Melange: Creating a "Functional" Internet

Anil Madhavapeddy[†‡], Alex Ho[†♡], Tim Deegan[†‡],
David Scott[‡] and Ripduman Sohan[†]

[†]Computer Laboratory, University of Cambridge [‡]XenSource Inc. [♡]Arastra Inc.

Abstract

Most implementations of critical Internet protocols are written in type-unsafe languages such as C or C++ and are regularly vulnerable to serious security and reliability problems. Type-safe languages eliminate many errors but are not used to due to the perceived performance overheads.

We combine two techniques to eliminate this performance penalty in a practical fashion: strong static typing and generative meta-programming. Static typing eliminates run-time type information by checking safety at compile-time and minimises dynamic checks. Meta-programming uses a single specification to abstract the low-level code required to transmit and receive packets.

Our domain-specific language, MPL, describes Internet packet protocols and compiles into fast, zero-copy code for both parsing and creating these packets. MPL is designed for implementing quirky Internet protocols ranging from the low-level: Ethernet, IPv4, ICMP and TCP; to the complex application-level: SSH, DNS and BGP; and even file-system protocols such as 9P.

We report on fully-featured SSH and DNS servers constructed using MPL and our OCaml framework MELANGE, and measure greater throughput, lower latency, better flexibility and more succinct source code than their C equivalents OpenSSH and BIND. Our quantitative analysis shows that the benefits of MPL-generated code overcomes the additional overheads of automatic garbage collection and dynamic bounds checking. Qualitatively, the flexibility of our approach shows that dramatic optimisations are easily possible.

1. INTRODUCTION

The rate of attacks against Internet hosts from malware continues to rise steadily, annually costing millions of dollars in damage and recovery costs. Remarkably, many of the vulnerabilities are still caused by low-level errors in buffer management and marshalling code, despite decades of research into compiler technology which can protect programs from this class of fault.

EuroSys'07, March 21–23, 2007, Lisboa, Portugal.

Table 1 shows recent vulnerabilities in OpenSSH, a widely-used implementation [46] of the SSH protocol written in C. Almost half of these vulnerabilities are in the packet parsing and marshalling code. OpenSSH is especially noteworthy since it is a *security* service and so was written with particular care for safety [45]; despite the best efforts of the developers it has been undone by the sheer complexity of implementing the full protocol in an unsafe language.

It is well known that many low-level errors in buffer management and marshalling code could be eliminated if the software was rewritten in a language which is type-safe [43]. For example, the FoxNet [4, 5] project implemented an entire TCP/IP stack in the language Standard ML. Although undeniably elegant, FoxNet ultimately did not deliver in terms of performance; they reported a 10x performance loss over a conventional TCP/IP stack, and required compiler modifications to handle low-level bit-shifting.

In this paper we demonstrate how it is possible to combine two techniques, *strong static typing* and *generative meta-programming* in a way which both shields Internet servers from these low-level vulnerabilities and which, unlike FoxNet, *introduces no performance penalty*. Our MELANGE framework[1] comprises the Meta Packet Language (MPL), together with a compiler and suite of libraries which target the Objective Caml (OCaml) [29] language.

MPL is a high-level, domain-specific language that describes binary network protocols in a succinct specification and compiles into type-safe, efficient code to manipulate network payloads. The MPL compiler relieves the programmer of the tedious and error-prone task of writing verbose marshalling and unmarshalling code by hand. The generated code exposes a safe external interface while still exploiting techniques such as zero-copy packet handling and in-place update for efficiency. Crucially, the generated code is carefully designed to interact well with automatic garbage collectors like the generational collector in the OCaml system.

We report on fully-featured SSH and DNS servers constructed using MELANGE, and measure greater throughput, lower latency, better flexibility and more succinct source code than their C equivalents OpenSSH and BIND. Our quantitative analysis shows that the benefits of MPL-generated code overcomes the additional overheads of automatic garbage collection and dynamic bounds checking, producing a net performance gain. Qualitatively, the flexibility of our approach shows that dramatic optimisations are easily possible.

[1]The full source code is available online at:
`http://melange.recoil.org/`

VU#	Description
40,327	OpenSSH UseLogin allows remote root execution
945,216	**CRC32 attack detection integer overflow**
655,259	OpenSSH allows arbitrary file deletion
797,027	OpenSSH allows PAM restrictions to be bypassed
905,795	OpenSSH fails to properly apply access control
157,447	OpenSSH UseLogin permits privilege escalation
408,419	**OpenSSH contains overflow in channel handling**
369,347	**OpenSSH vulnerabilities in challenge-response**
389,665	**SSH transport layer vulnerabilities in kexinit**
978,316	Vulnerability in OpenSSH daemon (sshd)
209,807	**OpenSSH server PAM auth stack corruption**
333,628	**OpenSSH contains buffer management errors**
602,204	OpenSSH PAM challenge authentication failure

Table 1: Recent CERT vulnerabilities for OpenSSH, with packet parsing security issues in bold *(source: kb.cert.org)*

2. ARCHITECTURE

In this section we define the details of the MELANGE application framework. It adopts Objective Caml (OCaml) [29] as our implementation language and supports the Meta Packet Language (MPL), which adds support for control of low-level data layout and efficient marshalling and handling of protocol data.

2.1 Objective Caml

OCaml is a modern programming language from the ML family and supports automatic memory management and strong static typing while allowing a mix of functional, imperative and object-oriented programming styles in the same program. Dynamic type-casting is forbidden, and all normal string or array accesses employ bounds-checking at run-time.

Provided a program has no external C bindings and uses none of the small set of built-in OCaml *unsafe functions* then the program is guaranteed to be type- and memory-safe; it cannot be made to overwrite its stack or any unallocated part of memory. OCaml supports concurrency via system threads, although it has a single-threaded garbage collector. The tool-chain is well-developed and supports both interpreted byte-code and fast native-code output on multiple CPU architectures (e.g. i386, Alpha, Sparc, PowerPC and AMD64).

OCaml has steadily gained popularity in the systems research community with projects like CIL [40], Ensemble [22] and Microsoft's Terminator [11] all using it. It is not just static type-safety that makes it an attractive language for systems programming, but also its simplicity. The lack of dynamic type information results in a very lightweight run-time with a consistent block-based heap structure that greatly simplifies writing foreign-language bindings compared to (for example) the Java native code interface. The compiler itself performs only relatively simple code optimisations, leading to greater levels of stability and predictability in the tool-chain.

2.1.1 Garbage Collection

The OCaml run-time includes a fast garbage collector (GC) [14] to manage the heap of OCaml programs automatically. The GC is generational and splits the heap into a *minor heap* for small and short-lived objects and a *major heap* for larger or longer-lived objects. When a small object is allocated it is placed first into the minor heap. When the minor heap is full, a mark-and-sweep garbage collection frees any unreferenced objects. Remaining objects are copied to the major heap, and the minor heap is left completely empty. The major heap is also regularly collected and compacted, but this operation can take significantly longer than the minor heap due to the larger size of objects. The collections happen incrementally to minimise pauses, and new large objects (over 1K in size) are put directly in the major heap in the hope that they will be long lived.

This generational collector handles a typical network server design well. The minor heap, containing small new objects, is ideal for allocating temporary data in the control plane. The major heap, containing older and larger objects, is an ideal place to store the network packet buffers which are re-used by the application layer and thus longer-lived. To tune performance, OCaml provides an API to trigger garbage collection. This is ideal for network servers; it allows MPL to perform memory management between packets.

2.1.2 Network Code

Writing network packet parsing code directly in OCaml is tedious, error-prone and verbose and does not leverage any of the advanced features of the language. Hand-written parsing code in OCaml looks rather like the equivalent C only with more type-conversion functions. Some projects such as Ensemble [22] (discussed further later in §4) adopt a type-unsafe approach to network communication since they trust other network nodes, but this is not an option for Internet-facing network servers. Our Meta Packet Language (MPL) fixes this deficiency by auto-generating the required low-level OCaml from a simple high-level specification and exposes the results as high-level native OCaml types.

2.1.3 Quicker Bounds Checking

OCaml automatically introduces fast bounds checking code before every buffer or array access. However, it is possible for bounds checks to be selectively disabled through the use of an *unsafe function*; e.g. the `String.set` function has the bounds checks while the `String.unsafe_set` does not. Unsafe functions should only be used when there is some way of statically guaranteeing their safety, otherwise the program could suffer a memory fault. To ensure safety, none of our hand-written control-plane code uses these functions. However, the MPL compiler is able to analyse the packet specifications, determine at compile-time when some of the bounds checks may be removed, and emit calls to unsafe functions in the output code. This technique gives a large performance boost without compromising safety or requiring C bindings, as reported later in our evaluation.

2.2 Meta Packet Language

The Meta Packet Language (MPL) is a domain-specific language used to specify the wire format of existing binary network protocols. The specifications contain sufficient information to create bi-directional parsers that can transmit and receive well-formed network protocol packets. MPL specifications define a protocol wire format, and the compiler generates appropriate code and interfaces for that protocol; this is the opposite of conventional interface description languages such as CORBA IDL. Figure 1 illustrates how the use of MPL enforces a separation between the concerns of statefully manipulating packets (the control plane) and of the low-level parsing required to convert to and from a stream of network traffic (the data plane).

Crucially, rather than emitting machine code, the MPL compiler acts as a *meta-compiler* and outputs optimised code in high-level, garbage collected languages (currently only OCaml is fully supported, although we have designed experimental backends for Java

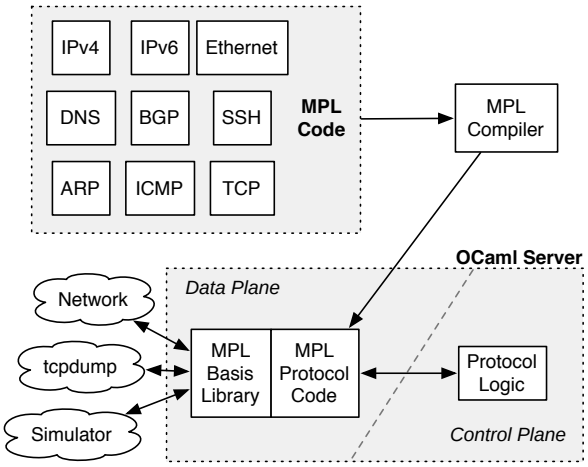


Figure 1: Architecture of an MPL-driven OCaml server

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 ( )**

Figure 2: EBNF grammar for MPL specifications

and Erlang in the past). The generated code itself is not designed to be human-readable and uses the capabilities of the target language to minimise memory allocation and bounds-checking overhead to maximise performance. The interfaces to the code are high-level and "zero-copy" so that accessing the contents of a packet provides a reference where possible and only copies data when necessary.

For example, the OCaml interfaces make use of language features such as polymorphic variants [19], functional objects [47], and ML pattern matching in order to provide a high level of flexibility and safety to the control logic. Internally, the OCaml code makes selective use of imperative, impure constructs to improve efficiency, but hides this from the external interface.

Text-based protocols such as HTTP or FTP are specified as BNF grammars and can mostly be parsed using existing tools such as `yacc`. MPL eases the process of implementing complex *binary* protocols such as SSH, DNS, or BGP. We use a non-lookahead decision-tree parsing algorithm that is simple enough to capture many binary Internet protocols while retaining a simple set of rules to ensure that specifications remain bijective.

MPL cannot express context-free grammars by design, since it has no stack. This has not proven to be a limitation, since most real-world binary Internet protocols are, perhaps due to their roots in early resource-constrained software stacks, simple (albeit quirky) grammars due to the evolutionary nature of Internet protocol design. When greater expressivity is required, MPL supports custom field types which can be written directly in the language backend, as we explain later in our DNS protocol implementation (§3.2.1).

2.2.1 Language

Figure 2 lists the Extended BNF grammar for MPL, and the rest of this section explains it in more detail. The simplest MPL specifications consist of an ordered list of named fields, each with three possible types: (*i*) *wire types* for the network representation of the field; (*ii*) *MPL types* used within the specification for classification and attributes (represented as strings in the grammar); and (*iii*) *language types* that are the native types of the field in the target programming language.

Internet protocols often use common mechanisms for representing values (e.g. 4 octets in big-endian byte order for a 32-bit unsigned integer), and this is captured by wire type definitions. Built-in MPL wire types include bit-fields, bytes, and unsigned fixed-precision integers and can be extended on a per-protocol basis. Section 3.2 containts an illustrative example for DNS. Each wire type is statically mapped onto a corresponding MPL type so the contents of the field may be manipulated within the specification (e.g. for classification). The MPL types are fixed-precision integers, strings, booleans, or "opaque" where the payloads are not parsed. Every wire type also has a corresponding language type—an unsigned 32-bit integer is mapped into the OCaml `int32` type, and a compressed DNS hostname (§3.2) is an OCaml `string list`.

The **classify** keyword permits parsing decisions to depend on the contents of a previously defined field. The packet classification syntax is similar to ML-style pattern-matching with the exception that each match has a text label attached that is used in the output interface to identify the packet type (e.g. "Ethernet-IPv4-ICMP-EchoReply"). Every field can include a set of attributes specifying constraints such as a default value, a constant value, or alignment restrictions. Since most network protocols use a set byte-order, the endian-ness is set via a flag to the basis library routines. It only needs to be changed for host-specific protocol parsing (e.g. our `libpcap` [24] file parser) or protocols which are specifically little-endian (e.g. the Plan 9 filesystem protocol [23]).

Figure 3 lists three MPL specifications for subsets of the Ethernet, IPv4, and ICMP protocols[2]. The examples illustrate how variable-length buffers are bound to previous fields in the header that specifies their length. For example, in IPv4, the `ihl` field is later used to calculate the length of the `options` variable-length buffer during packet parsing, and is automatically calculated when generating IPv4 packets using the MPL interfaces. We have also created MPL specifications for a number of additional protocols, including BGP, DNS, SSH, and DHCP (available on-line).

The **variant** attribute maps values to human-readable labels that are exposed in the external code interface; this is not only more readable but often more type-safe as they become variant algebraic types in ML or enumerations in Java. Many fields also define default attributes to make the code for packet creation more succinct

[2]We do not reiterate the network formats for Ethernet, IPv4 and ICMP for space reasons.

```
packet ethernet {
  dest_mac: byte[6];
  src_mac: byte[6];
  length: uint16 value(offset(eop)-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()];
  };
  eop: label;
}

packet ipv4 {
  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 → Inet_control |7 → Net_control
  };
  delay: bit[1] default(false);
  throughput: bit[1] default(false);
  reliability: bit[1] default(false);
  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_off: 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)];
}

packet icmp {
  ptype: byte;
  code: byte default(0);
  checksum: uint16 default(0);
  classify (ptype) {
  |0:"EchoReply" →
    identifier: uint16;
    sequence: uint16;
    data: byte[remaining()];
  |5:"Redirect" →
    gateway_ip: uint32;
    ip_header: byte[remaining()];
  |8:"EchoRequest" →
    identifier: uint16;
    sequence: uint16;
    data: byte[remaining()];
  };
}
```

Figure 3: MPL specifications for subsets of the Ethernet, IPv4 and ICMPv4 protocols

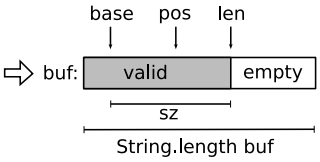
in the common case and afford the MPL compiler the opportunity to create "fast-path" unmarshalling code.

More complex protocols such as DNS or SSH also make use of additional MPL features such as the support for state variables, which are necessary to deal with protocol irregularities and compatibility issues, and boolean/string classifications. This paper does not seek to provide a rigorous definition of MPL, but instead to convey a feel for the succinctness and clarity of a typical real-world protocol specification. A complete user manual is available with more details [32].

2.2.2 OCaml Interface

The OCaml code generated by the MPL compiler does not communicate with the network directly; instead it makes a series of calls to a basis library that includes both I/O and buffer management functions. The library internally represents each packet as a single string to reduce data copying, and provides a light-weight *packet environment* record to represent fragments of packet data:

```
type env = {
  buf: string;
  len: int ref;
  base: int;
  mutable sz: int;
  mutable pos: int;
}
```



This structure uses the OCaml facility for references (essentially type-safe non-NULL pointers) and mutable data that can be destructively updated. A packet environment can be cloned to create a more restrictive view into the packet (e.g. during classification), which cheaply copies the meta-data in the packet environment and not the actual payload. The payload data is always represented by a single large string that, together with its length, is shared across all of the packet environments.

The style of programming found in the generated code is imperative and C-like and, if it were written by hand, could easily result in corrupted packet data. In this system, all the code is generated by the MPL compiler from the MPL specification, ensuring the code is both safe and efficient. The external OCaml interface exposes functional objects to represent each packet, with each classification branch being assigned a unique name based on the labels in the MPL specification.

The example below assumes the presence of checksumming functions that operate on ICMP, TCP or UDP packets and shows how ML pattern-matching can be used to manipulate network data in an elegant functional style with minimal overhead.

```
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")
```

If necessary, low-level code can be written directly using the basis library; the example below iterates over the payload of an ICMP packet environment to calculate the ICMP protocol checksum. Note that the code is 100% OCaml—no C bindings are required.

```
let ones_checksum sum =
    0xffff - ((sum lsr 16 + (sum land 0xffff)) mod 0xffff)
let icmp_checksum env =
    let header_sum = Uint16.unmarshal env in
    Stdlib.skip env 2;
    let body_sum = Uint16.dissect (+) 0 env in
    ones_checksum (header_sum + body_sum)
```

Finally, data copying is minimised while creating packets through the use of *packet suspensions*—closures that capture the arguments required for a packet and delaying the act of writing data to a packet environment. These suspension functions can be nested; higher-level protocol suspensions can contain references to lower-level protocol suspensions. Finally, when an output buffer is available, it is applied to the packet suspension, which writes out its contents to the buffer as one operation. The example below shows how an ICMP echo reply packet can be constructed when supplied with an incoming packet that has previously been classified into two views—`ip` for the IPv4 header and body and `icmp` for the ICMP subset.

```
(* env represents the packet environment *)
let icmp_fn env =
    (* Create ICMP packet suspension *)
    let reply = Icmp.EchoReply.t
        ~identifier:icmp#identifier
        ~sequence:icmp#sequence
        ~data:('Frag icmp#data_frag) env in
    (* Compute overall ICMP checksum *)
    reply#set_checksum (icmp_checksum reply)
in
(* Create the IPv4 suspension *)
let ipr = Ipv4.t ~id:ip#id ~ttl:255 ~proto:'ICMP
    ~src:ip#dest ~dest:ip#src ~options:'None
    ~data:('Sub icmp_fn) in
(* Apply IPv4 packet suspension to environment *)
let reply = ipr env in
let csum = ip_checksum (reply#header_end / 4) env in
reply#set_checksum csum
```

A packet suspension `icmp_fn` is created with information about the ICMP identifier, sequence number, and payload taken from the incoming ICMP packet. The identifier and sequence number are copied since they are integers, but the larger payload is preserved as a reference to the incoming packet. The ICMP suspension is then passed to an IPv4 creation function that copies some data from the incoming packet (e.g. the source and destination addresses) and calculates the checksum. The packet is evaluated "backwards" with the IPv4 closure marshalled, which evaluates the ICMP closure at the appropriate location in the packet. This makes packet creation composable; an Ethernet layer could be added by passing the IPv4 function as another packet suspension; all of the packet offsets would automatically be adjusted by the auto-generated MPL code.

The OCaml interface also supports modifying packets in place, as seen in the `set_checksum` example above. This permits proxies such as IPv4 routers or NAT software to unmarshal packets, safely modify fields in place and transmit the result without re-creating the entire packet. Further details are available separately [32].

2.2.3 Performance

We now evaluate the performance of the MPL/OCaml backend using ICMP, which allows hosts to transmit "ping" packets to other hosts, which send back echo responses. The transmitting host encodes in the request a timestamp that is checked when the response is received and used to determine the time-of-flight of the packet. This simple protocol requires little more than packet parsing, and the size of pings can be varied making it an excellent test for gauging how well MPL code performs.

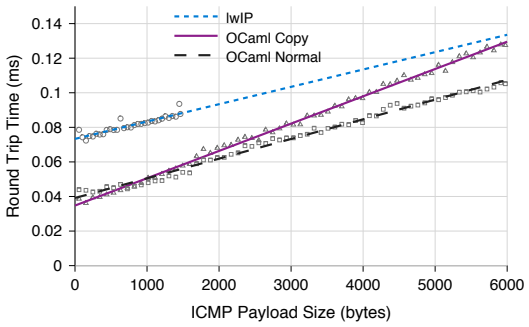


Figure 4: Latencies for lwIP vs OCaml "functional" version (*OCaml Copy*) which copies data and a normal MPL version (*OCaml Normal*) (*lower gradient is better*).

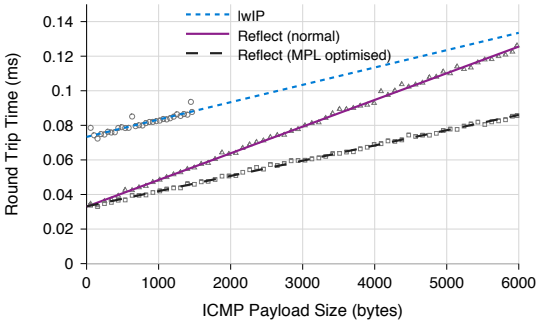


Figure 5: Latencies for lwIP vs the OCaml "reflector" with MPL bounds optimisation off (*Reflect normal*) and on (*Reflect MPL optimised*). The MPL optimised version is type-safe OCaml and as fast as lwIP.

The tests were run on a stock OpenBSD 3.8/i386 (`GENERIC`) kernel, on a 3.00GHz Pentium IV with 1GB of RAM, and all non-essential services disabled. The applications use the `tuntap` interface that allows userland applications to send and receive raw Ethernet in the `tap` mode or IPv4 packets in the `tun` mode. As a reference, we benchmark against the popular lwIP user-level networking stack[3], which is written in C and does not use automatic garbage collection or dynamic bounds checking. This is a good way to measure the throughput of our OCaml implementation versus a C equivalent.

Pings are transmitted on the same machine to eliminate variable network overhead. The Ethernet `tap` interface routes requests to the stack being tested. Our implementation uses the MPL specifications from Figure 3 to process the Ethernet, IPv4, and ICMP protocols, and is completely written in OCaml. The results are plotted over varying ICMP payload sizes; lwIP has a maximum MTU of 1500 so no larger results are available. Each test was repeated 150 times and the mean times plotted against the payload size. The 95% confidence interval is too small to show on the graphs. The gradient of the lines are of primary interest, as this reflects the amount of

[3] See `http://savannah.nongnu.org/projects/lwip`.

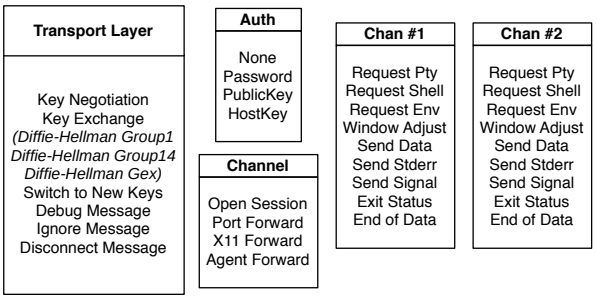


Figure 6: Various layers of the Secure Shell v2 protocol: a global transport, authentication and channel layer, and local channel states.

work done per byte and thus reveals how well the implementations scale with data size.

Figure 4 shows lwIP against two versions of the OCaml ICMP responder: (*i*) the *copying* version that copies the ICMP payload when parsing the packet, and again every time it encapsulates data in a new protocol layer (i.e. ICMP and IPv4), just as a conventional functional implementation would; and (*ii*) the *normal* version that uses the MPL (internally zero-copy) API and creates a new ICMP packet to respond with; it copies the payload exactly once. The copying server (performing 3 payload copies) clearly performs more work per byte than lwIP as reflected in the steeper gradient. The normal version is nearly parallel to the lwIP gradient; it is slightly slower as it re-calculates the ICMP checksum, whereas lwIP takes advantage of the IPv4 checksum algorithm and adjusts it in place. We conclude that minimising data copying—by using the MPL zero-copy API in this case—increases the network performance of the application.

In order to match the performance of lwIP, we implemented a "reflecting" OCaml version that matches its behaviour—the echo request packet is modified in-place and directly re-transmitted as an echo reply. The packet payload is thus read only once (to verify the IPv4 checksum) and not copied at all.

Figure 5 shows the performance of the reflecting OCaml server with every payload access bounds checked, as a manual implementation would, and another that uses the MPL auto-generated code with optimised bounds checks. The MPL-optimised version is as efficient as lwIP, while the version with redundant bounds checks is much slower. This test confirms that the MPL bounds checking optimisations make a significant different to the performance of the data plane code.

This optimisation could potentially be handled by the OCaml compiler itself, but the general case is still an active and complex area of type-theory research (e.g. dependent types [48]). Instead, we choose to solve it by integrating a domain-specific language in which the extra constraints are enforced, to generate optimised OCaml using unsafe constructs in a safe way; this approach is also used by the Coq theorem prover [30].

3. EVALUATION

We now describe two complex servers written using MELANGE: (*i*) a secure shell server, and (*ii*) a domain name server. We discuss the challenges of parsing the respective protocols and evaluate the throughput and latency of each server. We also show that using MPL/OCaml results in more compact code than C. Finally, we analyse the execution profiles and code sizes of the various DNS implementations.

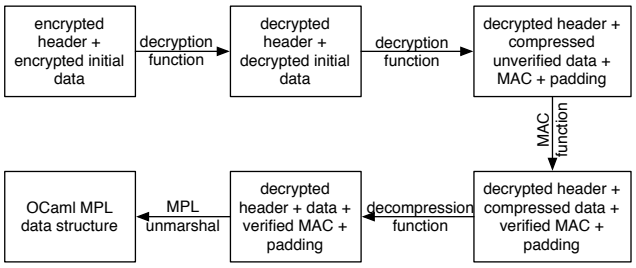


Figure 7: Illustrating the complex data flow of SSH wire traffic to plain text payload that can be parsed using MPL.

3.1 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 and confidentiality, and to multiplex data channels for interactive and bulk data transfer. The protocol has recently been standardised by the IETF[4]; Figure 6 illustrates the various layers: (*i*) a transport layer deals with establishing and maintaining encryption and compression via key exchange and regular re-keying; (*ii*) an authentication layer establishes credentials immediately after the transport layer is encrypted; and (*iii*) a connection protocol that provides data channels for interactive and bulk transfer.

The connection protocol has both global messages (e.g. for TCP/IP port forwarding) and channel-specific messages for individual sessions. Channels can be created and destroyed dynamically over a single connection, and data transfer can continue while new keys are established at the transport layer. The protocol also supports different cryptographic algorithms for the transmission and receipt of data. Extensions such as the use of DNS to store host keys and new authentication methods have also been published[5].

We have implemented a fully-featured SSH library—dubbed MLSSH—that supports both client and server operation. The library supports all 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. The OCaml Cryptokit library is the only external component, and no extra C bindings were used except for the small addition of pseudo-terminal functions (lacking from the OCaml standard UNIX library). Since C bindings are a source of type-unsafety, their complexity and size is kept as minimal as possible—the MLSSH C bindings are 140 lines.

In the remainder of this section, we discuss the challenges of parsing SSH traffic using MPL and evaluate the performance of MLSSH versus OpenSSH.

3.1.1 Packet Format

Constructing a control and data plane abstraction for the SSH protocol is rather more complex than our earlier ICMP case study. Packets are constructed in two stages: (*i*) a secure encapsulation layer for all packets that includes encryption, message integrity

[4]RFC 4251, 4252, 4253, and 4254
[5]RFC 4255, 4256, and 4344

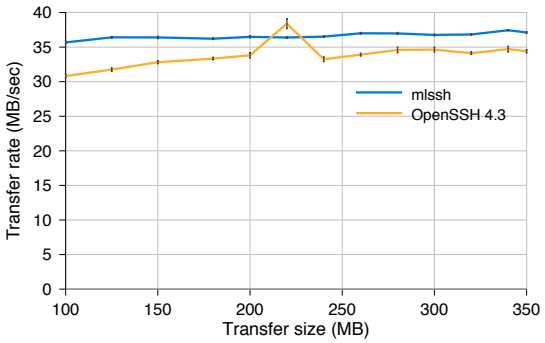


Figure 8: Throughput of OpenSSH vs MLSSH with encryption and message hashing disabled *(higher is better)*.

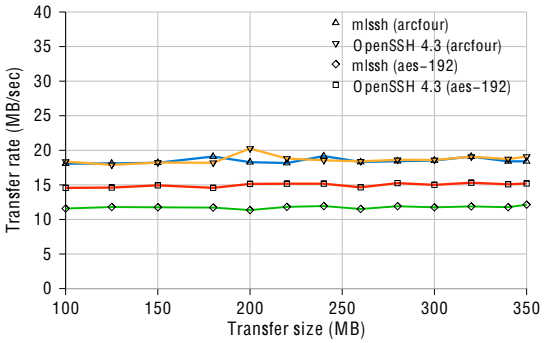


Figure 9: Throughput of OpenSSH vs MLSSH using stream and block ciphers *(higher is better)*.

hashes and random padding to foil traffic analysis; and (*ii*) classification rules for the decrypted packet payloads. Figure 7 illustrates the data flow; firstly a small chunk of data is read and decrypted from which the length of the rest of the packet is obtained. The remaining payload is read and decrypted, followed by an unencrypted message authentication code and random padding. Finally, this plain-text payload is passed onto the MPL classification functions for conversion into a packet object and processing by the control logic. The early implementations of MLSSH [33] did not use MPL and required a payload data copy at every stage of this computation. The latest (and much faster!) version using MPL requires only a single copy across all the stages.

The SSH protocol places high demands for flexibility on parsing tools. MPL-generated code be interfaced easily with hand-written code in order to: (*i*) handle protocol quirks (which exist due to specification errors or historical precedent); and (*ii*) call external library functions (e.g. encryption algorithms) without excessive data copying. MPL permits protocol quirks to be handled using state variables that are driven from the control plane logic. For example, a global SSH channel response can optionally include a "port" field, but only if it is replying to a TCP/IP port-forwarding request; an MPL state variable permits the control plane to instruct the data plane on which parsing action to follow.

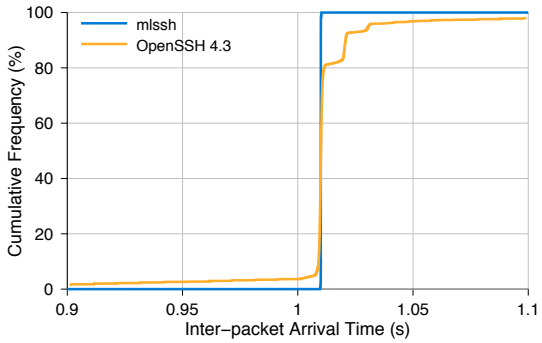


Figure 10: Cumulative Distribution Function of inter-packet arrival times of OpenSSH and MLSSH.

3.1.2 Performance

We measure the sustained throughput of an SSH session by repeatedly transferring large files through a single connection. The OpenSSH client is used to connect to either an MLSSH or OpenSSH server, with all logging and debug code disabled. A file of variable size (ranging from 100MB to 350MB) is transferred via the established SSH connection. This is repeated 100 times across the same connection by dynamically creating new channels, ensuring that at least 10GB of data are sent through every session to highlight any bottlenecks due to memory or resource leaks. Since the SSH protocol also mandates regular re-keying, our benchmarks reflect that cost as part of the overall results.

Figure 8 shows a plot of transfer rate (in MB/sec) versus the transfer size of the individual data chunks with encryption disabled. Each data point and error bar reflects the average time and 95% confidence interval over the 100 repeated invocations. MLSSH is slightly faster than OpenSSH and interestingly also has a smaller variation of transfer rates. In general, OpenSSH was more "jittery" as seen in the anomalously high transfer rate when transferring files in 220MB chunks (this was reproducible and attributed to cache behaviour).

Figure 9 shows the same experimental setup applied with encryption enabled and using HMAC-SHA1-160 as the message digest algorithm. Both servers have equivalent performance when using the Arcfour stream cipher, but due to the less optimised AES implementation MLSSH is slower when used with the AES-192 block cipher. Comparison of the different cryptographic libraries used (OpenSSL and Cryptokit) reveals that the OCaml AES implementation is less optimised and has potential for improvement.

We also measured the latency of established SSH connections to test if automatic garbage collection was introducing long pauses in MLSSH. The server is first heavily loaded with bulk data transfers as in the previous test, and then a "character generator" alternately transfers a single byte and sleeps for a second. The times between receiving these characters are plotted in Figure 10 as a cumulative distribution function.

The arrival times recorded through MLSSH are extremely consistent and clustered around the one second mark with little variance. In contrast, OpenSSH exhibits jitter within a range of $\pm$100ms; delays are being introduced within the server which cause it to disrupt the arrival times. This is surprising since: (*i*) OpenSSH is perform-

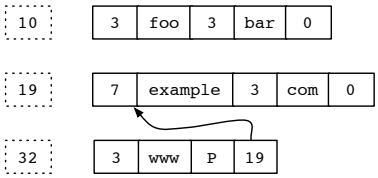


Figure 11: DNS label compression example, with `www.example.com` being encoded by a pointer. The dashed boxes are the offset from the start of the packet.

ing manual memory management which should be faster than automatic garbage collection; and (*ii*) MLSSH ought to have a wider distribution to reflect the cost of the occasional garbage collection introducing a delay.

Examination of the internals of the OpenBSD *malloc(3)* and *free(3)* routines reveal that modern memory management is as complex as the OCaml garbage collector routines. Allocation in OCaml is a simpler process than *malloc(3)* since only a single pointer needs to be incremented [14], as opposed to the more complex free-list management required by the `libc` functions. The presence of an incremental garbage collector which performs predictable slices of memory management at regular intervals is also better than the more ad-hoc caching of pages (to reduce the number of system calls) performed by *free(3)*. The minimised memory allocation of MPL means that the OCaml major heap is not over-used, and expensive compaction of the major heap is avoided, resulting in faster performance than the manual memory management routines.

3.2 Domain Name System (DNS)

The Domain Name System is a distributed database used to map textual names to information such as network addresses. The DNS consists of three components: (*i*) the Domain Name Space and Resource Records (RRs), which form a tree-structured namespace with associated data; (*ii*) name servers, which hold information about portions of the namespace and either act as authoritative sources or proxies; and (*iii*) resolvers in client network stacks, which 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 [1] serves over 70% of DNS second-level `.com` domains and over 99% of the servers are written in C [3, 38].

BIND has a long history of critical security vulnerabilities despite several complete re-writes. A statically type-safe and flexible DNS server would be useful not only for immediate deployment, but also to aid research into novel name systems (e.g. centralised name services [12]). Our authoritative server—dubbed DEENS—is written entirely in MPL and OCaml. DEENS also features a BIND-style zone file parser, and we have also written several variants such as a multicast DNS server, a `dig` client, and caching proxies.

3.2.1 DNS Packet Format

DNS was designed to be a low-latency, low-overhead protocol for resolving domain names. In order to avoid the time required to perform a 3-way TCP handshake, most DNS requests and responses can be encoded in a single UDP packet, normally 512 bytes or less. Due to tight resource restrictions, the original DNS specification employed a compressed binary packet format[6].

[6]RFC 1034, 1035

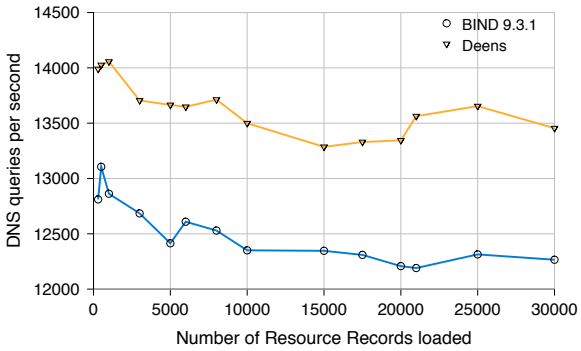


Figure 12: Throughput of BIND vs DEENS with random Zipf-distribution query sets *(higher is better)*.

The compression scheme works as follows. An uncompressed hostname is separated into a list of labels by splitting at each dot character. Each label is represented by a byte indicating its length followed by the contents. A length of 0 indicates the end of the hostname. To save space, duplicate labels are stored just once with pointers used to reference the shared copy; this duplication is common within response packets since the top-level portions of hostnames are often shared.

Figure 11 illustrates this compression—two hostnames `foo.bar` and `example.com` are defined in different areas of a DNS response (the dashed boxes indicate absolute offsets within the packet). When the hostname `www.example.com` is inserted later, the `www` label is inserted as normal, but the tail of the hostname is replaced by a pointer to the previous definition of `example.com`.

This compression scheme is challenging to implement securely and safely, and has been the cause of several serious bugs in other servers (e.g. from recursively following pointers while parsing DNS traffic). Recall that MPL supports custom field types in order to extend protocol descriptions. We define two new custom types for DNS: (*i*) `dns_label`; and (*ii*) `dns_label_comp`, where the latter indicates a compressible hostname. The custom types are implemented directly in OCaml as extensions to the basis library, and use a stateful symbol table to track the locations of pointers and labels. This permits DNS packets to be processed (for both creation and parsing) in a single pass, and the logic for handling these special labels is contained in a small MPL module.

3.2.2 Performance

We generated a large random data set using the freely available BIND DLZ tools[7], which generate both the source zone files for an authoritative server and also an appropriate query set that can be fed into the `queryperf` measurement tool from the BIND 9.3.1 distribution. The data was configured in a Zipf power-law distribution to match real-world DNS data sets [26].

Figure 12 measures the performance of BIND against DEENS in terms of queries per second against the data set size. The OCaml implementation is around 10% faster, and both servers exhibit level

[7]Available online at `http://bind-dlz.sf.net/`

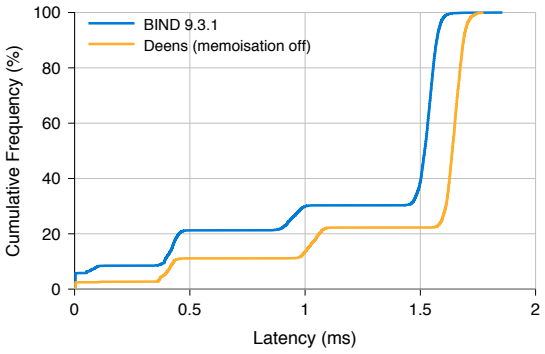


Figure 13: Cumulative Distribution Function of BIND vs DEENS **latencies with loaded servers *(lower is better)*.**

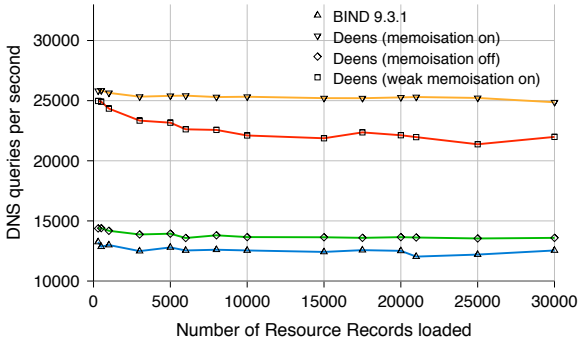


Figure 14: BIND vs DEENS **throughput with the strong and weak memoisation optimisations with random Zipf-distribution query sets *(higher is better)*.**

performance as the data set size increases. Figure 13 shows the cumulative distribution function for response latency. DEENS is consistently slightly faster than BIND, but the stair-step shape of the graph shows that the depth of the query dominates the implementation language.

However, the real benefit of using OCaml becomes obvious when we observe that the results of DNS queries are purely a function of the tuple `qclass` × `qname` × `qtype` of a DNS question, where `qclass` is the DNS class (most often "Internet"), `qname` is the domain name and `qtype` is the request record type. The exception to this rule is servers that perform arbitrary processing when calculating responses (e.g. DNS load balancing[8]), but this is a specialist feature we are not concerned with for the moment. The only variation is that the first two bytes in the response must be modified to reflect the DNS `id` field of the request.

As an optimisation, we add a memoisation query cache that captures a query answer in a string containing the raw DNS response and use the cached copy when possible. This requires changes to just 4 lines of code in DEENS, and to test the effectiveness we implemented two separate caching schemes: (*i*) a normal hash-table mapping the query fields to the marshalled packet; 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 will continue to grow. The weak hash table lies at the other extreme and is a cache that can be garbage collected and data may disappear at any time. Weak references are special data structures that do not count towards the reference counts of objects they point to for the purposes of reclamation and are often used as a safe mechanism to construct efficient purely functional data structures (known as "hash consing"). In our case we are using the weak data structure in isolation without any strong references pointing to it, and so it is cleared on every garbage collection cycle. Furthermore, it does not require any traditional cache management (e.g. least-recently-used checks) and can safely grow to any size—if the heap grows too large, a garbage collection will erase the cache.

Figure 14 shows a dramatic performance increase from our memoisation cache as DEENS is now twice as fast as BIND as a result of a small change in our OCaml code. This flexibility highlights the gains from re-implementing protocols using high-level languages—we can experiment with various data structures with relatively little effort, while maintaining type-safety.

[8]RFC 1794

3.3 Code Structure

In this section we analyse the code structure of MPL/OCaml applications, firstly via instruction profiling, and secondly by looking at the code size.

3.3.1 Profiling Analysis

Applications constructed using MPL/OCaml have very different run-time behaviour from applications written in C using manual memory management. In this section we present the results of detailed profiling of DEENS and BIND in order to understand these differences. The performance tests (§3.2.2) were repeated on a cluster of dual-CPU 2.4GHz (no-HT) Xeon machines, running Linux 2.6.17.9 and `oprofile`.

Using a combination of function call-graphs and cumulative-time profiling, we categorised the time spent by each application into: (*i*) **System** calls; (*ii*) **Network** packet handling code; (*iii*) **Libraries** (e.g. `libc`); (*iv*) **Memory** management (e.g. garbage collection); (*v*) **OCaml** run-time library; (*vi*) **Data** structure management (e.g. looking up a query); and (*vii*) **Other** code (e.g. thread management). For the OCaml applications, we assigned standard library functions depending on their invocation in the call graph where possible, and only into the more generic "OCaml" category if the use wasn't clear. For the purposes of our analysis, we combine the time spent in the OCaml run-time library and data management. Figure 15 shows the results for BIND and normal and memoised DEENS.

BIND spends most time in data management (49.5%) and network packet creation (23.2%) with little time in its memory management layer (4.9%). DEENS spends more time in data management due to the overhead of the OCaml run-time library (57.8%) and less time in packet processing due to the more efficient MPL-generated code (16.3%). Both servers spend approximately 14% in external libraries and 4.1% in system calls, indicating that there is no extra overhead to the userland/kernel interface when using MPL and OCaml.

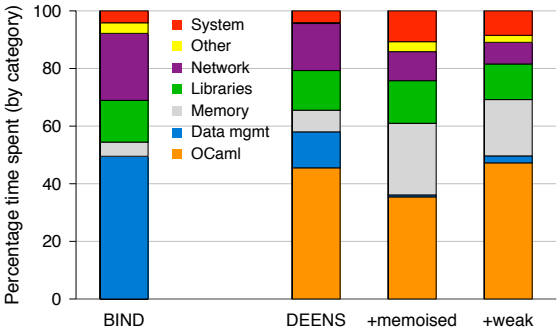


Figure 15: Normalised profiling results for the DNS servers, showing how each application spends its time serving queries.

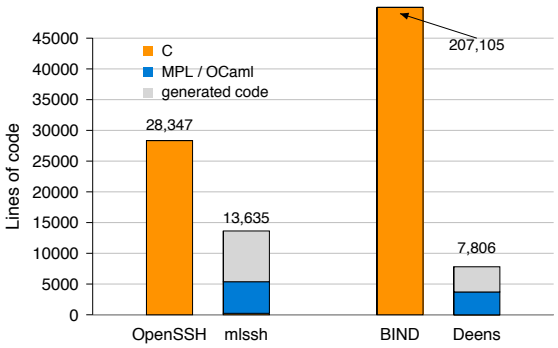


Figure 16: Relative code sizes for MPL/OCaml and C code *(lower is better)*.

Clearer differences arise when examining the memoized versions of DEENS. Recall (§3.2.2) that there are two versions—a strongly memoized cache which never releases cached entries and uses a larger heap in return for greater performance, and a weakly memoized cache which is erased on every garbage collection, but still maintains fast performance. Both versions spend less time processing network packets (12.35% and 14.4%) due to the cache hit rates, and more time in the garbage collector (19.5% and 22.8%) due to the extra use of the heap for storing cache entries. As expected, the strongly-memoized version spends more time in the garbage collector (by 3.3%) due to the larger heap requiring longer collection scanning times. The increased system call percentage (8.5% and 10.7%) is because the faster memoized versions are transmitting many more packets than the slower non-caching versions.

As an aside, the memoized DEENS saturated a GigE network line with responses during these tests, sustaining over 64,000 query responses per second (compared with around 20,000 for a non-caching DEENS, and less for BIND).

Memory Usage

In our tests, we loaded the DNS server with 30,000 resource records from approximately 2,200 zones. A recent survey of DNS name server density[9] shows the mean number of zones per server at 37.2 and the median 3.0, placing our experimental setup comfortably larger than an "average DNS server".

The memory hierarchy of modern servers is large enough to store a significant proportion of hot zone data in the processor cache. Our tests show a virtually 100% L2 data cache hit rate while running the benchmarks and DEENS having a slightly better instruction cache hit-rate than BIND due to its smaller code footprint. We have also explored ML DNS servers supporting millions of zones [13], although we do not cover that analysis in this paper.

3.3.2 Lines of Code

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 techniques.

[9]The Measurement Factory, June 2005. `http://dns.measurement-factory.com/surveys/200506.html`

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 through the `camlp4` pre-processor that reformats it to a consistent, well-tabulated style. External libraries were not included in the count (e.g. OpenSSL or Cryptokit).

Figure 16 plots the number of lines of C, OCaml and auto-generated code present in the applications. The figures for SSH 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 numbers for DNS reveal that DEENS is a remarkable 50 times smaller than the BIND 9.3.1. DEENS does lack some of the features of BIND such as DNSSEC support and so this should only be treated a rough metric. We are confident, particularly after our experiences with constructing MLSSH, that these extra features can be implemented without issue.

3.3.3 Configuration

The use of the MELANGE framework encourages the separation of data plane logic from control plane logic. The former is written in MPL and the latter in OCaml. A benefit of this split is that configuration information can easily be abstracted out by the control plane portion. In MLSSH, for example, all configuration decisions are represented as a functional object that is exported from the library and implemented by the main application. A sample snippet is shown next:

```
type user_auth =
    |Password |Public_key
    |Interactive |Host
type reason_code = |Protocol_error |Illegal_user [etc...]
type auth_resp = bool * user_auth list
type conn_resp =
    |Allow of connection_t
    |Deny of reason_code
class type server_config = object
   method connection_req : int32 → int32 → conn_resp
   method auth_methods_supported : user_auth list
   method auth_password : string → string → auth_resp
   method auth_public_key : string → Key.t → auth_resp
end
```

In this example, we define two new types: (*i*) for user authentication methods (`user_auth`); and (*ii*) for providing reasons for denying connections (`reason_code`). Both are part of the SSH protocol specification. The `server_config` object defines a *class signature*, which contains a list of call-back functions that are invoked at appropriate points in the protocol state machine. For example, the `auth_password` function is called when password authentication is required, and `connection_request` when a new connection with some window size parameters is requested by the client.

The result of this separation is that applications are now much more composable; e.g. embedding SSH functionality into other OCaml code is now a trivial matter of implementing this class type. This technique should be familiar to programmers who use object-oriented languages like Java or C++, but it can be implemented very succinctly in MELANGE because of OCaml's automatic type inference.

We have found that these configuration objects are very useful for rapidly prototyping novel network architectures. For example, we have combined MLSSH and DEENS with bindings to the user-level `tuntap` interface to experiment with new distributed naming algorithms in user-space, as well as constructing file systems that export their contents via the DNS (which involved specifying the Plan9 9P protocol [44] using MPL).

Composable configuration files can also be used with common logging and network setup modules; the notion of configuration unification between applications is an active research area in MELANGE. Our source code includes a domain-specific configuration language intended to bridge standard Unix `/etc`-style configuration files and the configuration objects described above.

3.3.4 MPL Custom Types

All of the implementations consist of a varying degree of code written in MPL and OCaml. This section discusses any custom field types required to implement a protocol using MELANGE.

SSH required defining one major custom type: the multiple precision integer. This has a very precise wire representation (defined in RFC 4253), but does not require any state to be maintained and is straight-forward to implement. The majority of the SSH protocol parsing can be expressed directly in MPL, once the external encryption layer has been processed, as described earlier (§3.1.1).

DNS required defining custom types for the string compression format. This implementation is around 100 lines of OCaml code, using a symbol table to keep track of entries in the compression table.

IPv4 required no custom fields, but requires two specifications: one for the IPv4 packet, and another for the contents of the IPv4 options field.

ICMP required no custom fields.

UDP required no custom fields.

DHCP has variable-length option fields which cannot be expressed directly in MPL. However, each individual option can be parsed with an MPL specification, with custom OCaml code used to iterate over the list of options until it reaches the 'end-of-option' field.

In general, it proved difficult to generalise a "list" type in MPL, since the precise wire formats of lists vary between protocols (e.g. how they are terminated or how elements are represented).

4. RELATED WORK

The inspiration for MPL stems from the FoxNet project [4, 5], which implemented a TCP/IP stack using Standard ML. It made use of SML/NJ features such as parameterised modules to separate out protocol elements into a series of abstract signatures. A combination of these protocol signatures resulted in a TCP/IP implementation, and other permutations included TCP over Ethernet Their work highlighted layering violations in the design of TCP such as the pseudo-header used in checksums. FoxNet was one of the first attempts to apply a functional language to a low-level "systems" problem such as network protocol implementation.

Although undeniably elegant, FoxNet ultimately did not deliver in terms of performance; they reported a 10x performance loss over a conventional TCP/IP stack, and required compiler modifications to handle low-level bit-shifting. We attribute MELANGE's superior performance to: (*i*) advances in hardware such as faster processors, larger caches, and more memory to increase efficiency of garbage collection; (*ii*) built-in support in recent OCaml compilers for low-level coding; and (*iii*) integrating meta-programming techniques via MPL. FoxNet did not tackle complex application-level protocol implementations such as SSH or DNS.

Another successful networking project that used OCaml is the Ensemble network protocol architecture [22]. Ensemble is designed to simplify the construction of group membership and communications protocol stacks by composing simple *micro-protocols*, which can be re-arranged depending on the exact application needs. They cite reduced memory allocation and avoiding the use of foreign function bindings as key to a successful OCaml system. When compared to MPL-based code, a crucial difference is that Ensemble assumes that all nodes are trusted and directly transfers OCaml heap structures into C `iovecs` to be transmitted over the network, a technique Ensemble calls *direct marshalling*. This means that a malicious node, or any heterogeneous node running on a different CPU architecture or OCaml version, can corrupt heap structures by sending malformed data. In contrast, MPL uses well-defined existing protocols and bounds checks to ensure that all traffic is valid. MELANGE is designed to simplify the implementation of network protocols; it does not tackle the high-level distributed communications issues that Ensemble solves. Billings et al. have created HashCaml [7] to permit type-safe marshalling in OCaml when the precise wire format used does not matter to the application.

Other data description languages for network protocols are PACKETTYPES [35] and Prolac [27], which have both been used to create implementations of TCP/IP and similar low-level protocols. The key differentiation of MPL is that it outputs high-level, type-safe code instead of C, and its facility for state variables and custom field types permit the expression of more complex application-level protocols such as DNS, BGP or SSH. To our knowledge, we are the first to consider the wider spectrum of Internet protocols instead of the relatively simple low-level ones. The PADS language [17], using modern dependent typing techniques to describe ad-hoc data sources, looks promising as an alternative to MPL if an efficient type-safe language backend is developed.

Many packet filtering languages have been developed to specify rules for detecting patterns in network traffic, such as BPF [36]

and the more extensible FFPF [8]. An early stub-compiler was USC [41] which provided an extended form of ANSI C to succinctly describe low-level heard formats and generate near-optimal C code to parse them. More recently, Pang et al. designed *binpac* as an equivalent of `yacc` aimed at parsing binary protocols [42]. binpac can parse a number of complex protocols such HTTP, DNS, CIFS/SMB, and Sun RPC. The key difference of these parsers from MPL is that they are unidirectional, and cannot be used to also create network packets from the same specification, as MPL or PACKETTYPES can. The general problem of bidirectional parsing is still an active area of language research, most notably tackled by Foster et al. for tree-structured data [18].

MPL draws from research into constructing fast data paths through operating systems, such as Scout [39] or fbufs [15], which both perform the same task inside the kernel. MPL generates type-safe code to provide the same facility to user-level network applications.

Click [28] was developed to ease the process of creating extensible network routers. It is assembled from packet processing modules (e.g. for classification, queueing or scheduling) and a configuration language specifies a routing graph. It is written in C++ and primarily focuses on low-level network protocols, unlike our wider focus on application-level binary protocols as well.

Recently, there have been several more formal languages designed to deal with complex networking problems. P2 [31] implements a high-level declarative language for constructing peer-to-peer networks succinctly, and meta-routing [20] is an algebra for network routing. The implementations are currently written in C++, and MPL complements them by enabling a switch to a type-safe language without compromising performance.

Researchers have also been helping to evolve C code to a safer future; languages such as Cyclone [25] and Ivy [9] extend the semantics of C to be type-safe and require minimal modifications from existing code. We are attempting the opposite approach, by starting from a clean language and solving low-level performance problems, but acknowledge that the evolutionary approach is also essential given the large amount of legacy C code already written. However, we have shown by example that our philosophy is quite practical for the well-specified Internet standard protocols.

5. CONCLUSIONS

We have described the Meta Packet Language (MPL), which permits the high-level specification of Internet protocols, and have described a compiler that transforms these high-level specifications into type-safe code. MPL is part of the MELANGE framework written in OCaml, and we used it to specify many standard Internet protocols ranging from the low-level IPv4, Ethernet and ICMP, to higher-level application protocols like SSH and DNS. Our approach of using a domain-specific packet specification language helped solve the issues reported by previous research projects such as FoxNet and Ensemble, such as the creation of high-performance servers without the need for foreign-function bindings.

We described and evaluated our implementations of the SSH and DNS protocols and found our type-safe versions to perform better than their counterparts written in type-unsound C. MLSSH sustained a higher bulk-throughput with less inter-packet jitter than OpenSSH, and DEENS handled double the queries per second with lower latency than BIND. Furthermore, both contain fewer lines of code and were easier to enhance and optimise.

5.1 MPL Enhancements

Modern kernels perform the minimum data copying required to process data as it passes through the network stack (e.g. the FreeBSD `mbuf` [37]) and strive for a zero-copy data-flow of network payloads directly from the hardware to user-space applications [10]. `mbuf`s can also be chained to support *scatter-gather* I/O.

The current MPL mechanism for packet suspensions consists of a set of operations which must be executed to construct a network packet (e.g. write some fields at an offset, or copy a payload from another packet environment). When the packet suspension is evaluated, it results in a single buffer which is transmitted using the *write(2)* or *sendto(2)* system call. The basis library could be extended to eliminate the need for the final copy into a single buffer, and instead internally maintain a data structure suitable for scatter-gather transmission via the *writev(2)* system call instead. Thus far however, the single final copy for marshalling a packet has not been shown to be a great overhead, and is much simpler to implement.

5.2 Future Research

Our approach of software reconstruction has opened up many interesting avenues of research. We have implemented a prototype operating system in OCaml to run MELANGE applications directly as guest operating systems over the Xen hypervisor [2], thus skipping the overhead of a general-purpose operating system written in in a type-unsafe language.

Our creation of a code-base of networking applications written in OCaml is helping with the integration of quasi-linear types [16] into OCaml to statically enforce the control/data abstractions. The use of MPL is also spreading beyond networking, being used as a convenient description language for creating bindings between kernel modules and safe OCaml daemons in user-space for file-system services [6], in the style of icTCP [21].

The complete MELANGE framework includes the content described in this paper as well as tools for integrating formal model-checking techniques into the applications [33, 34, 32]. The source code is available under a BSD-style license at `http://melange.recoil.org/`, and contributions and bug reports are welcomed!

6. ACKNOWLEDGEMENTS

We would like to thank Tim Griffin, John Billings, Jon Crowcroft, David Greaves, Steven Hand, Christian Kreibich, Evangelia Kalyvianaki, Andrew Warfield, and Euan Harris for many hours of discussions, reviews and cups of strong coffee with the authors. This work was partially funded by Intel Research Cambridge.

7. REFERENCES

[1] ALBITZ, P., AND LIU, C. *DNS and BIND*, fourth ed. O'Reilly, 2001.

[2] 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 *Proc. of 19th SOSP* (Bolton Landing, NY, 2003), pp. 164–177.

[3] BERNSTEIN, D. J. DNS server survey [online]. 2002. `http://cr.yp.to/surveys/dns1.html`.

[4] BIAGIONI, E. A structured TCP in Standard ML. In *Proc. of SIGCOMM* (London, UK, 1994), pp. 36–45.

[5] BIAGIONI, E., HARPER, R., AND LEE, P. A network protocol stack in Standard ML. *Higher Order Symbolic Computing 14*, 4 (2001), 309–356.

[6] BILLINGS, J., FRASER, A., AND SCOTT, D. Orion: Named flows with access control. Tech. rep., Fraser Research, Princeton, NJ, USA, 2005.

[7] BILLINGS, J., SEWELL, P., SHINWELL, M., AND STRNISA, R. Type-safe distributed programming for OCaml. In *Proceedings of the 2006 ACM-SIGPLAN Workshop on ML* (2006).

[8] BOS, H., DE BRUIJN, W., CRISTEA, M., NGUYEN, T., AND PORTOKALIDIS, G. FFPF: Fairly Fast Packet Filters. In *Proc. of 6th Symp. on OSDI* (2004), pp. 347–363.

[9] BREWER, E., CONDIT, J., MCCLOSKEY, B., AND ZHOU, F. Thirty years is long enough: Getting beyond C. In *Proc. of 10th HotOS Workshop* (2005).

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

[11] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Termination proofs for systems code. In *Proc. of PLDI* (2006).

[12] DEEGAN, T., CROWCROFT, J., AND WARFIELD, A. The main name system: an exercise in centralized computing. *Computer Communications Review 35*, 5 (2005), 5–14.

[13] DEEGAN, T. J. *The Main Name System*. PhD thesis, University of Cambridge, 2006.

[14] DOLIGEZ, D., AND LEROY, X. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. of the 20th Symp. on PoPL* (Charleston, South Carolina, 1993), pp. 113–123.

[15] DRUSCHEL, P., AND PETERSON, L. L. Fbufs: a high-bandwidth cross-domain transfer facility. In *Proc. of 14th SOSP* (1993), pp. 189–202.

[16] ENNALS, R., SHARP, R., AND MYCROFT, A. Linear types for packet processing. In *13th European Symp. on Programming* (Barcelona, Spain, 2004), pp. 204–218.

[17] FISHER, K., AND GRUBER, R. PADS: a domain-specific language for processing ad hoc data. In *Proc. of 2005 Conf. on PLDI* (Chicago, IL, 2005), pp. 295–304.

[18] 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* (New York, NY, USA, 2005), ACM Press, pp. 233–246.

[19] GARRIGUE, J. Programming with polymorphic variants. In *SIGPLAN Workshop on ML* (Baltimore, MD, 1998).

[20] GRIFFIN, T. G., AND SOBRINHO, J. L. Metarouting. In *Proc. of SIGCOMM* (Philadelphia, PA, 2005), pp. 1–12.

[21] GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Deploying safe user-level network services with icTCP. In *Proc. of 6th Symp. on OSDI* (2004), pp. 317–332.

[22] HAYDEN, M. *The Ensemble System*. TR98-1662, Cornell University, 1998.

[23] HENSBERGEN, E. V. Plan 9 remote resource protocol (experimental-draft-9p2000-protocol), March 2005. `http://v9fs.sourceforge.net/rfc/`.

[24] JACOBSON, V., LERES, C., AND MCCANNE, S. Packet capture with tcpdump and pcap [online]. `http://www.tcpdump.org/`.

[25] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *Proceedings of USENIX Annual Tech. Conf. (General Track)* (2002), pp. 275–288.

[26] JUNG, J., SIT, E., BALAKRISHNAN, H., AND MORRIS, R. DNS performance and the effectiveness of caching. In *Proc. of 1st Workshop on Internet Measurement* (San Francisco, CA, 2001), pp. 153–167.

[27] KOHLER, E., KAASHOEK, M. F., AND MONTGOMERY, D. R. A readable TCP in the Prolac protocol language. In *Proc. of SIGCOMM* (Cambridge, MA, 1999), pp. 3–13.

[28] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Trans. on Computer Systems 18*, 3 (2000), 263–297.

[29] LEROY, X., DOLIGEZ, D., GARRIGUE, J., RÉMY, D., AND VOUILLON, J. The Objective Caml system [online]. 2005. `http://caml.inria.fr/`.

[30] LETOUZEY, P. Exécution de termes de preuves: une nouvelle méthode d'extraction pour le Calcul des Constructions Inductives. Université Paris VI, 2000.

[31] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing declarative overlays. In *Proc. of 20th SOSP* (2005), pp. 75–90.

[32] MADHAVAPEDDY, A. *Creating High-Performance Statically Type-Safe Network Applications*. PhD thesis, University of Cambridge, 2006.

[33] MADHAVAPEDDY, A., AND SCOTT, D. On the challenge of delivering high-performance, dependable, model-checked internet servers. In *First Workshop on Hot Topics in System Dependability* (2005).

[34] MADHAVAPEDDY, A., SCOTT, D., AND SHARP, R. SPLAT: A tool for model-checking and dynamically enforcing abstractions. In *12th Int'l SPIN Workshop on Model Checking of Software* (2005), pp. 277–281.

[35] MCCANN, P. J., AND CHANDRA, S. Packet types: Abstract specification of network protocol messages. In *Proc. of SIGCOMM* (2000), pp. 321–333.

[36] 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), pp. 259–270.

[37] MCKUSICK, M. K., AND NEVILLE-NEIL, G. V. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional Computing Series, August 2004.

[38] MOORE, D. DNS server survey [online]. 2004. `mydns.bboy.net/survey/`.

[39] MOSBERGER, D., AND PETERSON, L. L. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (1996), pp. 153–167.

[40] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy code. In *Proc. of 29th Symp. on POPL* (Portland, OR, 2002), pp. 128–139.

[41] 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* (New York, NY, USA, 1994), ACM Press, pp. 295–306.

[42] PANG, R., PAXSON, V., SOMMER, R., AND PETERSON, L. binpac: A yacc for writing application protocol parsers. In *Proceedings of the Internet Measurement Conference* (2006).

[43] PFLEEGER, S. L., AND HATTON, L. Investigating the influence of formal methods. *Computer 30*, 2 (1997), 33–43.

[44] PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. The use of name spaces in Plan 9. *Operating Systems Review 27*, 2 (1993), 72–76.

[45] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *Proc. of 12th USENIX Security Symposium* (2003), pp. 231–242.

[46] PROVOS, N., AND HONEYMAN, P. ScanSSH: Scanning the internet for SSH servers. In *Proc. of 15th LISA* (San Diego, CA, 2001), pp. 25–30.

[47] RÉMY, D., AND VOUILLON, J. Objective ML: a simple object-oriented extension of ML. In *Proc. of 24th Symp. on POPL* (1997), pp. 40–53.

[48] XI, H., AND PFENNING, F. Eliminating array bound checking through dependent types. In *Proc. of Conf. on PLDI* (1998), pp. 249–257.