From C to Interaction Trees
Specifying, Verifying, and Testing a Networked Server

Nicolas Koh†, Yao Li‡, Yishuai Li†, Li-yao Xia†, Lennart Beringer§, Wolf Honore*, William Mansky‡, Benjamin C. Pierce†, Steve Zdancewic†
University of Pennsylvania†, Princeton University§, Yale University‡, University of Illinois at Chicago†

Abstract
We present the first formal verification of a networked server implemented in C. Interaction trees, a general structure for representing reactive computations, are used to tie together disparate verification and testing tools (Coq, VST, and QuickChick) and to axiomatize the behavior of the operating system on which the server runs (CertiKOS). The main theorem connects a specification of acceptable server behaviors, written in a straightforward “one client at a time” style, with the CompCert semantics of the C program. The variability introduced by low-level buffering of messages and interleaving of multiple TCP connections is captured using network refinement, a variant of observational refinement.

Keywords formal verification, testing, TCP, interaction trees, network refinement, VST, QuickChick

1 Introduction
The Science of Deep Specification [Appel et al. 2017] is an ambitious experiment in specification, rigorous testing, and formal verification of real-world systems such as web servers “from internet RFCs all the way to transistors.” The principal challenges lie in integrating disparate specification styles, legacy specifications, and testing and verification tools to build and reason about complex, multi-layered systems.

We report here on a first step toward realizing this vision: an in-depth case study demonstrating how to specify, test, and verify a simple networked server with the same fundamental interaction model as more sophisticated ones—it communicates with multiple clients via ordered, reliable TCP connections. Our server is implemented in C and verified, using the Verified Software Toolchain [Appel 2014], against a formal “implementation model” written in Coq [2018]; this is further verified (in Coq) against a linear “one client at a time” specification of allowed behaviors. The main property we prove is that any trace that can be observed by a collection of concurrent clients interacting with the server over the network can be rearranged into a trace that is allowed by the linear specification. We also show how property-based random testing using Coq’s QuickChick plug-in [Lampropoulos and Pierce 2018] can be deployed in this setting. We compile the server code with the CompCert verified compiler [Leroy 2009] and run it on CertiKOS [Gu et al. 2016], a verified operating system with support for TCP socket operations.

Our verified server provides a simple “swap” interface that allows clients to send a new bytestring to the server and receive the currently stored one in exchange. It is simpler in many respects than a full-blown web server; in particular, it follows a much simpler protocol (no authentication, encryption, header parsing, etc.), which means that it can be implemented with much less code.

Moreover, the degree of vertical integration falls short of our ultimate ambitions for the DeepSpec project, since we stop at the CertiKOS interface (which we axiomatize) instead of going all the way down to transistors. On the other hand, the C implementation of our server is realistic enough that it offers a challenging test of how to integrate disparate Coq-based methodologies and tools for verifying and testing systems software. In particular, it uses a single-process, event-driven architecture [Pai et al. 1999], hides latency by buffering interleaved TCP communications from multiple clients, and is built on the POSIX socket API.

Contributions We describe our experiences integrating Coq, CompCert, VST, CertiKOS, and QuickChick to build a verified swap server. This is the first VST verification of a program that interacts with the external environment. It is also, to the best of our knowledge, the first verification of functional correctness of a networked server implemented in C. Our technical contributions are as follows:

First, we identify interaction trees (ITrees)—a Coq adaptation of structures known variously as “freer” [Kiselyov and Ishii 2015], “general” [McBride 2015], or “program” [Letan et al. 2018] monads—as a suitable unifying structure for expressing and relating specifications at different levels of abstraction (Section 3).

Second, we adapt standard notions of linearizability and observational refinement from the literature on concurrent data structures to give a simple specification methodology for networked servers that is suitable both for rigorous property-based testing and for formal verification. We call this variant network refinement (Section 4).

Third, we demonstrate practical techniques for both verifying (Section 5) and testing (Section 6) network refinement between a low-level implementation model and a simple linear specification. We also demonstrate testing against the compiled C implementation across a network interface.
Informally, the intended behavior of the swap server is straightforward. Any number of clients can connect and send "swap requests," each containing a fixed-size message. The server acts as a one-element concurrent buffer: it retains the most recent message that it has received and, upon getting a swap request, updates its state with the new message and replies to the sender with

\[ \text{network-refines} \text{ the linear specification.} \]

The fact that the C implementation refines the linear specification is validated by a lower-level axiomatization in the style of CertiKOS, which is manually compared to the (unverified) TCP implementation. The implementation model, in turn, "network refines" the linear specification. The theorem that the C implementation network refines the linear specification is independently validated by property-based random testing.

In all the Coq models and specifications, interaction trees represent possible behaviors as an interaction tree (of enqueue trees). The linear specification is initialized with an empty set of connections and a message filled with zeros. The connection with a new client (\text{obs\_connect}) may be received out of order, and messages may be delayed indefinitely (dotted arrow). The "explanation" of the two runs in terms of the linear specification is the same.

The implementation model runs in terms of the linear specification is the same.

\[ \text{Figure 2. Swap server examples. On the left is a simple run that directly illustrates the linear specification. Each client in turn establishes a connection, sends a three-byte message, and receives the message currently stored on the server as a response. ('000' is the server's initial state.) On the right is another run illustrating internal buffering by the swap server and reordering by the network. Messages may be sent in multiple chunks, messages from different clients may be received out of order, and messages may be delayed indefinitely (dotted arrow). The "explanation" of the two runs in terms of the linear specification is the same.} \]

Lastly, the ITrees embedded into both VST's separation logic and CertiKOS's socket model allow us to make progress on connecting the two developments. Though completing the formal proofs remains for future work, we identify the challenges and describe preliminary results in Section 7.

Section 2 summarizes the whole development. Sections 8 and 9 discuss related and future work. A tarball containing all our Coq and C code has been provided to the PC chairs.

\[ \text{Figure 3. Linear specification of the server. In the linear spec' loop, the parameter \text{conns} maintains the list of open connections, while \text{last\_msg} holds the message received from the last client (which will be sent back to the next client). The linear specification is initialized with an empty set of connections and a message filled with zeros.} \]

\[ \text{Definition linear\_spec := linear\_spec' [] zeros.} \]

\[ \text{Figure 2. Swap server examples. On the left is a simple run that directly illustrates the linear specification. Each client in turn establishes a connection, sends a three-byte message, and receives the message currently stored on the server as a response. ('000' is the server's initial state.) On the right is another run illustrating internal buffering by the swap server and reordering by the network. Messages may be sent in multiple chunks, messages from different clients may be received out of order, and messages may be delayed indefinitely (dotted arrow). The "explanation" of the two runs in terms of the linear specification is the same.} \]

The old one. The left-hand side of Figure 2 shows a simple example of correct behavior of a swap server. Figure 3 shows the linear specification of the server's behavior. It says that the server can either accept a connection with a new client (\text{obs\_connect}) or else receive a message from a client over some established connection (\text{obs\_msg\_to\_server c}), send back the current stored message (\text{obs\_msg\_from\_server c last\_msg}), and then start over with the last-received message as the current state. The set of possible behaviors is represented as an interaction tree (of type \text{itree\_specE unit}).

Our main correctness theorem should relate the actual behavior of our server (the CompCert semantics of the C code) to this linear description of its desired behavior. Informally:

\[ \text{Theorem 1. Any sequence of interactions with the swap server that can be observed by clients over the network could have been produced by the linear specification.} \]
Theorem swap_server_correct :

\[ \exists \text{impl}_\text{model}, \text{ext}_\text{behavior} \text{ C}_\text{prog} \text{ impl}_\text{model} \land \text{network_refines linear_spec impl}_\text{model} . \]

Figure 4. End-to-end swap server correctness theorem.

This theorem constrains the server to act as a swap server: it prevents the server from sending a message before it receives one, or while it has only received a partial message; it prevents it from sending an arbitrary value in response to a request, or replying multiple times with the same value that has only been received once; it prevents it from sending a response to a client from which it has not received a request. However, the “over the network” clause is a significant caveat: the server communicates with clients via TCP, and even a correct implementation might thus exhibit a number of undesirable behaviors from the clients’ point of view. The network might drop all packets after a certain point, causing the server to appear to have stopped running, so the theorem allows the server to stop running at any point. Similarly, the network might delay messages and might reorder messages on different connections, so the theorem allows the server to respond to an earlier request after responding to a later request. The right-hand side of Figure 2 shows another run of the system illustrating these possibilities; it should also be accepted by the top-level theorem.

Figure 4 shows the formal specification linking the linear specification (linear_spec), which describes interactions with one client at a time, to the C program (C_prog). It is split in two parts articulated around an implementation model (impl_model). It is another interaction tree that describes the network-level behavior of the C program more closely than the linear specification. Like the C program, the implementation model interleaves requests from multiple clients and accounts for the effects of the network. A refinement between the C program and the implementation model is formalized by the VST property ext_behavior. Then the implementation model is connected to the specification by a different network refinement layer (network_refines).

Network refinement The linear specification is short and easy to understand, but an implementation that strictly followed it would be obliged to serve clients sequentially, which is not what real servers (including ours) want to do. Moreover, as shown on the right-hand side of Figure 2, the network may delay and reorder messages, so that, for example, the first two bytes of a message from client 1 might be received after the first byte of a message from client 2. The server should be able to account for this by buffering messages until they are complete. The second part of our server specification loosens the linear specification to account for the effects of communicating over a network; this also permits realistic implementations that serve multiple clients concurrently.

Network refinement states that every possible behavior of the implementation model is allowed by the linear specification, while accounting for message reordering and buffering that might be introduced by the network and/or server. Section 4 explains this process in more detail.

C Implementation Our C implementation is a simple but reasonably performant server in a classical single-process, event-driven style [Pai et al. 1999]. The implementation maintains a list of connection structures, each representing a state machine for one connection. Specifically, a connection structure contains (1) a state, which may be RECVING, SENDING, or DELETED; (2) a buffer for storing bytes that have been received on the connection; and (3) a buffer for storing bytes to send on the connection.

The main body of the server is a non-terminating loop (Figure 5); in each iteration, it uses the select system call to check for pending connections to accept and for existing connections ready for receiving/sending bytes from/to, and processes them. A new connection is handled by initializing a new connection structure and adding it into the list, and an existing connection is processed by updating the read/write buffers and advancing the connection’s state appropriately. This buffering strategy lets the server interleave processing of multiple connections without having to wait for one client to send or receive a complete message.

Verifying the C code To prove that the C implementation refines the implementation model (that is, that every possible network behavior of the C program is allowed by the implementation model), we use VST, a tool for proving correctness of C programs using separation logic. The VST predicate ext_behavior C_prog impl_model in Figure 4 relates the operational semantics of the C program C_prog to the interaction tree description given by impl_model. Section 3 describes the implementation model in more detail.

VST’s model of program execution includes both conventional program state (memory, local variables, etc.) and external state, an abstract representation of the state of the environment in which the program is running. We connect the C program semantics to the implementation model by adding a predicate ITree(t) to VST’s separation logic, asserting that the environment expects the C program’s network behavior to match the interaction tree t. Section 5 describes this process.

Assumptions and modeling gaps We have a complete proof (using VST) that the C implementation compiled with CompCert network-refines the linear specification—that is, a complete proof of the claim in Figure 4. This proof is grounded in axiomatic specifications of the OS-level system calls, and some library functions. We rely on the soundness of the Coq proof assistant, plus the standard axioms of functional and propositional extensionality and proof irrelevance.

For this case study, our verification bottoms out at the interface between the application program and the operating system; we rely on the correctness of the OS’s socket library and of the OS itself. Since we are running on CertiKOS,
while(1 == 1) {
    ...  
    int num_ready = select(maxsock + 1, &rs, &ws, &es);  
    if (num_ready <= 0) { continue; }  
    int socket_ready = fd_isset_macro(server_socket, &rs);  
    if (socket_ready) {  
        /* Accept a new connection on the socket, create a  
        connection structure for it, and link it into the  
        head of the linked list of connections. */  
        accept_connection(server_socket, &head);  
    }  
    /* For each connection in the list pointed to by head,  
    read from or write to its buffer of data. */  
    process_connections(head, &rs, &ws, last_msg_store);  
}  

Figure 5. Main loop of swap server (in C)  

As the OS has actually been proved correct, but its correctness  
proofs and ours are not formally connected. That is, our  
specification of its socket API is axiomatized, but the axioms  
are partially validated by connection to the corresponding  
CertiKOS specifications (specifically, a VST specification of  
recv has been partly connected to the CertiKOS-level one;  
the other socket primitives remain to be connected). There  
are several remaining challenges with connecting VST to  
CertiKOS, ranging from the semantic—one critical technical-  
ity is connecting VST’s step-indexed view of memory with  
the flat memory model used by CertiKOS—to the technical—  
you use different versions of Coq. See Section 7 for a fuller  
description of what we have done to bridge these two formal-  
izations. Also, because CertiKOS currently does not provide  
a verified TCP implementation, the best it can do is mediate  
between the VST axioms and some, possibly lower-level, axi-  
omatization of the untrusted TCP stack. Filling these gaps  
is left to future work.

Testing network refinement For our long-term goal of  
building verified systems software like web servers, rigorous  
testing will be crucial, for two reasons. First, even small web  
servers are fairly complex programs, and they take signifi-  
cant effort to verify; streamlining this effort by catching as  
many bugs as possible before spending much time on verifica-  
tion makes good economic sense, especially if the code  
can be automatically tested against the very same specifi-  
cation that will later be used in the verification effort. Second,  
programs like web servers must often fit into an existing  
ecosystem—a verified web server that interpreted the HTTP  
RFCs (e.g., Belshie et al. [2015]) differently from Apache and  
Ngix would not be used. Testing can be used to validate the  
formal specification against existing implementations.

For the present case study, we use QuickChick [Lam-  
propoulos and Pierce 2018], a Coq plug-in for property-based  
testing based on the popular QuickCheck tool [Claessen and  
Hughes 2000]. We test both the compiled C code (by sending  
messages over a network interface) and the implementa-  
tion model (by exploring its behaviors within Coq) against  
the linear specification.

CoInductive itree (E : Type -> Type) (R : Type) :=  
| Ret (r : R) |  
| Vis (X : Type) (e : E X) (k : X -> itree E R) |  
| Tau (t : itree E R) |  

Inductive event (E : Type -> Type) : Type :=  
| Event : V X, E X -> X -> event E |  

Definition trace E := list (event E)  

Inductive is_trace E R :=  
| : ttree E R -> trace E -> option R -> Prop :=  
(* straightforward definition omitted *)  

Figure 6. Interaction trees and their traces of events.

Supporting property-based testing requires executable spec-  
ifications of the properties involved. Happily, interaction  
trees, which play a crucial role throughout our development,  
also work well with Coq-style program extraction, and hence  
with testing. Testing must also be performed “modulo net-  
work refinement” in the same way as verification. Section 6  
describes this in more detail.

3 Interaction Trees

Components that interact with their environment appear  
at many levels in our development (see Figure 1). We use  
interaction trees (ITrees) as a general-purpose structure for  
specifying such components. ITrees are a Coq adaptation of  
similar concepts known variously as “freer,” “general,” or  
“program” monads [Kiselyov and Ishtii 2015; Letan et al. 2018;  
McBride 2015]. We defer a deeper comparison until Section 8.

Constructing ITrees Figure 6 defines the type itree E R.  
The definition is coinductive, so that it can represent poten-  
tially infinite sequences of interactions, as well as divergent  
behaviors. The parameter E is a type of external interactions—  
it defines the interface by which a computation interacts  
with its context. R is the result of the computation: if the  
computation halts, it returns a value of type R.

There are three ways to construct an ITree. The Ret r  
constructor corresponds to the trivial computation that halts  
and yields the value r. The Tau t constructor corresponds to  
a silent step of computation, which does something internal  
that does not produce any visible effect and then continues  
as t. Representing silent steps explicitly with Tau allows us,  
for example, to represent diverging computation without  
violating Coq’s guardedness condition:

CoFixpoint spin (E R) : itree E R := Tau spin.  

The final, and most interesting, way to construct an ITree  
is with the Vis X e k constructor. Here, e : E X is a “visi-  
ble” external effect (including any outputs provided by the  
computation to its context) and X is the type of data that  
the context provides in response to the event. The constructor  
also specifies a continuation, k, which produces the rest of  
the computation given the response from the context. Vis
introduces branches into the interaction tree, because \( k \) can behave differently for distinct values of type \( X \).

Here is a small example that defines a type \( IO \) of output or input interactions, each of which works with natural numbers. It is then straightforward to define an ITree computation that loops forever, echoing each input received to the output:

\[
\begin{aligned}
\text{Variant } IO & : \text{Type} \rightarrow \text{Type} := \\
| \text{Input} & : IO \, \text{nat} \\
| \text{Output} & : \text{nat} \rightarrow IO () .
\end{aligned}
\]

\[
\begin{aligned}
\text{CoInductive } echo & : \text{itree } IO () := \\
& \text{Vis Input } (\text{fun } x \Rightarrow \text{Vis } (\text{Output } x)) (\text{fun } _{-} \Rightarrow echo)).
\end{aligned}
\]

**Working with ITrees** Several properties of ITrees make them appealing as a structure for representing interactive computations. First, they are *generic* in the sense that, by varying the \( E \) parameter, they can be instantiated to work with different external interfaces. Moreover, such interfaces can be built compositionally: for example, we can combine a computation with external effects in \( E1 \) with a different computation with effects in \( E2 \), yielding a computation with effects in \( E1 + E2 \), the disjoint union of \( E1 \) and \( E2 \); there is a natural inclusion of ITrees with interface \( E1 \) into ITrees with interface \( E1 + E2 \). This approach is reminiscent of *algebraic effects* [Plotkin and Power 2003]. Our development exploits this flexibility to easily combine generic functionality, such as a nondeterministic choice effect (which provides the \text{or} operator used by the linear specification of Figure 3) with domain-specific interactions such as the network send and receive events. As with algebraic effects, we can write a *handler* or *interpreter* for some or all of the external interactions in an interface, for example to narrow the effects \( E1 + E2 \) down to just those in \( E1 \). Typically, such a handler will process the events of \( E2 \) and “internalize” them by replacing them with \( \tau \) steps.

Second, the type \text{itree} \( E \) is a monad [Wadler 1992], which makes it convenient to structure effectful computations using the conventions and notations of functional programming. We package up the \text{ret} constructor as a \text{ret} (return) operation and use the sequencing notation \( x \leftarrow e ;; k \) for the monad’s bind. With a bit of wrapping and a loop combinator \text{forever}, we can rewrite the echo example with less syntactic clutter:

\[
\begin{aligned}
\text{Definition } echo & : \text{itree } IO () := \\
& \text{forever } (x \leftarrow \text{input} ;; \text{output } x)
\end{aligned}
\]

Third, the ITree definition works well with Coq’s extraction mechanism, allowing us to represent computations as ITrees and run them for testing purposes. Here again, the ability to provide a separate interpretation of events is useful, since its meaning can be defined outside of Coq. In the echo example, \text{Output} events could be linked to a console output or to an OS’s network-send system call. ITrees thus provide *executable* specifications.

**Equivalence and Refinement** Intuitively, ITrees that encode the same computation should be considered equivalent. In particular, we want to equate ITrees that agree on their terminal behavior (they return the same value) and on \text{Vis} events; they may differ by inserting or removing any finite number of \( \tau \) constructors. This “equivalence up to \( \tau \)” is a form of weak bisimulation. We write \( t \equiv u \) when \( t \) and \( u \) are equivalent up to \( \tau \). The monad laws for ITrees also hold modulo this notion of equivalence. (Some of the laws used in our development are shown in Figure 7.)

ITrees that contain nondeterministic effects or that receive inputs from the environment denote a set of possible *traces*—(finite prefixes of) execution sequences that record each visible event together with the environment’s response. The definitions of \text{trace} and the predicate \text{is_trace}, which asserts that a trace belongs to an ITree, are shown in Figure 6. Subset inclusion of behaviors gives rise to a natural notion of ITree *refinement*, written \( t \subseteq u \), which says that the traces of \( t \) are a subset of those allowed by \( u \). We use this refinement relation to allow an implementation to exhibit fewer behaviors than those permitted by its specification. Note that \( t \equiv u \) implies \( t \subseteq u \).

**ITrees as Specifications: the Linear Specification** Interaction trees provide a convenient yet rigorous way of formalizing specifications. We have already seen them in the linear specification of the swap server in Figure 3. The \text{itree spec} type there is an instance of \text{itree} whose visible events include nondeterministic choice as well as observations of swap request and response messages, which are events that include message content and connection ID information. The specification itself looks like a standard functional program that uses an effects monad to capture network interactions.

**ITrees as Specifications: the Implementation Model** We use the same \text{itree} datatype, this time instantiated with a socket API interface included in \text{implE}, to define the implementation model, which is a lower-level (but still purely functional) specification of the swap server that more closely resembles the C code. Figure 8 shows the body of the main loop from the implementation model.

In contrast to the linear specification, the implementation model maintains a list of connection structures instead of bare connection identifiers. Each structure records the state for some connection. The state indicates whether the server should be \text{SENDING} or \text{RECVING} on the connection (or whether the connection is closed). The state also records the contents of receive and send buffers. In each iteration of the loop,
We show a "network refinement" relation between the implementation model and the linear specification. At a high level, this property is a form of observational refinement [He et al. 1986]: the behaviors of the implementation that can be observed from across the network are included in those of the specification. Intuitively, this property is also an analog, in the network setting, of linearizability for concurrent data structures; we compare them in detail in Section 8.

**The network** We model a simple subset of the TCP socket interface, where connections carry bytestreams (the bytes sent on an individual connection are ordered); they are bidirectional (both ends can send bytes) and reliable (what is received is a prefix of what was sent). This network model is represented by a nondeterministic state machine where each connection carries a pair of buffers of “in flight” bytes, with labeled transitions for a client to open a connection, a server to accept it, and either party to send and receive bytes (Figures 9 and 10). For example, there is a transition from network state \( ns \) to state \( ns' \), labeled `FromServer c b := Map.lookup ns c in` the connection state \( cs \) with \( \text{ACCEPTED} \Rightarrow \text{let c}' := \text{update_out} \) (connection_outbytes cs ++ [b]) ns in \( ns' := \text{Map.update c}' \text{ ns} \) |...| Closed |False |...| Other two cases |End.

**Figure 8.** Loop body of the implementation model

the server either accepts a new connection or services a connection that is in the `SENDING` or `RECVING` state. Servicing a connection in the `SENDING` state means sending some prefix of the bytes in the send buffer; servicing a connection in the `RECVING` state means receiving some bytes on the connection.

Note that the control flow of this model differs from both the linear specification and the C implementation. The linear specification bundles together request–response pairs and totally abstracts away from the details of buffering and interleaving communications among multiple clients. The relationship between the implementation model and the linear specification is given by network refinement, as we explain in the next section. For the C implementation, a single iteration of the main server loop in Figure 5 corresponds to multiple iterations of the select loop body of the model. Nevertheless, we can prove that the C behavior is a refinement of the implementation model, as we describe in Section 5.

4 Network Refinement

We show a "network refinement" relation between the implementation model and the linear specification. At a high level, this property is a form of observational refinement [He et al. 1986]: the behaviors of the implementation that can be observed from across the network are included in those of the specification. Intuitively, this property is also an analog, in the network setting, of linearizability for concurrent data structures; we compare them in detail in Section 8.

**The network** We model a simple subset of the TCP socket interface, where connections carry bytestreams (the bytes sent on an individual connection are ordered); they are bidirectional (both ends can send bytes) and reliable (what is received is a prefix of what was sent). This network model is represented by a nondeterministic state machine where each connection carries a pair of buffers of “in flight” bytes, with labeled transitions for a client to open a connection, a server to accept it, and either party to send and receive bytes (Figures 9 and 10). For example, there is a transition from network state \( ns \) to state \( ns' \), labeled `FromServer c b := Map.lookup ns c in` the connection state \( cs \) with \( \text{ACCEPTED} \Rightarrow \text{let c}' := \text{update_out} \) (connection_outbytes cs ++ [b]) ns in \( ns' := \text{Map.update c}' \text{ ns} \) |...| Closed |False |...| Other two cases |End.

**Figure 9.** Types for events and traces observed over the network. `network_event` maps to `event` values to form traces for both the specification and the implementation model.

**Figure 10.** Network transitions labeled by `network_event`, showing only the case where the server sends a byte.

is represented by a nondeterministic state machine where each connection carries a pair of buffers of “in flight” bytes, with labeled transitions for a client to open a connection, a server to accept it, and either party to send and receive bytes (Figures 9 and 10). For example, there is a transition from network state \( ns \) to state \( ns' \), labeled `FromServer c b := Map.lookup ns c in` the connection state \( cs \) with \( \text{ACCEPTED} \Rightarrow \text{let c}' := \text{update_out} \) (connection_outbytes cs ++ [b]) ns in \( ns' := \text{Map.update c}' \text{ ns} \) |...| Closed |False |...| Other two cases |End.

We define a relation `network_reordered ns ts tc : Prop` between server- and client-side traces of network events `ts` and `tc`, which holds if they can be produced by an execution of the network starting from state `ns`. For the initial state with all connections closed, we define `network_reordered ns tc := network_reordered_initial ns ts tc`. The trace `tc` is a "disordering" of `ts`—i.e., `tc` is one possible trace a client may observe if the server generated the trace `ts`. Conversely, `ts` is a "reordering" of `tc`.

**Network behavior of ITrees** As mentioned in Section 3, ITrees such as the implementation model (of type `itree implE`) and the linear specification (`itree specE`) define sets of event traces. From across the network, those events can appear disordered to the client, so the network behavior of an ITree is the set of possible disorderings of its traces (defined using `network reordered`). Finally, the ITree `impl_model network refines` the `linear_spec` when the former’s network behavior is included in the latter’s; see Figure 11.

**Proving network refinement** In order to prove that our implementation model network refines the linear specification, we establish logical proof rules for a generalization of
661 Definition impl_behavior (impl : itree implE unit) : network_trace → Prop := 662 fun tr → ∃ tr_impl, is_impl_trace impl tr_impl ∧ 663 network_reordered_impl tr tr.
664
665 Definition spec_behavior (spec : itree specE unit) : network_trace → Prop := 666 fun tr → ∃ tr_spec, is_spec_trace spec tr_spec ∧ 667 network_reordered_spec tr tr.
668
669 Definition network_refines impl spec : Prop := 670 ∀ tr, impl_behavior impl tr → spec_behavior spec tr.
671
672 Figure 11. Definition of network refinement in Coq. The functions is_impl_trace and is_spec_trace are thin wrappers around is_trace that convert between traces of different (but isomorphic) event types.
673
674 Record state := { get_ns : network_state; 675 get_spec : itree specE unit; ... }. 676
677
678 Definition nrefines_ (z : nat) (s : state) 679 (impl : itree implE unit) : Prop := 680 ∀ dtr : network_trace, 681 network_reordered_ (get_ns s) dtr tr ∧ 682 is_spec_trace (get_spec s) dtr.
683
684 Figure 12. Refinement relation generalized for reasoning
685 network_refines, named nrefines_, (Figure 12). The nrefines_- 686 relation is step-indexed (z : nat) to handle the server’s nonterminating loop; it relates a subtree of the implementation model impl to a record s of the current state of the network (get_ns s : network_state) and a subtree of the specification ITrees (get_spec s : itree specE unit).
687
688 Two example proof rules are shown in Figure 13. When
689 the server performs a network operation, for example when
690 it receives a byte on a connection c, we use a lemma such 691 as nrefines_recv_byte_: we must prove that the connection c 692 is open, and we then prove the nrefines_ relation on the 693 continuation k b, with an updated network state in s’.
694
695 At any point in the proof, we can also generate part of the 696 reordered trace from the linear specification ITrees get_spec 697 s, using the lemma nrefines_network_transition_. We actually 698 use this rule at exactly two “linearization points” in the 699 implementation model: right after the server accepts a new 700 connection, and after it receives a complete message from a 701 client and swaps it with the last stored message.
702
703 Using these rules, we prove the proposition ∀ z, nrefines_ 704 z s0 impl_model, where s0 is defined so that get_spec s0 705 = linear_spec and get_ns s0 is the initial network state, 706 where all connections are closed; we can show this implies 707 the second clause of the correctness theorem (Figure 4).
708
709 5 Verification
710
711 Embedding ITrees in VST VST is a framework for prov- 712 ing separation logic specifications of C programs, based on 713 the C semantics of the CompCert compiler. Its separation 714 logic comes with a proof automation system, Floyd, that 715
716
717 Lemma nrefines_recv_byte_z s
718 (c : connection_id) (k : byte → itree implE unit) 719 : in (get_status s c) [PENDING ; ACCEPTED] → 720 (∀ b s’, s’ = append_inbytes c [b] s →
721 nrefines_ z s’ (k b)) →
722 nrefines_ z s (b ← recv_byte c ;; k b).
723
724 Lemma nrefines_network_transition_z s obs ns’ t
725 (dtr : network_trace) : (∃ dtr’, is_spec_trace obs’ dtr’ →
726 is_spec_trace (get_observer s) (dtr ++ dtr’)) →
727 server_transitions dtr (get_ns s) ns’ →
728 nrefines_ z (set_ns ns’ (set_observer obs’ s)) t →
729 nrefines_ z s t.
730
731 Figure 13. Example proof rules for nrefines_.
732
733 supplies tactics for symbolically executing a program while maintaining its pre- and postcondition [Cao et al. 2018]. To support reasoning about external behavior in general—and the swap server’s invocations of OS/network primitives in particular—we extend VST’s logic with two abstract predicates [Penninckx et al. 2015]; these are separation logic predicates that behave like resources but do not have a footprint in concrete memory. Instead they connect to VST’s model of external state, which in this case represents the allowed network behavior of the program. To make this possible, we made a small modification to the internals of VST to enable it to refer to the external state in assertions.
735
736 The first abstract predicate, ITrees(t), injects an interaction
737 tree t into a VST assertion (an npre): 738
739 Definition ITrees (R) (t : itree implE R) : npre :=
740 EX t’ : itree impl E, !(t ⊆ t’) && has_ext t’.
741
742 ITrees t asserts that the observation traces of t (i.e., the traces 
743 that may be produced by a program satisfying the assertion 
744 ITrees t) are included in the traces that are permitted by the 
745 external environment (here, the OS). The has_ext predicate asserts that the external state (here representing the network behavior the OS expects from the program) is exactly t’. The notation || lifts an ordinary Coq predicate p to a VST separation logic predicate, and && and EX are logical conjunction and existential quantification at the level of separation logic assertions.
748
749 While a detailed description of VST’s support for external 
750 state is beyond the scope of the present paper, we give some 
751 key properties of this embedding. Internal code execution 
752 does not depend on or alter external state, so every program 
753 step that is not a call to the socket API leaves the ITrees 
754 predicate unchanged. The monad and equivalence laws from 
755 the abstract theory of interaction trees are reflected as (provable) entailments between ITrees predicates (recall the refinement relation of Figure 7):
758
759 t ⊆ u
760 ITrees u ⊆ ITrees t
762
763 This rule is contravariant because we can conform to the 
764 ITrees u by producing some subset of its allowed behavior.
with specifications that reflect the evolution of interaction states to socket states to file descriptors to socket states closed, opened, bound, listening, or connected. Bound and listening states are associated with an endpoint identifier in the network model, and connected states are associated with a connection identifier in the network model. The reason for modularly separating socket states from interaction trees is that the latter describe truly external behavior while the former concern the (private) contract between the server program and the OS. Specifically, the functions for creating sockets, binding them to addresses, and closing sockets are not visible at the other end of the network and are hence specified to only operate over SOCKAPI abstract predicates. In general, system calls like `recv` that affect the network state carry specifications of the form

$$\exists \mathbf{st} \cdot \text{SOCKAPI} \mathbf{st} \cdot \text{ITree} \mathbf{t} \cdot \text{VST axiom for the recv system call.}$$

External calls to network and OS functions are equipped with specifications that reflect the evolution of interaction trees, in resource-consuming fashion: actions are “peeled off” from the ITree as execution proceeds, so that the interaction tree in the postcondition of an external function specification is a subtree of the tree in the precondition. The ITree found in the outermost precondition of a program is thus a sound approximation of all the program’s external interactions.

**Hoare-logic specifications of system calls** This use of the ITree predicate can be seen in the VST axiom for the `recv` system call in Figure 14. The precondition of this rule requires that the ITree \((r \leftarrow \text{recv client_conn} \ldots) ; k r\), which starts with a `recv` event, be among the allowed behaviors of \(t\), so a legal implementation of this specification is allowed to perform a `recv` call next. The postcondition either leaves the interaction tree \(t\) untouched, in the case that the call to `recv` failed, or says that the implementation may continue as \(k \text{msg}\), in the case that the call to `recv` successfully returned a message `msg`.

Most of the remaining constraints relate the program variables and the variables in the interaction tree to the corresponding state in memory. For example, the predicate `data_at alloc_len buf_ptr` says that `buf_ptr` points to a buffer of length `alloc_len`. The constraint `lookup_socket st fd = ConnectedSocket client_conn` says that the socket with identifier `fd` is in the `CONNECTED` state according to the API and is associated with the connection identifier `client_conn` appearing in the interaction tree.

This socket information is tracked by a second abstract predicate, `SOCKAPI(st)`, which asserts that the external socket API memory can be abstracted as \(st\), mapping file descriptors to socket states closed, opened, bound, listening, or connected. Bound and listening states are associated with an endpoint identifier in the network model, and connected

where \(\phi\) is a boolean predicate distinguishing ITree-advancing (successful) invocations from failed invocations (which leave the ITree unmodified), by inspection of the implicitly quantified return value `r`.

**Verifying the C implementation** Having defined the abstract predicates we need to describe the network behavior of the server, we can now prove that the C implementation refines the implementation model using VST’s separation logic. The goal is to prove that the implementation model `impl_model` is an envelope around the possible network behaviors of the C program, i.e., every execution of the C program performs only the socket operations described in `impl_model`; this is expressed by the predicate `ext_behavior C prog impl_model`. This proof then composes with the network refinement proof between `impl_model` and the linear specification to give us the main theorem in Figure 4.

We prove `ext_behavior C prog impl_model` by specifying and proving a Hoare triple for each function in the C implementation. We begin with axiomatized Hoare triples for the library functions, in particular those from the POSIX socket API; these triples modify the `SOCKAPI` state and possibly consume operations from the ITree, as described above.

We then specify Hoare triples for functions in the program, including embedded interaction trees where appropriate. Verification proceeds as in standard Hoare logic, including formulating an appropriate invariant for each loop. The most interesting invariant is for the main loop, shown in Figure 5; among other things, the invariant states that `head` points to a linked list \(l\) of connection structures, `last_msg_store` points to a buffer storing a message \(M\), and the interaction tree under `ITree` is an infinite loop of `select_loop_body` (Figure 8) started on \((l, m)\); the server address and buffer size are constants.

Note that it is not immediate that the C loop body refines `select_loop_body`. The former iterates over all ready connections in `process_connections`, while the latter works on only one connection per iteration. However, each iteration in `process_connections` is itself an iteration of `select_loop_body`, so the inner invariant carries the same
interaction tree. Conceptually, one iteration of the main loop in C corresponds to multiple iterations of the model.

6 Testing

Our overall approach to verifying software includes testing for errors in code and specifications before we invest too much effort in verification. For the swap server, we used QuickChick [Lampropoulos and Pierce 2018], a property-based testing tool in Coq, to test both whether the C implementation satisfies the linear specification, and whether the implementation model refines the linear specification. These tests help establish confidence in all three artifacts.

Test setup Our testbed consists of a simple hand-written client, the server to be tested, and the linear specification that the server should satisfy. The client opens multiple TCP connections to simulate multiple clients communicating with the server over the network.

The testing process is straightforward: First, the client generates a random sequence of messages along randomly chosen TCP connections. The client then collects a trace of its interactions with the server—the messages that it sent and the responses that it received in return on each connection. Finally, the checker attempts to “explain” this trace by enumerating all of the possible reorderings of the real trace and checking whether any of them is, in fact, a trace of the linear specification. If such a trace is found, this test case passes, and another trace is generated. If none of the reorderings satisfies the specification, the tester reports that it has found a counterexample. Before actually displaying the counterexample, the tester attempts to shrink it using a greedy search process modeled on the one used in Haskell’s QuickCheck tool, successively throwing away bits of the counterexample and rechecking to see whether the remainder still fails.

We can also test that the implementation model refines the linear specification. The setup here is similar to the one for the C program, but simpler because we can execute both the client and server within a single Coq program rather than extracting a client from Coq and running it with the server and a network.

Testing the tester The proofs connecting our C implementation, implementation model, and linear specification were well along before we completed the testing framework; this meant that these artifacts were already thoroughly debugged, and testing was not able to find any additional bugs.

To assess how effective testing might have been if it had been deployed earlier in the process, we used QuickChick’s mutation testing mode [DeMillo et al. 1978] to inject 12 different “plausible bugs” (of the sort commonly found in C: pointer errors, bad initialization, off-by-one errors, etc.) into the code and check that each could be detected during testing. The bugs are added to the C program as comments marking a section of “good code” and a “mutant” that can be substituted for it. QuickChick performs this substitution for each of the mutants in turn, generates random tests as usual, and reports how many tests it took to find a counterexample for each of the mutants.

We analyzed the running time and number of tests needed to capture the bugs, by repeating QuickChick for 29 times on each mutant. For five of the 12 mutants (changing the initial message buffer, sending extra bytes from the response buffer, responding with wrong message, computing wrong connection state, and skipping the completeness check), the wrong behavior was caught by the very first test in each run. Six of the mutants passed the first test in some runs, but always failed by the second test (sending wrong number of bytes, storing to wrong message buffer, handling partial messages incorrectly, dropping one byte of message, copying response from wrong buffer, and skip populating response). The most interesting mutant was changing the return value of the recv call. 3/4 of the runs caught the bug within four rounds, but others took up to nine rounds. This mutant sometimes causes the server not to respond, which is trivially correct because our specification does not deal with liveness. As a result, the tester discarded up to three thousand test cases where the server did not respond, and ran for up to five minutes before failing. The other mutants could fail within 0.4 second with 95% confidence.

It is hard to draw definite conclusions about the effectiveness of testing from a case study of this size, but the fact that we are able to detect a dozen different bugs, most quite quickly, is an encouraging sign that this approach to testing will provide significant value as the codebase and its specification become more complex. Reports in the literature of property-based random testing of similar kinds of systems (e.g., Dropbox [Hughes et al. 2016]) are also encouraging.

7 Connecting to CertiKOS

A key pillar of the proof of correctness of the C implementation is the specification of the socket operations such as send and recv. We took these specifications as axioms when proving the implementation model, but because we are running the server on top of CertiKOS, which has its own formal specification, we should be able to go one step better: we would like to prove that the socket operations as specified by CertiKOS satisfy the axioms used in the VST proof. This part of the case study is still in progress; we report here on what we’ve achieved so far and identify the challenges that remain.

The Socket API in CertiKOS CertiKOS provides its own axiomatized specifications for the POSIX socket API. Unlike VST specifications, which are expressed as Hoare triples, CertiKOS specifications are written as state transition functions on the OS abstract state. This state is a record with a
The network component of the CertiKOS OS state is now defined as an ITree, which stores information about the network connections and their states. This represents a significant improvement over a “free monad” type, which one might use as an abstraction for network communication. It is proposed as an axiomatic “freer monad”, to describe the external network behavior. This simplifies the task of connecting the VST and CertiKOS specifiers: we just need to relate the interaction tree to some component of the OS state, and translate an assertion on a single piece of memory into the flat memory model and back. (The other socket operations do not involve any changes to user memory, though they do modify kernel memory, which is abstracted to the C program via the `SOCKAPI` predicate.)

We have explored this approach by sketching the correspondence between the VST specification of `recv` and its CertiKOS specification. We translated the VST pre- and postcondition for `recv` into step-index-free predicates on CompCert memories and interaction trees by hand, and proved the correctness of the translation using the underlying logic of VST. We then wrote functions that transfer a single block of memory between the CompCert model and the flat model, and adapted the CertiKOS OS component representing the network state to use interaction trees, so that the two systems have a common language to describe network operations. The network component of the CertiKOS OS state is now a map that, for each user process, holds an interaction tree describing the network communication that that process is allowed to perform. Finally, we are in the process of proving that the CertiKOS specification for `recv` satisfies the step-index-free, flattened versions of the VST pre- and post-condition. This gives us a path to validating the axiomatized specifications of the socket API that we rely on for the correctness of the C implementation: they can be substantiated by connection to the (axiomatized) behavior of the socket operations in the underlying operating system.

### 8 Related Work

**Interaction trees** As mentioned in Section 3, our “interaction trees” are a Coq-compatible variant of ideas found elsewhere. Kiseliov and Ishii [2015] present a similar concept under the name “freer monad”. It is proposed as an improvement over a “free monad” type, which one might hope to define in Coq as follows:

```coq
Inductive free (E : Type → Type) (R : Type) :=
  | Ret : R → free E R
  | Vis : E (free E R) → free E R. (* NOT PERMITTED!! *)
```
Unfortunately, the recursive occurrence of free in the Vis constructor is not strictly positive, so this definition will be rejected by Coq. Thus in a total language, the choice for the Vis constructor to separate the effect E X from the continuation X → itree E R is largely driven by necessity, whereas the work on freer monads proposes it as a matter of convenience and performance.

The McBride [2015] variant, which builds on earlier work by Hancock [2000], is called the “general monad.” It is defined inductively, and its effects interface replaces our single E : Type → Type parameter with a single S : Type and a type family S → Type to calculate the result type. It was introduced as a way to implement general recursive programs in a total language (Agda), by representing recursive calls as effects (i.e., Vis nodes). Our coinductively defined interaction trees also support a general (monadic) fixpoint combinator.

Letan et al. [2018] present the “program monad” to model components of complex computing systems. Like the general monad, it is defined inductively. Whereas our interpretation of iTrees is based on traces, they use a coinductively defined notion of “operational semantics” to provide the context in which to interpret programs, describing the state transitions and results associated with method calls/effects.

Our choice to use coinduction and the Tau constructor gives us a way to account for “silent” (internal) computation steps, and hence allows us to semantically distinguish terminating from silently-diverging computations (which is not easy with trace-based semantics, at least not without adding a “diverges” (terminal component to some of the traces). Although liveness is explicitly not part of our correctness specification in this project (the spec is conditioned on there being visible output), it is conceivable to strengthen the specifications and account for Tau transitions as part of the C semantics, which might allow one to prove liveness properties (although VST does not currently support that). However, there are also costs to working with coinduction: our top-level programs are defined by CoFixpoint, and coinduction is generally not as easy to use in Coq as it could be [Hur et al. 2013].

Verifying effectful systems A common approach to reasoning about effectful programs is to provide a model of the state of the outside world, with access mediated strictly through external functions. These functions may be given (possibly non-deterministic) semantics directly [Chlipala 2015], or indirectly through an oracle [Féreé et al. 2018; Gu et al. 2016]. For example, in Fèreé et al. [2018], external functions are called through a Foreign Function Interface (FFI), and specification/verification is done with respect to an instantiated FFI oracle that records external calls and defines the state of the environment and the semantics of external functions. In their work, a TextIO library was verified with respect to a model of the file system. Similarly, our specifications in terms of Hoare triples assume a model of external socket API memory, i.e., the state under the SOCAPI predicate, and describe how this state is transformed.

Stronger specifications of effectful programs can involve dynamics (“what has happened”) rather than statics (“what is the final state”). In such cases, a model of the external state is commonly extended with (or taken to be) a trace or history of past events, and specifications involve these traces. Chajed et al. [2018]; Hawblitzel et al. [2015]; Leroy [2009]; Malecha et al. [2011], etc. use this approach.

Our specifications are based on interaction trees (which can be construed as sets of traces), with one major difference: interaction trees specify “what is allowed to happen.” Rather than reasoning about lists of events that have occurred in the past, our reasoning is based on the trees of events that are allowed to be produced in the future. One main advantage of using interaction trees is that it gives us a unifying structure for specification, testing, and verification, as detailed in Section 3. A similar underlying structure to interaction trees is used as specifications of distributed systems in an early version of F” [Swamy et al. 2011], but they did not show how to use them for testing or how to do refinement.

Linearizability Network refinement is closely related to linearizability [Herlihy and Wing 1990], a correctness criterion for concurrent data structures. A data structure implementation is linearizable if, for every possible collection of client threads, the behavior of the data structure is indistinguishable from the behavior of a sequential implementation of the structure. Filipovic et al. [2009] related linearizability to contextual refinement. Network refinement is essentially the same idea of contextual refinement, but with network effects playing the role of relaxed memory. Our network model closely resembles TSO, and network refinement is similar to TSO-linearizability [Burckhardt et al. 2012].

Verifying networked servers In one early attempt at server verification, Black [1998] verified security properties of the httppd web server, based on axiomatized C semantics. That work did not establish the functional correctness of the web server, the axiomatic semantics was not testable, and it did not consider the effects of network reordering.

IronFleet [Hawblitzel et al. 2015] is a methodology for verifying distributed system implementations and it is similar to our approach in several ways: both verify the functional correctness of a networked system; both use a “one client at a time” style specification at the top-level; and both verify the correctness of a system implementation which interleaves its operations via linearizability. However, there are several major differences between IronFleet and our work: (1) We are concerned with testing, as it allows us to find implementation bugs early, and it also allows us to use the same specification for blackbox-testing of existing implementations. For these reasons, we choose the executable interaction trees to represent the specification. IronFleet focuses instead on reducing the burden of verification. It uses
non-executable state machines, and it relies on tools such as IDEs to support rapid verification. (2) Our work verifies C implementations. VST and CompCert ensure that the properties we have proved at the source-code level are preserved after the program has been compiled to assembly code. IronFleet verifies programs written in Dafny [Leino 2010], and extracts them to C#. This means that both the extraction engine and the .NET compiler must be trusted. The authors of IronFleet also suggest an alternative strategy to reduce the trusted computing base, by first translating the programs to assembly code, and verifying the assembly code using an automatically translated specification [Hawblitzel et al. 2014]. However, that still requires the specification translator to be trusted. (3) IronFleet is based on UDP, while our works is based on TCP. Nevertheless, we both need to consider packet reordering. The difference is that messages will not be reordered on each individual connection. (4) IronFleet uses TLA+ [Lamport 2002] to prove liveness properties. The partial-correctness approach of separation logic makes it more difficult to reason about liveness. CSPEC [Chajed et al. 2018] is a framework for verifying concurrent software. CSPEC focuses on reducing the number of interleavings a verifier must consider. To do that, it provides a general verification framework built on mover types [Lipton 1975]. We may be able to use mover types to simplify the process of proving network refinement.

Verdi [Wilcox et al. 2015] is a framework for verified distributed systems that work under different fault and network models. Verified System Transformers transform a distributed system verified under one model to one that works in another. In particular, the Raft system transformer [Woo et al. 2016] transforms a given state machine (server) into a distributed system of servers that synchronize state using Raft messages, over a network that may drop, reorder, or duplicate messages. Any trace of Raft I/O messages produced by the distributed system can be linearized to an I/O trace of the input state machine. Distributed systems and transformers are written in Coq and extracted to OCaml.

Ridge [2009] verified the functional correctness and linearizability of a networked, persistent message queue written in OCaml using the HOL4 theorem prover. In contrast to Verdi and Ridge’s work, our methodology focuses on testing and verifying C implementations, dealing with the full complexity of low-level programming including memory allocation and pointer aliasing.

For simplicity, our work builds on a small subset of axiomatized TCP specifications. A rigorous and experimentally-validated specification of TCP can be found in Bishop et al. [2005a,b]; Ridge et al. [2009].

Testing There is more research on testing linearizability of concurrent or distributed systems than we can summarize here, including Burckhardt et al. [2010]; Scott et al. [2016]; Shacham et al. [2011]; Vechev et al. [2009]. Our work is distinguished by the focus on unifying testing and verification in the same framework. The QuickChick property-based testing methodology has been shown to be useful in formal verification [Lampropoulos and Pierce 2018]. There are also many accounts of successfully applying property-based random testing to real-world systems. For example, Hughes and Bolinder [2011] used QuickCheck to test for race conditions in dets, a vital component of the Mnesia distributed database system; Arts et al. [2015] have applied the methodology to test the AUTOSAR Basic Software for Volvo Cars; and Hughes et al. [2016] tested the linearizability of Dropbox, the distributed synchronization service.

9 Conclusions and Future Work

Starting from a C implementation and a "one client at a time" specification of swap server behavior, we have proved that every execution of the implementation correctly follows the specification. The proof breaks down into layers of refinements: from the C program to an implementation-level interaction tree, and from there, via network refinement to the linear interaction tree. We use VST to verify the C code, pure Coq to relate the trees, QuickChick to test our specifications and implementations, and CertiKOS to validate our specifications of network communication. The result is a proof of the correctness of the swap server from the linear specification down to the interface between the C program and the operating system.

Although this work represents significant progress towards the Deep Specification project’s goal of formally-verified systems software, much remains to be done. The verification of the swap server has tested the limits of VST, in terms of both scale and style of specifications. Previous VST verifications were self-contained libraries, but this swap server interacts with the OS through the socket API, requiring us to develop new features (the external assertions) that should be useful for verifying a variety of more realistic programs.

A clear next step is to fully verify the socket API used by the server, by completing the proof that each VST socket axiom follows from the specification of the corresponding operation in CertiKOS. Doing so will require several more proofs along the lines of our verification of recv, bridging the gap between VST’s step-indexed memory and CertiKOS’s flat memory, as well as defining a suitable C-level abstraction of the kernel memory related to the socket operations. This will further extend the reach of our result, so that we rely only on the correctness of the operating system’s model of the socket API.

References


