Abstract

We present a networked key-value server, implemented in C and formally verified in Coq. The server interacts with clients using a subset of the HTTP/1.1 protocol and is specified and verified using interaction trees and the Verified Software Toolchain. The codebase includes a reusable and fully verified C string library that provides 17 standard POSIX string functions and 17 general purpose non-POSIX string functions. For the KVServer socket system calls, we establish a refinement relation between specifications at user-space level and at CertiKOS kernel-space level.

1 Introduction

The Science of Deep Specification launched a series of experiments in formal verification of real-world systems [3]. Among these, Koh et al. [25] demonstrated how to verify a simple networked server (called a “swap server”), written in C, against an implementation model written with interaction trees [42], using the Verified Software Toolchain (VST) [4]. The
main result was a guarantee that any trace of observed external communications with the server is included in an interaction tree describing intended server behavior.

In the work described here, we scale these verification techniques to a more realistic example: a Key-Value server (KVServer) running over a minimal but practical subset of the HTTP/1.1 protocol. The KVServer provides clients with a $\text{Get}(\text{key})$ and $\text{Put}(\text{key}, \text{value})$ interface that uses HTTP/1.1 features including GET requests, POST requests, and Content-Length encoding. It runs on the verified operating system CertiKOS [18] or any other OS with a POSIX-compatible socket interface.

Besides significantly scaling up server features and protocol complexity, the present work reduces the set of trusted axioms compared to that of Koh et al. [25]. The network interface of the earlier swap server is described by a set of Hoare triples for the socket system calls, which are assumed to be satisfied by the host operating system. In this work, we apply recently developed techniques for proving refinement relations between CertiKOS' kernel-level system call specifications and user-level VST system call specifications, and prove that network interface assumptions from [25] are consistent with CertiKOS' system call specifications [32]. Such proofs are necessary because the specification styles of VST and CertiKOS are different enough that it is not obvious that two specifications describe the same behaviors. One significant difference stems from the logics used by VST and CertiKOS. Another distinction is in their representations of the system state and the external effects of socket operations. The user-level VST socket specifications use interaction trees to describe external effects as observed “on the wire”. The CertiKOS specifications, on the other hand, capture these external effects in a logical log of events, while also describing the internal effects on the kernel state, which are invisible to the user-level code.

By proving refinement between the VST and CertiKOS models of socket system calls, we demonstrate that the kernel- and user-level specifications agree on the behavior of sockets. The kernel implementation of the socket system calls in CertiKOS is currently unverified with respect to CertiKOS socket specifications. Our work does not attempt to fill this gap (which requires modeling and verifying a TCP/IP stack), but instead proves a refinement between the CertiKOS and VST socket specifications. This guarantees at least that the operating system and the server agree on how sockets are expected to behave, thus removing this interface from the trusted computing base and leaving only the kernel’s implementation.

The main challenges in developing the new KVServer stem from the significant increase in feature complexity across all levels of the server. At the connection management level, the KVServer needs verified data structures to maintain incoming and outgoing buffers for multiple concurrent connections. At the protocol level, the KVServer requires a verified parser to deserialize HTTP/1.1 requests and a verified printer to serialize HTTP/1.1 responses. The parser and printer depend on a verified C string library. At the application level, the KVServer needs a verified in-memory map for storing key-value pairs.

This blow-up in feature complexity also calls for a modular approach that can (1) contain the implementation and verification complexity within each module and (2) reduce total proof checking time through parallelization. We achieve this by dividing the entire KVServer into eight independent verified modules. Each module comes with VST specifications for all exposed functions. A module that depends on a lower-level module only needs the lower-level module’s C API and VST specifications, while implementation and proof complexity remain hidden away. This modular separation of the KVServer also produces general purpose and reusable low-level modules. Furthermore, we keep function specifications separate from code and avoid intermingling of code and proof information (e.g., loop invariants), as the latter is typically specification-dependent. This organization sets our development apart from the methodologies of verification tools such as Frama-C, VeriFast, KeY, and Dafny [23, 22, 28, 1] and enables us to distribute the verification of individual function bodies into different files.
that can be processed in parallel. We discuss related verification projects in more detail in Section 6.

Our main contributions are:
1. We demonstrate a verified network server, implemented in C and communicating using a subset of HTTP/1.1. The termination-insensitive specification and proofs are formulated using VST and interaction trees (Section 3).
2. We prove that the network operations in KVServer interaction trees correspond directly to I/O operations performed by the operating system. We use the verification methods of Mansky et al. [32] to demonstrate a refinement between two disparate specifications of the socket interface written in two different specification languages, by abstracting the lower kernel-level CCAL specification on kernel socket states and the logical log of socket operations into a higher user-level VST specification on externally observable network effects. Our work is the first to formally bridge the gap between user- and kernel-level specifications of POSIX network operations (Section 4).
3. We present a reusable and fully verified C string module offering 34 verified C string functions. 17 of these belong to the POSIX string library; the rest are general-purpose string functions used by the KVServer (Section 5).

2 Background

2.1 Interaction Trees

Interaction trees (ITrees) are a data structure and an accompanying Coq library for representing and reasoning about effectful and potentially non-terminating programs [42]. A significant part of the verification work for KVServer involves proving that the server performs various socket system calls in an expected way. We streamlined this effort by using ITree programs to specify the socket-level behavior of KVServer.

To write an ITree program, one must first define the set of visible effects the program can perform. In KVServer, the socket-level network effects are modeled by the following Coq datatype.

```coq
Variant networkE : Type -> Type :=
| Listen : endpoint_id -> networkE unit
| Accept : endpoint_id -> networkE connection_id
| Shutdown : connection_id -> networkE unit
| RecvByte : connection_id -> networkE byte
| SendByte : connection_id -> byte -> networkE unit.
```

Each constructor describes a network effect parameterized by its argument types (the parameters to the constructor) and its result type (the type argument to networkE at the end of each line). For example, the Listen constructor represents the server beginning to listen for incoming client connections on an endpoint_id (a network address identifier). This operation does not return any meaningful data so its return type is unit. We specify how this effect corresponds to the POSIX listen system call in the C program in Section 2.3.

ITree programs may involve multiple sets of effects. For example, runtime errors can be expressed in ITree programs through the exceptE effect type from the ITree library...

```coq
Variant exceptE (Err : Type) : Type -> Type :=
| Throw : Err -> exceptE Err void.
```

... and we can use the binary operator +’ to create a larger effect type: networkE +' (exceptE string) is a composition of two kinds of effects, which together allow an ITree program to perform socket effects and throw string-valued errors.
The ITree library also provides a mechanism for expressing effect subsumption relations between effect types, leveraging Coq’s typeclass mechanism to automatically resolve subsumption constraints. This mechanism can be used to automatically “lift” a concrete, monomorphic effect type into a polymorphic one. For example, we can use it to define a helper function that generalizes the \texttt{Listen} effect:

\begin{verbatim}
Definition listen `{networkE -< E} : endpoint_id -> itree E unit := embed Listen.
\end{verbatim}

Here, the effect type \texttt{E} represents some possibly larger set of effects satisfying the subsumption relation \texttt{networkE -< E}, and \texttt{embed} is an ITree library function that performs the lifting from \texttt{networkE} into the subsuming type \texttt{E}. The signature of \texttt{listen} says that it is a function that receives a network identifier and produces an ITree of type \texttt{itree E unit} that, intuitively, calls the “listen” kernel function and returns the unit value once the effect completes.

ITrees satisfy the interface of monads [42], a standard mechanism for composing effectful programs in a pure functional programming context. The monad interface consists of two combinators:

\begin{verbatim}
Definition bind `{Monad m} {a b : Type} : m a -> (a -> m b) -> m b.
Definition ret `{Monad m} {a : Type} : a -> m a.
\end{verbatim}

Intuitively, the \texttt{bind} combinator builds a computation that runs the effectful computation in its first argument (with type \texttt{m a}) to produce a result of type \texttt{a}, passes the result to the continuation in the second argument (with type \texttt{a -> m b}), and returns an effectful computation with type \texttt{m b}. The \texttt{ret} combinator injects an effectless value of type \texttt{a} into a computation that may have effects. ITree programmers can use these two combinators to compose ITree values; for convenience, the ITree library provides the notation \texttt{a <- m;; k} for the expression \texttt{bind m (fun a => k)}.

2.2 Verified Software Toolchain

The Verified Software Toolchain [4] is a Coq framework for verifying C programs using concurrent separation logic [37, 35]. To verify a piece of code, a user employs Coq’s programming features to define assertions, connects partial-correctness specifications to function definitions in CompCert’s Clight program representation, and finally applies forward symbolic execution tactics to verify the corresponding function bodies. For readability, specifications in this paper are presented in informal notation rather than in VST’s Coq-based syntax.

As an example, consider this string library function, which allocates space for a string of a given length.

\begin{verbatim}
unsigned char* new_string(uint32_t len);
\end{verbatim}

Drawing upon predicates \texttt{Mem} and \texttt{Mtok} from VST’s verified malloc/free library [5], this function’s specification

\begin{verbatim}
{ { 0 <= max _unsigned } \land \langle Mem gv \} } new_string(l) \begin{cases} \{ p, if p = null then Mem gv \} \land CHStringN(Ews, [], l + 1, p) \land Mtok(Ews, uchar[l + 1] = p) \land Mem gv \} \end{cases}
\end{verbatim}

asserts that the result \texttt{p} of a call to \texttt{new_string} (with a suitable argument \texttt{l}), is either null or is a pointer to some fresh region of memory that satisfies the predicate \texttt{CHStringN}. The “deallocation token” \texttt{Mtok} represents the fact that the client not only gains read/write access (represented by the exclusive-write-share \texttt{Ews}) to the freshly allocated region but may also free it. In addition to these predicates – whose precise definitions we elide – the specification makes use of VST’s operators for separating (\texttt{\ast}) and ordinary (\texttt{\&\&}) conjunction, and an
operator $!!$ that injects a pure Coq proposition into VST’s category of assertions. Finally, $\text{uchar}_n$ is a shorthand for a length-annotated specialization of CompCert’s representation of the function’s return type when interpreted as an array. Note that the malloc/free library assertion $\text{Mem}\_\text{M}$ must be present but is not modified by the call, and that no fresh memory is allocated if the return value is null.

2.3 Specifying Effects with VST and ITrees

We not only use VST for verifying memory safety and application logic, but also rely on a combination of VST and ITrees to verify externally observable effects of C code.

Consider the $\text{listen}$ system call from the POSIX socket API:

```
int listen(int sockfd, int backlog);
```

Recalling the lifted $\text{Listen}$ network effect from Section 2.1, we specify the $\text{listen()}$ system call using two abstract predicates. The predicate $\text{ITREE } t$ states that effects described by the interaction tree $t$ are included in the overall effects exhibited by the host OS. The predicate $\text{SOCKAPI } st$ states that the host OS has sockets in states corresponding to $st$, a map from socket identifiers to states (bound, open, listening, etc.).

The specification of $\text{listen()}$ quantifies over two ITrees, $t$ and $k$, that respectively describe the sequence of effects performed by the KVServer before and after running this $\text{listen()}$ network call.

```
{ !!((\text{listen } addr; k) \sqsubseteq t) \& \& \text{ITREE } t * \text{SOCKAPI } st} \\
\text{listen}(fd, backlog) \\
\{ \forall r. \text{EX } t'. !((-1 \leq r \leq 0 \wedge \text{post\_listen } t k st fd addr r t' st')} \\
\& \& \text{ITREE } t' * \text{SOCKAPI } st' \}
```

Informally, the precondition says that the observed effects in $t$ must first be a $\text{listen}$ effect, followed by effects observable in $k$, and that $st$ is the current internal state of the sockets being managed by the kernel. We think of the tree $t$ as permission to perform certain sequences of external operations, from the perspective of an observer that checks off operations one by one as they are performed by the program. The ITree value $k$ is a continuation that models effects following this $\text{listen}$ effect. The relation $\text{post\_listen}$ specifies how $t$ and $st$ evolve to the ITree value $t'$ modeling the remaining observable effects and the updated socket state $st'$, depending on whether the $\text{listen()}$ system call succeeds. Specifically, if the $\text{listen()}$ system call fails ($r = -1$), then $\text{post\_listen}$ states that $t'$ is the same as $t$. (The specification implies that the actual side effect of $\text{listen()}$ does not occur when the system call fails, and this design leads to a more straightforward connection to the CertiKOS socket specification compared to some alternative designs. We discuss this detail in Section 4.) Note that $\text{post\_listen}$ is purely propositional: it is independent of the memory.

Formally, the predicates $\text{ITREE}$ and $\text{SOCKAPI}$ are defined as assertions on the external ghost state of VST [32], which is kept in sync with a piece of external state in the OS. In this case, the external state is the log of socket communications and the set of currently active sockets; Section 4 will detail how this ghost state is related to CertiKOS’ kernel state.

2.4 HTTP/1.1

HTTP/1.1 is a standard network protocol that allows a client (e.g., a web browser) to access and modify resources (e.g., HTML files, databases) stored on a remote server. A client
initiates the communication with a request formatted as: (1) a request line consisting of a method (e.g., GET, PUT, POST) indicating the desired action, and the resource on which to perform it; (2) a sequence of header fields that specify extra options; (3) a blank line; and (4) an optional body whose meaning depends on the request line. After handling the request, the server responds with a message comprising (1) a status line that includes the numeric status code (e.g., 404) and a textual message (e.g., “Not Found”); (2) a sequence of header fields with additional information; (3) a blank line; and (4) an optional body [16].

The full HTTP/1.1 specification includes nine methods and many possible headers whose effects range from setting the acceptable language of the response to specifying compression and caching behaviors. In practice, though, only a relatively small subset is necessary for common operations such as retrieving or updating resources: in particular, the only methods the KVServer needs to implement are GET for looking up keys and POST for updating or inserting key-value pairs, and the only header that the server needs to recognize is Content-Length, which indicates the size of the message body. The following is a sample request-response pair for a successful retrieval of the key-value pair $\texttt{foo} \mapsto \texttt{bar}$. The $\leftarrow$ symbol represents an ASCII carriage return and line feed (CRLF) sequence.

```
GET /foo HTTP/1.1

HTTP/1.1 200 OK
Content-Length: 3

bar
```

Though lean, this subset is sufficient to build a non-trivial application and demonstrate the effectiveness of our methodology. Two real-world webservers\(^1\) also implement just this lean subset of HTTP with only GET and POST support.

## 3 Components

The implementation of our HTTP server is divided into eight verified modules. We verify memory safety and (termination-insensitive) functional correctness of each.

### 3.1 Infrastructure Modules

**StringAPI.** The KVServer presents a string-based key-value store over HTTP, and its implementation uses C strings throughout. Our lowest-level infrastructure module is therefore a reusable verified string library with implementations and specifications of many common string functions, plus some useful variants. Due to the details of C memory semantics, idiomatic C string programming introduces proof obligations for side conditions that programmers typically gloss over. We therefore provide alternative implementations and specifications for commonly used string functions to hide these proof obligations from dependent modules. Section 5 describes the string library in more detail.

**BufferAPI.** This module provides a dynamically allocated resizable byte-buffer data structure. These byte buffers are used to both accumulate data received from clients and to construct data to be sent to clients. We provide verified functions to create, resize, append to, and deallocate byte buffers. Our specifications assert that BufferAPI operations allocate and deallocate memory correctly and do not perform any invalid memory reads or writes.

---

SocketAPI. This module provides the VST specifications of POSIX system calls for creating sockets, binding sockets to network addresses, accepting connections, writing and reading data on sockets, and learning which sockets are ready to read or write. This module bridges the CertiKOS kernel and the rest of the KVServer by establishing refinement proofs of the CertiKOS socket specifications against the VST socket specifications. Since these socket operations have network effects that are observable from outside the KVServer, their VST specifications describe the effects they may trigger using the technique introduced in Section 2.3.

The earlier “swap server” described by Koh et al. [25] had similar specifications for the network operations used by our KVServer. However, the specifications of network operations in that work did not have a refinement relation with the CertiKOS socket specifications. In our work, we push this verification boundary lower into CertiKOS with the SocketAPI refinement proofs. We discuss this improvement in Section 4.

3.2 Application Modules

ParseAPI. This module provides functions that parse the subset of HTTP/1.1 requests accepted by the KVServer and functions that serialize standard HTTP responses for KVServer clients. The top-level specification for the parser is a relation between the input string, the parsed request, and the remaining unused input string. This parser specification describes how the parsed request and the unused portion of the input string can be reassembled to recover the original input string. Specifically, the subset of HTTP/1.1 requests that the KVServer accepts are GET and POST requests, and POST requests must have a string as payload in the Content-Length encoding (the length of the string payload must be equal to the value in the Content-Length header).

We also develop an executable parser in Gallina, Coq’s internal functional programming language, for the subset of requests KVServer supports, plus refinement proofs between the VST specification of the C parser and the Gallina parser, showing that the C parser is a refinement of the pure Gallina parser.

KeyValueAPI. This module is a thin wrapper around the ParseAPI module that interprets HTTP requests to the KVServer as read/write operations on the KeyValue storage. An HTTP GET request at some \texttt{url} is translated into a read request to the KeyValue storage, with the key being the specified \texttt{url}. Similarly, an HTTP POST request at some \texttt{url} with some \texttt{payload} is a write request that puts the value of \texttt{payload} under the key \texttt{url}. The KeyValueAPI specifications assert that GET and POST requests are correctly translated into key-value read and write requests.

HashtableAPI. This module implements a hash table for string-valued keys and values, which acts as the in-memory KeyValue storage. This module uses a pure Gallina implementation of a string-valued hash table as its specification; we establish the refinement between the Gallina implementation and the C implementation using VST. The hash table uses a verified hash function that computes the hash value of a string by arithmetic manipulations on the ASCII values of the characters in the string. The HashtableAPI specifications assert that the C hash table implementation strictly follows the Gallina implementation model.
3.3 Server I/O Modules

ConnectionAPI. This module pulls in both the application modules and the infrastructure modules and provides an interface for managing and communicating with logical client connections.

The KVServer is a single-threaded event-driven server. The event-driven I/O model allows a single-threaded server to manage multiple concurrent connections. Under this scheme, the server repeatedly tries to receive network data in a loop, buffers available network data from all clients and checks the client buffers for pending requests in each loop iteration.

ConnectionAPI uses the BufferAPI module to maintain both incoming and outgoing byte buffers for each connection. It also relies on the ParseAPI module to abstract over the underlying byte stream by repeatedly trying to parse the accumulated bytes in the receiving buffer until a complete HTTP request has been parsed. This request is then interpreted as a KeyValue request by the KeyValueAPI module, and the corresponding KeyValue operations are executed on the hash table storage.

ConnectionAPI uses a linked-list data structure to manage a collection of connections. Furthermore, ConnectionAPI abstracts over select and provides an interface for focusing on connections ready for network I/O.

ServerAPI. This module implements the main event-loop of the entire KVServer. The main loop relies on operations from the ConnectionAPI module to focus on connections that have pending requests to be processed, performs the operations encoded in these pending requests, appends the serialized responses to the outgoing buffers for relevant connections, and flushes outgoing buffers for connections that are ready for outgoing communication. The ServerAPI specification asserts that the main loop does not cause the connection data structures to become invalid and that the hash table storage’s content is updated correctly upon client requests.

The ServerAPI module exposes a top-level VST specification for the main() function of the entire server and relates the C implementation of the server’s main loop to its interaction-tree specification.

```
!!(consistent_world st) &&
\{
  (ITREE ITree.iter (run_server) ([], empty_table); tt k) *
  SOCKAPI st * MemM gv
\}
main()

{st'. ITree k tt * SOCKAPI st' * MemM gv}
```

This states that, starting from a valid state of OS sockets modeled by st, the effect of running the main() function is reflected by the interaction tree that iterates the server specification run_server in a loop, and that the loop starts with the initial empty state ([], empty_table). The empty list is the initial empty list of client connections, and the empty table is an initial empty key-value storage table.

The ITree function run_server is the specification of a single event-loop iteration of the main server. This iteration is repeated using an ITree combinator ITree.iter. The composed specification ITree.iter run_server is a program that runs forever, unless the server encounters an irrecoverable error (e.g., memory allocation failure). The proof of this VST triple uses data structure invariants for the hash table and connection list and relies on lemmas that prove each server operation preserves these data structures’ invariants. However, these proof details need not be exposed in the top-level VST specification.
### Summary

To give an idea of the scale of the KVServer and its verification, Figure 1 lists some of the important functions from each module along with the number of lines of C code and Coq specification and proof each required. Figure 2 compares the total number of lines to the earlier Swap Server [25] to highlight the significant increase in scale.

### 4 Socket API

#### 4.1 Connecting VST Specifications to CertiKOS Socket Calls

The KVServer communicates with clients using POSIX socket system calls: \texttt{bind}, \texttt{accept}, \texttt{send}, \texttt{recv}, and so on. These system calls are provided by the verified operating system CertiKOS, which includes functional specifications (either verified or axiomatized) for each system call. Using a technique due to Mansky et al. [32], we connect the VST specifications (separation logic pre- and postconditions) for the socket calls to the Certified Concurrent Abstraction Layers (CCAL) specifications of socket calls provided by CertiKOS, strengthening our confidence in the correctness of our server’s network communication.

The basic approach of Mansky et al. [32] is shown in Figure 3. We connect VST specifications of each call to CertiKOS by means of an intermediate-level first-order predicate on CompCert memories (maps from memory locations to values) and external state. Mansky
et al. refer to this intermediate-level specification as a “dry specification”. The dry specification serves as a translation layer between the corresponding VST specification and CertiKOS specification; this allows us to prove a round-trip theorem stating that the VST specification follows from the guarantees provided by CertiKOS. For instance, recall the VST specification of the listen system call from Section 2.3. The CertiKOS specification for listen is:

```lean
Definition listen_spec (abd : RData) (fd : Z) : option (SysRet Z) :=
if negb (kern_init abd) then None else
let socks := ZMap.get (curid abd) abd.(sockets) in
let log := ZMap.get (curid abd) abd.(socket_log) in
match is_bound (ZMap.get fd socks) with
| Some port =>
| let socks' := ZMap.set fd (ListeningSocket port) socks in
| let log' := SysSockListen port :: log in
| Some (abd {sockets: ZMap.set (curid abd) socks', abd.(sockets)})
| {socket_log: ZMap.set (curid abd) log', abd.(socket_log)},
| OK
| _ => Some (abd, ERR EBADF) (* Invalid socket state *)
end.
```

This specification mentions various pieces of OS state that are invisible to the C programmer, including a record of the state of the sockets (i.e., `abd.(sockets)`) that is modified during the call and a log of socket operations performed (i.e., `abd.(socket_log)`). The OS socket states should correspond to the SOCKAPI in the VST specification, while operations appended to the log should be reflected in events removed from the ITREE. We connect the two layers by writing a dry specification for `listen`, in which the assertions of the VST specification are converted to first-order predicates on memory and external state. The dry pre- and postcondition take the parameters of the VST specification as arguments, along with the memory `m`, external state `z`, and – in the case of the postcondition – the initial memory `m₀` and return value `r`. They capture the requirements on the external state and reflect the fact that `listen` has no effect on user memory:

```plaintext
Pre((t, k, st, addr, fd, backlog), m, z) ≜ (listen addr; ; k) ⊑ t ∧ z = (t, st) ∧ st fd = BoundSocket addr
Post((t, k, st, addr, fd, backlog), m₀, m, z, r) ≜ m = m₀ ∧ ∃ t' st'. z = (t', st') ∧ consistent_world st' ∧ −1 ≤ r ≤ 0 ∧ post_listen t k st fd addr r t' st'
```

In particular, note that the ITREE and SOCKAPI predicates are no longer present, and their contents are translated into assertions on the external state `z`. We then complete
the refinement by showing that the VST precondition implies Pre, that Post implies the VST postcondition, and that if Pre is true of the user memory and external state given to the CertiKOS listen_spec, then Post is true of the corresponding parts of the output – thereby translating the assertions on \( z \) to effects on \( \text{abd.}(\text{sockets}) \) and \( \text{abd.}(\text{socket_log}) \), the OS-level representation of sockets and communication.

The server implementation uses the system calls socket, bind, listen, accept, send, recv, close, shutdown, and htons. For each of these, we prove a connection between the VST proof rule used in the verification of the server and the CertiKOS axiom for the call. The proofs follow the pattern of prior work. Most of the calls only affect external state (socket state and/or interaction tree), while send and recv also use or change the contents of a single buffer in memory; both of these patterns were illustrated by Mansky et al. [32]. These refinement proofs significantly strengthen our confidence in the correctness of the server by removing the VST socket specifications from the trusted computing base and replacing them with the abstractions provided by the operating system. Indeed, while carrying out the refinement proof, we discovered some correctness conditions that were missing from the original VST specifications. For instance, the bind call is only guaranteed to return a valid result when the provided port number is between 0 and 65535; the original VST specification omitted this range requirement.

### 4.2 Granularity of ITree Events

The VST specification we used for the listen call looks like this:

\[
\{!!((\text{listen addr};, k) \sqsubseteq t) \&\& \\text{ITREE} \ t \star \text{SOCKAPI} \ s t\} \\
\text{listen}(f d, \text{backlog}) \\
\{ \\
\quad r, \text{EX} \ t', s t'. \text{!!}((-1 < = r < = 0 < \text{post}_{-}\text{listen} \ t \ k \ s t \ f d \ a d d r \ r \ t' \ s t') \\
\quad \&\& \\text{ITREE} \ t' \star \text{SOCKAPI} \ s t' \\
\}
\]

Most of the details of what listen actually does are hidden in post_listen, which says that either the call succeeds and \( t' \) is the continuation \( k \) (i.e., \( t \) minus the listen event), or the call fails and \( t' = t \). In other words, the listen event in the ITree represents a successful call to listen, and on failure the server must retry the call before moving on to \( k \). This is only one possible approach to representing communication events with an ITree: we could imagine using the listen event to represent an invocation of the listen system call, successful or not, or even a permission guaranteeing that, if the server calls listen at this point, then the call will succeed. In the former case, the event would be removed from the ITree regardless of the result; in the latter case, the error result (and the return value of \(-1\)) would not appear in the specification at all.

In the swap server of Koh et al. [25], it was (somewhat arbitrarily) decided that events should represent successful communications, leading to the current style of specification. Now that we have connected the ITree to the operating system’s log, this choice is no longer arbitrary: each ITree event corresponds to exactly one socket_log event, and the OS does not add an event to its log when the call fails. We could build and validate specifications in the other styles (by writing a wrapper function that calls listen until it succeeds, or by providing a token from the OS that somehow guarantees that the next listen will not fail), but our specification style leads to the most direct translation of the CertiKOS specification into VST. If a user writes a program that assumes listen will always succeed and tries to verify it using VST, the gap between their assumptions and the guarantees of the OS will show up in the verification.
C Strings in VST

A C string is a contiguous array of non-zero unsigned bytes (1-255), terminated by a null (0) byte. To avoid confusion between a C string value and a Coq string value, we write C strings here with array notation. For example, ['K', 'V', 'S', 'e', 'r', 'v', 'e', 'r', '\0'] is a C string that can be modeled by the Coq string "KVServer".

Programs manipulate C strings through pointers to these byte arrays. For example, our implementation of the standard function strstr() takes two pointers of type const unsigned char *, representing a “haystack” and a “needle,” and searches for the needle in the haystack.

    const unsigned char* strstr(const unsigned char* hstk, const unsigned char* ndl);

Two key properties of a C string are its contiguous memory layout and its terminating null byte; violating these can cause unexpected behaviors and subtle memory bugs [43, 14]. For example, if hstk is not null-terminated, strstr may read beyond the allocated memory region, possibly leaking secret information or crashing the program.

We use two VST predicates CUStringN(sh, s, n, p) and CUString(sh, s, p) to model C strings. The predicate CUStringN is defined in terms of an access-permission share, the list of bytes in the string s, and a pointer p to an array of size n. CUStringN(sh, s, n, p) states that:
1. the list of bytes s does not contain a null byte;
2. the length of s is strictly smaller than the array’s size (n);
3. the pointer p points to a contiguous memory region that starts with the contents of s;
4. the pointer p points to a value with the type uchar (i.e., an array of n unsigned bytes); and
5. the contents of s are immediately followed by a null byte.

The leftover space may hold arbitrary data. The share parameter sh controls whether read or write accesses are allowed on the memory where the string is located. In KVServer, we use the share values Ews and Tsh, which respectively mark heap-allocated read-write memory and stack-allocated read-write memory.

We then further refine CUStringN with CUString by requiring that the length of the array at the pointer p is exactly the length of s plus 1 (the extra byte is for the terminating null). For example, the C string ['K', 'V', 'S', 'e', 'r', 'v', 'e', 'r', '\0'] allocated at pointer p with both read and write permissions can be specified, in Coq, as CUString(Ews, "KVServer", p).

5.1 Specifying strstr

To specify a string function like strstr, we need to formally describe two parts:
1. Memory safety: the memory-layout assumptions that the function makes about its inputs.
   In this case, strstr only requires both inputs to be valid C strings.
2. Functional correctness. In this case, if hstk contains ndl, then strstr (either diverges or) returns a pointer to a substring of hstk whose prefix is ndl. Otherwise, strstr returns NULL.

For functions that return a substring of one of their arguments, the convention in the standard C string library is to return a pointer at some offset from the input. For example, when strstr succeeds, it returns a pointer at some offset i into the haystack C string. However, in many instances (especially when working with constant strings), we are primarily interested in the index at which the substring begins, rather than the substring itself.
In principle, the returned pointer from `strstr` implicitly encodes a non-negative offset into the haystack string where the needle string can be found: if the haystack string is at pointer `p` and the returned pointer is `n`, then `offset = n - p`. Although this offset is trivial to compute, it adds proof obligations to convince VST that the arithmetic uses only well-defined operations according to the CompCert memory model. While programmers usually think of memory in C programs as a big array of data indexed by memory addresses, and while memory addresses can obviously be subtracted from one another to compute the offset between them, the C standard as reflected in the CompCert memory model is more structured. In VST, memory regions allocated by different calls to `malloc` are considered disjoint, and it is undefined behavior to take a pointer `p_A` that points to region `A` and add an offset `x` to `p_A` such that `p_A + x` points to a separate memory region `B` starting at some pointer `p_B`, even if arithmetically `p_A + x = p_B` [29].

Thus, a pointer subtraction like `p_1 - p_2` induces an extra proof obligation that `p_1` and `p_2` point to memory addresses within the same memory region. Users of `strstr`'s specification must deal with such proof obligations if what they really want is the offset. Since computing the offset is indeed a common pattern throughout KVServer, we provide an alternative “indexed” version of `strstr` called `strstr_idx` that packages up the pointer subtraction proof and directly returns the offset.

```c
int strstr_idx(const unsigned char* hstk, const unsigned char* ndl) {
    const unsigned char* s = strstr(hstk, ndl);
    if (s == NULL) {
        return -1;
    }
    int i = s - hstk;
    return i;
}
```

The specification of `strstr_idx` also reflects some logical simplifications that come with working with offsets instead of pointers:

```latex
\begin{align*}
\text{CUString}(\text{sh1}, \text{hstk}, \text{hstk}_{\text{ptr}}) \times \text{CUString}(\text{sh2}, \text{ndl}, \text{ndl}_{\text{ptr}}) \triangleq \begin{cases} 
    \text{} & \text{strstr_idx}(\text{hstk}_{\text{ptr}}, \text{ndl}_{\text{ptr}}) \cr
    \begin{aligned}
    & i. !((-1 \leq i < \text{length(hstk)}) \land \text{post_strstr_idx hstk ptr1 hstk ndl i}) \land \\
    & \text{CUString}(\text{sh1}, \text{hstk}, \text{hstk}_{\text{ptr}}) = \text{CUString}(\text{sh2}, \text{ndl}, \text{ndl}_{\text{ptr}})
    \end{aligned}
\end{cases}
\end{align*}
```

The proposition `post_strstr_idx` used in the postcondition is defined as follows:

```latex
\begin{verbatim}
Variant post_strstr_idx (ptr1 : val) (s1 s2 : list byte) : Z -> Prop :=
\begin{array}{c}
| StrStr_Idx_Not_Found: \\
\quad \text{~(is_sublist s2 s1)} \\
\quad \text{post_strstr_idx ptr1 s1 s2 (-1)} \\
| StrStr_Idx_Empty: \\
\quad s2 = nil \\
\quad \text{post_strstr_idx ptr1 s1 s2 0} \\
| StrStr_Idx_Found (r : Z): \\
\quad 0 \leq r < Z\text{length s1} \\
\quad \text{~ (is_sublist s2 \text{firstn (Z.to_nat r s2) \text{skimn (Z.to_nat r + \text{List.length s2} - 1) s1})}} \\
\quad \text{post_strstr_idx ptr1 s1 s2 r.}
\end{array}
\end{verbatim}
```

The preconditions state the inputs are valid C strings stored in readable memory, and the postconditions state that the returned value `i` is an integer between `-1` (inclusive) and the length of the “haystack” (exclusive) and that the model haystack string, needle string, and returned value `i` satisfy a relation `post_strstr_idx`. This relation is split into three cases:
1. The needle is not in the haystack and $i = -1$.
2. The needle is empty and $i = 0$.
3. The needle appears at offset $i$ in the haystack, and $i$ is in the range $[0, \text{length(hstk)})$.

The extra proof obligation induced by pointer subtraction is then handled once and for all in the verification of $\text{strstr}_{\text{idx}}$.

We applied this technique for three functions in our string library: $\text{strstr}$, $\text{strchr}$, and $\text{strcasestr}$. Each of these has a _idx version with a simplified specification, and higher-level modules that depend on the C string module all use the indexed versions instead of the raw pointer versions.

6 Related Work

Verifying networked servers. There are many papers on verifying networked servers, including HTTP servers [11], distributed systems [20, 41], and mail servers [12]. Koh et al. provide a detailed discussion of this previous work [25].

The goals and techniques of our work have much in common with those of Koh et al. [25]. The primary methodology in both projects is to refine a C program against an interaction tree specification using separation logic and VST based on the Clight semantics of CompCert. The KVServer extends the scale of this earlier effort in several dimensions. One is the complexity of the server’s state and behavior: The swap server by Koh et al. [25] simply remembers the last integer it received, where our KVServer manages arbitrarily many mappings, which requires operations such as growing and shrinking buffers and string hashing. Another difference is the protocols used for server-client communications. The swap server assumes requests are always 4-byte integers, while the KVServer understands a subset of HTTP/1.1, a ubiquitous industry standard. Handling HTTP requires verified parsing and C string libraries, most of which are generic and could be reused in other verified projects.

Additionally, our work significantly strengthens the connection to CertiKOS. Although Koh et al. [25] discussed connecting user- and kernel-level socket specifications, at the time there was only a work-in-progress proof for $\text{recv}$, whereas we provide complete proofs for a significant portion of the POSIX socket interface. The relation between user and kernel state in the swap server also ignored some important details, such as the translation between virtual and physical memory addresses, which are handled correctly in our work.

One limitation of our work compared to the swap server [25] is that we do not provide a full refinement proof connecting the implementation to a high-level “linear specification” that models the server’s behavior at the level of whole HTTP requests, hiding the low-level details of parsing and buffering. We believe that the “network refinement” relation between the low-level implementation and such a top-level linear specification can be formulated in terms of linearizability [21]. The additional complexity in the KVServer compared to the swap server arises from the fact that requests and responses are not atomic, since they may be split by the network. As future work, we plan to formalize the connection to linearizability and prove network refinement.

There is a great deal of previous work on verified parsers for network protocols. For example, TRX [26] is a parser interpreter that can be used to extract HTTP parsers with total correctness guarantees, and EverParse [36] is a framework for generating secure parsers and has been used to implement a parser for TLS. Instead of generating a verified parser, our work focuses on verifying hand-written C programs that use the standard, low-level C string library to implement HTTP parsing.
Perennial [13] is a new framework for verifying concurrent, crash-safe systems that has been used to implement a mail server. Whereas Perennial focuses on reasoning about crash-safety of concurrent programs, our work focuses on building a networked server whose specification is connected to the host operating system. A potential next step of our work is to incorporate Perennial’s crash safety reasoning methodology.

There are also many prior efforts to specify the POSIX socket interface [8, 9, 10, 38]. Since KVServer only requires a subset of the POSIX socket interface, our specification is not as complete as these. However, the KVServer socket specification is formally verified against the specification provided by the host CertiKOS (Section 4), while [8], [9], [10] and [38] all considered the verification against their specifications out of scope.

**Modular verification.** Beringer and Appel [7, 6] extended VST to support data representation abstraction and separation logic specification subsumption. These improvements made it possible to modularly specify and verify abstract data structures that hide their internal layouts and their operations with support from VST. KVServer uses an earlier release of VST that does not have these features yet, and all internal data structures used by KVServer in fact expose their implementation layouts – all C structs are defined in C header files. Although we manually ensure higher-level modules only access lower-level data structures through their verified C APIs to avoid breaking abstractions or introducing spurious dependencies, this manual discipline could be mechanically checked by leveraging these new features in the latest VST.

CCAL [17, 19] is a modular verification technique used in the CertiKOS project. CCAL is a formal calculus that enforces clear separation between the interface of a verified module and its implementation. CCAL gives a formal semantics of horizontal module composition within abstraction layers and vertical composition between abstraction layers. The composition of KVServer modules is similar to the horizontal composition of CCAL modules. However, KVServer does not employ vertical composition or abstraction layers, since these features do not exist yet in VST.

**Verifying C strings.** The VeriSoft project [2] verified a custom string library [39] based on the C0 semantics, a restricted version of ANSI C99 [27]. Moy and Marché [34] verified 22 functions from the standard C string library of MINIX 3 (https://www.minix3.org/); however, they only checked basic safety properties of these functions (e.g., absence of memory access errors), not functional correctness. Efremov et al. [15] verified the functional correctness of 26 string functions from the Linux kernel via deductive verification in Frama-C. Our work includes the functional correctness specifications and proofs of 34 functions (see Figure 1) based on the Clight semantics of CompCert [29]. 17 out of the 34 functions are POSIX compliant.

**Interaction trees.** Our specifications are phrased in terms of interaction trees, a general-purpose data structure for representing the behaviors of recursive programs that interact with their environments [42]. They are a coinductive variant of “freer monads” [24]; similar data structures include the program monads of Letan et al. [30], the general monads of McBride [33], and the action trees of Swamy et al. [40]. Interaction trees were also used in the specification of the swap server [25].
Connecting user-space and kernel-space specifications. Mansky et al. [32] demonstrated how to connect higher-order specifications with external effects written in VST with first-order specifications written in CCAL. This technique removes a verification gap between user-space programs and the host kernel. We apply Mansky et al.’s verification methods, and prove a refinement between the KVServer and CertiKOS socket specifications.

7 Conclusions and Future Work

We have verified a networked key-value server based on a subset of the HTTP/1.1 protocol, using VST and interaction trees to verify memory safety and functional correctness of the C implementation for each module. We also deepened the connection between KVServer and CertiKOS by proving that the user-level socket specifications agree with kernel-level socket specifications. The resulting proof guarantees the termination-insensitive correctness of the KVServer down to the kernel level, reducing the trusted computing base to the unverified POSIX socket system calls provided by CertiKOS.

As discussed in Section 6, an important future project is to define a high-level specification similar to the “linear specification” of Koh et al. [25] and prove the associated refinement, which can be viewed as a form of linearizability [21].

Specifying servers with interaction trees allows us to test server implementations against the specification [31]. We have written a top-level linear specification for testing purposes, whose relationship with the VST specification is still to be proven. From the testable specification, we have automatically derived a “testing client” that interacts with servers and checks whether they violate the specification. When developing the verified KVServer, we ran it against the derived tester, which has helped shake out a liveness-related bug – when a client pipelines more than one request in a single `send()`, the client connection may hang without immediately processing the latter requests. This liveness bug was out of scope for the verification of the KVServer due to the partial-correctness nature of VST specifications, but we have patched the server implementation and related proofs to correctly handle pipelined requests.

References


Verifying an HTTP Key-Value Server with Interaction Trees and VST


