On the Challenge of Delivering High-Performance, Dependable, Model-Checked Internet Servers

Anil Madhavapeddy[1] and David Scott[2]
University of Cambridge Computer Laboratory[1], Fraser Research[2]
avsm2@cl.cam.ac.uk[1], djs@fraserresearch.org[2]

1 Introduction

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 dependability issues ranging from low-level buffer overflows to subtle protocol logic errors. 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++, often due to a desire for performance or portability.

We set ourselves a challenge: re-implement a critical Internet service with a history of security problems which meets both the traditional goals of systems research (high performance, portable and scalable servers), and those of the dependable systems community (well-specified, formally-verified, fault-tolerant code). We chose to implement SSH v2.

We believe our SSHv2 implementation is a good proof of concept because: (*i*) the protocol is widely used as the modern "remote shell", relied upon to gain access to machines across a hostile network; (*ii*) the most common implementations, such as OpenSSH or SSH.com, are written in C and have had a steady stream of critical security vulnerabilities and bugs in recent years; and (*iii*) the SSHv2 protocol is in the process of being documented by the IETF.

We build upon Objective Caml [5] — an existing high-level language which provides strong memory-safety guarantees — but *crucially*, we also address performance and logical correctness concerns by setting ourselves two goals for the server: (*i*) it must be at least as fast as the current best of breed industrial implementation (in this case, OpenSSH); and (*ii*) it must be tractable to model-check important aspects of the server behaviour.

In Section 2, we show that low-level packet parsing can be neatly expressed as a *protocol decision tree* in order to: (*i*) generate efficient, statically type-safe functions suitable for marshalling, unmarshalling and pretty-printing individual packets; (*ii*) sanity-check the actual protocol specification for inconsistencies before generating any code. Our checks of the SSHv2 protocol using this technique revealed real errors in the Internet Drafts being published by the SECSH working group [13], and resulted in corrections being made.

In Section 3, we describe the notion of a *statecall* and we demonstrate that the use of type-safe languages permits the introduction of *inline automata*, allowing a simplified representation of the server process to be statically model-checked and dynamically enforced at run-time, with extremely small performance overhead. Our inline automata enable arbitrarily fine-grained models of the program to be checked, in contrast to techniques such as privilege separation [10], which incurs a per-message IPC overhead due to the use of multiple processes.

In Section 3.1, we describe the Statecall Policy Language (SPL), used to specify the inline automata using a powerful, expressive, and familiar 'C'-like syntax. The SPL compiler outputs automaton code which links in directly to the server source code, and also targets PROMELA, the input language of the SPIN model checker [4] allowing us to mechanically verify important safety properties. Related work is discussed in Section 4 while Section 5 provides current results, discusses future work and concludes.

Problem 1: *Servers need to guarantee memory safety to guard against trivial buffer overflows.*

Two common techniques to guarantee memory safety found in modern programming languages are *type-systems* and *dynamic bounds checks*. Type-systems prevent programs mis-interpreting data values and allow data to be *encapsulated* behind a set of interface functions. Dynamic bounds checking is a simpler technique which prevents a program accidentally illegally accessing memory outside a finite buffer. Since we have set ourselves an ambitious performance goal for our servers, we eliminate the class of dynamically typed languages such as Java, LISP, Python, Perl etc.

We settled on the use of Objective Caml (OCaml) as a

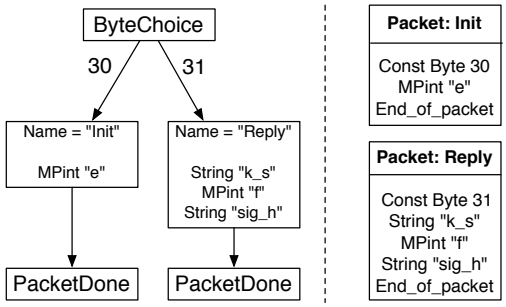

Figure 1. Specification Decision Tree for the SSHv2 Diffie-Hellman Group1-SHA1 key exchange packets

pragmatic language solution. OCaml is a modern variant of the ML family, and offers strong static type checking, type inference, a compact run-time garbage collector with minimal overhead, direct native code generation to a variety of CPU architectures, and a portable byte-code for any platforms with an ANSI C compiler. OCaml's static type system reduces the overhead involved with tracking Run-Time Type Information (RTTI) for variables and functions which dynamically typed languages such as Java or C++ must do.

2 Packet Parsing

Low-level packet parsing has been a constant source of bugs in traditional Internet servers. For example, the SSHv2 protocol specification defines a regular format for packets consisting of a header, sequence of typed fields and random padding. Two popular implementations—PuTTY and OpenSSH—have had errors in their parsing code where they failed to properly check the results of signed/unsigned integer arithmetic leading to heap corruption.

Problem 2: *Server network traffic needs to be parsed safely and consistently, while avoiding run-time performance overhead.*

To overcome this, we define two components: (*i*) a "contract module" which marshals, unmarshals and performs basic operations on low-level packet fields (in the case of SSH, multiple precision integers etc.); and (*ii*) an abstract specification of the parsing process as a *decision tree* which captures informal rules in specification about how sequences of fields should be parsed.

Consider the simple data structure in Figure 1, again using the SSHv2 protocol as an example. On the left is the decision tree whose root node is a `ByteChoice` indicating that the rest of the parsing process depends on the value of the first field interpreted as a byte. Assuming the byte has value 30, the left branch of the tree is followed. The left branch (an "Init" packet) indicates the rest of the packet contains a single field of type `MPint` followed by an end-of-packet marker. The above example was chosen for simplicity; that other parts of the SSH specification require much larger decision trees.

Rather than dynamically interpret the decision tree, we apply meta-programming techniques to generate concrete OCaml code from the specification. This generated code does three things: (*i*) defines one unique data type per SSH packet; (*ii*) defines one pretty-print function per SSH packet; and (*iii*) defines strongly-typed marshalling and demarshalling functions to/from a network stream using efficient ML pattern-matching.

In addition to auto-generating code, the abstract specification helped highlight design inconsistencies in the original SSH specification. For example, SSH packets exist to request functions such as X11 and TCP/IP port forwarding and the specification allows multiple requests to be issued asynchronously. Each type of request has a reply packet with different fields, but none of the reply packets refer to the originating request and the specification allows the replies to come back in any order. This ambiguity was solved by a correction to the SSH draft specification through the IETF SECSH working group, which required that replies be sent in exactly the same order that the requests are received. Out of the hundreds of SSH packets we specified, three such errors were found and corrected in our packet specification.

This technique of generating high-performance statically-typed packet parsing code from decision-trees is applicable to many other common network protocols such as DNS, BGP, and NTP, which have well-defined fields and parsing procedures.

3 Inline Automata

The SSH protocol state machine, like many other real-world protocols, proves to be rather complex as it deals with establishing a secure transport, performing user authentication, and opening up multiplexed, flow-controlled channels to transmit data.

Problem 3: *Internet protocols often have complex state machines which servers must implement accurately, safely and efficiently*.

Our complete SSH server code is too large to formalize without a great deal of effort and complexity [8]; rather, we seek a mechanism which allows critical operations (such as sending and receiving packets) to be abstracted out and analyzed separately. OpenSSH uses the technique of *privilege separation* [10] to provide some level of protection and enforcement of this state machine, by splitting out critical operations into a separate process.

Privilege separation is limited in how fine-grained the messages between the parent and child can be, due to the

overheads of inter-process communication. We observe that the requirement for two processes comes from the fact that the child process cannot make a guarantee of memory safety; that is, errors in the child can result from a malicious attacker overwriting process memory. However, the OCaml type system does provide a guarantee of memory safety, given that foreign function bindings are safe.

Inspired by Schneider's security automata [11], we introduce the notion of *inline automata* and *statecalls*. Inline automata serve two purposes: (*i*) they help protect a server against attack and (*ii*) they can be analysed statically against a set of assertions. We define an OCaml automaton module, exposing a single *tick* function which advances an internal state machine (hidden from the rest of the program by the type system). The *tick* function accepts a single *statecall* argument, which is a special data type representing some action taken by the main program. Statecalls can be defined to an arbitrary granularity, since the only overhead in calling them is a single function call in the program (unlike privsep which requires socket communication).

The introduction of statecalls into the main server source can be done by: (*i*) automatically including statecalls in generated code; (*ii*) using program slicing [3] to introduce statecalls across API boundaries; and (*iii*) manual annotation of statecalls in the server source code. In our implementation of SSH, we modified our packet-parsing decision tree code generator to introduce two unique statecalls per SSH packet (e.g. `ReceiveAuthPasswordRequest` or `TransmitTransportKexInit`). The result is that for every packet transmitted or received, a unique statecall is triggered on the inline automaton. In addition to network packet-related statecalls, there exist "computation statecalls" which are executed by the server when some significant action has been taken. These are currently manually annotated by the programmer, although we plan to use the CamlP4 syntax-extender to provide us with convenient syntactic sugar.

The SSH state machine also has a dynamic element, as channels can be created and destroyed on the fly as part of the protocol. The various protocol layers also operate in parallel; for example, the transport layer can perform key re-exchange during a data transfer. We take advantage of this separation by allowing multiple inline automata to operate simultaneously. The individual automaton can operate through: (*i*) specifically allowing a set of statecalls (i.e. whitelisting); or (*ii*) allowing all statecalls except a specifically banned set (i.e. blacklisting). The utility of dynamically spawning inline automata is extremely useful beyond our SSH implementation—they are of use in multi-threaded servers (an automaton per thread), and in complex protocols allowing multiple aspects of the server to be checked in parallel.

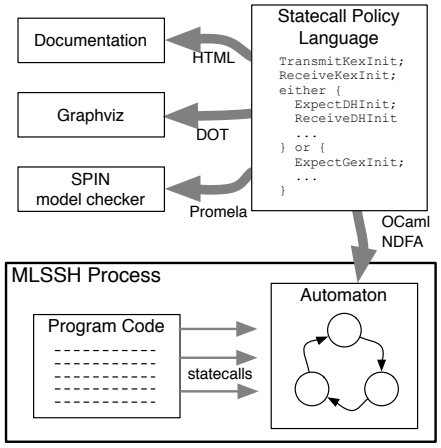

Figure 2. Overall server architecture with statecalls and inline automata specified using SPL.

3.1 The Statecall Policy Language (SPL)

Recall that an inline automaton is a state machine which represents a server protocol in an abstract form. Note that a protocol state machine can be written directly by the programmer with two important drawbacks: (*i*) traditional programming languages lack useful constructs such as non-deterministic choice; and (*ii*) there would be no guarantee that any such state machine was correct.

Problem 4: *A policy language to conveniently express a protocol state-machine and compile it into a non-deterministic finite state automaton suitable for formal analysis is required.*

We define the "Statecall Policy Language": a domain-specific policy language which facilitates the description of readable, maintainable and expressive policies by providing many of the features of a general-purpose programming language (imperative programming style, functions, namespaces) together with new features designed to simplify the specification of protocols.

SPL policies may be regarded as non-deterministic finite state automata, specified in a familiar 'C'-like syntax. Statecalls are represented by capitalized identifiers while semicolons are used for sequencing. Non-deterministic choice is written using a special construct "`either/or`" similar to Occam's `ALT`. Out-of-band messages are handled using a construct called `always_allow` while `multiple` indicates that a block may be repeated zero or more times. Finally, the `during/handle` construct allows for exception and signal handling. Due to space constraints, we cannot formally specify SPL here; instead we devote the remainder of this section with an example of how it can be

applied to the SSH transport layer. Consider the following simplified SPL snippet taken from our SSH server:

```
function transport (bool encrypted, bool serv_auth)
{
  always_allow (Receive_Debug, Receive_Ignore) {
    multiple {
L1:   either {
        Transmit_Transport_KexInit;
        Receive_Transport_KexInit;
        either {
          Expect_DHGroupSHA1;
          Receive_DHGroup1SHA1_Init;
          Transmit_DHGroup1SHA1_Reply;
        } or {
          Expect.DHGexSHA1;
          Receive.DHGexSHA1_Request;
          Transmit.DHGexSHA1_Group;
          Receive.DHGexSHA1_Init;
          Transmit.DHGexSHA1_Reply;
        }
        Transmit_Transport_NewKeys;
        Receive_Transport_NewKeys;
        encrypted = true;
L2:  } or (encrypted && !serv_auth) {
        Receive_Transport_ServiceAuth;
        Transmit_Transport_ServiceAuthOK;
        serv_auth = true;
      }
    }
  }
}

function start_transport()
{
    during {
        transport(false, false)
    } handle {
        Signal.QUIT;
        Transmit.Transport.QUIT;
    } handle {
        Notify.KeyExchangeFailure;
        Transmit.Transport.QUIT;
    }
}
```

The transport layer first exchanges messages to set up an encrypted channel between two hosts. Consider the code at label `L1`. Initially, both client and server exchange key exchange packets which contain the capabilities of both sides. The choice of key exchange algorithm is determined from these packets; a typical choice being Diffie-Hellman with a fixed public group. Afterwards, both sides derive their shared secret, and signal the intent to use it for subsequent traffic via a "new keys" message. After this, the channel is encrypted, and the `encrypted` state variable is set to `true`.

Once the encrypted channel has been set up, the client can then request access to the authentication service. The server must check two things: (*i*) that the channel is encrypted, preventing passwords being sent in cleartext; and (*ii*) that the authentication service is never activated more than once. Complicating matters further, at any time either end can request a renegotiation of encryption key. The code at label `L2` handles these concerns.

At any point the server can receive a UNIX signal (e.g. requesting clean shutdown); this is represented in SPL by the `during/handle` clause in the function `start_transport`.

3.2 Model Checking SPL policies

Protocol specifications often contain informal assertions about the operation of the protocol which the programmer must ponder carefully while writing the server. For example, the SSH protocol specification includes three assertions: (*i*) authentication never begins before the transport is encrypted (*ii*) channels will never be opened before authentication is successful; and (*iii*) a request to open a channel will always result in a success, failure, or connection termination being sent back. Since implementation code is often complex, ensuring these assertions are never violated is a very difficult task. Previous implementations of SSH have had bugs where clients could bypass public key authentication and directly open shells or turn off encryption altogether by sending a server packets it did not expect.

Problem 5: *The programmer needs to be able to formally express high-level assertions about security-relevant behaviour of the program.*

Recall that our SPL compiler converts an SPL policy specification into an OCaml automaton which dynamically checks that the policy is followed. Since an SPL policy is fundamentally a non-deterministic finite state automaton it can also be compiled into a language such as PROMELA, suitable for model-checking. Once in PROMELA, the programmer can specify and mechanically verify (using the model-checker SPIN [4]) assertions about the server written in the temporal logic LTL. Therefore informal assertions from the protocol spec like those mentioned earlier can be written formally and automatically verified. For example, consider the following LTL formula:

$$\texttt{encrypted} \Rightarrow \Box\texttt{encrypted}$$

The formula simply states that if the variable `encrypted` is ever true then it must remain true forever i.e. that encryption can never be turned off.

In addition to the PROMELA output, SPL can also generate DOT graphs of the state machine (see Figure 2), suitable for output as Postscript or HTML. These graphs serve as useful documentation in the style of *literate programming*, allowing the programmer to see how the server behaves without looking directly at any source-code.

4 Related Work

Producing secure software has long been an active focus of research. Much work has focused on ensuring the integrity of existing source code, often written in C. For example, *Systrace* [9] and *Model Carrying Code* [12] are examples of systems which allow policies to be written governing the system call behaviour of existing programs. While these systems have the advantage of working with existing application binaries, we are more interested in building dependable applications from scratch.

Systrace [9] limits the *system calls* (i.e. calls from the application to the OS kernel such as open a file or bind a socket) made by an application. Applications which violate the installed policy may have their system calls blocked or may be suspended or killed. Policies can be created either in advance of execution or may be interactively with the help of the user. The main limitation of systrace is that it operates entirely at the level of system calls, while the system proposed here operates at any arbitrary granularity. For example, SPL policies for an SSH server may operate in terms of actual protocol messages while a systrace policy for the same system will only ever see repeated calls to the system call `read` returning encrypted blobs of data.. SPL is built on our earlier work in system call protection [6]; however, it has been extended with conditional guards and more constructs to facilitate the construction of complex inline automata.

Privilege separation [10] is a design technique which splits a server into two components: a small monitor process running with root privileges and a larger process with everything else. Inline automata and privsep are complementary techniques, since privsep allows for coarse OS-level permissions protections (e.g. process UIDs and chroot), while inline automata offers fine-grained policy enforcement.

The use of statically typed languages such as OCaml to create high-performance servers has also been gaining credence in recent years. The FoxNet project [2] implemented the TCP/IP networking protocol stack by using the Standard ML (SML) type system. However, since their implementation was in user-space, it is difficult to draw performance comparisons against the commonly used kernel-based networking implementations. Web servers have also been written in Haskell [7] and Occam [1]; our techniques such as inline automata take advantage of the underlying safety of these languages to provide even more protection against programmer errors.

5 Future Work and Conclusions

We started this position paper by introducing a challenge: to create a dependable and analyzable server that matches the stringent performance and portability requirements of a current best-of-breed implementation. We decided to implement an SSH server to meet this challenge. Did we succeed? The answer is—almost. Our prototype server implementation currently clocks in transfer rates at around 75% of the speed when using OpenSSH. Of course, our implementation does guarantee protection against exploits such as buffer overflows, integer overflows, and logic bugs which are covered by the SPIN LTL assertions.

In addition, the implementation experience has been valuable, and we have identified two key techniques which significantly sped up our implementation and have much broader appeal across different services. Firstly, the packet-parsing decision tree makes programmers more formally specify how they plan to handle network packets, and automatically generates statically type-safe code to allow for the safe marshalling and unmarshalling of network traffic. Secondly, *inline automata* allow the embedding and enforcement of light-weight state-machines within a server process to an arbitrary granularity and degree of parallelism chosen by the programmer. Finally, our *Statecall Policy Language* allows these automata to be specified using a powerful, expressive, 'C'-like syntax, and also has also shown that it is feasible to model check the automaton by specifying high-level LTL assertions.

Moving forward, we are confident that we can completely succeed at our performance challenge while integrating more formal methods. Interesting areas still to be examined are: (*i*) the effect of garbage collection in a high-performance server; (*ii*) whether SPL can be a complementary solution to privilege separation (by adding in language primitives to automate the privsep process); or (*iii*) combining multiple parallel automata into a theorem prover such as HOL. We expect to release a first version of our SSH server soon under a BSD-style license, and hope to gather feedback from the dependable systems community on further practical techniques we can use to improve the performance, dependability and security of our implementation.

References

[1] F. Barnes. occwserv: An occam web-server. In J. Broenink and G. Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 251–268, Amsterdam, The Netherlands, September 2003. IOS Press.

[2] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in Standard ML. *Higher Order Symbol. Comput.*, 14(4):309–356, 2001.

[3] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[4] G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual title*. Pearson Educational, 2003.

[5] X. Leroy. Objective Caml (OCaml). `http://caml.inria.fr/`.

[6] A. Madhavapeddy, A. Mycroft, D. Scott, and R. Sharp. The case for abstracting security policies. In *International Conference on Security and Management*, June 2003.

[7] S. Marlow. Developing a high-performance web server in concurrent haskell. *Journal of Functional Programming*, 12(4+5):359–374, July 2002.

[8] M. Norrish, P. Sewell, and K. Wansbrough. Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. In *10th ACM SIGOPS European Workshop*, pages 49–53, September 2002.

[9] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington D.C., USA, Aug. 2003.

[10] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, Aug. 2003.

[11] F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[12] R. Sekar, V. Venkatakrishnan, S. Basu, S. B. kar, and D. C. DuVarney. Model carrying code: A practical approach for safe execution of unt rusted applications. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[13] B. Sommerfeld. IETF Secure Shell Working Group (secsh). `http://ietf.org/html.charters/secsh-charter.html`.