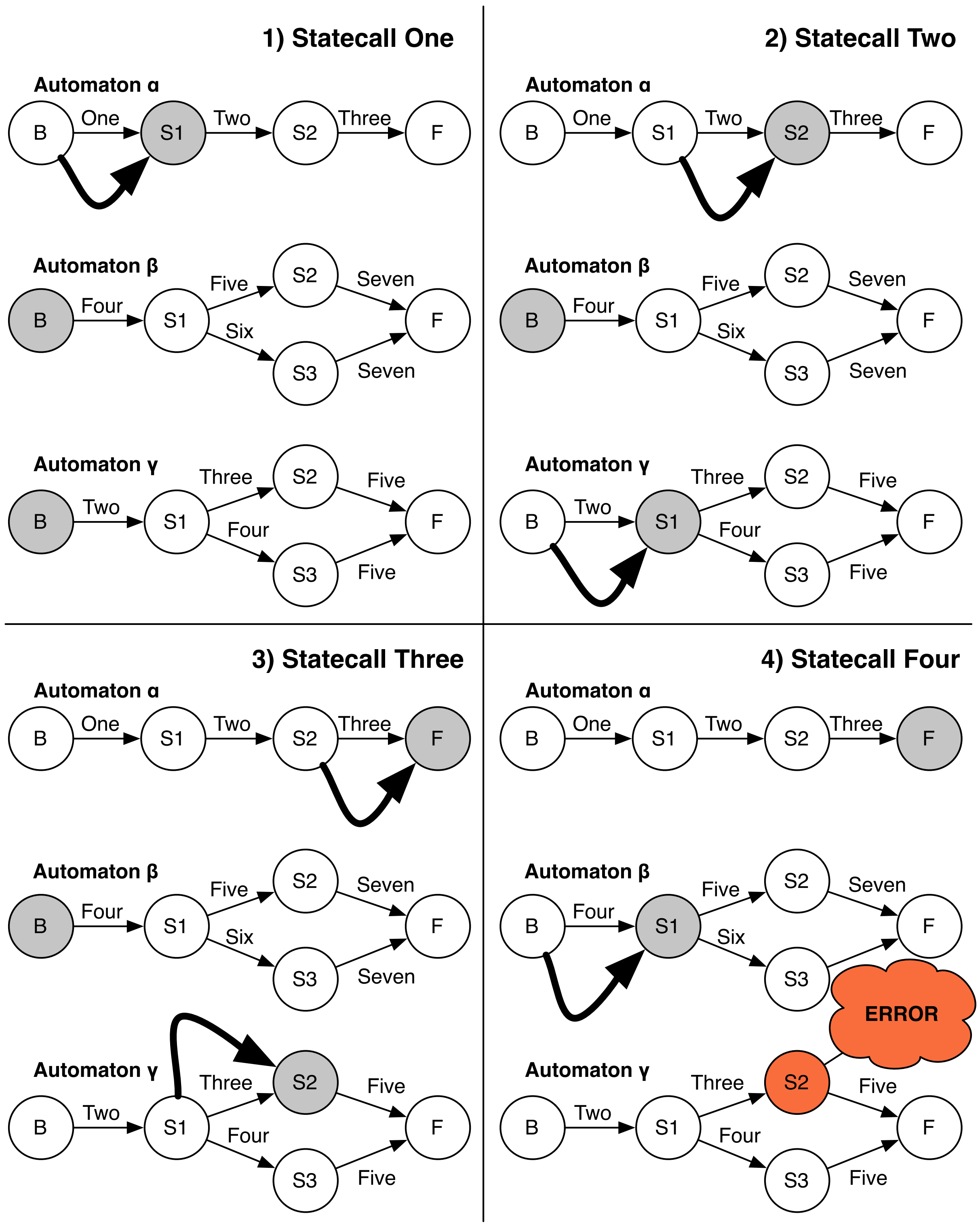
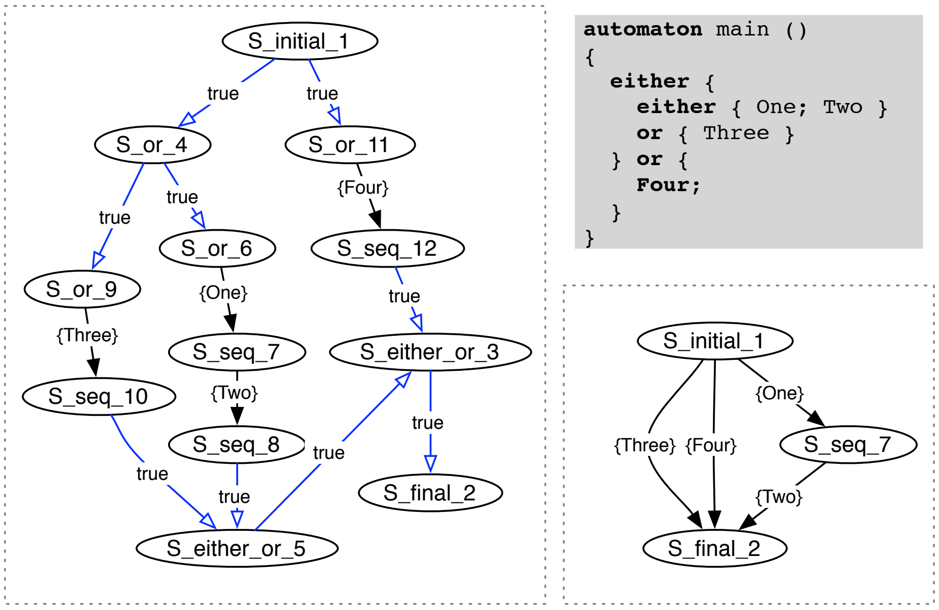


Figure 6.7: State transitions for our sample SPL specification with 4 input statecalls



```
automaton main ()
{
  either {
    either { One; Two }
    or { Three }
  } or {
    Four;
  }
}
```

Figure 6.8: Conditional elimination on a CFA *(left)* before, and *(right)* after

with booleans and integers, the folding is a simple recursive pattern match over the abstract syntax tree.

Once the constant folding is complete, the CFA is traversed to eliminate redundant nodes. A node is considered redundant if: (*i*) for a node Q_i, all edges from the node are of the form $Q_i \xrightarrow{if(true)} Q_o$ or (*ii*) for a node Q_o, all edges pointing to the node are of the form $Q_i \xrightarrow{if(true)} Q_o$. The initial state of the automaton is left unoptimised, in order to maintain each automaton as having only a single entry point for simplicity.

The conditional elimination optimisation is particularly important for CFA graphs generated from SPL source code, since the compiler deliberately inserts a number of redundant blocks around basic blocks in order to simplify the code generation algorithms. Eliminating these in a separate optimisation phase makes the compiler more modular and permits unit testing of the code generator and optimiser separately. Figure 6.8 shows an SPL code fragment before and after conditional elimination has been applied.

6.3 Outputs

This section describes the transformation of the CFA graph into various languages: (*i*) OCaml to be embedded as a dynamic safety monitor (§6.3.1); (*ii*) PROMELA to statically verify safety properties using a model checker such as SPIN (§6.3.2); and (*iii*) HTML/Javascript to permit high-level debugging of SPL models embedded in an executing application (§6.3.3).

Although we specifically describe an OCaml interface here, the compiler can also be easily extended to other type-safe languages (e.g. Java or C#), allowing application authors to write programs in their language of choice and still use the SPL tool-chain. In the case where languages do not make strong enough memory-safety guarantees to protect the safety monitor from the main program (e.g. C/C++), the compiler must output code which runs in a separate process [230] and stub code which allows the server to communicate with the monitor via IPC. This approach will be slower for larger SPL policies due to the overhead in performing inter-process communication rather than simply calling a function as the OCaml interface does.

```
exception Bad_statecall                                                    OCAML
type t
type s = [ ‘One | ‘Two | ‘Three | ‘Four | ‘Five | ‘Six | ‘Seven ]
val init : unit → t
val tick : t → s → t
```

```
exception Bad_statecall                                                    OCAML
module Alpha : sig
    type state = { state : int; }
    type states = (int, state list) Hashtbl.t
    type s = [ ‘One | ‘Two | ‘Three ]
    val tick : states → s → states
    val init : unit → states
end
module Beta : sig
    type state = { state : int; }
    type states = (int, state list) Hashtbl.t
    type s = [ ‘Five | ‘Four | ‘Seven | ‘Six ]
    val tick : states → s → states
    val init : unit → states
end
module Gamma : sig
    type state = { state : int; }
    type states = (int, state list) Hashtbl.t
    type s = [ ‘Two | ‘Three | ‘Four | ‘Five ]
    val tick : states → s → states
    val init : unit → states
end
type t = { alpha : Alpha.states; gamma : Gamma.states;
    beta : Beta.states }
type s = [ ‘One | ‘Two | ‘Three | ‘Four | ‘Five | ‘Six | ‘Seven ]
val init : unit → t
val tick : t → s → t
```

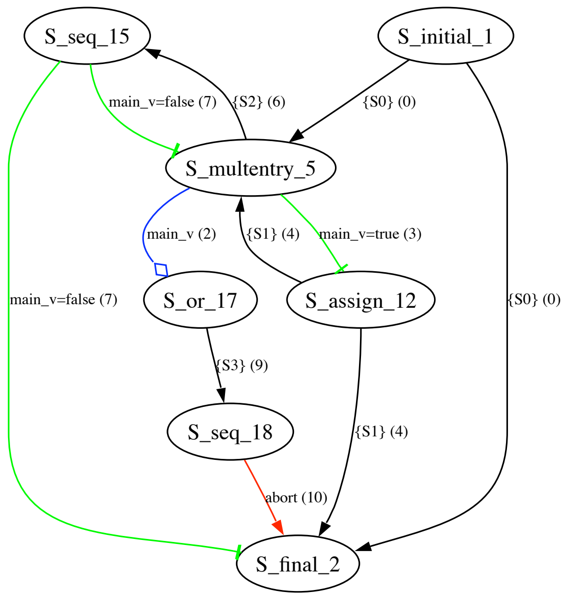
Figure 6.9: External *(top)* and internal *(bottom)* OCaml interface for the example SPL specification from Figure 6.6

6.3.1 OCaml

The OCaml output from the SPL compiler is designed to: (*i*) dynamically enforce the SPL model and raise an exception if it is violated; and (*ii*) provide real-time monitoring, debugging and logging of the SPL models. The SPL compiler generates OCaml code with a very simple external interface, shown in Figure 6.9 *(top)*. The polymorphic variant `s` represents all the statecalls, and `t` is an abstract type which represent the state of the automaton. Two methods manipulate this state—`init` returns a fresh instance of the automaton and `tick` accepts a statecall and automaton state, and returns the new state or raises the exception `Bad_statecall` in the event of a violation. The interface is purely functional and the returned state is independent of the input state (thus allowing an automaton to be “rolled back” by keeping a list of previous automaton values).

Internally, the implementation is structured into a sequence of modules for each automaton

```
automaton main(bool v)
{
  S0;
  multiple {
    either {
      v = true;
      S1;
    } or {
      S2;
      v = false;
    } or (v) {
      S3;
      abort;
    }
  }
}
```



```
|6, ‘S1 (⋆ S_assign_12 ⋆) →                                            OCAML
    register_state 0 x h; (⋆ S_final_2 ⋆)
    register_state 4 x h; (⋆ S_multentry_5 ⋆)
    if x.main_v then begin
        register_state 5 x h; (⋆ S_or_17 ⋆)
     end;
    begin (⋆ main_v ← true in ⋆)
    register_state 6 {x with main_v=true} h; (⋆ S_assign_12 ⋆)
    end;
|5, ‘S3 (⋆ S_or_17 ⋆) → raise Bad_statecall
|4, ‘S2 (⋆ S_multentry_5 ⋆) →
    begin (⋆ main_v ← false in ⋆)
        register_state 4 {x with main_v=false} h; (⋆ S_multentry_5 ⋆)
        (⋆ skipped false conditional ⋆)
        begin (⋆ main_v ← true in ⋆)
            register_state 6 {main_v=true} h; (⋆ S_assign_12 ⋆)
        end;
        register_state 0 {x with main_v=false} h; (⋆ S_final_2 ⋆)
    end;
|2, ‘S0 (⋆ S_initial_1 ⋆) →
     register_state 0 x h; (⋆ S_final_2 ⋆)
     register_state 4 x h; (⋆ S_multentry_5 ⋆)
     if x.main_v then begin
        register_state 5 x h; (⋆ S_or_17 ⋆)
     end;
     begin (⋆ main_v ← true in ⋆)
        register_state 6 {x with main_v=true} h; (⋆ S_assign_12 ⋆)
     end
```

Figure 6.10: An example SPL specification *(top left)*, its CFA graph *(top right)* and a fragment of the OCaml executable automaton *(bottom)*

defined in the SPL specification. A top-level `tick` function dispatches incoming statecalls to the correct modules according to the rules specified earlier (§6.2.2). The SPL compiler assigns every node in the CFA graph a unique integer label[1], and each automaton module defines a record `state` to represent state variables (it has type `unit` if no state variables are present). The full automaton *state descriptor* with type `states` is a hash-table mapping state labels to a list of `state` records. Figure 6.9 (*bottom*) shows the internal signature for the example SPL automata from Figure 6.8.

The internal implementation takes several steps to make transitions as fast as possible. Since the only edges in the CFA which can "block" during execution are the statecall edges, all other edges are statically unrolled during compile-time code generation. The `tick` function for each automaton allocates an empty state descriptor and populates it by applying the input statecall over all states in the old state descriptor and registering any resultant states in the new state descriptor. If the result is an empty state descriptor after all the input states have been iterated over, a `Bad_statecall` exception is raised.

When unrolling non-statecall edges during code generation, assignment operations are statically tracked by the SPL compiler in a symbol table. This permits the compiler to apply constant folding when the resultant expressions are used as part of conditional nodes (or when creating new state descriptors). Multiple conditional checks involving the same variable are grouped into a single pattern match (this is useful in SPL specs with **during/handle** clauses). These optimisations are necessary even when using an optimising OCaml compiler, since they represent constraints present in the SPL specification which are difficult to spot in the more low-level OCaml code output.

Figure 6.10 shows an SPL specification *(top left)*, the associated CFA graph *(top right)*, and the complete pattern match for all valid statecall transitions *(bottom)*. Each pattern match has a comment beside it with the human-readable state name (which matches up with the node labels in the CFA graph). Observe that only four nodes have a pattern match, as the other nodes do not have any statecall edges and are statically unrolled at compile-time. Assignment optimisation can be seen in the ($\texttt{S\_multentry\_5} \xrightarrow{S2} \texttt{S\_seq\_15} \xrightarrow{v=false} \texttt{S\_multentry\_5} \xrightarrow{v=true} \texttt{S\_assign\_12}$) transition sequence which results in only a single assignment to `v`. Similarly, the conditional check for `v` is also statically skipped in the ($\texttt{S\_multentry\_5} \xrightarrow{S2} \texttt{S\_seq\_15} \xrightarrow{v=false} \texttt{S\_multentry\_5} \xrightarrow{if(v)} \texttt{S\_or\_17}$) transition sequence since it is known to be `false`.

```
automaton main(int v) {                                          SPL
    S0;
    multiple {
        v = v+1;
        optional { S2; }
    };
    S1;
}
```

This optimisation also has the useful side-effect of detecting poorly written SPL policies which would result in a large state explosion at run-time, such as the one above. If an excessive depth of recursion through the CFA graph is detected, the SPL compiler terminates with an error indicating the location of the state explosion. In practise, we used integer variables extremely

[1]Variant types were not directly used here since OCaml imposes a limitation of 255 labels per type; polymorphic variants remove this limitation, but are not needed since the labels are only used internally.

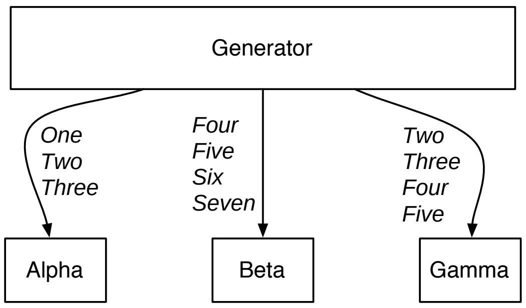


Figure 6.11: The structure of the output PROMELA from our example in Figure 6.6. Each box represents a separate PROMELA process

rarely in protocol state machines; the main source is internal variables created from statements such as **multiple** or **during/handle** (which are safely used).

6.3.2 PROMELA

The CFA graph structure is already a non-deterministic finite-state automaton, and thus maps very easily into PROMELA code. Firstly, two global boolean variables `err` and `gen` are defined to represent an error occurring or the generator process (described later) shutting down. Each statecall is mapped into an **mtype** (or an integer if there are more than 255 statecalls, due to implementation limits in SPIN), and every node is assigned a unique integer label as with the OCaml output. Each automaton in the SPL specification is defined as a separate PROMELA process and its state variables are declared globally (with unique names obtained by prepending the automaton name). Every automaton also has a rendezvous channel (a channel with a message buffer of size 0) through which it receives statecall messages.

The automaton processes execute continuously in a `do` :: `od` loop, and outside this loop two labels `Bad_statecall` and `End_automaton` are defined to represent an error or normal termination respectively. The only way a process can exit the loop is by jumping to one of these labels (this is guaranteed by an assertion between the loop and the labels). A `generator` process continuously transmits statecalls non-deterministically to each of the other processes using their respective rendezvous channels. Each statecall is dispatched only to the automata which expect it, according to the semantics described earlier (§6.2.2). Although this conversion seems straightforward, it is carefully designed to overcome several practical problems with expressing models in PROMELA or checking them with SPIN.

- The generated model will *always* pass the SPIN checks for valid end-states, progress and assertion checks by default for any valid SPL models. A manually constructed PROMELA model can easily dead-lock due to programmer error when handling rendezvous channels, since every attempt at receiving a statecall on a rendezvous channel must also guard against automaton error by checking that the generator is still transmitting messages via the `gen` variable. Since this check must be performed at every potential blocking point in an automaton and PROMELA lacks the high-level programming constructs to abstract it in the language, it is best introduced mechanically by the SPL compiler output.

- By default, PROMELA assumes that statements in different processes can be executed with arbitrary interleaving. This can lead to a very large state explosion if many processes are involved. However, the SPL output constrains this interleaving via the **atomic** and **d_step** keywords to the same semantics of the OCaml safety monitor. This reduction in state space has reduced the verification times of some SPL models from the order of weeks to minutes, but without introducing the risk of deadlocks (due to the point above).

- The contents of messages transmitted across rendezvous channels cannot be specified in a **trace** clause in SPIN. To overcome this, every message transmitted by the generator is also assigned to a global variable which can be tested in LTL formulae.
- Key automaton local variables such as whether it has terminated, the current state, and the value of state variables are exported as global variables, but constrained using the **local** keyword. This enables compatibility with partial order reduction in SPIN [139] (which is normally disabled if local variables are accessed inside LTL formulae and never-claims). The variables names are also guaranteed to be unique, which means that the potential danger of incorrect verification by inadvertently using a variable marked local in more than one process is guaranteed not to happen.
- The limitation of 255 **mtype** values is overcome by converting them to integers if more statecalls are present and instrumenting the PROMELA output to print human-readable strings in the Message Sequence Chart output. Normally the **mtype** variable is the only way to obtain human-readable strings inside simulation runs, severely limiting the utility of PROMELA code which does not use it.

Running SPIN over the PROMELA output allows the model checker to exhaustively calculate the maximum ranges of state variables in the SPL specification. SPIN uses this information to optimise its verification algorithms, but it also gives developers more assurance that the OCaml safety monitors will have a reasonable size at run-time if integer variables are being used. The PROMELA models can also be further constrained via safety assertions such as never-claims or using the convenient SPIN LTL formulae converter provided by SPIN. We give concrete examples of some LTL formulae applied to the SPL models defined for our implementation of the SSH protocol later (§7.1.5).

6.3.3 HTML and Javascript

AJAX (Asynchronous Javascript and XML) is a group of technologies for creating interactive applications running directly in a Web browser [295], consisting of: (*i*) XHTML[2] and CSS[3] to markup and style content; (*ii*) the DOM[4] tree which can be modified by the client-side scripting language JavaScript to dynamically alter a web page; and (*iii*) the `XMLHttpRequest` object to exchange data asynchronously with a web server. AJAX is an effective method of deploying cross-platform applications with no dependencies beyond a standard web browser such as Mozilla [212] being present on the host.

The SPL compiler can optionally embed an AJAX-based debugger in an executing SPL safety monitor, permitting a standard web-browser to connect to the monitor and perform high-level debugging with respect to the SPL specifications. For security reasons, a web-browser can only make HTTP requests using the `XMLHttpRequest` object to the same server from which it obtained the web-page and the responses to these requests must be short-lived or the browser runs out of memory rather quickly.

[2]eXtensible HyperText Markup Language, see `http://www.w3.org/TR/xhtml1/`
[3]Cascading Style Sheets, see `http://www.w3.org/Style/CSS/`
[4]Document Object Model, see `http://www.w3.org/DOM/`

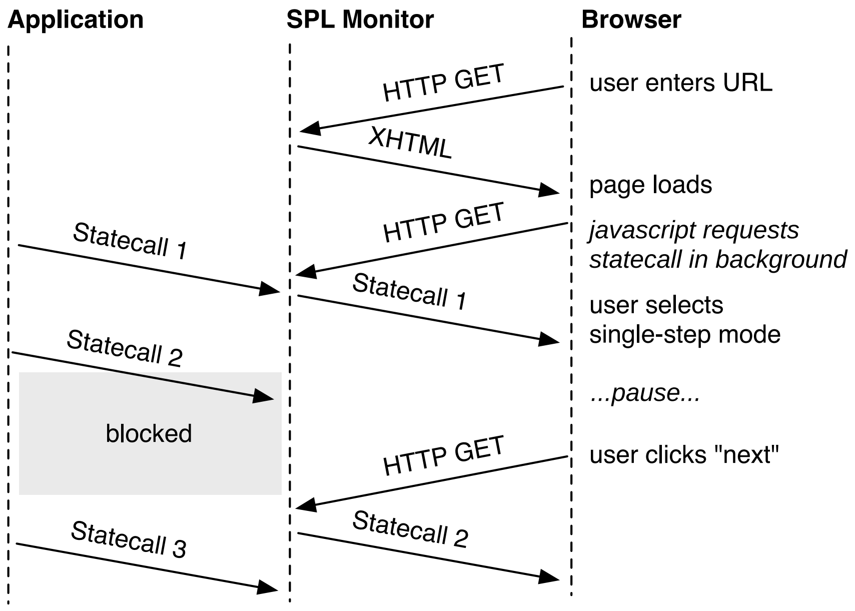


Figure 6.12: Message sequence chart showing the asynchronous interactions between a web browser, an SPL safety monitor its associated application.

Figure 6.12 shows the interactions between a web-browser, an SPL monitor, and the application the monitor is embedded in. Every SPL safety monitors listens on a unique TCP port[5] and operates as normal until an HTTP request is received on this port. When the monitor receives an HTTP request, it switches into debug mode and replies with a web-page embedded in it as a string generated by the SPL compiler. The web-page contains a pretty-printed version of the SPL specification rendered using XHTML and CSS. As soon as the page loads in the browser, the Javascript in it begins to the poll the safety monitor for updates. The safety monitor accepts the connection and holds it open until it receives a statecall from the application, after which it replies with the statecall and its internal state, which is parsed by Javascript in the browser and updated on the client browser in real-time.

This scheme permits the easy implementation of single-step monitoring of the target application with respect to the statecalls it is outputting by delaying the web-browser HTTP GET requests. This in turn causes the SPL monitor to block in the *accept(2)* system call when it receives a statecall from the main application, until the user clicks on a "next" button in the web-browser and triggers the transmission of an HTTP GET. After this, the application can execute as normal until it needs to issue another statecall.

The AJAX debugger proved to be extremely useful when developing the SSH implementation, which embeds multiple SPL safety monitors and invokes statecalls on every packet being sent or received. We show screen-shots of the debugger in action when we describe the SSH server (§7.1).

6.4 Summary

In this chapter we have presented the Statecall Policy Language (SPL), a language which simplifies the task of specifying non-deterministic state machines (§6.1). The state machines are

[5]The actual IP address/port a safety monitor is listening on can be discovered in a variety of ways; e.g. multicast DNS, the UNIX `portmap` utility, or simply by logging a randomly chosen port to `syslog`.

compiled into an intermediate representation which separates the front-end SPL language from the compiler outputs (§6.2). SPL trades extreme formal safety in favour of flexible, light-weight dynamic enforcement of the models via safety monitors in languages such as OCaml, but also makes model-checking easier by outputting well-formed PROMELA code which can be verified using the SPIN model checker (§6.3). Applications using SPL models also benefit from high-level run-time monitoring support embedded within safety monitors and exposed to any standard web browser via AJAX techniques.

CHAPTER 7

Case Studies

Applicants must also have extensive knowledge of UNIX, although they should have sufficiently good programming taste to not consider this an achievement.
MIT JOB ADVERTISEMENT

We have defined the MELANGE architecture, which consists of two domain specific languages (MPL and SPL) which output OCaml code to handle packet parsing and embedding state-machines. In this chapter we describe two implementations of Internet protocol servers using this architecture and evaluate the performance of each against the industry-standard reference implementations. We chose two protocols for implementation: (*i*) Secure Shell version 2 (SSH), used for securely connecting to remote machines across an untrusted network (§7.1); and (*ii*) the Domain Name Service (DNS), which provides an Internet directory service, e.g. used to map human-readable names to IP addresses (§7.2).

The choice of protocols is important due to their differing characteristics. SSH is a highly flexible protocol which deals with authentication, access control, encryption, and channel multiplexing and flow control for both interactive and bulk transfer sessions. DNS is primarily a stateless control protocol and deals in very small packets but, unusually for control protocols, must also be high-performance due to the large number of DNS lookups performed during common operations such as e-mail delivery or web surfing.

Our performance measurement methodologies emphasise the throughput and latency characteristics of the complete application. During the initial stages of testing, we instrumented the OCaml garbage collector to output detailed statistics about its behaviour. However, examination of the C applications we benchmark against revealed that they quite often implement their own memory management routines (e.g. pool or slab allocators [42]). Measurement using standard system calls such as *getrusage(2)* is complicated by the fact that even the system *malloc(3)* and *free(3)* routines implement their own internal caching of memory depending on their exact implementation[1] and so obtaining a precise number for the amount of live memory used by an application is difficult without altering its performance (e.g. forcing a garbage collection

[1]On OpenBSD, `libc` memory allocation is handled by the *mmap(2)* system call, and a common alternative on other UNIX-like operating systems is to use *sbrk(2)*.

Table 7.1: Some CERT Vulnerabilities for OpenSSH from 2000 to 2003, with the crosses marking parsing related security issues *(source: kb.cert.org)*

	VU#	Date	Description
	40327	6/2000	OpenSSH UseLogin allows remote execution as root
$\otimes$	945216	2/2001	SSH CRC32 attack detection contains remote integer overflow
	655259	6/2001	OpenSSH allows arbitrary file deletion via symlink redirection
	797027	6/2001	OpenSSH allows PAM restrictions to be bypassed
	905795	9/2001	OpenSSH fails to properly apply source IP based access control
	157447	12/2001	OpenSSH UseLogin directive permits privilege escalation
$\otimes$	408419	3/2002	OpenSSH contains a one-off array overflow in channel handling
$\otimes$	369347	6/2002	OpenSSH vulnerabilities in challenge response handling
$\otimes$	389665	16/2002	SSH transport layer vulnerabilities in key exchange and init
	978316	6/2003	Vulnerability in OpenSSH daemon (sshd)
$\otimes$	209807	9/2003	Portable OpenSSH server PAM conversion stack corruption
$\otimes$	333628	9/2003	OpenSSH contains buffer management errors
	602204	9/2003	OpenSSH PAM challenge authentication failure

or `libc` to flush its caches). The OCaml garbage collector is also characterised by providing faster memory allocation than the standard C functions since it simply increments a pointer on the heap and relies on the garbage collection phase to free it.

We do not actually care about the precise internal workings of memory allocation, but rather the overall performance and latency of the system (e.g. whether the presence of a garbage collection phase introduces long pauses which disrupts network traffic). In all the performance results presented in this section, we have used default system resource limits in OpenBSD and ensured that tests run long enough to permit thousands of full garbage collection cycles to occur in the OCaml applications so that the latency data will point to any "hotspots".

7.1 Secure Shell (SSH)

SSH is a widely used protocol for providing secure login over a potentially hostile network. It uses strong cryptography to provide authentication, confidentiality and multiplexed data channels for interactive and bulk data transfer. Provos and Honeymoon developed `scanssh` to rapidly scan large portions of the Internet for SSH servers and determine the versions of deployed SSH servers [231]. As we showed earlier in Figure 1.1, OpenSSH [262] is the dominant server used by over 90% of Internet hosts which are running an SSH server. OpenSSH is written in C and developed in two versions: a "core version" developed on OpenBSD, and a "portable" release consisting of a patchset for numerous other operating systems.

SSH is a complex protocol, with hundreds of different packet types, combined with cryptography and channel multiplexing. Interestingly, despite the complex cryptography requirements OpenSSH has also suffered from a significant number of relatively straightforward parsing-related security issues in recent years. Table 7.1 shows the important security issues in OpenSSH since 2000, and out of the 14 listed, 6 (43%) could be attributed to bugs in the parsing of network traffic. Some of the problems such as the CRC32 integer overflow[2] were easily and remotely

[2]CERT VU#945216 available from `http://www.kb.cert.org/vuls/id/945216`

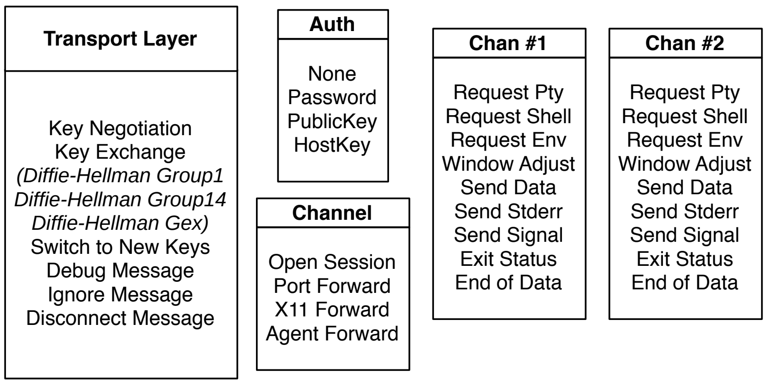


Figure 7.1: Diagram of the various aspects of the SSHv2 protocol with sample messages inside each box

exploitable, leading to a wave of attacks being reported to CERT [62].

The potential for more SSH worms is worrying since the service is so widespread across the Internet and is difficult to hide from attackers since it is often used as a management protocol to administer firewalls. OpenSSH is becoming more secure through the use of techniques such as privilege separation [230] and host hashing [240] to reduce the impact of vulnerabilities. However the server is still written in C, with the associated risks of future security issues through code errors.

The SSH protocol has recently been standardised in a series of RFCs by the IETF, starting with its architecture [293]. Figure 7.1 illustrates its various layers: (*i*) a transport layer [294] which deals with establishing and maintaining encryption and compression via key exchange and regular re-keying; (*ii*) an authentication layer [291] which is used immediately after the transport layer is encrypted to establish credentials; and (*iii*) a connection protocol [292] which deals with multiplexing data channels for interactive and bulk data transfer sessions. The connection protocol has both global messages (e.g. to create TCP/IP port forwardings) and channel-specific messages which must be dispatched appropriately. Channels can be created and destroyed dynamically over a single connection, data transfer can continue while re-keying over the transport layer is in progress, and the protocol even permits different cryptographic mechanisms to be used for transmission and receipt of data. Extensions such as the use of DNS to store host keys and new authentication methods have also been published as RFCs [241, 87, 29].

We implemented a fully-featured SSH library—dubbed MLSSH—which supports both client and server operation. The library supports the essential features of an SSH session including key exchange, negotiation and re-keying, various authentication modes (e.g. password, public-key and interactive), and dynamic channel multiplexing for interactive and bulk data transfer. The implementation is quite succinct, consisting of around 4500 lines of OCaml code, 350 lines of MPL and 400 lines of SPL specifications. When the MPL and SPL compilers have auto-generated their respective OCaml modules, the total size rises considerably to 18000 lines. The only external component used was Xavier Leroy's Cryptokit library[3] to handle the cryptography. The code is pure OCaml with no C bindings except a small library for pseudo-terminal allocation via *openpty(3)* which is not supported by the OCaml standard library.

We now present the performance evaluation of MLSSH versus OpenSSH (§7.1.1), describe how SSH packets are parsed using MPL (§7.1.2), the use of SPL to model the SSH protocol

[3]Available from `http://pauillac.inria.fr/~xleroy/software.html`

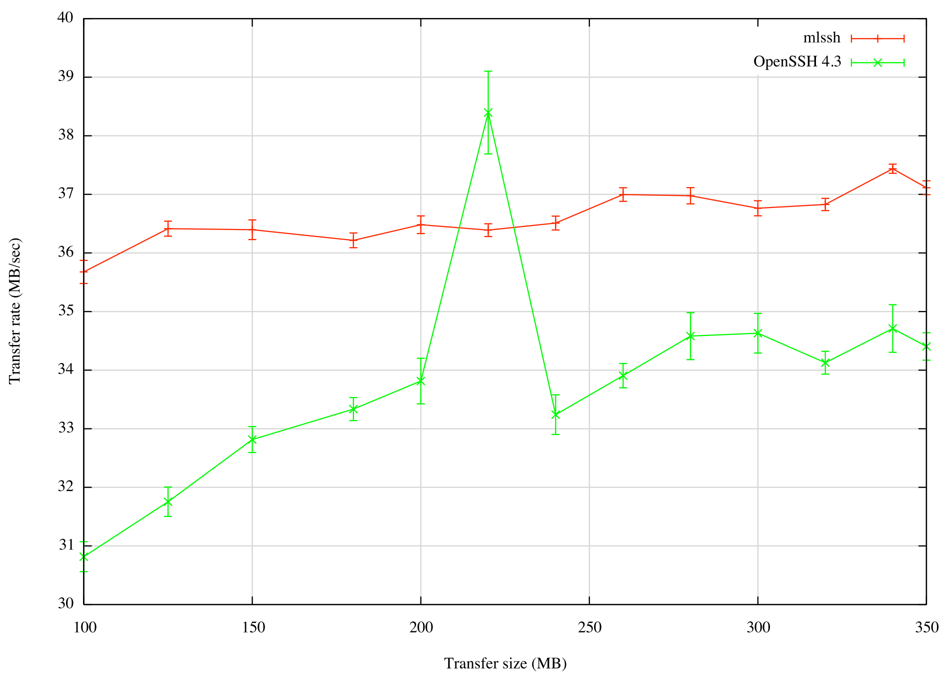


Figure 7.2: Transfer size vs transfer rate for MLSSH and OpenSSH servers using null encryption and MAC cipher (rekeying via Diffie-Hellman every million packets is enabled). The anomolous OpenSSH value is attributed to caching effects and is reproducable.

state machine (§7.1.3), the AJAX debugger (§7.1.4) and finally some LTL formulae we applied against the PROMELA generated from the SPL compiler (§7.1.5).

7.1.1 Performance

In this section we measure the performance and latency characteristics of MLSSH against OpenSSH 4.3, as included in the standard distribution of OpenBSD 3.8. We first perform our measurements with cryptography turned off (both data ciphers and the MAC verification codes) in order to isolate the core SSH engine.

Throughput

First, we measure the sustained throughput of an SSH session by running repeated transfers of large files through a single connection. A connection using the standard OpenSSH client is established to either an MLSSH or OpenSSH server, with all logging disabled. A file of a variable size (ranging from 100MB to 300MB) is created on a memory file system and transferred via the established SSH connection to another memory file system on the same host. This process is repeated 100 times across the same connection by dynamically creating a new channel, ensuring that at least 10 gigabytes of data are sent through every connection to highlight any bottlenecks due to memory or resource leaks. Since the SSH protocol also mandates re-keying every million packets or so, our benchmarks reflect that cost as part of the overall result (despite disabling encryption, session keys were still derived).

Figure 7.2 shows a plot of transfer rate (in megabytes/second) vs the transfer size of the individual chunks of data. Each data point and error bar reflects the average time and 95% confidence interval over the 100 repeated invocations across one SSH connection. OpenSSH is slightly slower than MLSSH and interestingly also has a larger variation of transfer times over

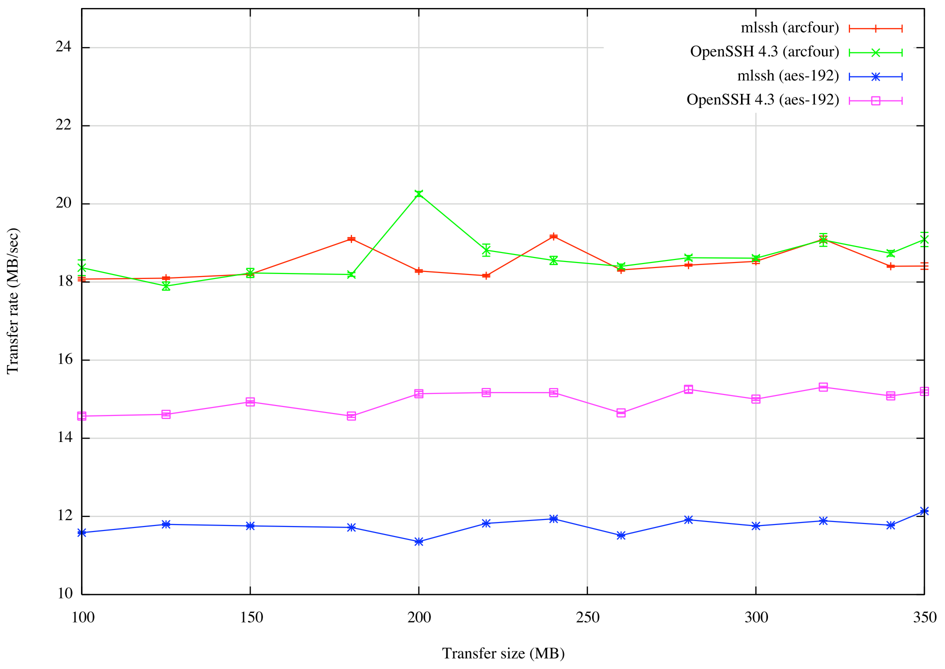


Figure 7.3: Transfer size vs transfer rate for MLSSH and OpenSSH servers, using either the stream Arcfour or block AES ciphers

the more consistent MLSSH. We attribute this to the regular garbage collection and memory compaction cycle used by MLSSH when compared to the more ad-hoc manual allocation and deallocation used by OpenSSH. This result is a very encouraging start since our MPL standard library is written in pure OCaml, and there remains a large scope for further low-level optimisations in the field-parsing code for even more speed in the future.

Figure 7.3 shows the same experimental setup applied to MLSSH and OpenSSH servers with encryption enabled and using HMAC-SHA1-160 as the message digest algorithm for both graphs. The transfer rates are of course slower than the previous plain-text cipher, and MLSSH and OpenSSH demonstrate similar transfer speeds when using a stream-based cipher. However, MLSSH is only 75% of the speed of OpenSSH when using the more computationally intensive AES-192 cipher. Examination of OpenSSL [264] (the cryptography library used by OpenSSH) and Cryptokit (the OCaml library we are using) reveals that OpenSSL uses optimised, hand-written assembly language code for their implementation of AES, where Cryptokit is a combination of portable C and OCaml. Both Cryptokit and OpenSSL include informal regression tests in their source distributions, and running them revealed that Cryptokit was roughly 75% slower than OpenSSL on our test machines, indicating that the problem lies with the external library used and not our SSH implementation. Since AES is becoming a commonly used cipher, we intend to investigate speeding up Cryptokit as part of our future work.

Connection Latency

We also measured the latency of established SSH connections to MLSSH and OpenSSH, with the measurement architecture shown in Figure 7.4. First, a master connection is started to either server and a file transfer is repeatedly looped through it as in our earlier throughput experiments. Once this continuous transfer has settled down (we left it for 5 minutes), another connection is established from an OpenSSH client to the session. A character generator process is begun on

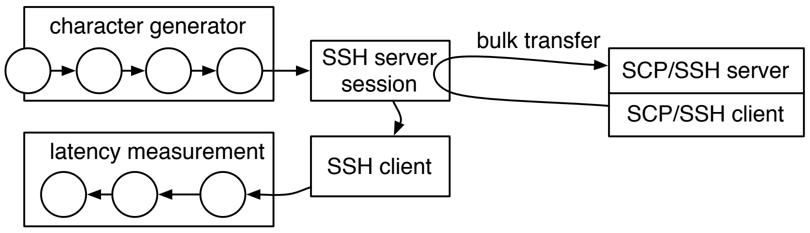


Figure 7.4: Architecture of our latency tests which send a regular stream of characters through an established and heavily loaded SSH session to another client to measure latencies between the received characters.

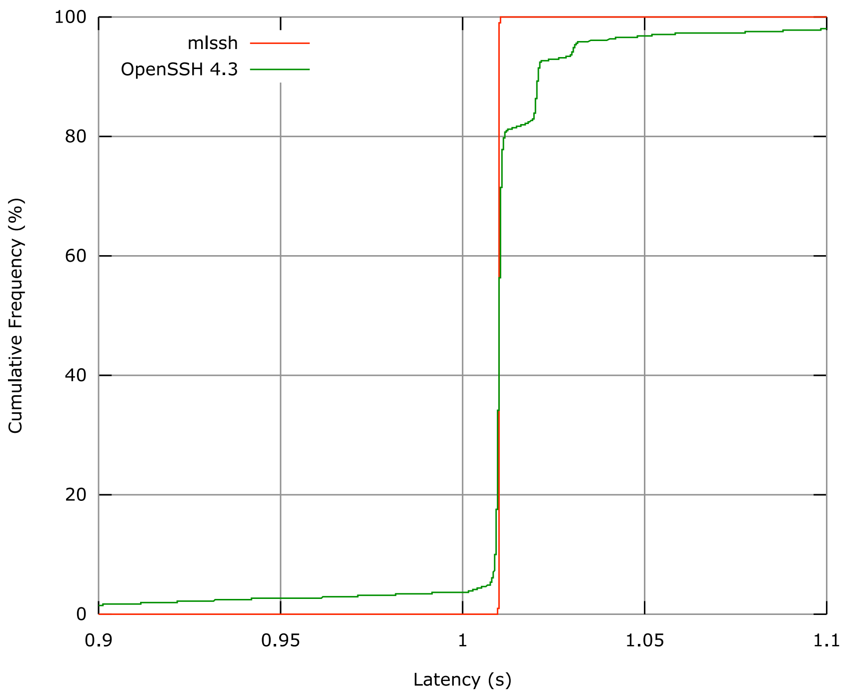


Figure 7.5: Cumulative distribution function for SSH connection latency against a heavily loaded SSH server session

the server side which transmits a single character every second[4] back to the client. The client measures the time between characters being received back—in an "ideal" server this would be a regular one second but if delays are introduced in the pipe-line (e.g. network problems) then the inter-packet latency will vary. Since we are using the `localhost` interface which does not introduce any network latencies, the main source of delay will be the server process itself (along with scheduling inconsistencies which are common to all the tests).

Figure 7.5 shows the cumulative distribution function of receiving 10000 characters over an SSH connection loaded down with bulk transfers of 200MB files in the background. The latencies recorded through MLSSH are extremely consistent and clustered around the one second mark with very little variance. In contrast OpenSSH exhibits jitter within a range of $\pm$100ms indicating that delays are being introduced within the server which cause it to disrupt the inter-packet arrival times. This is surprising for two reasons: (*i*) OpenSSH is performing manual

[4]The TCP `NO_DELAY` socket option is active on both servers being measured to ensure that TCP buffers do not introduce buffering delays.

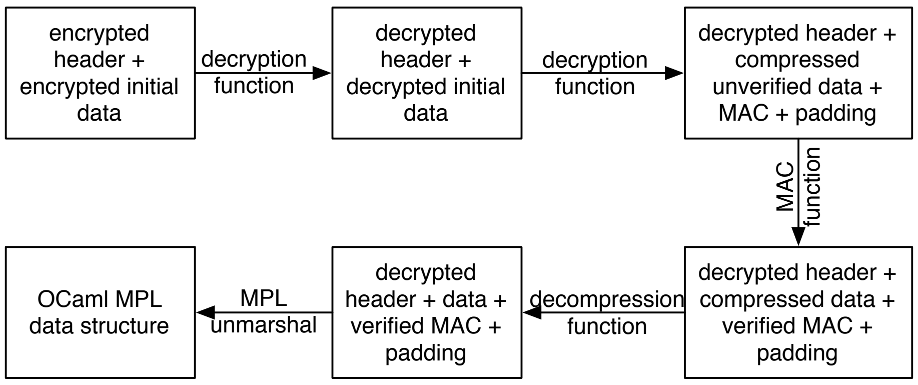

encrypted header + encrypted initial data
decryption function
decrypted header + decrypted initial data
decryption function
decrypted header + compressed unverified data + MAC + padding
MAC function
decrypted header + compressed data + verified MAC + padding
decompression function
decrypted header + data + verified MAC + padding
MPL unmarshal
OCaml MPL data structure

Figure 7.6: SSH unmarshal path for a single packet which shows the sequence of decryption, decompression, MAC verification and finally classification of the contents with MPL

memory management which "should" be faster than automatic garbage collection; and (*ii*) MLSSH ought to have a more bi-modal distribution to reflect the cost of the occasional garbage collection introducing a delay.

The first point is easily debunked by examining the internals of the OpenBSD *malloc(3)* and *free(3)* functions, which are almost as complex as the OCaml garbage collector routines. Allocation in OCaml is a simpler process than *malloc(3)* since only a single pointer needs tp be incremented as opposed to the more complex free-list management required by the `libc` functions. Recall that OCaml has two distinct heaps—one for longer-lived data which requires compaction (an expensive operation) and a smaller one for the more common short-lived data [17]. Our approach of separating the data and control paths of servers via MPL (§7.1.2) is clearly validated here, since we generate much less "large" garbage (i.e. data packet payloads) that would normally require collection from the major heap. Instead, by re-using packet environments the only real garbage is generated in the smaller heap which introduces negligible pauses in the application.

This is an interesting area we intend to re-visit in future work by measuring performance across a variety of *malloc(3)* implementations. For the purposes of our thesis however, we have shown that a well-constructed network application in a garbage-collected language can perform as well as an application manually managing its memory.

7.1.2 SSH Packet Format

The SSH protocol constructs data packets in two stages: (*i*) a secure encapsulation layer for all packets (which includes encryption and a message hash to ensure integrity) and random padding; and (*ii*) classification rules for the decrypted packet payloads. We implement the encapsulation layer directly in OCaml and and parse the decrypted payloads with MPL specifications. Figure 7.6 shows the sequence of functions called to convert the encapsulated encrypted data into plain-text data. Firstly a small chunk of data is read and decrypted from which the length of the rest of the packet is obtained. The packet payload is then read and decrypted, followed by an unencrypted MAC string and some random padding. The MAC is verified by recalculating it over the decrypted packet payload, and then the payload is decompressed. Finally, this plain-text payload is passed onto the MPL classification functions for conversion into an OCaml data structure.

The early implementations of MLSSH performed a data copy of the payload for every stage of this computation by allocating a new buffer. The latest (and much faster!) version requires the payload to be copied only a single time to decrypt it into a new buffer (if the Cryptokit library were extended to support in-place decryption, this copy would no longer be needed). The reduced data copies are possible due to the MPL standard library support for "fill functions", which are closures invoked when the MPL unmarshal code requires more data that has not yet been placed into the packet environment. Fill functions are normally simple calls to *read(2)* or *recvfrom(2)*, but in the case of SSH perform the stages shown in Figure 7.6 before placing the decrypted data directly into the packet environment at the correct location.

The MPL classification for SSH required the definition of several custom types: (*i*) `string32` for variable-length strings which have a 32-bit length identifier; (*ii*) `mpint` for multiple precision integers required for establishing cryptographic keys; and (*iii*) `boolean` for the wire representation of SSH binary flags. Some of the MPL specifications for SSH packets are shown in Appendix C.5; the full MPL specification defines over 250 different packet types defined across the many SSH RFCs. The support for MPL string and boolean types proved essential since SSH frequently uses both of these to classify packets (in addition to the more conventional integer labels).

When we first began to implement MLSSH, the RFCs were still in draft stage and our more formal encoding of the packet parsing portion of the protocol led us to find some inconsistencies in the drafts which we submitted as corrections to the final RFC.

- The identifier for a "password change request" packet clashes with that of a "public key confirmation" packet. An MPL state variable deals with this ambiguity by determining if a previous request to change the password is outstanding or not.

- The replies to global channel requests (e.g. to open a new session) do not specify the request to which they are replying. We notified the RFC working group, who modified the specification to mandate that replies (either successes or failure) had to be sent in the order they were received.

- A global channel response can optionally include a "port" field, but only if a previous request had requested a port forwarding. A state variable was added to the MPL specification to cope with this ambiguity.

Our MPL specification was useful to pin down poorly specified portions of the SSH RFCs with respect to packet parsing and shows that with some small modifications to eliminate the above ambiguities, SSH implementations could be made simpler by reducing the amount of external application state required to parse network packets.

7.1.3 SSH State Machines

We chose to use statecalls to specify and enforce: (*i*) the sequences of network packets being transmitted or received, identified by `Transmit` or `Receive` in front of the statecalls name; and (*ii*) the results of significant computation within the server, identified by `Expect` prefixed to the statecall name. The network packet statecalls are automatically provided by the MPL compiler and the `Expect` statecalls are manually inserted into key points in the server code. We have two distinct SPL specifications—a global automaton to deal with the transport layer, authentication layer and global channel messages, and a channel-local automaton which is spawned for every dynamic channel that is created. Statecalls are "routed" to the appropriate automaton via a

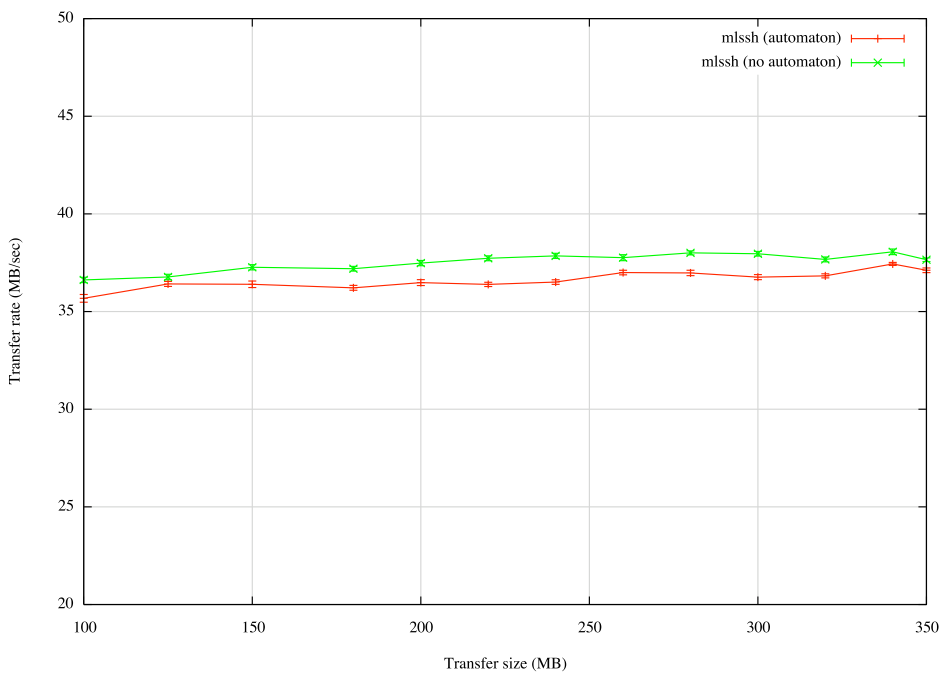


Figure 7.7: Transfer rate of MLSSH with the dynamic enforcement of the SSH SPL automata turned on and off, using null ciphers.

classification function which inspects the incoming or outgoing packet type and dispatches it to either the global automaton or the appropriate per-channel automaton.

The global SPL specification (Appendix D) contains multiple automata for the transport, authentication and global channel packets. These execute in parallel according to the semantics defined for multiple automata earlier (§6.2.2). Some statecalls are shared between automata, such as `Transmit_Transport_ServiceAccept_UserAuth` which is used in the SSH protocol to "unlock" the authentication service. It can be transmitted in the transport layer, but is also used as the first transition in the authentication automaton. This succinctly ensures that no authentication progress is allowed until the service is opened by the transport layer. A similar mechanism unlocks the global channel automaton by waiting for a `Transmit_Auth_Success` packet from the authentication layer.

The per-channel automaton ensures some operations can be done only once per channel, such as pseudo-terminal allocation and requesting a command execution (either as a shell or a specified binary). Once command execution has completed, data packets are allowed to be sent or received along with window adjust packets used for flow control. An "EOF" message which indicates that one side of a channel has closed is also enforced, making sure that data transmission packets are not sent after an EOF is transmitted.

The SPL specifications strike a balance between the complete formalisation of the packet state-machine and completely informal server code. The simplicity of the specifications, combined with the efficient OCaml code output from the SPL compiler results in a negligible performance loss from dynamically enforcing the SPL automata. Figure 7.7 shows the result of running the performance tests described earlier on a version of MLSSH with the SPL automata enabled and disabled. Note that all of the other performance and latency tests reported in this chapter are conducted with the dynamic automata enforcement turned on.

7.1.4 AJAX Debugger

Figure 7.8 shows a screen capture of the SPL AJAX debugger single-stepping through the global SPL automaton. The MLSSH server is blocked waiting for password authentication, having previously attempted to authenticate via null and public-key authentication. In our experience, the debugger was a valuable tool to debug complex protocol bugs in our implementation, as the single-stepping view via this debugger is significantly higher level then the alternative provided by either the native OCaml debugger or the GNU debugger `gdb`.

We also implemented the facility to turn the debugging mode on and off via a UNIX signal, enabling individual SSH sessions to be debugged without affecting the other connections (our MLSSH server executes a *fork(2)* on every unique session since it needs to switch user-id during the course of a session, so threading is not sufficient).

7.1.5 Model Checking

The SPL specifications described earlier are also output as PROMELA models by the SPL compiler. The specification for the transport, authentication and global channel handling is a complex state machine, and an exhaustive safety verification in SPIN without any additional LTL constraints (i.e. testing assertions and invalid end-states) requires around 400MB of RAM and one minute to verify on a dual-G5 1.25GHz PowerMac running MacOS X 10.4.5. SPIN reports the following statistics:

```
State-vector 48 byte, depth reached 78709, errors: 0
1.41264e+07 states, stored (1.99736e+07 visited)
2.59918e+07 states, matched
4.59654e+07 transitions (= visited+matched)
7.85945e+07 atomic steps
```

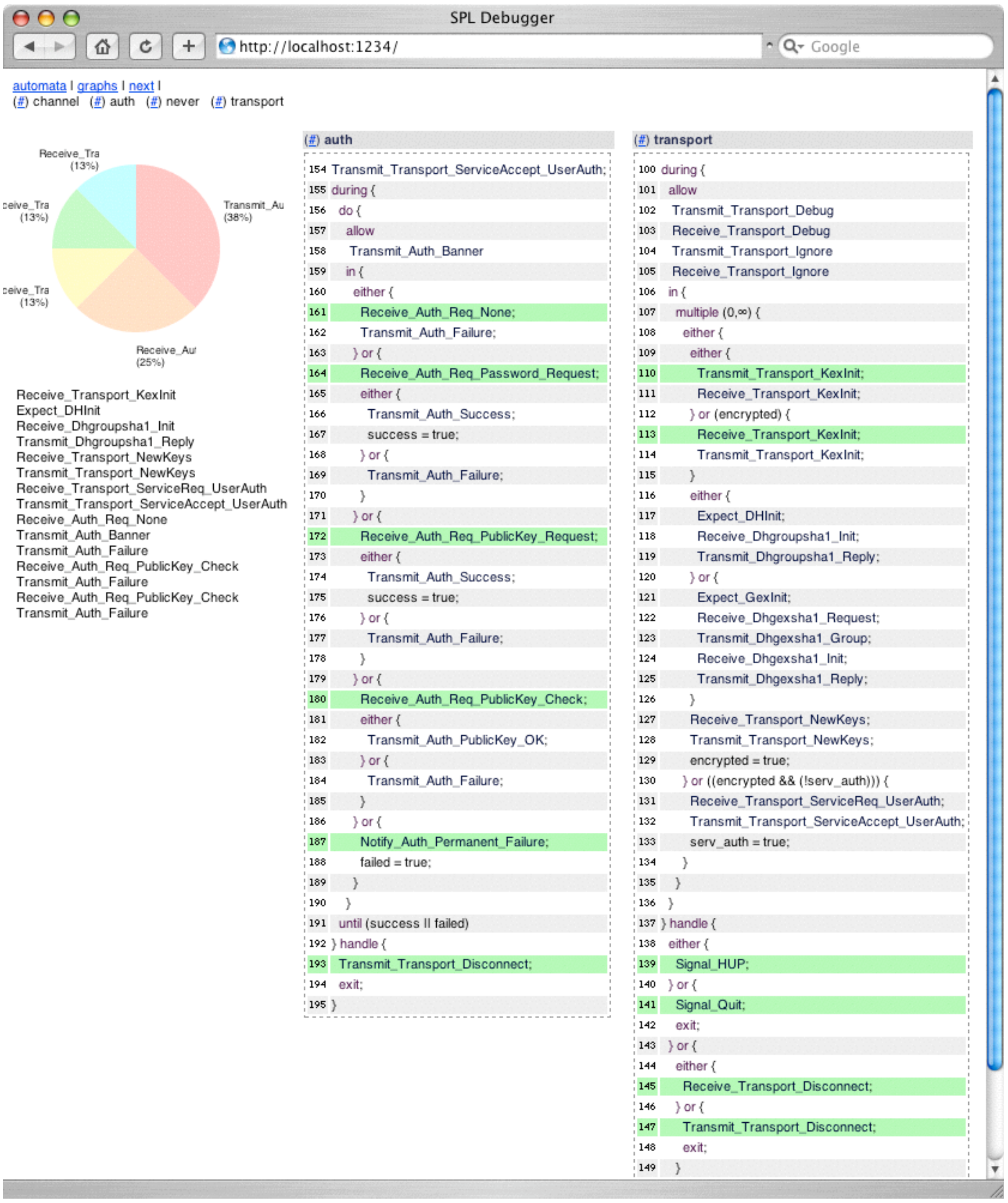
The large number of atomic steps show the complexity reduction which results from the SPL compiler inserting `atomic` statements in the generated PROMELA to simulate the execution semantics of the OCaml safety monitors. Before this optimisation, messages would unnecessarily be interleaved and verification took orders of magnitude longer.

We now list some of the LTL formulae applied to the PROMELA output of the SSH global automaton. Unlike some other tools which translate state machine languages into PROMELA (e.g. Scott's SWIL language for interface descriptions [246]), we never require the manual modification of the PROMELA code (which would be dangerous since the equivalence between the model and the dynamically enforced SPL automaton would not be guaranteed any more). Instead, globally allocated state variables[5] are exposed within the model which can be referenced with LTL formulae, as shown below:

- $\Box(\mathtt{a} \rightarrow \Box\mathtt{a})$ where ($\mathtt{a} \leftarrow$ `transport_encrypted`) which informally reads "once `transport_encrypte`
  is true, it remains true forever". This check ensures that the SPL specification never sets the encrypted state variable to `false` once a secure transport has been established.

- $\Box(\mathtt{a} \rightarrow \Box(\mathtt{a}\ \&\&\ \mathtt{b}))$ where ($\mathtt{a} \leftarrow$ `transport_serv_auth`) and ($\mathtt{b} \leftarrow$ `transport_encrypted`) which informally reads "in the `transport` automaton, once `serv_auth` is `true`, both `serv_auth` and `encrypted` remain `true` forever".

[5]Recall that SPIN does not support partial order evaluation over local variables, so the SPL compiler safely promotes automaton-local variables to a global scope.



automata | graphs | next |
(#) channel (#) auth (#) never (#) transport

Receive_Transport_KexInit
Expect_DHInit
Receive_Dhgroupsha1_Init
Transmit_Dhgroupsha1_Reply
Receive_Transport_NewKeys
Transmit_Transport_NewKeys
Receive_Transport_ServiceReq_UserAuth
Transmit_Transport_ServiceAccept_UserAuth
Receive_Auth_Req_None
Transmit_Auth_Banner
Transmit_Auth_Failure
Receive_Auth_Req_PublicKey_Check
Transmit_Auth_Failure
Receive_Auth_Req_PublicKey_Check
Transmit_Auth_Failure

(#) auth

```
Transmit_Transport_ServiceAccept_UserAuth;
during {
 do {
  allow
   Transmit_Auth_Banner
  in {
   either {
    Receive_Auth_Req_None;
    Transmit_Auth_Failure;
   } or {
    Receive_Auth_Req_Password_Request;
    either {
     Transmit_Auth_Success;
     success = true;
    } or {
     Transmit_Auth_Failure;
    }
   } or {
    Receive_Auth_Req_PublicKey_Request;
    either {
     Transmit_Auth_Success;
     success = true;
    } or {
     Transmit_Auth_Failure;
    }
   } or {
    Receive_Auth_Req_PublicKey_Check;
    either {
     Transmit_Auth_PublicKey_OK;
    } or {
     Transmit_Auth_Failure;
    }
   } or {
    Notify_Auth_Permanent_Failure;
    failed = true;
   }
  }
 until (success || failed)
} handle {
 Transmit_Transport_Disconnect;
 exit;
}
```

(#) transport

```
during {
 allow
  Transmit_Transport_Debug
  Receive_Transport_Debug
  Transmit_Transport_Ignore
  Receive_Transport_Ignore
 in {
  multiple (0,∞) {
   either {
    either {
     Transmit_Transport_KexInit;
     Receive_Transport_KexInit;
    } or (encrypted) {
     Receive_Transport_KexInit;
     Transmit_Transport_KexInit;
    }
    either {
     Expect_DHInit;
     Receive_Dhgroupsha1_Init;
     Transmit_Dhgroupsha1_Reply;
    } or {
     Expect_GexInit;
     Receive_Dhgexsha1_Request;
     Transmit_Dhgexsha1_Group;
     Receive_Dhgexsha1_Init;
     Transmit_Dhgexsha1_Reply;
    }
    Receive_Transport_NewKeys;
    Transmit_Transport_NewKeys;
    encrypted = true;
   } or ((encrypted && (!serv_auth))) {
    Receive_Transport_ServiceReq_UserAuth;
    Transmit_Transport_ServiceAccept_UserAuth;
    serv_auth = true;
   }
  }
 }
} handle {
 either {
  Signal_HUP;
 } or {
  Signal_Quit;
  exit;
 } or {
  either {
   Receive_Transport_Disconnect;
  } or {
   Transmit_Transport_Disconnect;
   exit;
  }
```

Figure 7.8: Screen capture of the AJAX debugger embedded into the SSH daemon, showing the global SPL automaton. The green states are valid statecalls, the pie chart shows the 5 most popular statecalls in real time, and the list on the left show recent statecalls.

- $\Box \mathtt{a}$ where ($\mathtt{a} \leftarrow \mathtt{auth\_success} + \mathtt{auth\_failed} < 2$) informally reads "in the `auth` automaton, `success` and `failure` must never simultaneously be `true`". This restriction lets us use two boolean variables instead of a larger integer to store the 3 values for undecided, success or failure authentication states.

- $\Box(\mathtt{a} \rightarrow X(\mathtt{b} \,||\, \Box\Diamond\mathtt{c}))$ where ($\mathtt{a} \leftarrow \mathtt{p} == \mathtt{Transmit\_Auth\_Success}$) and ($\mathtt{b} \leftarrow \mathtt{auth\_success}$) and ($\mathtt{c} \leftarrow \mathtt{err}$) informally reads "when an authentication success packet is transmitted, it must immediately be followed by the `success` variable being `true` or always eventually lead to an error."

- $\Box(\mathtt{a} \rightarrow (\mathtt{b} \,||\, \Box\Diamond\mathtt{c}))$ where ($\mathtt{a} \leftarrow \mathtt{p} == \mathtt{Transmit\_Transport\_ServiceAccept\_UserAuth}$) and ($\mathtt{b} \leftarrow \mathtt{transport\_encrypted}$) and ($\mathtt{c} \leftarrow \mathtt{err}$) which informally reads "if the authentication service is unlocked then the transport layer must be encrypted or an error always eventually occurs". This matches the security considerations section of the SSH authentication specification in RFC4252 [291] which states that "it assumed *(sic)* that this runs over a secure transport layer protocol, which has already authenticated the server machine, established an encrypted communications channel [...]".

- $\Box(\mathtt{a} \rightarrow (\mathtt{b} \,||\, \Box\Diamond\mathtt{c}))$ where ($\mathtt{a} \leftarrow \mathtt{p} == \mathtt{Receive\_Channel\_Open\_Session}$) and ($\mathtt{b} \leftarrow \mathtt{auth\_success}$) and ($\mathtt{c} \leftarrow \mathtt{err}$) which informally reads "requests to open a new channel are only allowed when authentication has been successful, or an error state is always eventually reached". This is in line with the security considerations section of the SSH connection specification in RFC4254 [292] which states that "this protocol is assumed to run on top of a secure, authenticated transport".

The SPIN guided traces which result from violations of these policies are very easy to follow. For example, consider this fragment from the authentication automaton:

```
SPL
either {
    Receive_Auth_Req_None;
    Transmit_Auth_Success;
} or {
    Receive_Auth_Req_Password_Request;
    either {
        Transmit_Auth_Success;
        success = true;
    } or {
        Transmit_Auth_Failure;
    }
}
```

When the LTL constraints described above are applied, SPIN reports a violation and generates the guided trace seen in Figure 7.9. The trace consists of an entire successful key exchange and the receipt of two authentication packets. The mistake is now obvious; we have mistakenly placed an authentication success packet following the request for null authentication, instead of a failure. This error was caught by the LTL requirement that all success packets be immediately followed by setting the `success` state variable to true. In reality of course, this simple bug would have caught during testing (unless the bug was mirrored in the implementation), but by using a model checker we can eliminate the error even before the main application is created.

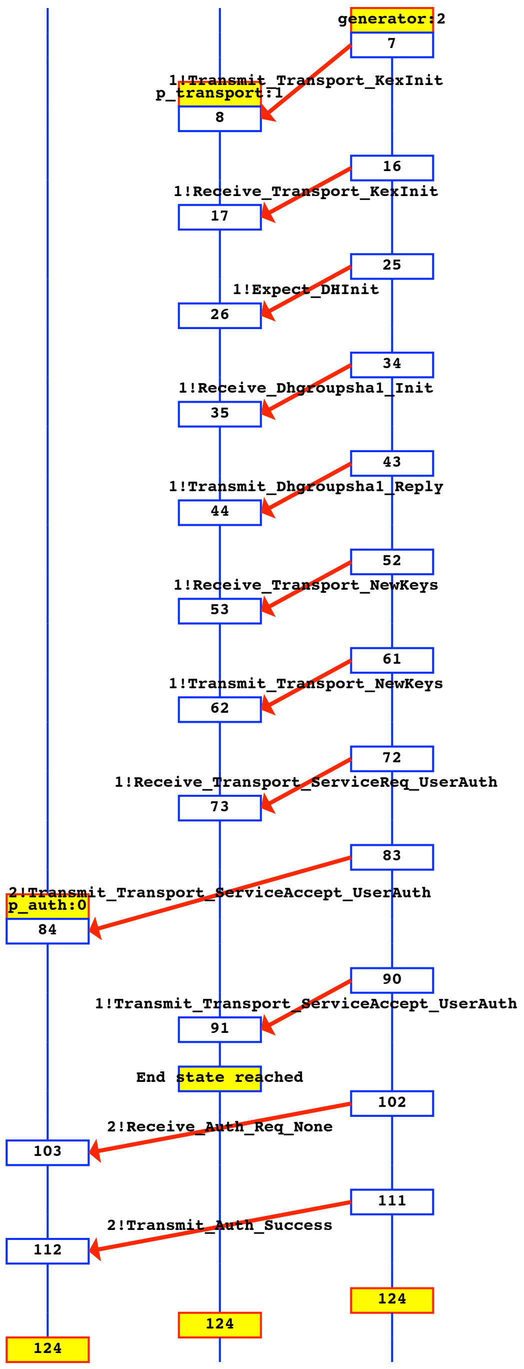

generator:2
7
1!Transmit_Transport_KexInit
p_transport:1
8
16
1!Receive_Transport_KexInit
17
25
1!Expect_DHInit
26
34
1!Receive_Dhgroupsha1_Init
35
43
1!Transmit_Dhgroupsha1_Reply
44
52
1!Receive_Transport_NewKeys
53
61
1!Transmit_Transport_NewKeys
62
72
1!Receive_Transport_ServiceReq_UserAuth
73
83
2!Transmit_Transport_ServiceAccept_UserAuth
p_auth:0
84
90
1!Transmit_Transport_ServiceAccept_UserAuth
91
End state reached
102
2!Receive_Auth_Req_None
103
111
2!Transmit_Auth_Success
112
124
124
124

Figure 7.9: The SPIN guided backtrace for the shortest possible violation of an LTL specification (see text for details)

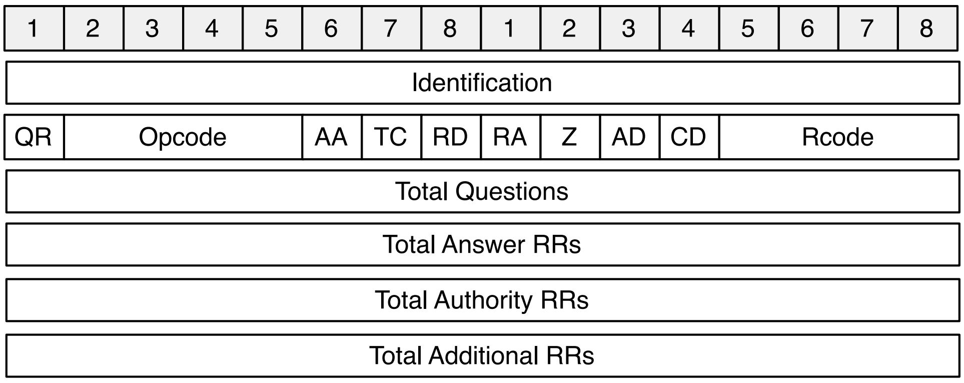


Figure 7.10: Format of a DNS Header message *(source: RFC1035 [208])*

7.2 Domain Name System

The Domain Name System (DNS) is a distributed database used to map textual names to information such as network addresses, mail forwarders, administrative contacts and even physical location. According to RFC1034 [207] the DNS consists of three components: (*i*) the Domain Name Space and Resource Records (RRs) which form a tree-structured namespace and the data associated with each name; (*ii*) name servers which hold information about portions of the domain name space, and can either act as an authoritative source for data or as a proxy which obtains and caches information from other name servers; and (*iii*) resolvers are generally part of client network stacks and manage the interface between client DNS requests and the local network name server.

Surveys of DNS name server deployment on the Internet have revealed that BIND [4] serves over 70% of DNS second-level `.com` domains and over 99% of the servers are written in C, according to surveys by Bernstein [32] and Moore [209]. BIND has had a long history of critical security vulnerabilities[6] and has been the target of worms[7] which exploit these vulnerabilities to self-propagate across hosts.

In the rest of this section we first describe the DNS packet format (§7.2.1), the architecture of our OCaml authoritative DNS server (§7.2.2) and a performance evaluation of our implementation against the widely deployed BIND (§7.2.3).

7.2.1 DNS Packet Format

DNS is designed to be a low-latency, low-overhead protocol for resolving domain names. In order to avoid the time taken to perform the 3-way TCP handshake most DNS requests and responses are encoded in a single UDP packet. This packet is normally restricted to be of 512 bytes or less (unless certain extensions, such as EDNS [276] are in use).

Due to these size restrictions, the original DNS specifications [207, 208] allocated a small number of bits to a various fields in the protocol header to pack them into a smaller space, as shown in Figure 7.10. The `identification` field is an unsigned 16-bit integer used to uniquely identify a question and the corresponding answer (necessary since the underlying transport

[6]The CERT Knowledge Base has these vulnerability ids, among others: VU#13145, VU#196945, VU#229595, VU#327533, VU#542971, VU#572183, VU#738331. Another list is available at `http://www.isc.org/index.pl?/sw/bind/bind-security.php`

[7]An anatomy of the “li0n” worm available from `http://www.cse.msu.edu/~enbody/virus/lionFuad.pdf` and `http://www.sans.org/y2k/lion.htm`

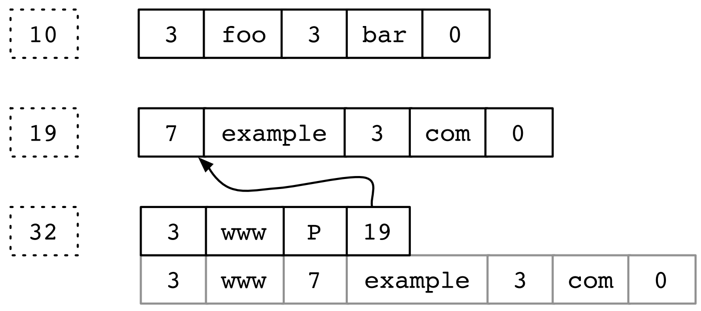


Figure 7.11: DNS label compression example, with `www.example.com` being encoded by a pointer. The dashed boxes are the offset from the start of the packet.

protocol can be connectionless, such as UDP). The next 16 bits contain a series of status flags such as whether the packet is a query or response, or an authoritative or truncated reply. Once the packet flags have been parsed, the next 4 fields are unsigned 16-bit integers containing the number of *Resource Records* (or RRs) which follow in their respective sections (query, answer, authority and additional). The RRs describe a DNS entry of a specific type, such as an `A` record for a hostname to IPv4 address mapping or an `MX` record for mapping domain names to their e-mail servers. The sections are used to distinguish different classes of answers: (*i*) the Answer section contains either authoritative or cached answers (depending on the value of the `aa` flag in the header); (*ii*) the Authority section contains meta-data about why the answers are authoritative by including the `NS` records for the authoritative name servers; and (*iii*) the Additional section provides any "glue" records which may be helpful for the client (e.g. the IP address of an authoritative name server referenced in the Authority section).

DNS packet parsing is made significantly more complex due to the use of a compression mechanism for hostnames. An uncompressed hostname is separated into a list of labels by splitting across the dot characters[8]. Each label is represented by a byte indicating its length following by the contents of the label. A length of `0` indicates the end of the hostname. The designers of DNS decided that if a sequence of labels occurred previously in the DNS packet it should be referenced instead of being duplicated (this repetition is quite common within hostnames in responses since at least the top-level portions are often the same between RR sets). Figure 7.11 illustrates how this compression works—two hostnames `foo.bar` and `example.com` are defined in different areas of a DNS response (the dashed boxes indicate the absolute offset within the packet). When the hostname `www.example.com` is encoded later, the first `www` is inserted as normal, but the tail of the hostname is replaced by a pointer to the previous definition of `example.com`. A pointer can only be used for the tail of a hostname but can lead to other strings which are also terminated by a pointer. Due to backward compatibility concerns, pointers may only point from hostnames contained in RRs defined in the original DNS RFCs [207, 208] so that an implementation is guaranteed to be capable of decompressing a label. The complexity of this compression scheme has directly led to bugs in many DNS implementations[9], for example by constructing a label with a pointer directed at itself.

We parse DNS labels via two MPL custom types `dns_label` and `dns_label_comp`, where the latter indicates a compressible hostname. Unlike most of our other custom types the imple-

[8]Confusingly the DNS specification does not forbid dot characters *within* labels, leaving it up to implementations to decide if this is valid or not!

[9]See CERT VU#23495 at `http://www.kb.cert.org/vuls/id/23495`

mentation of these requires stateful parsing actions and the MPL parser initialisation functions *must* be invoked carefully (normally these functions are identity functions and their omission may not be noticed). Internally hostnames are stored as a tuple of (`int` × `string list`) listing the encoded hostname size and the list of labels. The parsing maps very easily onto the standard OCaml hash-table, with pointer references only permitted to reference a previous offset[10].

These two custom types (along with the `string8` type described earlier) were sufficient to capture DNS packets in a single pass, thus avoiding the overhead of a two-pass resolution approach for hostnames. Appendix C.4 lists the MPL specification for the packet format shown in Figure 7.10. The most notable aspect is the use of the MPL **array** and **packet** keywords to capture RR sections without duplicating code. The RR specification (only common types are shown for the sake of brevity) is also listed, and it can be seen that the custom types abstract away the stateful parsing portions of DNS very effectively.

7.2.2 An Authoritative Deens Server

We implemented an OCaml authoritative DNS server in order to test the performance of a complete application against the reference BIND implementation. The OCaml server—dubbed `deens`—was created in collaboration with Tim Deegan who implemented efficient representations for large numbers of DNS records as part of his work on a more centralised name system [91]. This work included a DNS zone file parser and the OCaml data structures to store zones in-memory with a compressed trie representation.

`deens` is single-threaded; once zone file loading is complete it operates in a continuous loop which listens on an unconnected UDP socket using *recvfrom(2)* for queries and transmits responses back using *sendto(2)*. Both queries and answers are generated using only the automatically generated MPL interfaces, and no external C bindings were required beyond the functions provided by the OCaml standard library.

During our tests, we observed that the results of DNS queries are often idempotent with respect to the (`qclass` × `qname` × `qtype`) of a DNS question, where `qclass` is the DNS class (e.g. most often "Internet"), `qname` is the domain name and `qtype` is the type of RR being requested. The exception to this rule are servers which perform arbitrary processing when calculating responses (e.g. DNS load balancing [49]), but this is a specialist feature we are not concerned with for the moment. Features such as wildcard domains and DNS updates can be supported via standard functional data structure techniques [218] and thus also our memoisation cache. The only variation in response packets is that the first two bytes of the response must be modified to reflect the DNS `id` field of the request.

As an optimisation, we implemented a "memoisation" query cache which captured a query answer in a string containing the raw DNS response, and used the cached copy for further questions which were the same. The modifications required to `deens` were trivial, and to test the effectiveness of the technique we implemented two separate caching schemes: (*i*) a hash-table mapping the query fields to the marshalled packet bytes which is never removed from the hastable; and (*ii*) a "weak" hash-table (using the standard `Weak.Hashtbl` functor) of the query fields to the packet bytes.

The normal hash table simulates an ideal cache when large amounts of memory are available, since it performs no cache management and can thus leak memory. The weak hash table lies at the other extreme and is a cache which can be garbage collected and disappear at any

[10]The DNS specifications are not entirely clear on whether an forward pointers are allowed, but none of the implementations we have examined support this nor have any of the tested packet traces exhibited this.

time. Weak references are special data structures which do not count towards the reference counts of objects they point to for the purposes of garbage collection, and are often used as a safe mechanism to construct efficient purely functional data structures (known as “hash consing”[11]). In our case we are using the weak data structure itself as a cache without pointing it to anything, meaning that it is extremely transient and will be cleared on every garbage collection cycle. However, it also does not require any traditional cache management (e.g. least-recently-used checks) and can safely grow to any size since if the heap grows too large the cache will simply be erased.

7.2.3 Performance

We used the freely available tools from the BIND DLZ project[12] to evaluate the performance of `deens`. These tools generate both the source data for an authoritative server in the form of zone files and also an appropriate query set which can be fed into `queryperf` measurement tool from the BIND distribution. The query generation tools were configured to create zones and RRs in a rough Zipf power-law distribution[13], so that some hosts were more popular than others in the query set.

Our benchmarks are performed on the OpenBSD 3.8 operating system running on a 3.00GHz Pentium IV with 1GB RAM. The standard installation of BIND (9.3.1) included with OpenBSD was used with no modifications and configured to run a single instance, and `deens` compiled with OCaml 3.08.4 as native-code. Zone files are randomly generated for a variable number of domains and output as zone files which can be loaded into either BIND or `deens`. The `queryperf` tool from the BIND distribution was run against the DNS server on the same machine for 30 1-minute intervals for each set of zone data (the query data sets are re-randomised for each of the 1-minute runs), and the average and standard distribution of the reported queries-per-second calculated.

Throughput

The first test measured the performance of BIND against the `deens` server with query memoisation turned off. These results are shown in Figure 7.12. The results of the first test show that `deens` is slightly (around 10%) faster than a stock BIND installation for authoritative-only data. Figure 7.13 shows the results of another test run under the same conditions with memoisation turned on. There is a very large performance increase when using the space-leaking cache as `deens` takes advantage of caching its query results to double in performance and become significantly faster than BIND, at the expense of larger memory usage.

The results from weak memoisation are more interesting—it represents an extremely transient cache with a low hit rate, but with no increase in memory footprint since it can be eliminated safely at any time. The weak cache, although slower than the space-leaking cache is still significantly faster than the non-caching servers. This result is justified by the analysis Jung et al. performed on the effectiveness of DNS caching [160], in which they note that lowering the caching of RRs to even a few hundred seconds does not significantly reduce cache hits. Similarly, our simple experiment with weak hash-tables (which was a 3-line modification to the original non-caching `deens` server) demonstrates how our approach of reconstructing Internet protocol servers in a higher-level language can not only increase security and reliability, but

[11]An implementation and explanation for OCaml is available from `http://www.lri.fr/~filliatr/software.en.html`.

[12]`http://bind-dlz.sourceforge.net/`

[13]The correlation of DNS queries to a Zipf power-law distribution is well-established [160, 47]

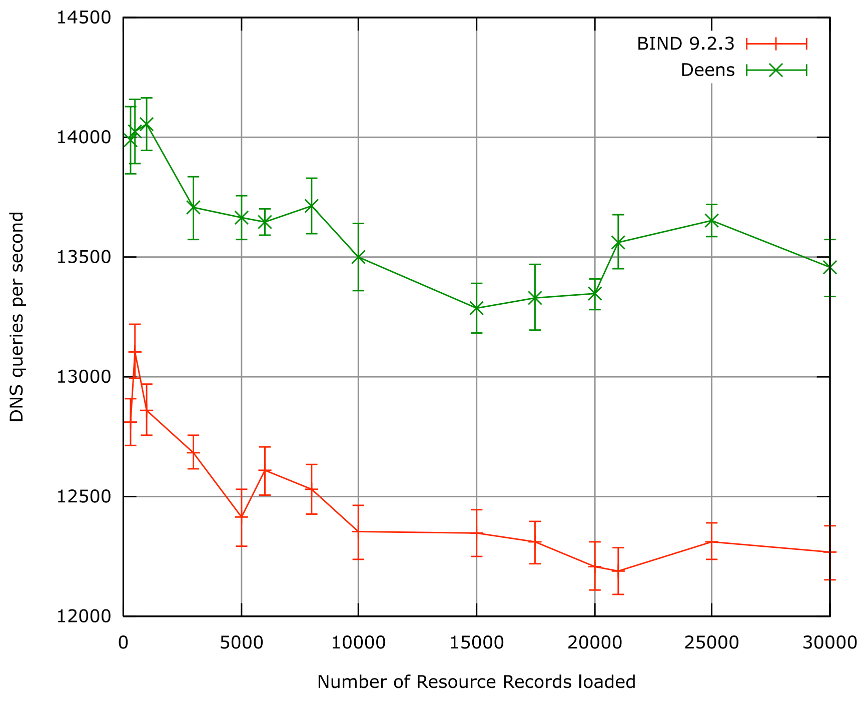


Figure 7.12: Query performance of BIND 9.3.1 versus deens with memoisation turned off

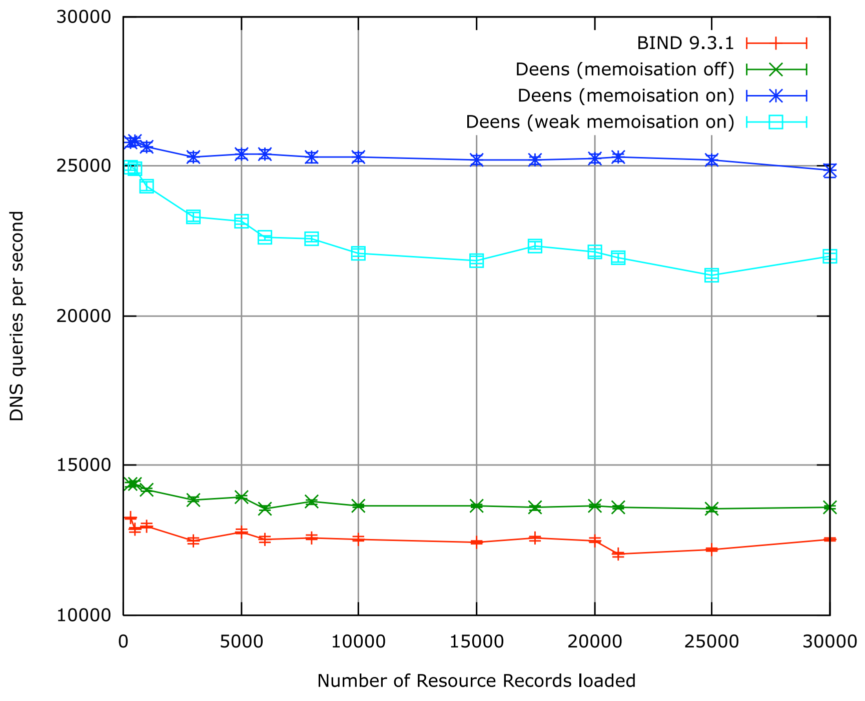


Figure 7.13: Query performance of BIND 9.3.1 versus deens running in three modes: (*i*) memoisation off; (*ii*) memoisation using a space-leaking hash-table; and (*iii*) memoisation using a safe weak hash-table cache.

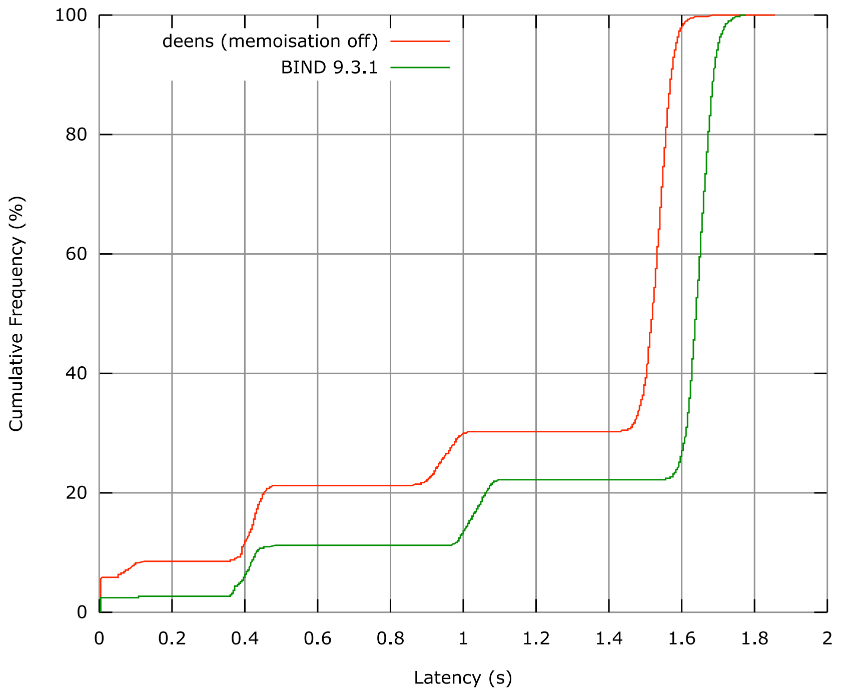


Figure 7.14: Cumulative distribution function for DNS query latency against a heavily loaded DNS server

also performance.

Latency

To test the latency of DNS responses, we first ran either `deens` or BIND under sustained load by running `queryperf` against it as in the previous throughput experiments. This load ensures that the OCaml `deens` server is undergoing a regular cycle of garbage collection while answering queries. Then the `nsping` utility[14] sends queries at 0.5 second intervals to the server and measures the latency of the returned response. The queries are run for an hour for each server, and the results shown in Figure 7.14 as a cumulative distribution graph. The results are remarkably consistent between `deens` and BIND, with `deens` being slightly faster in returning responses under load, and certainly not exhibiting any deviant behaviour due to overhead from the use of OCaml.

7.3 Code Size

A primary benefit of our approach is the smaller amount of code required to construct network applications. By reducing the difficulty and time required to rapidly implement Internet protocols (much as `yacc` simplified the task of writing language grammars), we hope to increase the adoption of type-safe programming languages such as OCaml.

To justify this claim of simplicity, we analyse the lines of code in our protocol implementations against their C equivalents. The C code is first pre-processed through `unifdef` to remove platform portability code that would artificially increase its size, but otherwise unmodified. The OCaml code is run though the `camlp4` pre-processor that reformats it to a consistent, well-tabulated style. External libraries such as OpenSSL or Cryptokit were not included in the count.

[14]Available in the OpenBSD ports tree in `net/nsping`

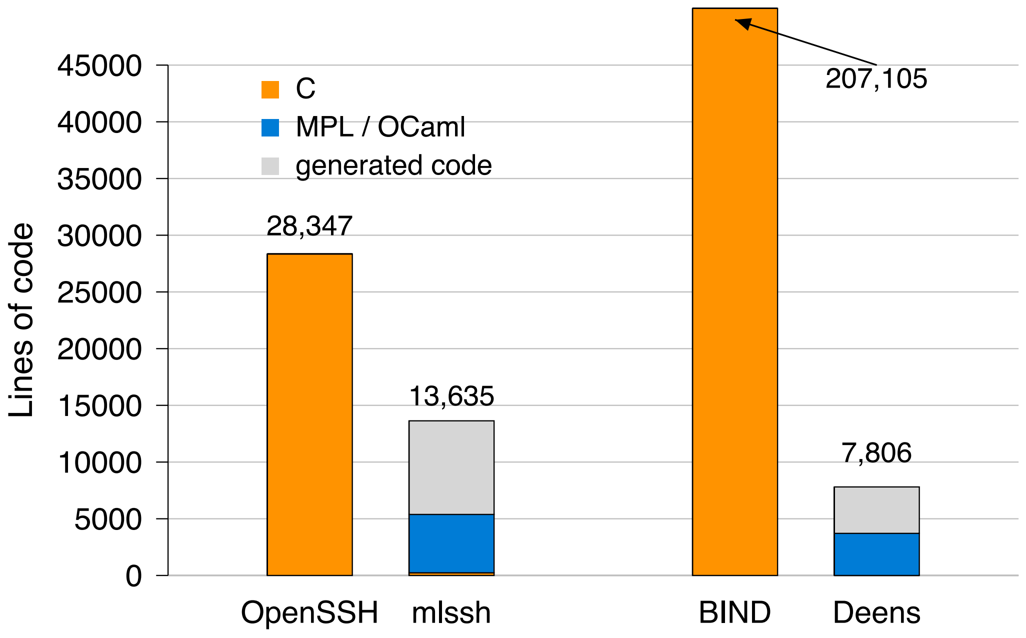


Figure 7.15: Relative code sizes for MPL/OCaml and C code *(lower is better)*.

Figure 7.15 plots the number of lines of C, OCaml and auto-generated code present in the applications. The figures for SH show that OpenSSH is nearly 3 times larger than the total lines of OCaml in MLSSH, and 6 times larger when considering only the hand-written OCaml.

The number for DNS reveal that `deens` is a remarkable 50 times smaller than the BIND 9.3.1 source code. `deens` does lack some of the features of BIND such as DNSSEC support and so this should only be treated as a rough metric. We are confident, particularly after our experiences with constructing MLSSH, that these features can be implemented with issue or dramatically increasing the OCaml code size.

7.4 Summary

In this chapter we have described two servers—MLSSH and `deens`—we constructed using the MELANGE application architecture. We evaluated the performance and latency characteristics of both of these implementations against their widely-deployed counterparts written in C and found that the OCaml versions perform equivalently under heavy load for sustained periods of time. In some cases, our implementations are significantly better—notably for SSH connection latency which exhibits extremely low variance compared to OpenSSH and for DNS peak throughput which performs *twice* as well as BIND with the trivial addition of a memoisation cache to our implementation. The code sizes of MLSSH and `deens` are also significantly smaller than their C equivalents.

This chapter seeks to dispel the widely-held assumption that garbage collection imposes inherent overhead for network applications when compared to manual memory management. We argue—and demonstrate via the above performance results—that with a clean separation of the data and control paths in a network application, garbage collection can actually be a positive feature for performance as well as for the more conventional safety and reliability aspects.

CHAPTER 8

Conclusions

Whenever people agree with me I always feel I must be wrong.
OSCAR WILDE

Recall from Chapter 1 our initial thesis statement that:

> Applications which communicate via standard Internet protocols must be rewritten to take advantage of developments in formal methods to increase their security and reliability, but still be high-performance, portable and practical in order to make continued deployment on the Internet feasible.

Let us start by considering the first part of this statement that "*applications which communicate via standard Internet protocols must be rewritten*". We justify this statement by arguing that the current trend of containment is not effective enough due to the persistence of attackers in finding new attacks that bypass them (§2.1) and that the rate of incidents and vulnerabilities to malicious attacks (§2.1.3) continues to grow (§2.1.4). We also note that the open RFC process which specifies Internet protocols (§2.1.1) makes it possible to write replacement applications which communicate via the same protocols.

Our thesis statement states that this reconstruction is necessary "*to take advantage of developments in formal methods to increase their security and reliability*". In our background work (§2) we described the history of functional languages which promote a more rigourous and well-specified programming style than the currently dominant C/C++ (§2.2), and in particular Objective Caml which provides a mature implementation on which to base our reconstruction efforts (§2.3). We also described the SPIN model checker which exhaustively verifies safety properties of abstract models of complex software systems (§2.4). Our survey of related work (§3) confirms that there exists a large amount of literature on constructing reliable network applications which is not currently being used in real applications deployed on the Internet (§1.1.3).

The thesis statement continues that the architecture must "*still be high-performance, portable and practical*". We note in our related work that pioneering projects such as FoxNet (§3.3.3) were very elegant examples of constructing network applications using a functional language, but fell short of the performance and portability requirements needed for deployment on the Internet. Therefore, our design goals (§4.1) and concrete MELANGE architecture (§4.2) made these a priority, and established the abstraction of a "data" and "control" plane for constructing network applications.

Our approach of constructing entirely new source code rather than relying on existing applications gave us the flexibility to re-examine conventional techniques for engineering network applications; in particular we imposed a requirement that the *entire* source code consist of type-safe OCaml as a new base-line for safety and security. We designed two domain-specific languages to implement the control and data abstractions and our architecture uses OCaml to implement the complex "glue" between the two planes. Both domain-specific languages are implemented in OCaml and output OCaml code in order to maintain the portability requirement that no compiler modifications be required.

The Meta Packet Language (MPL) handles the data plane by capturing protocol wire formats in succinct specifications and outputs high-performance and safe OCaml code which processes them with low overhead and no unnecessary data copying (§5). Unlike other data description languages which output C code, MPL demonstrates the feasibility of directly outputting statically type-safe code and thus the possibility of fully constructing high-performance network applications entirely in a high-level language. Our validation of this approach will be encouraging to generative meta-programming research (§3.3.2) which seeks to generalise the concept, and also to the field of data description languages which are becoming more formalised and feature-complete (§3.2.1) but still persist in outputting C code.

The Statecall Policy Language (SPL) specifies non-deterministic finite state automata that can be dynamically enforced with low overhead in an OCaml application and statically verified using the SPIN model checker (§6). Unlike conventional uses of model checking for systems code which involve model extraction from existing source code, SPL permits the developer to specify *both* the source code and the abstract models and decide how to hook them together. Failure of the main application to follow the abstract model results in a software exception being raised, from which termination or error recovery may occur. We argue that our approach is more appropriate to integrating model checking into software since it preserves high-level restrictions (e.g. LTL or CTL) across the evolution of the main application, and also permits the developer to directly choose their levels of abstraction without having to go through procedures such as counter-example refinement (§3.1.2).

We confirm the performance assertion by constructing implementations of two complex Internet protocols which have not previously been implemented with good performance (to our knowledge) in a high-level functional language such as OCaml. Our implementations of SSH (§7.1) and DNS (§7.2) were evaluated against their industry standard implementations OpenSSH and BIND respectively. In both cases our MELANGE implementations were at least as good as their C counterparts in terms of throughput and latency, and we demonstrated how the trivial use of features built into OCaml such as weak references dramatically increased the throughput of our DNS implementation to be twice as fast as BIND for authoritative DNS responses. Remarkably, the latency characteristics of our SSH implementation show that it is *more* stable than OpenSSH, revealing that the complexity of manual memory management can introduce more uncertainty into a network application than automatic garbage collection.

Our thesis statement concludes that our software was constructed "*to make continued deployment on the Internet feasible.*" We have released all of the source code for the compilers and applications described in this dissertation under a BSD-style open-source license[1] to ensure that the work can continue to be developed and deployed.

8.1 Future Work

Our approach of reconstructing software from scratch has involved re-evaluating many historical choices in the context of using modern languages and techniques, and some interesting future work in this area consists of: (*i*) continuing to develop new protocol implementations (e.g. DHCP and BGP servers) with a view for production deployment on the Internet; (*ii*) increase the integration of the applications with the operating system software stack; and (*iii*) raise our understanding of the software engineering process through which these applications are being constructed.

The development of new protocol implementations using functional languages is interesting as it eases the separation of algorithmic concerns from low-level protocol details. This is essential to avoid mistakes while evolving existing protocols to meet new demands, and is starting to be recognised as important by the networking community (e.g. meta-routing [126]) as the complexity of the Internet continues to increase. Another pragmatic reason for developing new implementations is that it allows the exploration of how to effectively *configure* complex network software. The current ad-hoc approaches makes reasoning of global network properties very difficult to machine verify.

UNIX (and its derivatives) are primarily operating systems which are written in C and exist to safely execute applications also written in C. More recent innovations such as high-level language run-times integrate poorly with the POSIX APIs exposed to them, and a lot of efficiency is lost due to interactions between the different layers. For example, when an OCaml application executes the *fork(2)* system call, the garbage collector managing its memory also splits since it is just part of the application run-time. If both processes subsequently perform a garbage collection, the efficient copy-on-write memory which both processes have as a result of the *fork(2)* will be copied and resources wasted as a result. If the garbage collectors were integrated directly into the operating system kernel, this inefficiency would be unnecessary.

Continuing this line of thought, many other features of conventional operating systems are not needed with applications which provide static typing guarantees. The use of separate virtual address spaces for isolation can be eliminated, especially on 64-bit architectures. Abstractions for concurrency based on time-quanta (e.g. threads) which are often used for convenience due to poor I/O APIs can be replaced by direct support for alternative high-level APIs (e.g. continuation passing style).

The recent ascent of *para-virtualisation* via systems such as Xen [23] makes it possible to construct entire light-weight operating systems without the need to support a wide range of hardware—a barrier which has led to the obsolescence of many past research efforts in novel operating systems. We are currently developing a prototype system, dubbed MLX, which executes MELANGE applications directly as a guest operating system running under Xen. This reduces the overall complexity of the system by removing a layer of software and leverages new features such as live migration [69] without requiring an entire OS to be also transferred with the application.

Although a perfectly secure Internet will probably never be realised, we anticipate that the lines of research begun in this dissertation will make life happier for system administrators, more miserable for virus authors, and more fun for programmers.

[1] Available at `http://melange.recoil.org/`

Bibliography

[1] ABADI, M., AND CARDELLI, L. *A Theory of Objects*. Springer-Verlag, New York, USA, 1996. Ref: page 26, 47

[2] AGHA, G., DE CINDIO, F., AND ROZENBERG, G., Eds. *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets* (2001), vol. 2001 of *Lecture Notes in Computer Science*, Springer. Ref: page 41

[3] AHO, A. V., AND ULLMAN, J. D. *Principles of Compiler Design*. Computer Science and Information Processing. Addison-Wesley, Reading, MA, USA, August 1977. Ref: page 99

[4] ALBITZ, P., AND LIU, C. *DNS and BIND*, fourth ed. O'Reilly, April 2001. Ref: page 9, 122

[5] ALEPH ONE. Smashing the stack for fun and profit. *Phrack 7*, 49 (1996), 14. Available from: `http://www.phrack.org/phrack/49/P49-14`. Ref: page 16

[6] ALEXANDER, D. S., MENAGE, P. B., KEROMYTIS, A. D., ARBAUGH, W. A., ANAGNOSTAKIS, K. G., AND SMITH, J. M. The price of safety in an active network. *Journal of Communications and Networks 3*, 1 (March 2001), 4–18. Ref: page 46

[7] ALTENKIRCH, T., MCBRIDE, C., AND MCKINNA, J. Why dependent types matter. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (January 2006). Available from: `http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`. Ref: page 22

[8] ALUR, R., AND WANG, B.-Y. Verifying network protocol implementations by symbolic refinement checking. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)* (London, UK, 2001), Springer-Verlag, pp. 169–181. Ref: page 43

[9] AMD64 TEAM. AMD64 architecture programmer's manual volume 1: Application programming. Tech. Rep. 24592, Advanced Micro Devices, 2005. Available from: `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf`. Ref: page 19

[10] APACHE FOUNDATION. The Apache web server [online]. 2006. Available from: `http://httpd.apache.org/`. Ref: page 9

[11] ARMSTRONG, J. The development of Erlang. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 1997), ACM Press, pp. 196–203. doi:10.1145/258948.258967. Ref: page 22, 26

[12] BACK, G. DataScript - a specification and scripting language for binary data. In *The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE)* (London, UK, 2002), Springer-Verlag, pp. 66–77. Ref: page 45

[13] BACK, R.-J. *On the Correctness of Refinement Steps in Program Development.* Department of computer science, University of Helsinki, Helsinki, Finland, 1978. Ref: page 48

[14] BACK, R.-J., AND WRIGHT, J. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, New York, USA, May 1998. Ref: page 48

[15] BACKUS, J. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM 21*, 8 (1978), 613–641. doi:10.1145/359576.359579. Ref: page 21

[16] BACKUS, J. W., BAUER, F. L., GREEN, J., KATZ, C., MCCARTHY, J., PERLIS, A. J., RUTISHAUSER, H., SAMELSON, K., VAUQUOIS, B., WEGSTEIN, J. H., VAN WIJNGAARDEN, A., AND WOODGER, M. Revised report on the algorithm language ALGOL 60. *Communications of the ACM 6*, 1 (1963), 1–17. doi:10.1145/366193.366201. Ref: page 70, 91

[17] BAKER, H. G. Infant mortality and generational garbage collection. *SIGPLAN Notices 28*, 4 (1993), 55–57. doi:10.1145/152739.152747. Ref: page 115

[18] BALL, T., COOK, B., DAS, S., AND RAJAMANI, S. K. Refining approximations in software predicate abstraction. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (April 2004), vol. 2988 of *Lecture Notes in Computer Science*, Springer-Verlag GmbH, pp. 388–403. Ref: page 42

[19] BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. K. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2001), ACM Press, pp. 203–213. doi:10.1145/378795.378846. Ref: page 37, 42

[20] BALL, T., AND RAJAMANI, S. K. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software* (New York, NY, USA, 2001), Springer-Verlag, pp. 103–122. Ref: page 37

[21] BALL, T., AND RAJAMANI, S. K. SLIC: A specification language for interface checking (of C). MSR-TR 2001-21, Microsoft Research, 2001. Ref: page 42

[22] BARENDREGT, H. P. *The Lambda Calculus: Its Syntax and Semantics*, 2nd ed. Studies in Logic and the Foundation of Mathematics. Elsevier B.V., 1997. Ref: page 20

[23] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (New York, NY, USA, 2003), ACM Press, pp. 164–177. doi:10.1145/945445.945462. Ref: page 10, 131

[24] BASET, S. A., AND SCHULZRINNE, H. An analysis of the skype peer-to-peer internet telephony protocol. Tech. Rep. CUCS-039-04, Columbia University, 2004. Ref: page 19

[25] BAUER, M. New covert channels in HTTP: adding unwitting web browsers to anonymity sets. In *Proceedings of the 2003 ACM Workshop on Privacy in the Electronic Society (WPES)* (New York, NY, USA, 2003), ACM Press, pp. 72–78. doi:10.1145/1005140.1005152. Ref: page 60

[26] BBC NEWS. UK "embraces digital technology" [online]. 2005. Available from: `http://news.bbc.co.uk/1/hi/entertainment/tv_and_radio/4679023.stm`. Ref: page 8

[27] BEAUDOUIN-LAFON, M., MACKAY, W. E., ANDERSEN, P., JANECEK, P., JENSEN, M., LASSEN, M., LUND, K., MORTENSEN, K., MUNCK, S., RATZER, A., RAVN, K., CHRISTENSEN, S., AND JENSEN, K. CPN/Tools: A post-WIMP interface for editing and simulating coloured Petri nets. In *Proceedings of the 22nd International Conference on the Application and Theory of Petri Nets* (2001), vol. 2075, Springer-Verlag, p. 71. Ref: page 41

[28] BEGEL, A., MCCANNE, S., AND GRAHAM, S. L. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. *SIGCOMM Computer Communications Review 29*, 4 (1999), 123–134. doi:10.1145/316194.316214. Ref: page 52

[29] BELLARE, M., KOHNO, T., AND NAMPREMPRE, C. The Secure Shell (SSH) Transport Layer Encryption Modes. RFC 4344 (Proposed Standard), Jan. 2006. Available from: `http://www.ietf.org/rfc/rfc4344.txt`. Ref: page 111

[30] BENZAKEN, V., CASTAGNA, G., AND FRISCH, A. CDuce: an XML-centric general-purpose language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP)* (New York, NY, USA, 2003), ACM Press, pp. 51–63. doi:10.1145/944705.944711. Ref: page 46

[31] BENZAKEN, V., AND DRIC MIACHON, G. C. C. A full pattern-based paradigm for xml query processing. In *7th International Symposium on the Practical Aspects of Declarative Languages (PADL)*, M. V. Hermenegildo and D. Cabeza, Eds., vol. 3350 of *Lecture Notes in Computer Science*. Springer, January 2005, pp. 235–252. Ref: page 46

[32] BERNSTEIN, D. J. Dns server survey [online]. 2002. Available from: `http://cr.yp.to/surveys/dns1.html`. Ref: page 9, 122

[33] BERNSTEIN, D. J. Cr.yp.to, home of qmail and djbdns [online]. 2006. Available from: `http://cr.yp.to/`. Ref: page 10

[34] BERRY, G. *The Foundations of Esterel: Proof, Language, and Interaction (Essay in Honor of Robin Milner)*. MIT Press, May 2000, ch. III, pp. 425–454. Ref: page 45, 88

[35] BEYER, D., CHLIPALA, A. J., HENZINGER, T. A., JHALA, R., AND MAJUMDAR, R. The BLAST query language for software verification. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)* (Verona, Italy, 2004), ACM Press, pp. 201–202. doi:10.1145/1014007.1014028. Ref: page 58

[36] BEYER, D., HENZINGER, T. A., JHALA, R., AND MAJUMDAR, R. Checking memory safety with blast. In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering: 8th International Conference (FASE)* (London, UK, 2005), M. Cerioli, Ed., vol. 3442 of *Lecture Notes in Computer Science*, Springer-Verlag GmbH, p. 2. doi:10.1007/b107062. Ref: page 42

[37] BIAGIONI, E. A structured TCP in Standard ML. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM)* (New York, NY, USA, 1994), ACM Press, pp. 36–45. doi:10.1145/190314.190318. Ref: page 48

[38] BIAGIONI, E., HARPER, R., AND LEE, P. A network protocol stack in Standard ML. *Higher Order Symbolic Computing 14*, 4 (2001), 309–356. doi:10.1023/A:1014403914699. Ref: page 10, 48, 49

[39] BIRD, R., AND WADLER, P. *Introduction to Functional Programming*. Series in Computer Science. Prentice Hall, 1998. Ref: page 21

[40] BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (2005), ACM Press, pp. 265–276. doi:10.1145/1080091.1080123. Ref: page 43

[41] BLATHERWICK, P., BELL, R., AND HOLLAND, P. Megaco IP Phone Media Gateway Application Profile. RFC 3054 (Informational), Jan. 2001. Available from: `http://www.ietf.org/rfc/rfc3054.txt`. Ref: page 44

[42] BONWICK, J. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Annual Technical Conference* (1994), USENIX, pp. 87–98. Ref: page 109

[43] BRADEN, R., BORMAN, D., AND PARTRIDGE, C. Computing the Internet checksum. RFC 1071, Sept. 1988. Updated by RFC 1141. Available from: `http://www.ietf.org/rfc/rfc1071.txt`. Ref: page 69

[44] BRADNER, S. The Internet Standards Process – Revision 3. RFC 2026 (Best Current Practice), Oct. 1996. Updated by RFCs 3667, 3668, 3932, 3979, 3978. Available from: `http://www.ietf.org/rfc/rfc2026.txt`. Ref: page 15

[45] BRADNER, S. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), Mar. 1997. Available from: `http://www.ietf.org/rfc/rfc2119.txt`. Ref: page 15

[46] BRAUNER, T. Introduction to linear logic. BRICS-LS 96-6, BRICS, 1996. Available from: `http://www.brics.dk/LS/96/6/BRICS-LS-96-6/BRICS-LS-96-6.html`. Ref: page 22

[47] BRESLAU, L., CAO, P., FAN, L., PHILLIPS, G., AND SHENKER, S. Web caching and Zipf-like distributions: evidence and implications. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)* (March 1999), pp. 126–134. Ref: page 125

[48] BREWER, E., CONDIT, J., MCCLOSKEY, B., AND ZHOU, F. Thirty years is long enough: Getting beyond C. In *Proceedings of the 10th Workshop of Hot Topics in Operating Systems (HOTOS)* (2005), USENIX. Ref: page 10

[49] BRISCO, T. DNS Support for Load Balancing. RFC 1794 (Informational), Apr. 1995. Available from: `http://www.ietf.org/rfc/rfc1794.txt`. Ref: page 124

[50] BRYANT, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers 35*, 8 (1986), 677–691. Ref: page 33

[51] BRYANT, R. E. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys 24*, 3 (1992), 293–318. doi:10.1145/136035.136043. Ref: page 33

[52] BUDKOWSKI, S., AND DEMBINSKI, P. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems 14*, 1 (1991), 3–24. Ref: page 37

[53] BUNKER, A., GOPALAKRISHNAN, G., AND MCKEE, S. A. Formal hardware specification languages for protocol compliance verification. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 9*, 1 (2004), 1–32. doi:10.1145/966137.966138. Ref: page 42

[54] BURSTALL, R. M., MACQUEEN, D. B., AND SANNELLA, D. T. Hope: An experimental applicative language. In *Conference Record of the 1980 LISP Conference* (Stanford University, Stanford, California, August 1980), ACM Press, pp. 136–143. Available from: `citeseer.ist.psu.edu/burstall80hope.html`. Ref: page 21, 25

[55] CALHOUN, P., AND PERKINS, C. Mobile IP Network Access Identifier Extension for IPv4. RFC 2794 (Proposed Standard), Mar. 2000. Available from: `http://www.ietf.org/rfc/rfc2794.txt`. Ref: page 44

[56] CALLAS, J., DONNERHACKE, L., FINNEY, H., AND THAYER, R. OpenPGP Message Format. RFC 2440 (Proposed Standard), Nov. 1998. Available from: `http://www.ietf.org/rfc/rfc2440.txt`. Ref: page 44

[57] CANTRILL, B., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference (General Track)* (June 2004), USENIX, pp. 15–28. Ref: page 43

[58] CARDELLI, L. Type systems. In *The Computer Science and Engineering Handbook*, A. B. Tucker, Ed. CRC Press, 1997, pp. 2208–2236. Ref: page 91

[59] CARDELLI, L., AND GORDON, A. D. Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure (FoSSaCS)* (London, UK, 1998), Springer-Verlag, pp. 140–155. Ref: page 40

[60] CERT coordination center [online]. Available from: `http://www.cert.org/`. Ref: page 17

[61] CERT COORDINATION CENTER. CERT knowledge base [online]. Available from: `http://www.cert.org/kb/`. Ref: page 17

[62] CERT COORDINATION CENTER. Incident note in-2001-12, November 2001. Ref: page 111

[63] CHAILLOUX, E., MANOURY, P., AND PAGANO, B. Developing applications with objective caml [online]. 2000. Available from: `http://caml.inria.fr/pub/docs/oreilly-book/`. Ref: page 27

[64] CHEN, H., DEAN, D., AND WAGNER, D. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, February 2004). Available from: `http://www.isoc.org/isoc/conferences/ndss/04/proceedings/Papers/Chen.pdf`. Ref: page 37

[65] CHEN, H., AND WAGNER, D. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)* (New York, NY, USA, 2002), ACM Press, pp. 235–244. doi:10.1145/586110.586142. Ref: page 37, 90

[66] CHEN, J., AND CUI, H. Translation from adapted UML to Promela for CORBA-based applications. In *Proceedings of the 11th Internation SPIN Workshop* (New York, USA, 2004), Springer-Verlag, pp. 234–251. Ref: page 37

[67] CHOI, I., SONG, M., PARK, C., AND PARK, N. An XML-based process definition language for integrated process management. *Computers in Industry 50*, 1 (2003), 85–102. doi:10.1016/S0166-3615(02)00139-2. Ref: page 41

[68] CHU, H. K. J. Zero-copy TCP in Solaris. In *Proceedings of the USENIX Annual Technical Conference* (1996), USENIX, pp. 253–264. Ref: page 52

[69] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd Symposium of Networked Systems Design and Implementation* (May 2005). Ref: page 131

[70] CLARK, K., AND GREGORY, S. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Language Systems 8*, 1 (1986), 1–49. doi:10.1145/5001.5390. Ref: page 22

[71] CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM 50*, 5 (2003), 752–794. doi:10.1145/876638.876643. Ref: page 42

[72] CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS) 8*, 2 (1986), 244–263. doi:10.1145/5397.5399. Ref: page 33

[73] COMPUTER INCIDENT ADVISORY CAPABILITY. I-092: SunOS ping buffer overflow vulnerability [online]. September 1998. Available from: `http://www.ciac.org/ciac/bulletins/i-092.shtml`. Ref: page 81

[74] CORBETT, J. C. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering 22*, 3 (1996), 161–180. doi:10.1109/32.489078. Ref: page 33

[75] CORBETT, J. C., DWYER, M. B., AND HATCLIFF, J. A language framework for expressing checkable properties of dynamic software. In *Proceedings of the SPIN Software Model Checking Workshop* (2000), K. Havelund, J. Penix, and W. Visser, Eds., vol. 1885 of *Lecture Notes in Computer Science*, Springer, pp. 205–223. Ref: page 37, 41

[76] CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., P&#259;S&#259;REANU, C. S., ROBBY, AND ZHENG, H. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)* (New York, NY, USA, 2000), ACM Press, pp. 439–448. doi:10.1145/337180.337234. Ref: page 41

[77] CORBETT, J. C., DWYER, M. B., HATCLIFF, J., AND ROBBY. Expressing checkable properties of dynamic systems: the Bandera Specification Language. *International Journal on Software Tools for Technology Transfer 4*, 1 (2002), 34–56. Ref: page 41

[78] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-end containment of internet worms. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), ACM Press, pp. 133–147. doi:10.1145/1095810.1095824. Ref: page 17

[79] COUSINEAU, G., AND MAUNY, M. *The Functional Approach to Programming*. Cambridge University Press, October 1998. Ref: page 22, 27

[80] COWAN, C., BEATTIE, S., WRIGHT, C., AND KROAH-HARTMAN, G. Raceguard: Kernel protection from temporary file race vulnerabilities. In *Proceedings of the 10th USENIX Security Conference* (2001), USENIX. Available from: `http://www.usenix.org/events/sec01/cowanbeattie.html`. Ref: page 10, 19

[81] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference* (January 1998), pp. 63–78. Ref: page 19

[82] CRANOR, C. D., AND PARULKAR, G. M. The UVM virtual memory system. In *Proceedings of the 1999 USENIX Annual Technical Conference (General Track)* (1999), USENIX, pp. 117–130. Ref: page 52

[83] CRANOR, L. F., AND LAMACCHIA, B. A. Spam! *Communications of the ACM 41*, 8 (1998), 74–83. doi:10.1145/280324.280336. Ref: page 60

[84] CREDENTIA. E-mail server survey results for April 2003 [online]. April 2003. Available from: `http://www.credentia.cc/research/surveys/smtp/200304/`. Ref: page 10

[85] CROCKER, D., AND OVERELL, P. Augmented BNF for Syntax Specifications: ABNF. RFC 2234 (Proposed Standard), Nov. 1997. Obsoleted by RFC 4234. Available from: `http://www.ietf.org/rfc/rfc2234.txt`. Ref: page 69

[86] CROSBY, S. A., AND WALLACH, D. S. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium* (August 2003), USENIX, pp. 29–44. Ref: page 60

[87] CUSACK, F., AND FORSSEN, M. Generic Message Exchange Authentication for the Secure Shell Protocol (SSH). RFC 4256 (Proposed Standard), Jan. 2006. Available from: `http://www.ietf.org/rfc/rfc4256.txt`. Ref: page 111

[88] DABBOUS, W., O'MALLEY, S., AND CASTELLUCCIA, C. Generating efficient protocol code from an abstract specification. In *Conference proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (New York, NY, USA, 1996), ACM Press, pp. 60–72. doi:10.1145/248156.248163. Ref: page 45, 49

[89] DAHL, O.-J., DIJKSTRA, E., AND HOARE, A. *Structured Programming*. Academic Press, June 1972. Ref: page 47

[90] DALLIEN, J., AND MACCAULL, W. Automated checking for stutter invariance of ltl formulas. In *Proceedings of the 28th Annual APICS Conference on Mathematics-Statistics-Computer Science* (October 2004). Available from: `http://www.unbsj.ca/conferences/apics/2004/DallienMacCaull.ps`. Ref: page 35

[91] DEEGAN, T., CROWCROFT, J., AND WARFIELD, A. The main name system: an exercise in centralized computing. *SIGCOMM Computer Communications Review 35*, 5 (2005), 5–14. doi:10.1145/1096536.1096538. Ref: page 124

[92] DHAMIJA, R., AND TYGAR, J. D. The battle against phishing: Dynamic security skins. In *Proceedings of the 2005 Symposium on Usable Privacy and Security (SOUPS)* (New York, NY, USA, 2005), ACM Press, pp. 77–88. doi:10.1145/1073001.1073009. Ref: page 60

[93] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Updated by RFC 3546. Available from: `http://www.ietf.org/rfc/rfc2246.txt`. Ref: page 8, 44

[94] DIJKSTRA, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 8 (1975), 453–457. doi:10.1145/360933.360975. Ref: page 48

[95] DROMS, R. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), Mar. 1997. Updated by RFC 3396. Available from: `http://www.ietf.org/rfc/rfc2131.txt`. Ref: page 36

[96] DSHIELD INC. Distributed intrusion detection system [online]. 2005. Available from: `http://www.dshield.com/`. Ref: page 17

[97] DUNKELS, A. lwIP - a lightweight TCP/IP stack [online]. Available from: `http://savannah.nongnu.org/projects/lwip/`. Ref: page 81, 82

[98] EADS/CRC TEAM. Integer overflow in skype client. CERT Advisory CVE-2005-3267, October 2005. Ref: page 19

[99] ELSMAN, M., AND HALLENBERG, N. Web programming with SMLserver. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL)* (London, UK, 2003), Springer-Verlag, pp. 74–91. Ref: page 49

[100] ENGLER, D., AND ASHCRAFT, K. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), ACM Press, pp. 237–252. doi:10.1145/945445.945468. Ref: page 37

[101] ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. Checking system rules using system-specific, programmer-written compiler extensions. In *In Proceedings of the 4th Symposium on Operating Systems Design and Implementation* (October 2000), USENIX. Available from: `http://www.usenix.org/events/osdi2000/engler.html`. Ref: page 48

[102] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. 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)* (New York, NY, USA, 2001), ACM Press, pp. 57–72. doi:10.1145/502034.502041. Ref: page 48

[103] ENNALS, R. *Adaptive Evaluation of Non-Strict Programs*. PhD thesis, University of Cambridge, 2004. Ref: page 21, 24

[104] ENNALS, R., SHARP, R., AND MYCROFT, A. Linear types for packet processing. In *13th European Symposium on Programming (ESOP), part of the Joint European Conferences on Theory and Practice of Software (ETAPS)* (Barcelona, Spain, April 2004), D. A. Schmidt, Ed., vol. 2986 of *Lecture Notes in Computer Science*, Springer, pp. 204–218. Ref: page 22

[105] ETOH, H. GCC extension for protecting applications from stack-smashing attacks. Tech. rep., IBM Research Japan, 2004. Available from: `http://www.research.ibm.com/trl/projects/security/ssp/`. Ref: page 19

[106] EVANS, D., AND LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE Software 19*, 1 (January–February 2002), 42–51. Ref: page 59

[107] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817. Available from: `http://www.ietf.org/rfc/rfc2616.txt`. Ref: page 69

[108] FINDLER, R. B., AND FELLEISEN, M. Contracts for higher-order functions. In *Proceedings of the International Conference in Functional Programming (ICFP)* (October 2002), ACM Press. Ref: page 23

[109] FISHER, K., AND GRUBER, R. Pads: a domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2005), ACM Press, pp. 295–304. doi:10.1145/1065010.1065046. Ref: page 45, 68

[110] FISHER, K., MANDELBAUM, Y., AND WALKER, D. The next 700 data description languages. In *Proceedings of the ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL)* (January 2006). Ref: page 45

[111] FLANAGHAN, C. Hybrid type checking. In *Principles of Programming Languages (POPL)* (2006). Ref: page 23

[112] FOSTER, J. N., GREENWALD, M. B., KIRKEGAARD, C., PIERCE, B. C., AND SCHMITT, A. Schema-directed data synchronization. Technical Report MS-CIS-05-02, University of Pennsylvania, March 2005. Supercedes MS-CIS-03-42. Available from: `http://www.cis.upenn.edu/~bcpierce/papers/sync-tr.pdf`. Ref: page 46

[113] FOSTER, J. N., GREENWALD, M. B., MOORE, J. T., PIERCE, B. C., AND SCHMITT, A. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (New York, NY, USA, 2005), ACM Press, pp. 233–246. doi:10.1145/1040305.1040325. Ref: page 47

[114] FRISCH, A. Regular tree language recognition with static information. In *Workshop on Programming Language Technologies for XML (PLAN-X)* (January 2004). Available from: `http://www.cduce.org/papers/reg.pdf`. Ref: page 46

[115] FURR, M., AND FOSTER, J. S. Checking type safety of foreign function calls. *SIGPLAN Notices 40*, 6 (2005), 62–72. Originally in PLDI 2005. doi:10.1145/1064978.1065019. Ref: page 59

[116] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 ed. Addison-Wesley Professional Computing Series, January 1995. Ref: page 47, 57

[117] GANSNER, E. R., AND NORTH, S. C. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience 30*, 11 (2000), 1203–1233. Ref: page 91

[118] GARRIGUE, J. Programming with polymorphic variants. In *1998 ACM SIGPLAN Workshop on ML* (Baltimore, Maryland, USA, September 1998). Available from: `http://www.math.nagoya-u.ac.jp/~garrigue/papers/variants.ps.gz`. Ref: page 29

[119] GARRIGUE, J. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering* (Sasaguri, Japan, November 2000). Available from: `http://www.math.nagoya-u.ac.jp/~garrigue/papers/fose2000.html`. Ref: page 29

[120] GARRIGUE, J. Private rows: abstracting the unnamed. (draft), June 2005. Available from: `http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/privaterows.pdf`. Ref: page 85

[121] GETCHELL, A., AND SATALURI, S. A Revised Catalog of Available X.500 Implementations. RFC 1632 (Informational), May 1994. Obsoleted by RFC 2116. Available from: `http://www.ietf.org/rfc/rfc1632.txt`. Ref: page 44

[122] GORDON, M. J. C., MILNER, R., MORRIS, L., NEWEY, M. C., AND WADSWORTH, C. P. A metalanguage for interactive proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)* (New York, NY, USA, 1978), ACM Press, pp. 119–130. doi:10.1145/512760.512773. Ref: page 21

[123] GOVINDAVAJHALA, S., AND APPEL, A. W. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP)* (Washington, DC, USA, 2003), IEEE Computer Society, p. 154. Ref: page 61

[124] GRAF, S., AND SAIDI, H. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)* (London, UK, 1997), Springer-Verlag, pp. 72–83. Ref: page 37, 42

[125] GREENBERG, A., HJALMTYSSON, G., MALTZ, D. A., MYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., AND ZHANG, H. A clean slate 4D approach to network control and management. *SIGCOMM Computer Communications Review 35*, 5 (2005), 41–54. doi:10.1145/1096536.1096541. Ref: page 11, 39

[126] GRIFFIN, T. G., AND SOBRINHO, J. L. Metarouting. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (New York, NY, USA, 2005), ACM Press, pp. 1–12. doi:10.1145/1080091.1080094. Ref: page 131

[127] GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Deploying safe user-level network services with icTCP. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (2004), USENIX, pp. 317—332. Ref: page 50

[128] GUSTAFSSON, P., AND SAGONAS, K. Native code compilation of Erlang's bit syntax. In *Proceedings of ACM SIGPLAN Erlang Workshop* (November 2002), ACM Press, pp. 6–15. Ref: page 26

[129] HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., AND A. SHTUL-TRAURING. Statemate: a working environment for the development of complex reactive systems. In *Proceedings of the 10th International Conference on Software Engineering (ICSE)* (Los Alamitos, CA, USA, 1988), IEEE Computer Society Press, pp. 396–406. Ref: page 88

[130] HAVELUND, K., AND PRESSBURGER, T. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer 2*, 4 (2000), 366–381. Ref: page 37

[131] HAYDEN, M. *The Ensemble System.* Tr98-1662, Cornell University, 1998. Available from: `http://www.nuprl.org/documents/Hayden/ensemblesystem.html`. Ref: page 10, 45, 49

[132] HENSBERGEN, E. V. Plan 9 remote resource protocol (experimental-draft-9p2000-protocol), March 2005. Available from: `http://v9fs.sourceforge.net/rfc/`. Ref: page 76

[133] HENZINGER, T. A., JHALA, R., MAJUMDAR, R., NECULA, G. C., SUTRE, G., AND WEIMER, W. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)* (London, UK, 2002), Springer-Verlag, pp. 526–538. Ref: page 37, 41, 88, 96

[134] HICKS, M., KAKKAR, P., MOORE, J. T., GUNTER, C. A., AND NETTLES, S. Plan: A packet language for active networks. *SIGPLAN Notices 34*, 1 (1999), 86–93. doi:10.1145/291251.289431. Ref: page 45

[135] HINDLEY, J. R. The principal type-scheme of an object in combinatory logic. *Transactions of American Mathematics Society 146* (1969), 29–60. Ref: page 21

[136] HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (1978), 666–677. doi:10.1145/359576.359585. Ref: page 40

[137] HOLLAND, D. A., LIM, A. T., AND SELTZER, M. I. An architecture a day keeps the hacker away. *SIGARCH Computer Architecture News 33*, 1 (2005), 34–41. doi:10.1145/1055626.1055632. Ref: page 10

[138] HOLZMANN, G. J. *The SPIN Model Checker.* Addison-Wesley, September 2003. Ref: page 32, 33, 37, 87

[139] HOLZMANN, G. J., AND PELED, D. Partial order reduction of the state space. In *Proceedings of the 1st SPIN Workshop on the Model Checking of Software* (1995). Ref: page 33, 106

[140] HOLZMANN, G. J., AND SMITH, M. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering 28*, 4 (April 2002), 364–377. Ref: page 37

[141] HOSOYA, H., AND PIERCE, B. C. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology 3*, 2 (2003), 117–148. doi:10.1145/767193.767195. Ref: page 46

[142] HUDAK, P. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys 21*, 3 (1989), 359–411. doi:10.1145/72551.72554. Ref: page 22

[143] HUDAK, P., AND WADLER, P. Report on the functional programming language Haskell. YALEU/DCS/RR 656, Yale University, 1988. Ref: page 21

[144] HUGHES, J. Why functional programming matters. *Computer Journal 32*, 2 (1989), 98–107. Ref: page 23

[145] HUTTON, G. Frequently asked questions for comp.lang.functional [online]. November 2002. Available from: `http://www.cs.nott.ac.uk/~gmh/faq.html`. Ref: page 20, 22

[146] INRIA-ROCQUENCOURT. The Coq proof assistant [online]. Available from: `http://coq.inria.fr/`. Ref: page 48, 57

[147] INTERNET ENGINEERING STEERING GROUP. Guidelines for the use of formal languages in IETF specifications [online]. October 2001. Available from: `http://www.ietf.org/IESG/STATEMENTS/pseudo-code-in-specs.txt`. Ref: page 15

[148] ISO. Estelle—a formal description technique based on an extended state transition model. ISO 9074, International Organisation for Standardization, Geneva, 1997. Ref: page 88

[149] JACOBSON, V. Congestion avoidance and control. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 1988), ACM Press, pp. 314–329. doi:10.1145/52324.52356. Ref: page 8

[150] JACOBSON, V., LERES, C., AND MCCANNE, S. Packet capture with tcpdump and pcap [online]. Available from: `http://www.tcpdump.org/`. Ref: page 76

[151] JENSEN, K. Coloured Petri Nets: a high level language for system design and analysis. In *Proceedings on Advances in Petri Nets (APN)* (New York, NY, USA, 1991), Springer-Verlag New York, pp. 342–416. Ref: page 41

[152] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference (General Track)* (June 2002), USENIX, pp. 275–288. Ref: page 10

[153] JOHNSON, C. A., AND TESCH, B. US eCommerce: 2005 to 2010. Tech. rep., Forrester Research, 2005. Available from: `http://www.forrester.com/Research/Document/Excerpt/0,7211,37626,00.html`. Ref: page 7

[154] JOHNSON, S. C. Yacc: Yet Another Compiler Compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, New Jersey, USA, 1975. Ref: page 44, 91

[155] JONES, G. *Programming in Occam.* Prentice-Hall, Hertfordshire, United Kingdom, 1986. Ref: page 34, 40, 89

[156] JONES, R., AND LINS, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, 2nd ed. John Wiley and Sons, 1999. Available from: `http://www.cs.kent.ac.uk/people/staff/rej/gcbook/gcbook.html`. Ref: page 22

[157] JONES, S. L. P., Ed. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003. Available from: `http://www.haskell.org/report/`. Ref: page 21

[158] JONES, S. L. P. Wearing the hair shirt: a retrospective on Haskell [online]. 2003. Available from: `http://research.microsoft.com/~simonpj/papers/haskell-retrospective/`. Ref: page 24

[159] JONES, S. P. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987. Available from: `http://research.microsoft.com/~simonpj/papers/slpj-book-1987/`. Ref: page 21

[160] JUNG, J., SIT, E., BALAKRISHNAN, H., AND MORRIS, R. DNS performance and the effectiveness of caching. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement (IMW)* (New York, NY, USA, 2001), ACM Press, pp. 153–167. doi:10.1145/505202.505223. Ref: page 125

[161] JÜNGER, M., AND MUTZEL, P., Eds. *Graph Drawing Software (Mathematics and Visualization)*, 1 ed. Springer, October 2003. Ref: page 58, 156

[162] KALOXYLOS, A. G. An Estelle to Promela compiler. Master's thesis, Heriot-Watt University, 1994. Ref: page 37

[163] KAY, J., AND PASQUALE, J. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 1993), ACM Press, pp. 259–268. doi:10.1145/166237.166262. Ref: page 81

[164] KC, G. S., AND KEROMYTIS, A. D. e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing. In *Proceedings of 21st Annual Computer Security Applications Conference (ACSAC)* (2005), IEEE Computer Society, pp. 286–302. Ref: page 10, 19

[165] KENNEDY, A., AND SYME, D. Design and implementation of generics for the .net common language runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2001), ACM Press, pp. 1–12. doi:10.1145/378795.378797. Ref: page 26

[166] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language*, 2nd ed. Prentice Hall, 1988. Ref: page 15

[167] KHALIDI, Y. A., AND THADANI, M. N. An efficient zero-copy I/O framework for UNIX. Tech. rep., Mountain View, CA, USA, 1995. Ref: page 52

[168] KLEENE, S. C. λ-definability and recursiveness. *Duke Mathematics Journal*, 2 (1936), 340–353. Ref: page 20

[169] KLENSIN, J. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), Apr. 2001. Available from: `http://www.ietf.org/rfc/rfc2821.txt`. Ref: page 69

[170] KOBAYASHI, N. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (New York, NY, USA, 1999), ACM Press, pp. 29–42. doi:10.1145/292540.292546. Ref: page 22, 46

[171] KOHLER, E., KAASHOEK, M. F., AND MONTGOMERY, D. R. A readable TCP in the Prolac protocol language. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 1999), ACM Press, pp. 3–13. doi:10.1145/316188.316200. Ref: page 45

[172] KUPERMAN, B. A., BRODLEY, C. E., OZDOGANOGLU, H., VIJAYKUMAR, T. N., AND JALOTE, A. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM 48*, 11 (2005), 50–56. doi:10.1145/1096000.1096004. Ref: page 19

[173] LANDIN, P. J. The next 700 programming languages. *Communications of the ACM 9*, 3 (1966), 157–166. doi:10.1145/365230.365257. Ref: page 20

[174] LEA, D., AND MARLOWE, J. Interface-based protocol specification of open systems using psl. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)* (London, UK, 1995), Springer-Verlag, pp. 374–398. Ref: page 48

[175] LEE, J., AND DEGENER, J. ANSI C yacc grammar [online]. 1995. Available from: `http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`. Ref: page 44

[176] LEE, P., AND LEONE, M. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 1996), ACM Press, pp. 137–148. doi:10.1145/231379.231407. Ref: page 48

[177] LEMON, T., AND CHESHIRE, S. Encoding Long Options in the Dynamic Host Configuration Protocol (DHCPv4). RFC 3396 (Proposed Standard), Nov. 2002. Available from: `http://www.ietf.org/rfc/rfc3396.txt`. Ref: page 36

[178] LEROY, X. The Zinc experiment: An economical implementation of the ML language. 117, INRIA, 1990. Ref: page 49

[179] LEROY, X. Formal certification of a compiler back-end, or programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)* (January 2006), p. to appear. Ref: page 48, 57, 58

[180] LEROY, X. OCaml-Call/CC: Continuations for OCaml [online]. 2006. Available from: `http://pauillac.inria.fr/~xleroy/software.html`. Ref: page 49

[181] LEROY, X., DOLIGEZ, D., GARRIGUE, J., RÉMY, D., AND VOUILLON, J. The Objective Caml system [online]. 2005. Available from: `http://caml.inria.fr/`. Ref: page 26, 27

[182] LHEE, K.-S., AND CHAPIN, S. J. Buffer overflow and format string overflow vulnerabilities. *Software—Practice and Experience 33*, 5 (2003), 423–460. Ref: page 19

[183] LITAN, A. Increased phishing and online attacks cause dip in consumer confidence. Tech. Rep. G00129146, Gartner, 2005. Available from: `http://gartner11.gartnerweb.com/DisplayDocument?doc_cd=129146`. Ref: page 7

[184] MADHAVAPEDDY, A., MYCROFT, A., SCOTT, D., AND SHARP, R. The case for abstracting security policies. In *The 2003 International Conference on Security and Management (SAM)* (June 2003). Ref: page 19, 43

[185] MALLORY, T., AND KULLBERG, A. Incremental updating of the Internet checksum. RFC 1141 (Informational), Jan. 1990. Updated by RFC 1624. Available from: `http://www.ietf.org/rfc/rfc1141.txt`. Ref: page 69

[186] MANNA, Z., AND PNUELI, A. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, New York, NY, USA, 1992. Ref: page 39

[187] MARLOW, S. Developing a high-performance web server in Concurrent Haskell. *Journal of Functional Programming 12*, 4+5 (July 2002), 359–374. Available from: `http://www.haskell.org/~simonmar/papers/web-server-jfp.pdf`. Ref: page 49

[188] MATSUMOTO, Y. The Ruby language [online]. 2006. Available from: `http://www.ruby-lang.org/`. Ref: page 26

[189] MCCANN, P. J., AND CHANDRA, S. Packet types: Abstract specification of network protocol messages. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2000), ACM Press, pp. 321–333. doi:10.1145/347059.347563. Ref: page 45, 69

[190] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter Technical Conference* (1993), USENIX, pp. 259–270. Ref: page 52

[191] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM 3*, 4 (1960), 184–195. doi:10.1145/367177.367199. Ref: page 20

[192] MCCARTHY, J. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems* (1963), 33–70. Ref: page 20

[193] MCCARTHY, J. History of LISP. In *The 1st ACM SIGPLAN Conference on History of Programming Languages* (New York, NY, USA, 1978), ACM Press, pp. 217–223. doi:10.1145/800025.808387. Ref: page 20

[194] MCKUSICK, M. K., AND NEVILLE-NEIL, G. V. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional Computing Series, August 2004. Ref: page 52

[195] MENAGE, P. B. *Resource Control of Untrusted Code in an Open Programmable Network*. PhD thesis, University of Cambridge, June 2000. Ref: page 46

[196] MEYER, B. *Eiffel: The Language*, 2nd edition ed. Prentice Hall, 1992. Ref: page 23

[197] MEYER, B. *Object-Oriented Software Construction*, 2nd edition ed. Prentice Hall Professional Technical Reference, 1997. Ref: page 23

[198] MEYER, R. A., MARTIN, J. M., AND BAGRODIA, R. L. Slow memory: the rising cost of optimism. In *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation (PADS)* (Washington, DC, USA, 2000), IEEE Computer Society, pp. 45–52. Ref: page 51

[199] MICROSOFT CORP. Microsoft Windows [online]. 2006. Available from: `http://www.microsoft.com/windows/`. Ref: page 17

[200] MILLER, T. C., AND DE RAADT, T. strlcpy and strlcat - consistent, safe, string copy and concatenation. In *USENIX Annual Technical Conference, FREENIX Track* (Monterey, California, USA, 1999), USENIX, pp. 175–178. Available from: `http://www.usenix.org/events/usenix99/`. Ref: page 83

[201] MILLS, H. D. Software development. *IEEE Transactions on Software Engineering 2*, 4 (1976), 265–273. Ref: page 47

[202] MILLS, H. D., AND LINGER, R. C. Data structured programming: Program design without arrays and pointers. *IEEE Transactions on Software Engineering 12*, 2 (1986), 192–197. Ref: page 47

[203] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (1978), 348–375. Ref: page 21, 22

[204] MILNER, R. *A Calculus of Communicating Systems*. Springer-Verlag New York, Secaucus, NJ, USA, 1982. Ref: page 40

[205] MILNER, R. *Communicating and Mobile Systems: The Pi Calculus*. Springer Verlag, May 1999. doi:10.2277/0521658691. Ref: page 40

[206] MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. *The Definition of Standard ML - Revised*, 2 ed. MIT Press, May 1997. Ref: page 21, 22, 26

[207] MOCKAPETRIS, P. Domain names - concepts and facilities. RFC 1034 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343. Available from: `http://www.ietf.org/rfc/rfc1034.txt`. Ref: page 122, 123

[208] MOCKAPETRIS, P. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343. Available from: `http://www.ietf.org/rfc/rfc1035.txt`. Ref: page 69, 122, 123

[209] MOORE, D. DNS server survey [online]. 2004. Available from: http://mydns.bboy.net/survey/. Ref: page 9, 122

[210] MOORE, D., PAXSON, V., SAVAGE, S., SHANNON, C., FORD, S. S., AND WEAVER, N. The spread of the sapphire/slammer worm. Available Online, 2003. Available from: http://www.cs.berkeley.edu/~nweaver/sapphire/. Ref: page 8

[211] MORGAN, C. *Programming from Specifications*, 2 ed. Prentice Hall, June 1994. Ref: page 48

[212] MOZILLA.ORG. Mozilla web browser [online]. 2006. Available from: http://www.mozilla.org/. Ref: page 106

[213] MURATA, T. Petri Nets: Properties, analysis and applications. In *Proceedings of the IEEE* (Apr. 1989), vol. 77, pp. 541–580. doi:10.1109/5.24143. Ref: page 41

[214] NECULA, G. C. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Paris, France, 1997), ACM Press, pp. 106–119. doi:10.1145/263699.263712. Ref: page 23, 42

[215] NECULA, G. C., MCPEAK, S., AND WEIMER, W. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Portland, Oregon, 2002), ACM Press, pp. 128–139. doi:10.1145/503272.503286. Ref: page 42

[216] NIELSON, F., AND NIELSON, H. R. Type and effect systems. In *Correct System Design* (1999), pp. 114–136. Available from: http://www.cs.ucla.edu/~palsberg/tba/papers/nielson-nielson-csd99.pdf. Ref: page 22

[217] ODLYZKO, A. M. Internet traffic growth: sources and implications. In *Optical Transmission Systems and Equipment for WDM Networking II* (August 2003), vol. 5247, International Society for Optical Engineering, pp. 1–15. Ref: page 7

[218] OKASAKI, C. *Purely Functional Data Structures.* Cambridge University Press, 1999. doi:10.1017/S0956796899009995. Ref: page 21, 24, 124

[219] O'MALLEY, S., PROEBSTING, T., AND MONTZ, A. B. Usc: a universal stub compiler. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM)* (New York, NY, USA, 1994), ACM Press, pp. 295–306. doi:10.1145/190314.190341. Ref: page 45

[220] PAXSON, V., Ed. *Proceedings of the 12th USENIX Security Symposium* (August 2003), USENIX. Ref: page 150

[221] PELAEZ, R. S. Linux kernel rootkits: Protecting the system's ring-zero. Giac unix security administrator (gcux), SANS Institute, 2004. Available from: http://www.sans.org/rr/whitepapers/honors/1500.php. Ref: page 16

[222] PETRI, C. A. *Kommunikation mit Automaten.* PhD thesis, Fakultt Matematik und Physik, Technische Universitt Darmstadt, 1962. Ref: page 41

[223] PIERCE, B. C. *Types and Programming Languages*. The MIT Press, 2002. Available from: `http://www.cis.upenn.edu/~bcpierce/tapl/`. Ref: page 22

[224] POSTEL, J. Internet Protocol. RFC 791 (Standard), Sept. 1981. Updated by RFC 1349. Available from: `http://www.ietf.org/rfc/rfc791.txt`. Ref: page 13, 69

[225] POSTEL, J. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFC 3168. Available from: `http://www.ietf.org/rfc/rfc793.txt`. Ref: page 13, 15, 49

[226] POSTEL, J., AND REYNOLDS, J. File Transfer Protocol. RFC 959 (Standard), Oct. 1985. Updated by RFCs 2228, 2640, 2773. Available from: `http://www.ietf.org/rfc/rfc959.txt`. Ref: page 69

[227] POTTIER, F., AND SIMONET, V. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS) 25*, 1 (2003), 117–158. doi:10.1145/596980.596983. Ref: page 61

[228] POULSON, K. Slammer worm crashed Ohio nuke plant net [online]. August 2003. Available from: `http://www.theregister.co.uk/2003/08/20/slammer_worm_crashed_ohio_nuke/`. Ref: page 8

[229] PROVOS, N. Improving host security with system call policies. In Paxson [220], pp. 257–272. Ref: page 10, 19, 43, 60

[230] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In Paxson [220], pp. 231–242. Ref: page 58, 101, 111

[231] PROVOS, N., AND HONEYMAN, P. Scanssh: Scanning the internet for ssh servers. In *Proceedings of the 15th Conference on Systems Administration (LISA)* (San Diego, California, USA, December 2001), USENIX, pp. 25–30. Ref: page 9, 110

[232] QIN, F., LU, S., AND ZHOU, Y. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 291–302. doi:10.1109/HPCA.2005.29. Ref: page 11

[233] RATZER, A. V., WELLS, L., LASSEN, H. M., LAURSEN, M., QVORTRUP, J. F., STISSING, M. S., WESTERGAARD, M., CHRISTENSEN, S., AND JENSEN, K. Cpn tools for editing, simulating, and analysing coloured petri nets. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets* (2003), vol. 2679, Springer-Verlag, pp. 450–462. Ref: page 41

[234] RAVENBROOK. The memory management reference [online]. Available from: `http://www.memorymanagement.org/`. Ref: page 22

[235] REISIG, W. Deterministic buffer synchronization of sequential processes. *Acta Informatica 18*, 2 (July 1982), 117–134. Ref: page 41

[236] REKHTER, Y., AND LI, T. A Border Gateway Protocol 4 (BGP-4). RFC 1771 (Draft Standard), Mar. 1995. Obsoleted by RFC 4271. Available from: `http://www.ietf.org/rfc/rfc1771.txt`. Ref: page 69

[237] REYNOLDS, J., AND POSTEL, J. Request For Comments reference guide. RFC 1000, Aug. 1987. Available from: `http://www.ietf.org/rfc/rfc1000.txt`. Ref: page 69

[238] SAIDI, H., AND SHANKAR, N. Abstract and model check while you prove. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)* (London, UK, 1999), Springer-Verlag, pp. 443–454. Ref: page 42

[239] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2*, 4 (1984), 277–288. doi:10.1145/357401.357402. Ref: page 8, 69, 87

[240] SCHECHTER, S. E., JUNG, J., STOCKWELL, W., AND MCLAIN, C. Inoculating ssh against address-harvesting. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS)* (San Diego, California, USA, February 2006), Internet Society. Ref: page 111

[241] SCHLYTER, J., AND GRIFFIN, W. Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints. RFC 4255 (Proposed Standard), Jan. 2006. Available from: `http://www.ietf.org/rfc/rfc4255.txt`. Ref: page 111

[242] SCHNEIDER, F. B. Enforceable security policies. *ACM Transactions on Information Systems Security 3*, 1 (2000), 30–50. doi:10.1145/353323.353382. Ref: page 42, 96

[243] SCHUBA, C. L., KRSUL, I. V., KUHN, M. G., SPAFFORD, E. H., SUNDARAM, A., AND ZAMBONI, D. Analysis of a denial of service attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy (SP)* (Washington, DC, USA, 1997), IEEE Computer Society, p. 208. Ref: page 60

[244] SCHWARZ, B., CHEN, H., WAGNER, D., LIN, J., TU, W., MORRISON, G., AND WEST, J. Model checking an entire Linux distribution for security violations. In *Proceedings of 21st Annual Computer Security Applications Conference (ACSAC)* (2005), IEEE Computer Society, pp. 13–22. Available from: `http://www.cs.berkeley.edu/~daw/papers/mops-full.pdf`. Ref: page 37

[245] SCOTT, D., AND SHARP, R. Abstracting application-level web security. In *Proceedings of the 11th International Conference on World Wide Web* (New York, NY, USA, 2002), ACM Press, pp. 396–407. doi:10.1145/511446.511498. Ref: page 8, 17, 60

[246] SCOTT, D. J. *Abstracting Application-Level Security Policy for Ubiquitous Computing*. PhD thesis, University of Cambridge, 2005. Ref: page 118

[247] SDL. SDL forum society. Tech. Rep. Recommendation Z.100, International Telecommunications Union, Geneva, 1993. Available from: `http://www.sdl-forum.org/`. Ref: page 88

[248] SEKAR, R., VENKATAKRISHNAN, V., BASU, S., BHATKAR, S., AND DUVARNEY, D. C. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the Nineteenth ACM symposium on Operating Systems Principles* (New York, NY, USA, 2003), ACM Press, pp. 15–28. doi:10.1145/945445.945448. Ref: page 43

[249] SENDMAIL CONSORTIUM. Sendmail [online]. 2006. Available from: `http://www.sendmail.org/`. Ref: page 10

[250] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)* (New York, NY, USA, 2004), ACM Press, pp. 298–307. doi:10.1145/1030083.1030124. Ref: page 19

[251] SHINWELL, M., BILLINGS, J., SEWELL, P., AND STRNISA, R. HashCaml: type-safe marshalling for O'Caml [online]. 2006. Available from: `http://www.cl.cam.ac.uk/~mrs30/talks/hashcaml.pdf`. Ref: page 50

[252] SIPSER, M. *Introduction to the Theory of Computation*, 1 ed. PWS Publishing, 1997. Ref: page 44, 70

[253] SKYPE TECHNOLOGIES S.A. Skype telephony software [online]. 2005. Available from: `http://www.skype.com`. Ref: page 8, 19

[254] SMITH, G., AND VOLPANO, D. Polymorphic typing of variables and references. *ACM Transactions on Programming Languages and Systems 18*, 3 (May 1996), 254–267. Ref: page 30

[255] SPAFFORD, E. H. The Internet worm program: an analysis. *SIGCOMM Computer Communications Review 19*, 1 (1989), 17–57. doi:10.1145/66093.66095. Ref: page 17

[256] STANIFORD, S., PAXSON, V., AND WEAVER, N. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium* (August 2002), D. Boneh, Ed., USENIX, pp. 149–167. Available from: `http://www.usenix.org/publications/library/proceedings/sec02/staniford.html`. Ref: page 17

[257] STEVENS, W. R., FENNER, B., AND RUDOFF, A. M. *Unix Network Programming: The Sockets Network API*, 1 ed. Addison Wesley, December 2003. Ref: page 43, 76

[258] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (New York, NY, USA, 2004), ACM Press, pp. 85–96. doi:10.1145/1024393.1024404. Ref: page 11

[259] SUN MICROSYSTEMS. Security vulnerability in ping [online]. November 2004. Available from: `http://sunsolve.sun.com/search/document.do?assetkey=1-26-57675-1`. Ref: page 81, 159

[260] TAHA, W. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation* (Dagstuhl Castle, Germany, March 2004), vol. 3016 of *Lecture Notes in Computer Science*, Springer, pp. 30–50. Ref: page 48

[261] TAKANEN, A., LAAKSO, M., ERONEN, J., AND RÖNING, J. Running malicious code by exploiting buffer overflows: A survey of publicly available exploits. In *Proceedings of the 1st European Anti-Malware Conference (EICAR)* (March 2000). Available from: `http://www.ee.oulu.fi/research/ouspg/protos/sota/EICAR2000-overflow-survey`. Ref: page 17

[262] THE OPENBSD PROJECT. OpenSSH [online]. Available from: `http://www.openssh.com/`. Ref: page 9, 110

[263] THE OPENBSD PROJECT. Systems using OpenSSH [online]. 2005. Available from: `http://www.openssh.com/users.html`. Ref: page 9

[264] THE OPENSSL PROJECT. Openssl: The open source toolkit for ssl/tls [online]. Available from: `http://www.openssl.org/`. Ref: page 113

[265] THOMPSON, K. Reflections on trusting trust. *Communications of the ACM 27*, 8 (1984), 761–763. doi:10.1145/358198.358210. Ref: page 61

[266] TREND MICRO. Vulnerability exploits break records. Tech. rep., Trend Micro, January 2004. Available from: `http://www.trendmicro.com/en/security/white-papers/overview.htm`. Ref: page 7

[267] TURING, A. M. On computable numbers with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society* (1936), no. 42 in 2, pp. 230–265. Ref: page 32, 40

[268] TURING, A. M. Computability and λ-definability. *Journal of Symbolic Logic*, 2 (1937), 153–163. Ref: page 20

[269] TURNER, D. A. The semantic elegance of applicative languages. In *Conference on Functional Programming Languages and Computer Architecture* (New York, NY, USA, 1981), ACM Press, pp. 85–92. Ref: page 21

[270] TURNER, D. A. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture* (New York, NY, USA, 1985), Springer-Verlag New York, pp. 1–16. Ref: page 21

[271] VAN RENESSE, R., BIRMAN, K., HAYDEN, M., VAYSBURD, A., AND KARR, D. Building adaptive systems using Ensemble. *Software—Practice and Experience 28*, 9 (1998), 963–979. Ref: page 10, 49

[272] VAN ROSSUM, G. The Python programming language [online]. Available from: `http://www.python.org`. Ref: page 20, 26

[273] VANINWEGEN, M. *The Machine-Assisted Proof Of Programming Language Properties*. PhD thesis, University of Pennsylvania, 1996. Ref: page 26

[274] VENKATRAMAN, B. R., AND NEWMAN-WOLFE, R. E. Capacity estimation and auditability of network covert channels. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy (SP)* (Washington, DC, USA, May 1995), IEEE Computer Society, pp. 186–198. Ref: page 60

[275] VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)* (Washington, DC, USA, 2000), IEEE Computer Society, p. 3. Ref: page 37

[276] VIXIE, P. Extension Mechanisms for DNS (EDNS0). RFC 2671 (Proposed Standard), Aug. 1999. Available from: `http://www.ietf.org/rfc/rfc2671.txt`. Ref: page 122

[277] WADLER, P. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming (ESOP)* (Amsterdam, The Netherlands, The Netherlands, 1988), North-Holland Publishing Co., pp. 231–248. doi:10.1016/0304-3975(90)90147-A. Ref: page 52

[278] WADSWORTH, C. P. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971. Ref: page 24

[279] WAGNER, D., AND SOTO, P. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)* (August 2002), V. Atluri, Ed., ACM, pp. 255–264. Ref: page 19, 43

[280] WAKEMAN, I., JEFFREY, A., OWEN, T., AND PEPPER, D. Safetynet: a language-based approach to programmable networks. *Computer Networks 36*, 1 (2001), 101–114. doi:10.1016/S1389-1286(01)00154-2. Ref: page 46

[281] WEAVER, N., PAXSON, V., STANIFORD, S., AND CUNNINGHAM, R. A taxonomy of computer worms. In *Proceedings of the 2003 ACM workshop on Rapid Malcode (WORM)* (New York, NY, USA, 2003), ACM Press, pp. 11–18. doi:10.1145/948187.948190. Ref: page 17

[282] WEAVER, N., STANIFORD, S., AND PAXSON, V. Very fast containment of scanning worms. In *Proceedings of the 13th USENIX Security Symposium* (August 2004), M. Blaze, Ed., USENIX, pp. 29–44. Ref: page 19

[283] WICKLINE, P., LEE, P., AND PFENNING, F. Run-time code generation and modal-ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 1998), ACM Press, pp. 224–235. doi:10.1145/277650.277727. Ref: page 48

[284] WIGER, U., ASK, G., AND BOORTZ, K. World-class product certification using Erlang. In *Proceedings of the ACM SIGPLAN workshop on Erlang* (New York, NY, USA, 2002), ACM Press, pp. 24–33. doi:10.1145/592849.592853. Ref: page 22

[285] WIRTH, N. Program development by stepwise refinement. *Communications of the ACM 14*, 4 (April 1971), 221–227. Available from: `http://www.acm.org/classics/dec95/`. Ref: page 47

[286] WOO, T. Y., BINDIGNAVLE, R., SU, S., AND LAM, S. S. SNP: An interface for secure network programming. In *Proceedings of the USENIX Summer Technical Conference* (August 1994), USENIX. Available from: http://www.usenix.org/publications/library/proceedings/bos94/woo.html. Ref: page 8

[287] XI, H. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, September 1998. Available from: http://www.cs.bu.edu/~hwxi/academic/papers/thesis.2.ps. Ref: page 22, 60

[288] XI, H., AND PFENNING, F. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 1998), ACM Press, pp. 249–257. doi:10.1145/277650.277732. Ref: page 60

[289] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004), pp. 273—288. Ref: page 37

[290] YEONG, W., HOWES, T., AND KILLE, S. X.500 Lightweight Directory Access Protocol. RFC 1487 (Historic), July 1993. Obsoleted by RFCs 1777, 3494. Available from: http://www.ietf.org/rfc/rfc1487.txt. Ref: page 44

[291] YLONEN, T., AND LONVICK, C. The Secure Shell (SSH) Authentication Protocol. RFC 4252 (Proposed Standard), Jan. 2006. Available from: http://www.ietf.org/rfc/rfc4252.txt. Ref: page 111, 120

[292] YLONEN, T., AND LONVICK, C. The Secure Shell (SSH) Connection Protocol. RFC 4254 (Proposed Standard), Jan. 2006. Available from: http://www.ietf.org/rfc/rfc4254.txt. Ref: page 111, 120

[293] YLONEN, T., AND LONVICK, C. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006. Available from: http://www.ietf.org/rfc/rfc4251.txt. Ref: page 69, 111

[294] YLONEN, T., AND LONVICK, C. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), Jan. 2006. Available from: http://www.ietf.org/rfc/rfc4253.txt. Ref: page 111

[295] ZAKAS, N. C., MCPEAK, J., AND FAWCETT, J. *Professional Ajax*, 1 ed. Wrox, February 2006. Ref: page 106

[296] ZEILENGA, K. Lightweight Directory Access Protocol version 2 (LDAPv2) to Historic Status. RFC 3494 (Informational), Mar. 2003. Available from: http://www.ietf.org/rfc/rfc3494.txt. Ref: page 44

APPENDIX A

Sample Application: ping

This Appendix describes how all the tools described in this dissertation can fit together to construct a simple network `ping` client. The client includes an MPL specification to parse IPv4 and ICMP headers, and an simple SPL specification to enforce the ping automaton.

We first define the network format of the ping packets. Our application will read using the raw socket interface (as the usual C implementations also do), which provides a packet consisting of the IPv4 header, the ICMP header, and then the ping payload. The MPL definitions are listed later, for IPv4 (§C.2) and ICMP (§C.3).

We then define a simple automaton for the application which gives it three main modes of operation: (*i*) initialising; (*ii*) transmitting a packet; or (*iii*) waiting for a packet or a timeout. We also define a separate "never" automaton which defines the packets which should never be received by the client (at least until support is added for them in the main code-base). The state-machine graphical rendering (using Graphviz [161] and the SPL compiler DOT output) can be seen in Figure A.1.

```
SPL
automaton ping () {
    Init;
    multiple {
        Transmit_Icmp_EchoRequest;
        either {
            Timeout;
        } or {
            Receive_Icmp_EchoReply;
        }
    }
}
automaton never () {
    either { Receive_Icmp_DestinationUnreachable; }
    or { Receive_Icmp_EchoRequest; }
    or { Receive_Icmp_Redirect; }
    or { Receive_Icmp_RouterAdvertisement; }
    or { Receive_Icmp_RouterSolicitation; }
    or { Receive_Icmp_SourceQuench; }
    or { Receive_Icmp_TimeExceeded; }
    or { Receive_Icmp_TimestampRequest; }
}
```

Next, we define some utility functions to handle initialising and driving the automaton based on network activity:

```
OCAML
let auto = ref (Automaton.init ())
let tick s = auto := Automaton.tick !auto s
let icmp_xmit_and_tick o s addr =
    tick (o#xmit_statecall :> Statecalls.t);
    Mpl_stdlib.sendto o#env s addr
let icmp_recv_and_tick env =
    let o = Icmp.unmarshal env in
    tick (Icmp.recv_statecall o);
    o
let timeout_read fd timeout fn =
    match Unix.select [fd] [] [] timeout with
    |[s],_,_ → fn ()
    |_ → tick ‘Timeout
```

The `tick` function accepts a statecall and updates the global automaton with a tick, raising an exception if the transition is invalid, or just continuing otherwise. The next two functions `icmp_xmit_and_tick` and `icmp_recv_and_tick` ensure that all network traffic via the MPL interface also ticks the automaton. The final function `timeout_read` waits for data to be available, and ticks the automaton with a timeout message if none is, and otherwise calls the function argument. This group of functions represents the entire MPL/SPL “bridge” interface, which makes sure that all network traffic drives the embedded automaton. Even if the application is extended later to handle more packet types, this bridge does not need to be modified since the auto-generated code from the MPL and SPL compilers is compatible and uses extensible polymorphic variants.

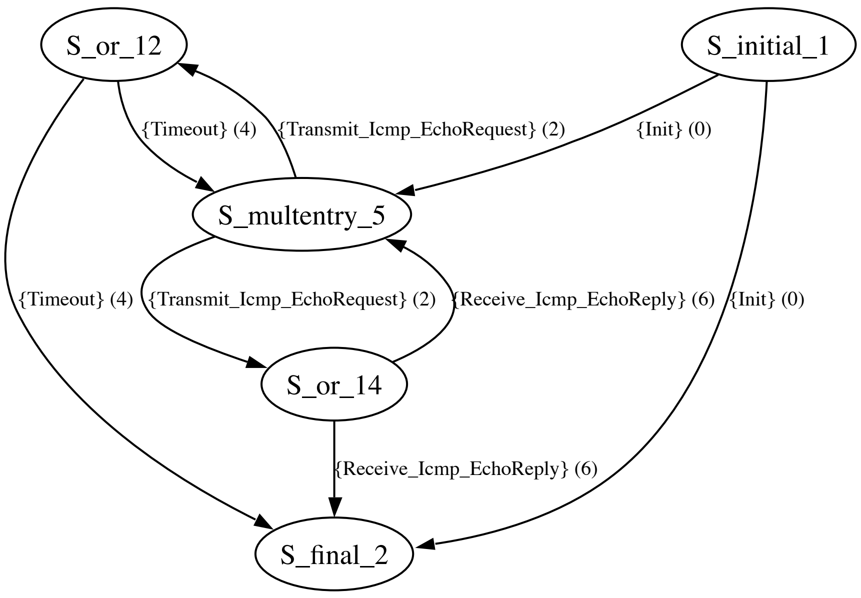

S_or_12
S_initial_1
{Timeout} (4)
{Transmit_Icmp_EchoRequest} (2)
{Init} (0)
S_multentry_5
{Timeout} (4)
{Transmit_Icmp_EchoRequest} (2)
{Receive_Icmp_EchoReply} (6)
{Init} (0)
S_or_14
{Receive_Icmp_EchoReply} (6)
S_final_2

Figure A.1: Automaton for the sample ping application

```
let sequence = ref 0                                          OCAML
let create_ping env =
    Mpl_stdlib.reset env;
    incr sequence;
    let icdata = Icmp.EchoRequest.t
        code:0  checksum:0  identifier:345  sequence:!sequence
        data:(`Str "foobar") env in
    let csum,_ = Ip_utils.icmp_checksum icdata#env in
    icdata#set_checksum csum;
    icdata
```

We also define the `create_ping` function to construct ICMP echo request packets. The above fragment is obviously simplified—in reality, the function would also parameterise the packet contents. Finally, we define the main body of code to handle sending pings:

```
let s = socket PF_INET SOCK_RAW 1 in (* 1 is IPPROTO_ICMP *)                OCAML
let addr = ADDR_INET (inet_addr_of_string !ip, 0) in
let senv = Mpl_stdlib.new_env (String.create 4000) in
let renv = Mpl_stdlib.new_env (String.create 4000) in
tick ‘Init;
while true do
    icmp_xmit_and_tick (create_ping senv) s addr;
    timeout_read s 4. (fun () →
    let faddr = Mpl_stdlib.recvfrom renv s [] in
    let ip = Ipv4.unmarshal renv in
        Ip_utils.data_env ip (fun env →
        match ip#protocol with
        |‘ICMP → begin
            match icmp_recv_and_tick env with
            |‘EchoReply icmp →
                icmp#prettyprint;
            |_ → raise Unknown_ICMP_Packet
        end
        |_ → raise Unknown_IP_Packet
        );
    );
    sleep 1;
done
```

The program first opens a raw socket, creates MPL environments, and invokes the *Init* statecall. Recall from the SPL automaton that once the *Init* statecall is called, it cannot happen again for the lifetime of the automaton. Although not of prime importance in this fragment, it becomes important when dealing with privilege dropping[1].

Finally, the ping enters an infinite loop, where it constructs a ping packet and transmits it, waits for a reply, and pattern-matches the response to determine if it is an Echo Reply packet, using the standard OCaml construct for this purpose.

This is all that is needed to create a simple `ping` client. In a real implementation, of course, there would be more code for argument parsing for the command-line and better error handling, but these are left as an exercise to the reader.

[1]`ping` is traditionally run with root privileges to be able to open raw sockets, and should drop those privileges after initialisation since they are no longer required [259].

APPENDIX B

MPL User Manual

This Appendix describes the MPL language and compiler in more detail. The MPL compiler internally represents the specification using the Abstract Syntax Tree (AST) shown in Table B.1. In our notation, $(a \times b)$ represents a tuple, and $ident$ represents a capitalized string (e.g. Foo).

B.1 Well-Formed Specifications

Although MPL is a domain-specific language which prohibits constructs such as user-defined function calls or recursion, it is still possible to specify malformed packets (i.e. impossible conversions or incomplete information) by using the syntax presented above. The MPL compiler performs static checks to ensure that: (*i*) types and expressions used in a packet are self-consistent (i.e. a string field is not classified as an integer); (*ii*) names are unique at all scopes to avoid aliasing issues; and (*iii*) enough information has been specified to perform both marshalling and unmarshalling of packets precisely. The checks for a well-formed specification are done in two phases; the first checks that all variables, statements and attributes except for **value** (which has different scoping rules described in §B.2). The second phase performs a global scope check to ensure that **value** expressions can be resolved correctly during packet creation.

An MPL specification consists of one or more MPL packets, and each packet has an optional list of state variables. State variables of are of type `int` (32-bit) or `bool`, and can *only* be used as guards in **classify** clauses. Names of state variables are distinct from the other variable bindings discussed below.

A packet consists of a sequence of named fields, each of which have a single *wire type*. The wire type represents the network representation of a value, and maps to one of the following MPL types: `bool`, `int`, `string`, and `opaque`. Some wire types are built into MPL and others can be added on a per-protocol basis, as shown in Table B.2. Mapping to an MPL type allows the contents of that variable to be used in MPL expressions such as length expressions or classification pattern matches. The `opaque` type represents abstract data (such as multiple-precision integers in SSH) that cannot be further manipulated in an MPL spec. As a special exception, a variable can have the wire type `label`, which binds it as a position marker in the packet at that point but does not modify the parsing state.

Variables of type `bit` must have a constant integer length also specified to represent the number of bits. Successive `bit` variables form a bit-field of any length, but it *must* be aligned

n	$\leftarrow$	string		$\mathit{variable\ name}$
e	$\leftarrow$	$n \mid \mathbf{and}(e_1, e_2) \mid \mathbf{or}(e_1, e_2) \mid \mathbf{not}(e)$		$\mathit{expression}$
	$\mid$	$\mathbf{true} \mid \mathbf{false} \mid \mathit{integer} \mid \mathit{string}$		
	$\mid$	$(e_1 > e_2) \mid (e_1 \geq e_2) \mid (e_1 < e_2) \mid (e_1 \leq e_2)$		
	$\mid$	$(e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 \times e_2) \mid (e_1 \div e_2)$		
	$\mid$	$(e_1 = e_2) \mid (e_1 .. e_2) \mid \mathit{builtin\_function}(n, e)$		
v	$\leftarrow$	$n \times \mathit{ident}$		$\mathit{variant\ type\ mapping}$
t	$\leftarrow$	$\mathit{byte} \mid \mathit{uint}16 \mid \mathit{uint}32 \mid \mathit{uint}64 \mid \ldots$		$\mathit{normal\ variable\ type}$
	$\mid$	$(\mathit{bit} \mid \mathit{byte} \mid \ldots) \times e$		$\mathit{array\ variable\ type}$
a	$\leftarrow$	$\mathbf{value}(e)$		$\mathit{value\ attribute}$
	$\mid$	$\mathbf{const}(e)$		$\mathit{constant\ attribute}$
	$\mid$	$\mathbf{align}(e)$		$\mathit{alignment\ attribute}$
	$\mid$	$\mathbf{min}(e)$		$\mathit{minimum\ attribute}$
	$\mid$	$\mathbf{max}(e)$		$\mathit{maximum\ attribute}$
	$\mid$	$\mathbf{default}(e)$		$\mathit{default\ attribute}$
	$\mid$	$\mathbf{variant}(v_1 \ldots v_l)$	$(l > 0)$	$\mathit{variant\ attribute}$
g	$\leftarrow$	$e \times n \times (f_1 \ldots f_n)$	$(n > 0)$	$\mathit{guard\ expression}$
f	$\leftarrow$	$\mathbf{var}(n \times t \times (a_0 \ldots a_k))$	$(k \geq 0)$	$\mathit{variable\ binding}$
	$\mid$	$\mathbf{classify}(n \times (g_1 \ldots g_j))$	$(j > 0)$	$\mathit{variable\ classification}$
	$\mid$	$\mathbf{array}(n \times (f_1 \ldots f_n))$	$(n > 0)$	$\mathit{array\ declaration}$
	$\mid$	$\mathbf{label}(n)$	$(n > 0)$	$\mathit{label\ marker}$
s	$\leftarrow$	$\mathit{int} \mid \mathit{bool}$		$\mathit{state\ variables}$
p	$\leftarrow$	$(s_1 \ldots s_i) \times (f_1 \ldots f_n)$	$(i \geq 0, n > 0)$	packet

Table B.1: Abstract Syntax Tree used to represent MPL specifications

`bit[x]`	built-in	int(x)
`byte[x]`	built-in	opaque
`byte`	built-in	int(8)
`uint16`	built-in	int(16)
`uint32`	built-in	int(32)
`uint64`	built-in	int(64)
`string8`	custom (DNS)	string
`string32`	custom (SSH)	string
`mpint`	custom (SSH)	opaque
`boolean`	custom (SSH)	bool

Table B.2: Mapping of wire types to MPL types

Table B.3: Primitive type rules for MPL expressions

`true`	bool	(unary) $-$	int $\rightarrow$ int
`false`	bool	(unary) $+$	int $\rightarrow$ int
string	string		
number	int	`and`	bool $\times$ bool $\rightarrow$ bool
		`or`	bool $\times$ bool $\rightarrow$ bool
`const`	$\forall\alpha.\alpha \times \alpha \rightarrow$ unit	`not`	bool $\rightarrow$ bool
`value`	$\forall\alpha.\alpha \times \alpha \rightarrow$ unit		
`default`	$\forall\alpha.\alpha \times \alpha \rightarrow$ unit	..	int $\times$ int $\rightarrow$ int $\times$ int
`align`	int $\times$ int $\rightarrow$ unit	$=$	int $\times$ int $\rightarrow$ bool
`min`	int $\times$ int $\rightarrow$ unit	$>$	int $\times$ int $\rightarrow$ bool
`max`	int $\times$ int $\rightarrow$ unit	$<$	int $\times$ int $\rightarrow$ bool
`variant`	$\forall\alpha.\alpha \times [\alpha] \rightarrow$unit	$>=$	int $\times$ int $\rightarrow$ bool
		$<=$	int $\times$ int $\rightarrow$ bool
`offset`	*variable* $\rightarrow$ int		
`sizeof`	*variable* $\rightarrow$ int	$+$	int $\times$ int $\rightarrow$ int
`array_length`	*variable* $\rightarrow$ int	$-$	int $\times$ int $\rightarrow$ int
`remaining`	*unit* $\rightarrow$ int	$*$	int $\times$ int $\rightarrow$ int
		$/$	int $\times$ int $\rightarrow$ int

to 8-bits when the bit-field ends (by a non-`bit` variable or the end of the packet). Bit-fields are allowed to contain **classify** clauses, but the bit-alignment at the end of each classification branch must be equal (note that bit-alignment is modulo-8 and so one classification branch could include many more bits, as long as the final alignment is the same as other branches). Variables of type `byte` can optionally have an integer length to convert them into a byte array. This length can be constant or an expression consisting of earlier `int` variables.

Every variable can be tagged with an optional list of *attributes*. The precise meanings of the attributes are discussed later (§B.2), but first we define how they can be well-formed. An array variable (e.g. `byte`[x]) can only have the **align** attribute, which accepts an integer argument which is a multiple of 8. For each normal variable with some type α, each the following attributes can optionally be specified: (*i*) **const**, **value** or **default** which contain an expression of type α; (*ii*) **min** or **max** which accept `int` arguments and are only allowed when ($\alpha \leftarrow$ `int`); and (*iii*) **variant** which contains a list of constant expressions of type α. Any `int` types also have a precision from 1–64 bits, and any constants specified in relation to that variable are range-checked appropriately.

Expressions can also include built-in functions (see bottom-left of Table B.3) from the following: (*i*) `remaining` takes no arguments and returns an `int`; (*ii*) `offset` and `sizeof` accepts a single variable name argument and returns an `int`; and (*iii*) `array_length` accepts a single variable name argument which is bound to an **array** variable (see below) and returns an `int`. `remaining` can only be used in the length specifier to arrays, and the other functions only in attribute expressions.

In addition to variable declarations, statements can also specify: (*i*) a **classify** clause; (*ii*) an **array**; (*iii*) a **packet**; or (*iv*) **unit** (representing no action). The **classify** clause accepts one argument, a variable of type ($\forall\alpha \mid \alpha \neq$ `opaque`), and a pattern match consisting of a list of named constant α expressions with an optional `bool` guard expression (guard expressions can only contain state variables). If the **classify** has type `int`, then the range operator (represented by "..") can also be used. All pattern match names *must* be distinct in a given **classify** clause. An **array** is named and accepts an `int` expression and a sub-block of statements. A **packet** represents an external MPL packet (which must be found in the path of the MPL compiler) and must specify a list of arguments of equivalent types to any state variables required by that packet.

Expressions are forbidden from referring to variables inside a **classify** clause[1]. Variable scoping in expressions is handled differently for **value** attributes from other expressions. Normal expressions can refer to previously declared variables, and **value** expressions (only used when creating new packets) can refer to any variables in the specification.

B.2 Semantics

We now describe the meaning of every element in an MPL specification. A specification consists of a list of statements (represented by f in the AST in Table B.1):

Variable Binding: A variable name n is bound to type t, which can either be a built-in or custom type (see Table B.2 and §5.2.3). The type can also have an optional size specifier, which indicates that it is a bit or $byte$ array of data. A variable can have the following attributes attached to it:

Value e**:** The expression e is always assigned as the value of the variable when a new packet is being created. The variable is no longer exported to the external code interface.

Const e_c**:** The constant expression e_c always represents the value of this variable when a new packet is being created, and may optionally be checked against received traffic. The variable is no longer exported to the external code interface.

Min/Max n**:** The integer n represents the minimum or maximum range of this variable when it is being created. The range can also be optionally enforced for received packets.

Default e_c**:** The constant expression e_c is offered as a convenient default value when a new packet is being created, but it can be overridden by the user if required.

Variant $v_1 \ldots v_n$**:** A list of mappings which convert expressions (of the type of the variable) into a string label. The labels are exposed in the external interface instead of the raw values themselves.

Align n**:** Ensures that created $byte$ arrays always end at the bit-boundary specified by n. A common value in Internet protocols is 32 to ensure alignment of data packets for efficiency reasons on 32-bit architectures. Added padding bytes always have the value `0`. This check can optionally be enabled for received traffic.

Classify $(n \times (g_1 \ldots g_j))$**:** The value of variable n (which must been previously bound with a variable binding) decides the structure of the packet that follows. Each pattern match

[1]This restriction is actually due to implementation limitations of our current compiler, and could be relaxed in the future if every path in a **classify** contained a variable of the same name and type.

g contains: (*i*) a constant expression e_c indicating the value to pattern match n against; (*ii*) a string label giving a name for this portion of the classified packet; (*iii*) an optional boolean guard expression which can use the values of the packet state variables to decide whether to pattern match or not; and (*iv*) a list of further statements $(f_1 \dots f_n)$ to evaluate upon a successful pattern match. Any statements which follow after the **classify** block are appended to each of the statement blocks $(f_1 \dots f_n)$ and evaluated after them.

Array $(e \times (f_1 \dots f_n))$**:** The integer expression e represents a fixed-length number of records. The records are represented by the statements $(f_1 \dots f_n)$.

Label: Labels are used to bind *markers* within a packet, and expose these markers to the external code interface. They are used (for example) to delimit the header and body portions of a packet, or to dynamically calculate the size of a classification block by placing labels before and after it.

Expressions have access to the following built-in functions (within the limits of our well-formedness rules in §B.1):

sizeof v**:** Returns the size in bytes of the variable v.

offset v**:** Returns the offset in bytes after the end of the variable v. The packet is assumed to start from offset 0.

remaining: Returns the number of bytes remaining in the packet currently being parsed. Can only be used in the size specifier to a $byte$ array.

array_length v**:** Returns the number of elements in an array variable bound with the **array** keyword. Is not meaningful with any other variable types.

APPENDIX C

MPL Protocol Listings

C.1 Ethernet

```
packet ethernet {                                                          MPL
        dest_mac: byte[6];
        src_mac: byte[6];
        length: uint16 value(offset(end_of_packet)-offset(length));
        classify (length) {
            |46..1500:"E802_2" →
                data: byte[length];
            |0x800:"IPv4" →
                data: byte[remaining()];
            |0x806:"Arp" →
                data: byte[remaining()];
            |0x86dd:"IPv6" →
                data: byte[remaining()];
        };
        end_of_packet: label;
    }
```

C.2 IPv4

```
packet ipv4 {                                                              MPL
        version: bit[4] const(4);
        ihl: bit[4] min(5) value(offset(options) / 4);
        tos_precedence: bit[3] variant {
            |0 ⇒ Routine |1 → Priority
            |2 → Immediate |3 → Flash
            |4 → Flash_override |5 → ECP
            |6 → Internetwork_control |7 → Network_control
        };
        tos_delay: bit[1] variant {|0 ⇒ Normal |1 → Low};
        tos_throughput: bit[1] variant {|0 ⇒ Normal |1 → Low};
        tos_reliability: bit[1] variant {|0 ⇒ Normal |1 → Low};
        tos_reserved: bit[2] const(0);
        length: uint16 value(offset(data));
        id: uint16;
        reserved: bit[1] const(0);
        dont_fragment: bit[1] default(0);
        can_fragment: bit[1] default(0);
        frag_offset: bit[13] default(0);
        ttl: byte;
        protocol: byte variant {|1→ICMP |2→IGMP |6→TCP |17→UDP};
        checksum: uint16 default(0);
        src: uint32;
        dest: uint32;
        options: byte[(ihl × 4) - offset(dest)] align(32);
        header_end: label;
        data: byte[length-(ihl×4)];
    }
```

C.3 ICMP

```
packet icmp {                                                    MPL
        ptype: byte;
        code: byte default(0);
        checksum: uint16 default(0);
        classify (ptype) {
        |0:“EchoReply” →
            identifier: uint16;
            sequence: uint16;
            data: byte[remaining()];
        |3:“DestinationUnreachable” →
            reserved: uint32 const(0);
            ip_header: byte[remaining()];
        |4:“SourceQuench” →
            reserved: uint32 const(0);
            ip_header: byte[remaining()];
        |5:“Redirect” →
            gateway_ip: uint32;
            ip_header: byte[remaining()];
        |8:“EchoRequest” →
            identifier: uint16;
            sequence: uint16;
            data: byte[remaining()];
        |9:“RouterAdvertisement” → ();
        |10:“RouterSolicitation” → ();
        |11:“TimeExceeded” →
            reserved: uint32 const(0);
            ip_header: byte[remaining()];
        |13:“TimestampRequest” →
            identifier: uint16;
            sequence: uint16;
            origin_timestamp: uint32;
            receive_timestamp: uint32;
            transmit_timestamp: uint32;
        };
    }
```

C.4 DNS

```
packet dns {                                                              MPL
        id: uint16;
        qr: bit[1] variant { |0 → Query |1 → Answer };
        opcode: bit[4] variant { |0 → Query |1 → IQuery |2 → Status
            |3 → Reserved |4 → Notify |5 → Update };
        authoritative: bit[1];
        truncation: bit[1];
        rd: bit[1];
        ra: bit[1];
        zv: bit[3] const(0);
        rcode: bit[4] variant {|0 ⇒ NoError |1 → FormErr
            |2 → ServFail |3 → NXDomain |4 → NotImp |5 → Refused
            |6 → YXDomain |7 → YXRRSet |8 → NXRRSet |9 → NotAuth
            |10 → NotZone |16 → BadVers |17 → BadKey |18 → BadTime
            |19 → BadMode |20 → BadName |21 → BadAlg};
        qdcount: uint16 value(array_length(questions));
        ancount: uint16 value(array_length(answers));
        nscount: uint16 value(array_length(authority));
        arcount: uint16 value(array_length(additional));
        questions: array (qdcount) {
            qname: dns_label;
            qtype: uint16 variant {|1 → A |2 → NS |3 → MD |4 → MF
                |5→ CNAME |6→ SOA |7 → MB |8 → MG |9 → MR
                |10 → NULL |11 → WKS |12 → PTR |13 → HINFO
                |14 → MINFO |15 → MX |16 → TXT |17 → RP
                |33 → SRV |38 → A6 |252 → AXFR |253 → MAILB
                |254 → MAILA |255 → ANY};
            qclass: uint16 variant {|1 ⇒ IN |2 → CSNET
                |3 → CHAOS |4 → HS |255 → ANY};
        };
        answers: array (ancount) {
            rr: packet dns_rr();
        };
        authority: array (nscount) {
            rr: packet dns_rr();
        };
        additional: array (arcount) {
            rr: packet dns_rr();
        };
    }
    packet dns_rr {                                                       MPL
        name: dns_label_comp;
        atype: uint16;
        aclass: uint16 variant {|1 ⇒ IN |2 → CSNET |3 → CHAOS |4 → HS };
        ttl: uint32;
        rdlength: uint16 value(offset(ans_end) - offset(ans_start));
        ans_start: label;
        classify (atype) {
        |1:"A" →
            ip: uint32;
        |2:"NS" →
            hostname: dns_label_comp;
        |3:"MD" →
            madname: dns_label;
        |5:"CNAME" →
            cname: dns_label;
        |6:"SOA" →
            primary_ns: dns_label_comp;
            admin_mb: dns_label_comp;
            serial: uint32;
            refresh: uint32;
            retry: uint32;
            expiration: uint32;
            minttl: uint32;
        |12:"PTR" →
            ptrdname: dns_label_comp;
        |15:"MX" →
            preference: uint16;
            hostname: dns_label_comp;
        |16:"TXT" →
            data: string8;
            misc: byte[rdlength - offset(data) + offset(ans_start)];
        |29:"LOC" →
            version: byte const(0);
            size: byte;
            horiz_pre: byte;
            vert_pre: byte;
            latitude: uint32;
            longitude: uint32;
            altitude: uint32;
        |_:"Unknown" →
            data: byte[rdlength];
        };
        ans_end: label;
    }
```

C.5 SSH

```
packet transport {                                                    MPL
        ptype: byte;
        classify (ptype) {
         |1:"Disconnect" → reason_code: uint32 variant {
                |1 → Host_not_allowed |2 → Protocol_error
                |3 → Kex_failed |4 → Reserved |5 → MAC_error
                |6 → Compression_error |7 → Service_not_available
                |8 → Protocol_unsupported |9 → Bad_host_key
                |10 → Connection_lost |11 → By_application
                |12 → Too_many_connections |13 → Auth_cancelled
                |14 → No_more_auth_methods |15 → Illegal_user_name };
            description: string32;
            language: string32;
        |2: "Ignore" → data: string32;
        |3: "Unimplemented" → seq_num: uint32;
        |4: "Debug" →
            always_display: boolean;
            message: string32;
            language: string32;
        |5: "ServiceReq" →
            stype: string32;
            classify (stype) {
             |"ssh-userauth": "UserAuth" → ();
             |"ssh-connection": "Connection" → ();
            };
        |6: "ServiceAccept" →
            stype: string32;
            classify (stype) {
             |"ssh-userauth": "UserAuth" → ();
             |"ssh-connection": "Connection" → ();
            };
        |20: "KexInit" →
            cookie: byte[16];
            kex_algorithms: string32;
            server_host_key_algorithms: string32;
            encryption_algorithms_client_to_server: string32;
            encryption_algorithms_server_to_client: string32;
            mac_algorithms_client_to_server: string32;
            mac_algorithms_server_to_client: string32;
            compression_algorithms_client_to_server: string32;
            compression_algorithms_server_to_client: string32;
            languages_client_to_server: string32;
            languages_server_to_client: string32;
            kex_packet_follows: boolean;
            reserved: uint32 const(0);
        |21: "NewKeys" → ();
        };
}
packet auth (bool passwd_ns) {                                        MPL
    ptype: byte;
    classify (ptype) {
        |50: "Req" →
            user_name: string32;
            service: string32;
            authtype: string32;
            classify (authtype) {
             |"none":"None" → ();
             |"publickey":"PublicKey" →
                bcheck: boolean;
                classify (bcheck) {
                 |false:"Check" →
                    algorithm: string32;
                    blob: string32;
                 |true:"Request" →
                    algorithm: string32;
                    publickey: string32;
                    signature: string32;
                };
             |"password":"Password" →
                bcheck: boolean;
                classify (bcheck) {
                 |false:"Request" →
                    password: string32;
                 |true:"Change" →
                    old_password: string32;
                    new_password: string32;
                };
            };
        |51:"Failure" →
            auth_continue: string32;
            partial_success: boolean;
        |52:"Success" → ();
        |53:"Banner" →
            banner: string32;
            language: string32;
        |60:"ChangeReq" when (passwd_ns) →
            prompt: string32;
            language: string32;
        |60:"PublicKey_OK" when (!passwd_ns) → ();
       };
}
```

APPENDIX D

SPL Specifications

D.1 SSH Transport and Authentication

SPL

```
automaton transport (bool encrypted, bool serv_auth)
  {
    during {
      always_allow (Transmit_Transport_Debug, Receive_Transport_Debug,
                    Transmit_Transport_Ignore, Receive_Transport_Ignore) {
        multiple {
          either {
            either {
              Transmit_Transport_KexInit;
              Receive_Transport_KexInit;
            } or (encrypted) {
              Receive_Transport_KexInit;
              Transmit_Transport_KexInit;
            }
            either {
              Expect_DHInit;
              Receive_Dhgroupsha1_Init;
              Transmit_Dhgroupsha1_Reply;
            } or {
              Expect_GexInit;
              Receive_Dhgexsha1_Request;
              Transmit_Dhgexsha1_Group;
              Receive_Dhgexsha1_Init;
              Transmit_Dhgexsha1_Reply;
            }
            Receive_Transport_NewKeys;
            Transmit_Transport_NewKeys;
            encrypted = true;
          } or (encrypted && !serv_auth) {
            Receive_Transport_ServiceReq_UserAuth;
```

```
                    Transmit_Transport_ServiceAccept_UserAuth;
                    serv_auth = true;
                }
            }
        }
    } handle {
        either {
            Signal_HUP;
        } or {
            either {
                Receive_Transport_Disconnect;
            } or {
                optional { Signal_QUIT; }
                Transmit_Transport_Disconnect;
            exit;
            }
        } or {
        Receive_Transport_Unimplemented;
        }
    }
}
automaton auth (bool success, bool failed)
{
    Transmit_Transport_ServiceAccept_UserAuth;
    during {
        do {
            always_allow (Transmit_Auth_Banner) {
                either {
                    Receive_Auth_Req_None;
                    Transmit_Auth_Failure;
                } or {
                    Receive_Auth_Req_Password_Request;
                    either {
                        Transmit_Auth_Success;
                        success = true;
                    } or {
                        Transmit_Auth_Failure;
                    }
                } or {
                    Receive_Auth_Req_PublicKey_Request;
                    either {
                        Transmit_Auth_Success;
                        success = true;
                    } or {
                        Transmit_Auth_Failure;
                    }
                } or {
                    Receive_Auth_Req_PublicKey_Check;
                    either {
                        Transmit_Auth_PublicKey_OK;
                    } or {
                        Transmit_Auth_Failure;
                    }
                } or {
                    Notify_Auth_Permanent_Failure;
                    failed = true;
                }
            }
        } until (success || failed);
    } handle {
        Transmit_Transport_Disconnect;
        exit;
    }
}
```

D.2 SSH Channels

SPL

```
// Automaton representing an interactive channel session
  automaton
  interactive (bool done_pty, bool done_exec, bool got_eof, bool sent_eof)
  {
      multiple {
          either (!done_pty) {
              Receive_Channel_Request_Pty;
              either {
                  Expect_Pty_Success;
                  optional { Transmit_Channel_Success; }
                  done_pty = true;
              } or {
                  Transmit_Channel_Failure;
              }
          } or (!done_exec) {
              Receive_Channel_Request_Shell;
              either {
                  Expect_Shell_Success;
                  optional { Transmit_Channel_Success; }
                  done_exec = true;
              } or {
                  Transmit_Channel_Failure;
              }
          } or (!done_exec) {
              Receive_Channel_Request_Exec;
              either {
                  Expect_Exec_Success;
                  optional { Transmit_Channel_Success; }
                  done_exec = true;
              } or {
                  Transmit_Channel_Failure;
              }
```

```
            } or (done_exec) {
                either {
                    Receive_Channel_WindowAdjust;
                } or {
                    Transmit_Channel_WindowAdjust;
                } or (!got_eof) {
                    Receive_Channel_Data;
                } or (!got_eof) {
                    Receive_Channel_ExtendedData;
                } or (!sent_eof) {
                    Transmit_Channel_Data;
                } or (!sent_eof) {
                    Transmit_Channel_ExtendedData;
                } or (!got_eof) {
                    Receive_Channel_EOF;
                    got_eof = true;
                } or (!sent_eof) {
                    Transmit_Channel_EOF;
                    sent_eof = true;
                } or {
                    Receive_Channel_Close;
                    optional { Transmit_Channel_Close; }
                    exit;
                } or {
                    Transmit_Channel_Close;
                    sent_eof = true;
                }
            }
        }
    }
    automaton never ()
    {
        either { Receive_Channel_OpenConfirmation; }
        or { Transmit_Channel_Request_Pty; }
        or { Receive_Channel_Success; }
        or { Transmit_Channel_Success; }
        or { Transmit_Channel_Request_Shell; }
        or { Transmit_Channel_Request_Exec; }
        or { Transmit_Channel_Request_X11; }
        or { Transmit_Channel_Request_ExitStatus; }
        or { Receive_Channel_Request_Env; }
        or { Receive_Channel_Request_ExitSignal; }
        or { Receive_Channel_Request_ExitStatus; }
        or { Receive_Channel_Request_LocalFlowControl; }
        or { Receive_Channel_Request_Signal; }
        or { Receive_Channel_Request_Subsystem; }
        or { Receive_Channel_Request_WindowChange; }
        or { Receive_Channel_Request_X11; }
        or { Receive_Channel_Failure; }
```

```
}
```