# SPLAT: A Tool for Model-Checking and Dynamically-Enforcing Abstractions

Anil Madhavapeddy[1], David Scott[2], and Richard Sharp[3]

[1] Computer Laboratory, University of Cambridge
[2] Fraser Research
[3] Intel Research Cambridge
avsm2@cl.cam.ac.uk, djs@fraserresearch.org, richard.sharp@intel.com

## 1 Introduction

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

SPLAT attempts to simplify the model specification and validation tasks with a view to making model checking more accessible to regular programmers. We provide a high-level modelling language, SPL, which enables developers to specify models in terms of allowable program events (e.g. valid sequences of received network packets). We have implemented a compiler that translates SPL into both PROMELA and a number of general purpose programming languages (e.g. C, OCaml, Java). The generated PROMELA can be used with SPIN [4] in order to check static properties of the model. The generated general purpose code provides an executable model in the form of a *safety monitor*: a program which dynamically checks whether the application's behaviour deviates from the specified model. A developer can link this safety monitor against their application in order to *dynamically* ensure that the application's behaviour does not deviate from the model. If the safety monitor detects that the application has violated the model then it logs this event and terminates the application.

Although this technique simplifies model specification and validation it is, of course, not appropriate for all systems. For example, dynamically shutting down a fly-by-wire control system when a model violation is detected is not an option. However, we observe that there *are* a large class of applications where dynamic termination, while not desirable, is preferable to (say) a security breach. It is these areas in which we believe SPLAT can deliver real benefits.

Our work currently focusses on implementing servers for common Internet protocols securely and correctly. None of the major industrial implementations of protocols such as HTTP (Apache), SMTP (Sendmail/Postfix), or DNS (BIND)

are model-checked by their development teams. All of them regularly suffer from serious security flaws ranging from low-level buffer overflows to subtle high-level protocol errors [2]. In this paper we describe how we used SPLAT in the development of `mlssh`: a complex, high-performance SSH2-compliant server written in OCaml. Our experiences in implementing `mlssh` lead us to believe that SPLAT is accessible to regular programmers without extensive model-checking experience.

## 2 Discussion

To demonstrate the benefits of SPLAT we chose to use it in the development of `mlssh`: an Objective Caml [1] SSHv2 server. SSH, currently being standardized by the IETF [6], is a complex protocol combining transport-level encryption, user authentication, multiplexed data channels and remote shells. We chose to implement `mlssh` in Objective Caml [1], since the strong static-type safety, good UNIX syscall interface, and fast native-code output were all essential to our goals of high performance and portability.

We used the SPL language to specify sequences of network messages allowed by the SSH protocol. SPL policies, which are written using a familiar 'C'-like syntax, represent non-deterministic finite state automata. An SPL automata's inputs are referred to as *statecalls*. In the case of `mlssh`, statecalls are generated when certain packets are received or transmitted, or some significant computation is performed by the server (e.g. deriving a shared secret via Diffie-Hellman key exchange). A simplified fragment of the `mlssh` SPL policy for the transport layer and authentication is shown in Figure 1.

```
automaton transport (encrypted, s_auth) {
  always_allow (Recv_ignore, Recv_debug) {
    multiple (1..) {
      either {
        Recv_kexinit; Xmit_kexinit;
        either {
          Expect_dh;      (... etc)
        } or {
          Expect_gex;     (... etc)
        }
        Recv_newkeys; Xmit_newkeys;
        encrypted = true;
      } or (encrypted && !s_auth) {
        Recv_serv_auth;
        Xmit_serv_auth_ok;
        service_auth = true;
      }
    }
  }
}
```

```
automaton auth (success, failed) {
  do {
    either {
      optional { Xmit_auth_banner; }
      either {
        Recv_auth_req_none;
        Xmit_auth_failure;
      } or
        Recv_auth_req_password;
        auth_decision (success);
      } or {
        Recv_auth_req_publickey;
        auth_decision (success);
      } or {
        Notify_auth_permanent_failure;
        failed = true;
      }
  } until (success || failed);
}
```

**Fig. 1.** Sample SPL fragment for an SSHv2 server

Statecalls are represented by capitalized identifiers, and SPL functions use lower-case identifiers. Semicolons are used to specify sequencing (e.g. `S1; S2`

specifies that state call, `S1`, must occur before state call, `S2`). Non-deterministic choice is represented by using the `either/or` construct. The `always_allow` block specifies out-of-band messages which are expected at any time but do not cause state transitions. The `multiple (1..)` block specifies that its body may occur one or more times, and `optional` allows a block to occur at most once. Although not in this example, SPL also supports a `during/handle` construct that models asynchronous message handling (particularly useful for UNIX signal handling). General recursion is prohibited, allowing us to statically allocate space for function arguments and return values. Internally, the SPLAT compiler transforms SPL into a Control Flow Automaton (CFA) [3] representation.

There are two automata specified in Figure 1: `transport` and `auth`. The `transport` automaton is parameterised over two variables: `encrypted` (representing that the channel has completed an initial key exchange and is encrypted) and `s_auth` (to indicate that the server has enabled the authentication service to the client). The `auth` automaton maintains two state variables to indicate either a successful authentication (`success`) or a permanent failure (`failed`). The `auth_decision` function call has been omitted for brevity.

Informally, the meaning of an SPL program is as follows. Each `automaton` executes in parallel and sees every statecall. If an `automaton` receives a statecall it was not expecting it reports an error. If *any* of the parallel automata report an error then the SPL model has been violated.

To make the SPL more readable, each automaton, $\mathcal{A}$, is surrounded by an implicit `always_allow` block that allows all statecalls not explicitly referenced in $\mathcal{A}$. More precisely, let $\mathcal{S}_{all}$ be the set of all statecalls referenced in the entire SPL policy. Let $\mathcal{S}(\mathcal{A})$ be the set of all statecalls referenced in the definition of automata $\mathcal{A}$. Then $\mathcal{A}$'s implicit `always_allow` block allows statecalls in the set $\mathcal{S}_{all} \setminus \mathcal{S}(\mathcal{A})$.

### 2.1 Executable model

In order to actually use and enforce this model in the target application, the SPLAT compiler outputs a safety monitor designed to be linked directly against the source code of the server. The safety monitor requires that the rest of the server program cannot compromise its internal state. If this were not the case then an attacker could (say) exploit a buffer overflow to manipulate the control-flow of the safety monitor. In the case of `mlssh`, the SPLAT compiler generates OCaml code for the safety monitor. The interface to the safety monitor is generated as follows:

```
module Automaton = struct
  exception Bad_statecall
  type statecall =
    |Xmit_ignore |Xmit_debug |Recv_kexinit |Xmit_kexinit (... etc)
  type state
  val init : unit -> state
  val tick : statecall -> state -> state
end
```

This interface allows `mlssh` to initialize the safety monitor (via `Automaton.state`), and to drive it by calling `tick`. If the safety monitor ever receives an invalid sequence of statecalls (passed via `tick` calls) then it generates the `Bad_statecall` exception, terminating the program.

Although we specifically describe an OCaml interface here, the compiler can also be easily extended to other language's type systems (e.g. Java objects), allowing server authors to write programs in their language of choice and still use the SPLAT tool-chain. In the case where languages do not make strong enough memory-safety guarantees to protect the safety monitor from the main program (e.g. C), the compiler outputs an automaton which runs in a separate UNIX process [5] and stub code which allows the server to communicate with the monitor via IPC. However, this approach is slower for more fine-grained SPL policies, as there is significantly more overhead in performing IPC than simply calling a function (as is the case with OCaml).

It is important to note that we do not enforce mechanisms for insertion of `tick` calls in server applications. In recent years, there have been a proliferation of languages being used for authoring Internet services (Python, Perl, Ruby, Erlang, Java, Eiffel, C/C++, Objective C etc.). Since different languages present such different mechanisms for writing servers, any such technique would be either too language-specific or too general to be of use. Instead, we allow SPL automata to be embedded into any of these languages and allow server authors to insert `tick`s as they see fit. In our implementation of `mlssh` we used a combination of: (*i*) manual `tick`-insertion within the server source; (*ii*) meta-programming techniques to automatically introduce `ticks` into generated code used for low-level packet parsing; and (*iii*) program slicing to automatically call `tick`s across API boundaries (e.g. into the cryptographic functions).

### 2.2 Promela Output

The SPL compiler transforms the SPL policy directly into PROMELA code, suitable for machine-checking using SPIN. In addition, message producer processes are created which continuously transmit statecalls to each automata. Model-checking this output is not very interesting beyond exhaustiveness testing to check that all the states are reachable. However, the programmer can specify additional LTL assertions which must hold for the SPL policy as a whole. These LTL assertions are checked by SPIN, and any counter-examples are translated by the compiler back into SPL line numbers (which were added to each state in the CFA by the compiler). In the code snippet shown previously, some LTL assertions used are: (*i*) `encrypted` $\Rightarrow \Box$`encrypted` and (*ii*) `s_auth` $\Rightarrow \Box$`encrypted`. These indicate that when encryption is enabled, it must remain enabled for the lifetime of the session. Similarly, when the authentication service is activated, encryption must always be enabled from then on.

Although the LTL assertions are currently specified outside of the SPL specification, we are currently integrating support for the assertions into the SPL grammar, thus allowing them to be dynamically enforced in the executable au-

tomaton in the event that a model-checker is not available in the local build environment.

The LTL assertions provide the programmer with a very useful mechanism for determining whether the often informal intuitions about their server actually hold true in their SPL model. For instance, UNIX signal handlers are a common source of errors due to their extremely asynchronous nature; since they can easily be expressed in SPL via a `during/handle` clause, the programmer can include them in LTL assertions and more formally check those intuitions.

## 3 Tool Demonstration

In our tool demonstration, we intend to show the SPLAT tool-chain as used in the `mlssh` SSHv2 protocol server, as well as the executable code output in other languages such as C where type-safety is not as strong. In addition, we will show how LTL assertions help the programmer enforce invariants that are currently informally mandated by the SSH specification. Finally, we hope to gather feedback from the model-checking community, as we plan to release SPLAT under a BSD-style license to encourage broader open-source adoption of model-checking techniques across multiple languages and coding styles.

## References


1. X. Leroy at al. Objective Caml. `http://caml.inria.fr/`.
2. CERT Coordination Center (CERT/CC). CERT knowledgebase. `http://www.cert.org/kb/`.
3. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, pages pp. 526–538. Lecture Notes in Computer Science 2404, Springer-Verlag, 2002.
4. Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual title*. Pearson Educational, 2003.
5. Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.
6. Bill Sommerfeld. IETF Secure Shell Working Group (secsh). `http://ietf.org/html.charters/secsh-charter.html`.