DeepSEA: A Language for Certified System Software

VILHELM SJÖBERG, CertiK and Yale University, USA
YUYANG SANG, Yale University, USA
SHU-CHUN WENG, Yale University, USA
ZHONG SHAO, Yale University, USA

Writing certifiably correct system software is still very labor-intensive, and current programming languages are not well suited for the task. Proof assistants work best on programs written in a high-level functional style, while operating systems need low-level control over the hardware. We present DeepSEA, a language which provides support for layered specification and abstraction refinement, effect encapsulation and composition, and full equational reasoning. A single DeepSEA program is automatically compiled into a certified “layer” consisting of a C program (which is then compiled into assembly by CompCert), a low-level functional Coq specification, and a formal (Coq) proof that the C program satisfies the specification. Multiple layers can be composed and interleaved with manual proofs to ascribe a high-level specification to a program by stepwise refinement. We evaluate the language by using it to reimplement two existing verified programs: a SHA-256 hash function and an OS kernel page table manager. This new style of programming language design can directly support the development of correct-by-construction system software.

CCS Concepts: • **Software and its engineering → Software verification**; *Abstraction, modeling and modularity*; *General programming languages.*

Additional Key Words and Phrases: verification, layered specification, refinement, certified software

ACM Reference Format:
Vilhelm Sjöberg, Yuyang Sang, Shu-chun Weng, and Zhong Shao. 2019. DeepSEA: A Language for Certified System Software. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 136 (October 2019), 27 pages. https://doi.org/10.1145/3360562

1 INTRODUCTION

Formal verification is the surest way to create bug-free secure programs, and low-level system software is a particularly important verification target because the correctness of all other software depends on the OS and language runtime. But writing such software remains very expensive.

Part of the problem is due to the lack of good programming language support. Program verification works best when programs are written at a high abstraction level and support equational reasoning, while systems programming uses effects and requires low-level control of hardware resources. Existing languages make different trade-offs between these two goals: C-like languages favor control; memory-safe languages, and functional programming languages such as OCaml and Haskell, are easier to reason about but force the programmer to, for example, use a garbage collector; and total, pure languages (as provided in proof assistants such as Coq [The Coq development team

Authors’ addresses: Vilhelm Sjöberg, CertiK and Yale University, USA, vilhelm.sjoberg@yale.edu; Yuyang Sang, Yale University, USA, yuyang.sang@yale.edu; Shu-chun Weng, Yale University, USA, scweng@gmail.com; Zhong Shao, Yale University, USA, zhong.shao@yale.edu.

2475-1421/2019/10-ART136
https://doi.org/10.1145/3360562

2014]) support full equational reasoning so they can directly express how a program's implementation satisfies its specification, but do not provide any support for low-level programming. A classic paper on programming language design [Hoare 1974] advocated that a *good* programming language should support both implementation and specification:

> *... The first, and very difficult, aspect of (program) design is deciding what the program is to do, and formulating this as a clear, precise, and acceptable specification. Often just as difficult is deciding how to do it: how to divide a complex task into simpler subtasks, and to specify the purpose of each part, and define clear, precise, and efficient interfaces between them. A good programming language should give assistance in expressing not only how the program is to run, but what it is intended to accomplish; and it should enable this to be expressed at various levels, from the overall strategy to the details of coding and data representation. ...*

More than four decades later, the community has widely adopted Hoare's approach of specifying a system at multiple levels, moving from low-level implementation to high-level specification. However, even today no single existing language seems ideal for this task. Instead, two recent projects on OS verification [Gu et al. 2015; Klein et al. 2014] essentially wrote the kernel twice: first in a C-like language, and then as a "model" or "specification" in a functional language that can be reasoned about in a proof assistant. A large part of the verification effort was then spent on proving that the C implementation indeed satisfied the specification.

We are interested in the kind of verification tasks exemplified by OS development: proving full functional correctness for systems composed of many modules and with nontrivial functionality. With current technology, this requires manual proofs, so we assume that our users will spend much of their time writing proofs in an interactive proof assistant like Coq, but we want to remove as much busy-work as possible, and help them structure their system into separate modules so that the rest of the system can be verified by using just the module interface and not its implementation.

In this paper, we present a new programming language named DeepSEA (for Deep Simulation of Executable Abstraction) that combines low-level programming with abstract specification. DeepSEA is organized around four ideas.

Equational reasoning. Each DeepSEA term can be desugared into a corresponding functional specification, a typed lambda expression, which can be reasoned about equationally in a proof assistant. Effects are modelled by monadic programs.

Layered specification. Like C, DeepSEA does not come with a built-in runtime system or garbage collector. This is kept managable by structuring the system in *abstraction layers*, to specify the program in terms of a simpler abstract datatype (e.g., unbounded integers instead of 32-bit words). When verifying higher level layers, one can assume that the lower layers behave according to the abstract specification, and the DeepSEA compiler will prove that the raw C implementation behaves accordingly.

Effect encapsulation and composition. Each DeepSEA layer consists of a set of objects that are built on top of another layer. Similar to object-oriented languages, the state of an object can only be changed by the methods defined in that object, in other words each DeepSEA object encapsulates its internal side-effects.

Abstraction refinement. One can not hope to completely automatically verify a large system with respect to a user-friendly specification: this must be done in steps, and sometimes with manual proofs. We structure this process as a series of refinement proofs, which are composed to show that the linked program refines the linked specification. In DeepSEA there are two ways to build new layers: One can either add new methods, which are automatically proved to refine their (automatically generated, low-level) functional specifications, or one can add new specifications for existing methods, which creates a manual proof goal to show that the old specification refines the new one. These steps can be interleaved in a stack of layers.

This style of verification in terms of layers, objects, and functional specifications is inspired by the CertiKOS group [Gu et al. 2015], who successfully used it for building multiple certified OS kernels. However, the current CertiKOS development effort requires manually writing both the Clight [Blazy and Leroy 2009] implementation and the Coq layer specification, and a large manual (or semi-automatic) proof relating the two, and then manually use a separate layer library (in Coq) to compose the layers. DeepSEA provides support for layer definition, implementation, refinement, and composition within a single language.

We begin by presenting the key ingredients of DeepSEA through a series of examples, and show how they can be used to reason equationally about low-level code. (Sec. 2). Then we proceed to explain in more detail our specific new contributions:

- We define a functional core language which is small enough to allow us to write a verifying compiler with a reasonable effort, yet feature-rich enough to support CertiKOS-style kernel development. (Sec. 3)
- We define a calculus of layer operations, which give a concrete syntactic way to describe the style of stepwise refinement which the CertiKOS team implemented “by hand” in Coq. (Sec. 3.1).
- We describe the design of our verified compiler, which provides an end-to-end verified compilation chain from functional specifications down to assembly code. (Sec. 4)
- In particular, the compiler includes a novel framework for protecting the user from details about low-level data representation by automatically generating a refinement relation between concrete and abstract data (Sec. 4.2), automatically generating proofs about data representation (Sec. 4.4), and letting the user specify assertions in order to guide the automatic tactics (Sec. 4.3).
- We re-implement two existing verified programs (a cryptographic hash function and an OS kernel memory allocator), and compare the proof effort. Using DeepSEA significantly reduces the proof effort while still allowing adequate low-level control. (Sec. 5)

Finally we discuss related work (Sec. 6) and conclude.

2 DEEPSEA BY EXAMPLE

Consider the task of writing a memory allocator. This is used as part of the virtual memory manager (VMM) in order to find an unused page to serve a fault. The following code is a simplified excerpt from the case study in Section 5.3, which re-implemented part of the CertiKOS VMM.

The central piece of state required is the *allocation table*, which records which physical pages are currently in use, and the main method is palloc which allocates a new page. The method palloc is implemented in terms of additional state and operations. These can be high-level, such as a set of numbered *containers* which record the memory quota each process is entitled to and how much it is currently using (in order to decide whether to honor or decline requests for additional memory). The state can also be very low-level, e.g. whether the CPU is in user or kernel mode.

In DeepSEA, all state is encapsulated inside *objects* and manipulated through *methods*. We begin by specifying the types of the methods using *object signatures*.

```
signature CPUSig = {
  initialized : unit -> bool;  ...
}
signature ContainerSig = {
  container_get_usage : int -> int;        (* returns number of pages used by process i *)
  container_alloc : int -> bool             (* request another page for process i, return
                                               false if exceeded its quota. *)
  container_split : int * int * int -> int; (*  decrement the quota for i by n pages,
                                                and give them to a new process j. *)
  ...
}
signature PMMSig = {
  mem_init : int -> unit;   (* Initialize the system. *)
  palloc : int -> int;      (* Allocate a new page for use by process i. *)
  ...
}
```

The corresponding implementation is given in the form of an object declaration. The algorithm is very simple: we maintain a statically allocated array (of size maxpage, a constant), which for each page tracks whether the page is in use (b), what type of memory it contains (t), and a reference count. The palloc method scans the array to find the first unused page. Additionally, before marking the page as used, we call container_alloc to check that process id has not used up its quota. Page 0 is always marked as in use, so palloc returns 0 to signal that the allocation failed, and callers will check the return value.

The object implementation assumes that a lower layer (or *underlay* for short) will provide some functionality, and similar to ML functors [MacQueen 1984], definitions are made with respect to a *signature* {iflags: CPUSig;...} which specifies what objects the underlay should provide. Every effectful expression is typed with respect to such a signature, which tracks which effectful operations are available at that point in the program. (This is similar to how effectful code in e.g. Haskell "lives inside" a particular monad). In this case, among other things we assume that the environment provides an object container which satisfies ContainerSig. DeepSEA supports special-purpose loops, here **first** x = .. to .. **suchthat** .. **then** .. **else** .. which searches a range for the first integer satisfying a test. Using a special- rather than a general-purpose loop aids proof automation.

```
type ATType = ATResv | ATKern| ATNorm
type ATInfo = {b : bool; t : ATType; c : int}

object PMM ({iflags: CPUSig; container:ContainerSig...}) : PMMSig {
  let nps : int := 0  (*number of pages*)
  let AT : array[maxpage] ATInfo := array_init

  let palloc id =
    assert iflags.initialized ();
    let n = nps in
    first i = 0 to n suchthat AT[i].b = false
      then begin
        let within_quota = container.container_alloc id in
        if within_quota then begin
           AT[i].b := true;
           i
        end else
          0
      end else
        0
  ...
}
```

2.1 Assertions

Methods often have preconditions. For example, the arguments must be in a certain range, or they may only be called after initialization. We express such preconditions using assert-statements. Semantically, the meaning of a failing assert is that the program execution becomes undefined. However, DeepSEA is set up so that this never happens: during verification the programmer proves that all asserts will be satisfied, so there is no need to check them at runtime and the compiler ignores them when generating code.

Unlike function preconditions in some other programming languages, assert statements can occur in the middle of a function body as well as at the beginning. This can be convenient for the programmer, and fits naturally with the semantics we use (the execution reaches that point and then aborts).

2.2 Equational Reasoning and Low-level Data Representation

Having defined palloc, we want to prove theorems about it. For example, we may want to prove that requests will be granted as long as the quota is not exceeded, or that calling palloc twice will return two different page numbers.

In order to enable this, DeepSEA automatically generates a *functional specification* of each method– a function with the same behavior, but written in pure Coq to enable equational reasoning.

Since DeepSEA itself is not pure, we need to encode effects functionally. We use the standard technique of writing code in a *monad.* As we have seen above, commands can have two side effects: modifying the object state, and triggering a failing assertion. We can capture that by modeling "a command that returns A" by using the monadic type $DS\ A \equiv T \to \text{Option}\ (T \times A)$ where T is the type of mutable state and implement the usual helper functions such as $\text{gets} : (T \to A) \to DS\ A$ and $\text{guard} : \text{Bool} \to DS()$.

```
PMM_palloc_opt (id : Z) : DS Z :=
 (v <- MContainerIFlags_initialized_opt;; guard v);;      (* assert iflags.initialized (); *)
 n <- gets nps;;                                          (* let n = nps in  *)
 s <- get;;
 first_spec                                               (* first i = ... *)
  (fun i : Z =>                                             (* suchthat AT[i].b = false *)
    b <- gets (fun s => (ZMap.get i (AT s)).(b));;
    ret (b==false))
  (fun i : Z =>                                           (* then *)
    within_quota <- MContainerContainer_container_alloc_opt id;;
    (if within_quota                                        (* if within_quota *)
     then                                                     (*then *)
      modify (fun s =>                                          (* AT[i].b := true; *)
         s {AT : ZMap.set n ((ZMap.get n (AT s)) {b : true }) (AT s)});;
      ret i                                                     (* i *)
     else ret 0))                                             (* else 0 *)
  (fun i => ret 0)                                        (* else 0 *)
  0 n s                                                   (* ... first i = 0 to n *)
```

The example shows how the various features of DeepSEA are handled. DeepSEA generates a Coq function for each method, e.g. palloc in the first line and container_alloc from the underlay. The name of the specification function ends with _opt because it uses the "option monad" to represent assertion failures. Specifically, assertions are handled using the helper function guard which returns a value in Option, either SOME or NONE. The mutable state of the object is handled by the state monad: the type DS bool implicitly takes an extra argument s, which is a record with a field for each variable in the program—e.g. the field accessor AT. The monad is quite simple thanks to the minimalistic feature set of DeepSEA (Sec. 3). In particular there is no dynamic object creation, so the set of record fields can be fixed. The "first" loop is translated by using a Coq function first_spec which applies its first argument function to each number between 0 and n and then calls the second argument function on the result.

Like all specifications generated by DeepSEA, PMM_palloc_opt is a deterministic function, as opposed to only specifying some nondeterministic set of possible return values (such as "this function returns a sorted list"). Ultimately the user will manually prove some correctness property about these specifications, but we believe in the approach of first proving a "deep" specification [Gu et al. 2015] which says exactly what the implementation does in a functional form, and DeepSEA largely automates this first step.

The generated spec follows the input program line by line, and in fact basically looks the same as the input except for the surface syntax. (In the above figure we added comments to show the corresponding lines in the source program.) The user will probably end up spending more time looking at this spec than at the input program, because proving anything interesting about it still requires a similar amount of work as proving something about a function originally written in Coq.

However, for the user doing the verification this is much better than directly reasoning about C programs in Coq. First, we are working with an ordinary Coq function, rather than having to reason indirectly about a formalization in Coq of the C operational semantics. Second, we have abstracted away a lot of the details of the data representation. Instead of C's 32-bit integers, the Coq program operates on mathematical unbounded integers Z. Instead of an array, the allocation table is represented by a mathematical finite map ZMap. Although the ATType will be represented as a C integer, the DeepSEA program defines it as an algebraic data type, which is good for verification because it concisely expresses what valid data looks like and lets us use pattern matching syntax to enforce that all the possible cases are covered. The programmer can annotate the datatype declaration to specify the concrete representation, which is useful when the x86 ISA requires particular bit patterns in the page table.

```
type ATType [[int]] =
  ATResv [[= 0]] | ATKern [[= 1]] | ATNorm [[= 2]]
```

This extra power requires additional invariants. For example, not every int corresponds to a valid ATType, only the ones between 0 and 2 do. To prove that the compiled code implements the functional specification, the compiler needs to maintain a nontrivial relation between the abstract state and the corresponding C program state. Similarly, each integer operation must be proven to not overflow its 32-bit format, and each array access must be proven to be in bounds. Behind the scenes, the DeepSEA compiler uses a library of Coq tactics to automatically prove these properties (Sec. 4.4). The tactics can deal with basic arithmetic operations, and values that are computed by straight-line code, but they may get stuck on more complicated problems. Here, DeepSEA's notion of assertions play a second role, by pushing the verification conditions to the user.

For example, in the page table init method, there is a somewhat complicated loop which analyses the parameters provided via the bootloading, and then assigns the variable nps. As mentioned above, we need that nps < maxpage, but we cannot expect an automated theorem prover to prove something about a number n computed by an arbitrary loop. Instead, we add an assertion.

```
let mem_init mbi =
  (* loop to compute n. *)
  assert n < maxpage;
  nps := n;
  (* more initialization code. *)
```

In the generated specification, this corresponds to one more guard statement, i.e. one more place where the function may be undefined (return NONE). So in order to prove something useful about the function, the programmer will need to invent a strong enough loop invariant to prove the inequality. Meanwhile, DeepSEA is set up so that any asserted formula is automatically added as an assumption for the automated tactics to use (Sec. 4.3), so the correctness of array accesses becomes easy to prove. The assertion moved the proof obligation from the machine to the human.

2.3 Invariants

It is also possible to specify that the state of an object satisfies an *invariant*, which must be preserved by all methods. Often this is needed in order to prove that the assert-statements in a method are satisfied if they refer to data from the object fields.

For example, one of the objects in the memory manager is the Container, which keeps track of how much memory each process is entitled to, and has methods to transfer some of a process's quota to its child processes. (See above for method type declarations.) It is implemented by keeping an array of structs:

```
type AgentContainer = {
  ac_quota : int;  ac_usage : int;
  ac_parent : int; ac_children : int_list; ac_used : bool
}

object Container (iflags : CPUSig) : ContainerSig {
  let AC : array[NUM_ID] AgentContainer  := array_init
  ...
}
```

We need many invariants about this data, for example that the parent and child ids and the quota numbers are always in-range. We do not duplicate Coq's facilities for writing mathematical definitions. Instead we define the invariant as a predicate in Coq, which we can later refer to in the DeepSEA source file. Note that this definition uses the abstracted representation of the data, e.g. the numbers are mathematical integers and the array is represented by a finite map (ZMap).

```
Record ContainerPool_valid (C : AgentContainer) : Prop := {
  cvalid_id : forall i, ac_used (ZMap.get i C) = true -> 0 <= i < NUM_ID;
  cvalid_quota : forall i, ac_used (ZMap.get i C) = true ->
                         ac_quota (ZMap.get i C) <= Int.max_unsigned;
  ...
}.
Definition MContainer_high_level_invariant data := ContainerPool_valid (AC data).
```

2.4 Layer Composition

Once the objects are defined, they can be grouped into *layers*. A layer provides a view of the state of the whole system at a suitable abstraction level.

For example, the low-level hardware model in CertiKOS includes an object iflags which exposes, for example, whether the CPU is in kernel or user mode, and also an object flat_mem which provides a raw view of the memory in the computer. We group these together into a layer called MBoot, and ascribe it a *layer signature* MBootSig. Both objects depend on the same lower-level interface (in this case the empty signature {}, because these are trusted layers at the lowest level and therefore make no imports). Programming against MBootSig is like writing a program to run immediately after the boot loader finishes, without any OS to help manage the computer.

```
layer signature MBootSig
 = { iflags : CPUSig; flat_mem : FlatMemSig }
layer MBoot : [ {} ]MBootSig
 = { iflags = MBootCPU; flat_mem = MBootFlatMem }
```

Typically, layers are defined on top of some non-empty signature. For example, the methods used to implement the `Container` object have preconditions about being in kernel mode, which are stated using an object `iflags`. We can now create a new layer `MContainerImpl`, which augments `MBoot` with an object tracking memory quotas. The `MBootSig` stated inside `[ ]` describes the layer right beneath it.

```
layer signature MContainerSig
 = { iflags : CPUSig; flat_mem : FlatMemSig; container : ContainerSig }
layer MContainerImpl : [MBootSig]MContainerSig
 = {  container = Container }
```

At this point, the layer expression can also be annotated with the invariant that the data in this layer satisfies.

```
layer MContainerImpl : [MBootSig]MContainerSig
 = {  container = Container } assert "MContainer_high_level_invariant"
```

Note that the typing rules for layers do not require the signatures to match exactly (the types can be “relaxed”). For example, `MContainerImpl` only uses the `iflags` object, but the underlying layer also provides `flat_mem`. Also, even though only `container` is mentioned in the layer definition, `iflags` and `flat_mem` are mentioned in the signature, and *passed through* from the underlay. This is a common idiom; intuitively, programming against `MContainerImpl` is like working with a new abstract machine which has been augmented with a memory accounting facility. If we don’t want to pass through an object, it can be hidden by omitting it from the layer signature.

Two layers defined against the same underlay can be composed *horizontally*, $L_1 \oplus L_2$, producing a new layer containing the objects of both. Similarly, two layers can be composed *vertically*, $L_1 @ L_2$, if the underlay signature required by L_1 matches the signature exported by L_2. For example, we can instantiate `MContainerImpl` with the underlay `MBoot` to create a ground layer (i.e. one making no imports). In a ground layer the specification can be evaluated fully (because all methods have known specifications), and of course only ground layers can be compiled to executable code. This is shown by the list of imports (in square brackets) being the empty set `{}`.

```
layer MContainer_impl : [ {} ]MContainerSig
   = MContainerImpl @ MBoot
```

2.5 Pure Layer Refinement

The point of abstraction layers is to present a view of the state of the program. As we saw above, one way to create such layers is the same as what programmers in conventional C-like languages do: write a new set of methods which call the API exposed by the lower layer. But DeepSEA also provides a second way: to keep the same set of methods, but change their specifications and the type of the program state.

Concretely, DeepSEA objects and layers can be marked as `logical`, meaning that we don’t expect to automatically generate C code from the methods in them. The methods can still be desugared into Coq functions in the same way as above, and additionally one can import arbitrary types and function definitions written directly in Coq. If L_1 is a logical layer, L_2 is some other layer, and the user has manually defined a binary relation R between the state of L_1 and L_2, then the expression L_1 :> L_2 `with` R defines an *pure refinement* L_2. The DeepSEA compiler will generate a proof obligation for each method, where the user has to manually show that the logical specification in L_1 can be simulated by the specifications in L_2.

For a simple example, consider the assert statement in `mem_init` mentioned above. We can define a new layer L_1 which has the same method except it omits the assert statement, and use the identity relation for R. Since we do not generate code for L_1 we do not need the assert statement to guide the automated tactics. The compiler will create an obligation asking us to prove that the methods `mem_init` in L_1 and L_2 always return the same value; which is trivial except we need to show that the assertion in L_2 is satisfied. The layer (L_1 :> L_2 `with` R) can then be used as an underlay for further layers, and the generated specifications will include the "better" specification, so this allows us to insert a manual proof (of the asserted formula) into a development which mostly used automated proofs.

For the `Container` object, we use pure layer refinement in a more interesting way to add a form of ghost state. In the executable code, we just need to track how many children each process has (if a process has nonzero children it can't be deleted yet), but for proving properties it is useful to have an explicit list of each process's children.

We handle this by defining two objects, the concrete `MContainerImpl` which stores an integer count, and the logical `MContainer` which keeps a (mathematical) Coq list. The user defines a relation `ContainerRefineDataRel` which states that the integer count corresponds to the length of the list, and for every method such as `container_split`, DeepSEA will ask the user to prove a set of theorems such as:

```
forall a0 a1 a2 d1 d2 d1' r,
  ContainerRefineDataRel d1 d2 ->
  MContainerContainer_container_split_spec a0 a1 a2 d1 = Some (r, d1') ->
  exists d2', MContainerContainerImpl_container_split_spec a0 a1 a2 d2 = Some (r, d2')
             /\ ContainerRefineDataRel d1' d2'
```

The theorem just says that the implementation method satisfies the new specification we are ascribing to it, but there are two interesting points to note. First, it's stated entirely in terms of the generated functional specifications, so when proving it we are insulated from the grubby details of C. (The function ..._`split_spec` is a simple wrapper around ..._`split_opt`.) Second, it's stated in the *downwards* direction, the implementation simulates the specification, which is generally easier to prove (this way we don't have to invent an abstract list). Because all our specifications are deterministic functions, we can use known techniques [Gu et al. 2016; Leroy 2009] to automatically also show an upwards simulation as a corollary, and by composing the simulations we get that the executable code refines the high-level specification.

Pure layer refinement lets us apply the refinement-based approach that previous groups have used, but in a more selective way. Instead of writing duplicate implementations and specifications for the entire program, we can generate most of the specifications automatically but manually add custom specifications for individual methods as required.

A single DeepSEA layer expression such as

```
layer MContainer_impl : [ { } ] MContainerSig =
  (MContainer :> MContainerImpl with "ContainerRefineDataRel") @ MContainerLow @ MBoot_impl
```

provides a short way to explain multiple proof steps: it takes a concrete layer `MContainerLow`, ascribes more informative specifications `MContainer` to some of the methods, and satisfies its dependencies using another layer `MBoot_impl`. In previous work, this involved writing hundreds of lines of Coq code spread over multiple files. In DeepSEA it is possible to see the large-scale structure of the proof at a glance.

3 THE DEEPSEA LANGUAGE

The DeepSEA language consists of a small typed expression and command language, and a layer calculus to glue them together.

The DeepSEA compiler works in two steps. It first parses and type-checks the input file and creates a typed intermediate term, which is finally *desugared* into a functional specification. The desugaring acts as the formal semantics for the DeepSEA language. As we will see, the desugaring is a very simple transformation, so the programmer can easily look at the generated specifications and work with them in an interactive proof assistant. The language shown in this section corresponds to abstract syntax after parsing, omitting a few convenience features provided by the type-checker.

Omitted features. In order to keep the implementation simple, the functionality of the language is kept minimal. There is no use of C pointers and no built-in support of dynamic memory allocation (every DeepSEA object is realized as a set of static variables), so programs that need dynamic allocation will have to implement it themselves (using indices into an array, like the page table in Sec. 2). The CertiKOS kernel was written in this style, avoiding the use of pointers in order to not have to reason about aliasing. There are also no higher-order functions.

DeepSEA also does not natively support concurrency, or low level features like interrupt handlers, access to system registers, etc. This means that one can not write an operating system entirely in DeepSEA, but the design is interoperable with existing CertiKOS-style software development, so the user can manually write C or assembly code for the parts that fall outside the scope of DeepSEA. This is analogous to how existing operating systems are written in a mixture of C and inline assembly.

3.1 The Typed Language

Expressions are typed in just a variable environment Γ, while commands are typed with respect to both Γ and a layer signature I. The signature I specifies the set of objects in the underlay (named by slot identifiers s), and for each underlying object it specifies the types of its methods (named by method names f_i). The expression/command typing rules are entirely standard for a simply-typed language, and the way commands and pure expressions are sequenced is very similar to other effectful calculi (such as the languages of Moggi [Moggi 1989] and Levy [Levy 1999]), so in this paper we omit the typing rules and just show the syntax of the language (Figure 1).

A and B range over types. These include defined type names τ, primitive types like int and bool, as well as pair types $A \times B$, tagged union types $A + B$, functions, arrays (of fixed size N), record types, and algebraic datatypes. We saw examples of the last two in Sec. 2 (ATInfo and ATType).

Next, we syntactically distinguish (side-effect-free) expressions e from (effectful) commands c. Expressions include variables, constants, arithmetic operations ($\pm$) and comparisons ($\lesseqgtr$), and operations like construction of pairs and tagged values. Arrays are constructed (as uninitialized memory) by array_init, and accessed by e[e']. Finally we can construct new records, access record fields, and construct datatype values.

Commands include returning a pure expression (val(e)), sequencing commands (let), reading and writing object fields (v), case-analyzing and destructing data (if, match), bounded loops (for), and invoking operations from the underlay interface ($s.f\ e$).

assert(c) states a method precondition. The asserted expression c is a boolean, so it's simply-typed and evaluable. This is in contrast to layer invariants ϕ, which are given as part of the layer definitions, and can be any proposition in the underlying theorem prover.

Layer expressions. The more novel part of the calculus is how the commands are organized into layers. A layer in DeepSEA is a description of the state of the system and a collection of methods to

(Types)

$$
\begin{array}{lll}
A, B ::= \tau \mid \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid A \times B \mid A + B \mid A \rightarrow B \\
\qquad \mid \texttt{array}[N]A \mid X^{\mathsf{A}}\{s_i \mapsto A_i\}^* \\
\qquad \mid X^{\mathsf{A}}\{s_1\ \bar{A}_1\ X^{\mathsf{s}}_1 \mid \cdots \mid s_n\ \bar{A}_n\ X^{\mathsf{s}}_n\}
\end{array}
$$

(Expressions and commands)

$$
\begin{array}{lll}
e & ::= x \mid n \mid e_1 \pm e_2 \mid e_1 \lesseqgtr e_2 \mid \texttt{true} \mid \texttt{false} & \\
 & \mid () \mid (e_1, e_2) \mid \texttt{inl}(e) \mid \texttt{inr}(e) & \\
 & \mid \texttt{array\_init} \mid e[e] \mid \{s_i \mapsto e_i\}^* \mid e.s \mid s\ \bar{e} & \\
c & ::= \texttt{val}(e) \mid v \mid v \leftarrow e \mid s.f\ e \mid \texttt{let}\ x = c_1\ \texttt{in}\ c_2 & \\
 & \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 & \\
 & \mid \texttt{match}\ e\ \texttt{with}\ \texttt{inl}(x) \Rightarrow c_1 \mid \texttt{inr}(y) \Rightarrow c_2 & \\
 & \mid \texttt{match}\ e\ \texttt{with}\ (x, y) \Rightarrow c & \\
 & \mid \texttt{match}\ e\ \texttt{with}\ s_1\ \bar{x}_1 \Rightarrow c_1 \mid \cdots \mid s_n\ \bar{x}_n \Rightarrow c_n & \\
 & \mid \texttt{assert}(c) & \\
 & \mid \texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{do}\ c & \\
S & ::= \{f_i : A_i\}^* \quad I ::= \{s_i \mapsto S_i\}^* & \\
D & ::= \texttt{object}\ w := O & \\
 & \mid \texttt{layer}\ l := C & \\
 & \mid \texttt{layer}\ l := C\ \texttt{assert}\ \phi & \\
 & \mid \texttt{type}\ \tau := A & \text{(Type definition)} \\
O & ::= \texttt{Obj}^I\{\{v_j : B_j := e_j\}^*; \{f_i\ x := c_i\}^*\} & \text{(Constructors)} \\
 & \mid ([I]S :> [I']S)\ w & \text{(Relaxations)} \\
C & ::= \{s_i \mapsto w_i\}^* & \text{(Constructors)} \\
 & \mid l_1 @ l_2 & \text{(Instantiations)} \\
 & \mid l_1 \oplus l_2 & \text{(Aggregations)} \\
 & \mid ([I_1]I_2 :> [I'_1]I'_2)\ l & \text{(Relaxations)} \\
 & \mid l_1 :> l_2\ \texttt{with}\ R & \text{(Refinements)} \\
P & ::= D; P \mid \emptyset & \text{(Programs)}
\end{array}
$$

Fig. 1. Syntax

manipulate that state. Typically a given method will only affect a small portion of the full system state (e.g. a disk driver will never touch the state of the network), and in order to express this we partition the state into several *objects*, where each object consists of some state and a set of methods. Semantically there is no difference between grouping states and methods into one object or several, but because the DeepSEA type system enforces encapsulation, the objects could enable better reasoning principles, e.g. in future work we may automatically generate "frame lemmas."

To define a new layer in DeepSEA, we first define several objects by definitions of the form

$$\texttt{object}\ w := \texttt{Obj}^I\{\{v_j : B_j := e_j\}^*; \{f_i\ x := c_i\}^*\}$$

Each object contains a set of data fields named v_j, with type B_j and initial value e_j, and a set of methods f_i. The method bodies c_i are checked with respect to the underlay I, and in an environment extended with the object fields. The judgment produces a new object type declaration $w : [I]\{f_i : A_i \rightarrow C_i\}^*\}$ which will be in scope when the subsequent definitions are checked.

If an object can be defined with respect to some underlay signature I, it will also work over any larger interface I'. We express this using *relaxation* $([I]S :> [I']S)w$. In the surface syntax, both object definition and relaxation can be combined in a single definition `object w (I) : S {...}`.

Then the various objects are assembled into layers. A layer which exports objects I_2 and is implemented on underlying methods I_1 is given the type $[I_1]I_2$. There are five ways to define such a layer. First, by directly specifying the objects ($\{s_i \mapsto w_i\}^*$). Second, if l_1 can be implemented on top of the interface exposed by l_2, then the two can be combined into a single layer $l_1@l_2$ by *vertical composition*. Third, two layers l_1 and l_2 which are defined for the same underlay interface I can similarly be combined into a single layer $l_1 \oplus l_2$ by *horizontal composition* as long as their defined names do not clash. Fourth, one can *relax* the type of an existing layer to a weaker type $(([I_1]I_2 :> [I_1']I_2')\ l)$. The syntax for relaxation is verbose in the AST because it is fully annotated, but in the source language the programmer can write just $l :> [I_1']I_2'$ and the elaborator will infer the original type $[I_1]I_2$.

Fifth, two layers can be combined when one *refines* the other ($l_1 :> l_2$ `with` R), as we explained in section 2.5. For the expression to be well typed the two layers must expose the same signature. In addition to type checking, the compiler will also create proof obligations requiring the programmer to prove that R is a simulation relation. Trying to ascribe a "wrong" specification l_1 to l_2 will yield a well-typed but unverifiable program.

In the figure the language gives a name to every object and layer expression (in other words, the layer expressions are in "A-normal form"). In the source language there is no such restriction; the elaborator will expand out compound layer expressions.

3.2 Desugaring

In order to verify a program, we want to express what it is doing in a form that is easy to reason about. To that end, we *desugar* each layer into simply-typed lambda calculus. To express the fact that a layer depends on its underlay, we assume that the typing context contains variables for each of the objects exported by the underlay.

Further, as mentioned in section 2.2, we write code in the state monad to encode effects functionally. We assume that the environment includes operators `ret`, `bind`, `set` $: T \to M^T$`unit`, and `get` $: M^T T$. Given these we can desugar `DeepSEA` terms into lambda calculus expressions. `DeepSEA` expressions are a subset of lambda calculus and are directly embedded when desugaring. Commands are rewritten using the monadic combinators as follows. Note that because `for`-loops are bounded, they can be translated using a total function `forM`.

$$\begin{aligned}
\mathit{spec}(\texttt{val}(e)) &= \texttt{ret}\ e \\
\mathit{spec}(v) &= \texttt{bind get}\ (\lambda t.\ t.v) \\
\mathit{spec}(v \leftarrow e) &= \texttt{bind get} \\
&\quad (\lambda t.\ \texttt{set}\ \{t\ \texttt{with}\ v = e\}) \\
\mathit{spec}(\texttt{let}\ x = c_1\ \texttt{in}\ c_2) &= \texttt{bind}\ \mathit{spec}(c_1)\ (\lambda x.\ \mathit{spec}(c_2)) \\
\mathit{spec}(s.f\ e) &= s.f\ e \\
\mathit{spec}(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2) &= \texttt{if}\ e\ \texttt{then}\ \mathit{spec}(c_1)\ \texttt{else}\ \mathit{spec}(c_2) \\
\mathit{spec}(\texttt{for}\ x =\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c) &= \texttt{forM}\ e_1\ e_2\ \mathit{spec}(c) \\
\mathit{spec}(\texttt{match}\ e\ \texttt{with}\ (x,y) \Rightarrow c) &= \texttt{match}\ e\ \texttt{with}\ (x,y) \Rightarrow \mathit{spec}(c) \\
\mathit{spec}(\texttt{assert}(c)) &= \texttt{guard}\ \mathit{spec}(c)
\end{aligned}$$

Given the functional specification for each method, we can define the meaning of the larger units (objects and layers). To do so, we translate `DeepSEA` layer expressions into the *layer calculus* of Gu et al. [2015]. Direct object definitions are translated to finite maps from identifiers to methods specifications, while the @, $\oplus$ and :> operations are translated into the corresponding layer

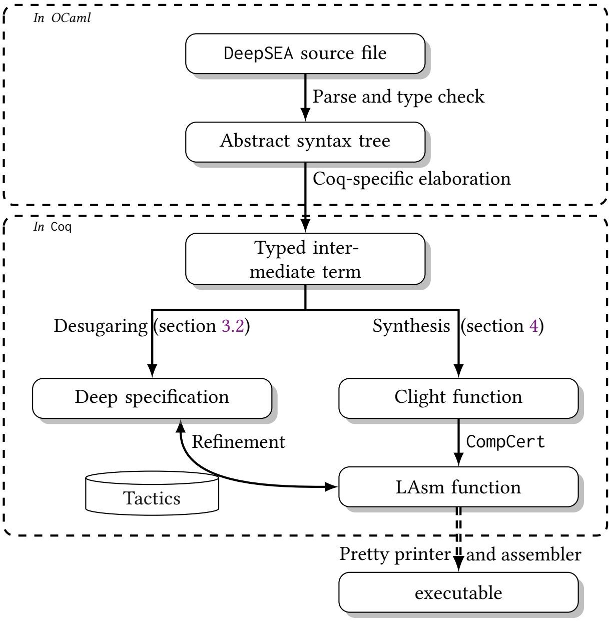

Fig. 2. The DeepSEA compilation pipeline

calculus combinators. The layer calculus Coq library automatically composes refinement proofs when building larger layers from smaller ones.

4 CERTIFIED COMPILATION

Our compiler (Fig. 2) takes an input file written in DeepSEA language and generates a C implementation, a functional Coq specification and a Coq proof that the implementation satisfies the specification. The first step is a frontend implemented in OCaml which typechecks the input and generates a directory of Coq files containing the AST of the program and the proofs for each layer. These files import the main functions and correctness theorems from the DeepSEA Coq implementation, so the full implementation of DeepSEA is 10,212 lines of OCaml and 18,235 lines (excluding comments) of Coq.

4.1 Generating Runnable Code

The generated specifications are produced by applying a Coq function to the AST. For the implementations, DeepSEA first applies a Coq function on the same AST representation to obtain a C AST defined in Coq, then applies CompCert to compile it into x86 assembly. Because we import CompCert (which is written in Coq) into our Coq project, we get an internalized end-to-end correctness theorem from the specifications down to assembly. This correctness theorem is stated using CompCert's operational semantics for the "Clight" C subset and for Assembly: we prove "if the functional spec evaluates to a value, then the generated C program evaluates to the same value", which composes with the CompCert correctness statement "if the C program evaluates to a value, then the compiled assembly evaluates to the same value". (In particular, this means that the evaluation of the C program is defined, with no C undefined behavior.)

In order to actually produce executable code the entire Coq project is compiled (using Coq's extraction mechanism) into an OCaml program which, if executed, will print out the assembly.

$$
\begin{array}{lrl}
\text{C type} & ty ::= & \texttt{uint32\_t} \\
 & & \mid \texttt{struct}(\{id_i \mapsto ty_i\}^*) \\
 & & \mid \texttt{array}(ty, n) \mid \texttt{void} \\
\text{C value} & vl ::= & \texttt{Vint}\ n \mid \texttt{Vptr}\ p \\
\text{Ext. C value} & cv ::= & \texttt{CVval}\ vl \\
 & & \mid \{id_i \mapsto cv_i\}^* \\
 & & \mid [cv_i]^*
\end{array}
$$

$$\boxed{p \overset{ty}{\underset{m}{\leadsto}} cv}$$

$$\frac{\forall n.\ 0 \le n < N \Rightarrow p + n \times \texttt{sizeof}(ty) \overset{ty}{\underset{m}{\leadsto}} cv_n}{p \overset{\texttt{array}(ty,N)}{\underset{m}{\leadsto}} [cv_i]^*}$$

$$\frac{\phi = (id_i, \tau_i)^* \quad \forall i.\ p + \texttt{field\_offset}(id_i, \phi) \overset{\tau}{\underset{m}{\leadsto}} cv_i}{p \overset{\texttt{struct}(\phi)}{\underset{m}{\leadsto}} \{s_i \mapsto cv_i\}^*}$$

$$\frac{}{p \overset{ty}{\underset{m}{\leadsto}} \texttt{CVval}(\texttt{load}(m, p))}$$

Fig. 3. Complex values and match relation.

Alternatively, this program can be given a command-line option to only pretty-print the C code without compiling it, and the user can then use a faster but less trustworthy compiler, like gcc.

This setup creates a very small trusted computing base. One must inspect the statement of the theorems one proves about the programs, and any specifications from a "trusted" layer at the bottom of the layer stack. And we rely on the x86 assembly semantics formalized by CompCert, the correctness of CompCert pretty printer phase (since the assembly semantics don't specify the binary instruction format), and the correctness of Coq (including extraction). On the other hand, although we use CompCert's C semantics we don't need to trust it since CompCert itself is verified. (Of course, if you use gcc, you also have to trust CompCert's C semantics, CompCert's pretty-printer, and the gcc implementation.) Similarly a bug in the DS frontend (written in OCaml) could make it generate a bogus Coq specification, but typically the generated Coq specifications are not the end product. For example, in the SHA case study (Sec. 5.2 below) we go on to prove that those specifications are equivalent to a nicer functional spec.

4.2 Datatype Synthesis

We now go on to describe the structure of the compilation and correctness proof in detail. First, in order to relate the "mathematical" types (e.g. unbounded integers, algebraic datatypes) in the generated spec with the C types in the code, the compiler must generate two things. First a *refinement relation* $\mathfrak{R}$, which relates a Coq record containing one mathematical value for each object field in the input program, to a Clight memory which concretely represents the program state. Second, for each object field v it must generate a set of operations to load and assign to the field, and each operation must be proven correct with respect to the refinement relation.

This task is done in two steps. First, Clight only has a notion of word-sized values (integers and pointers), which is inconveniently low-level. We define a grammar of *extended C values* (either

$$\begin{array}{ll}
\text{Type assign.} & rty : A \to ty \\
rty(\texttt{int}) & = \texttt{uint32\_t} \\
rty(\texttt{uint}) & = \texttt{uint32\_t} \\
rty(\texttt{bool}) & = \texttt{uint32\_t} \\
rty(\texttt{unit}) & = \texttt{void} \\
rty(A \times B) & = \texttt{struct}(\{id_A \mapsto rty(A), id_B \mapsto rty(B)\}) \\
rty(\{s_i \mapsto A_i\}) & = \texttt{struct}(\{s_i \mapsto rty(A_i)\}^*) \\
rty(\texttt{array[}N\texttt{]}A) & = \texttt{array}(rty(A), N)
\end{array}$$

$$\begin{array}{l}
\text{Type rel.} \qquad R_A \subseteq \mathcal{P}(A \times cv) \\
R_{\texttt{int}}\ n\ \texttt{CVval(Vint}\ n\texttt{)} \iff 0 \leq n < 2^{32} \\
R_{\texttt{uint}}\texttt{(Vint}\ n\texttt{)}\ \texttt{(CVval(Vint}\ n\texttt{))} \iff \texttt{True} \\
R_{\texttt{bool}}\texttt{true}\ \texttt{(Vint}\ n\texttt{))} \iff (n = 1) \\
R_{\texttt{bool}}\texttt{false}\ \texttt{(Vint}\ n\texttt{))} \iff (n = 0) \\
R_{\texttt{unit}}\ ()\ \texttt{(Vint}\ n\texttt{)} \iff (n = 0)
\end{array}$$

$$\boxed{R_A}$$

$$\frac{R_A a\ cv_a \qquad R_B b\ cv_b}{R_{A \times B}(a, b)\ \{id_A \mapsto cv_a, id_B \mapsto cv_b\}}$$

$$\frac{\forall i.\ R_{A_i}\ a_i\ cv_i}{R_{\{s_i \mapsto A_i\}}\{s_i \mapsto a_i\}^*\ \{s_i \mapsto cv_i\}^*} \qquad \frac{\forall i. \quad 0 \leq i < N \implies R_{A_i} a_i\ cv_i}{R_{\texttt{array[}N\texttt{]}A}[a_i]^*\ [cv_i]^*}$$

$$\boxed{\Re\ t\ (m, a)}$$

$$\frac{R_{A_i}\ t.v_i\ cv_i \qquad p_i \overset{rty(A_i)}{\underset{m}{\rightsquigarrow}} cv_i}{\Re\ t\ (m, a)} \quad \text{for each new field } v_i : A_i$$

Fig. 4. Datatype implementation and refinement relation

word-sized values, structs, or arrays), and inductively define the "match" relation $p \overset{ty}{\underset{m}{\rightsquigarrow}} cv$ which states that the C memory m and pointer p represents the complex value cv. (Fig. 3). For word-sized types ty, the match relation simply looks up the pointer in the memory using the operation $\texttt{load}(m, p)$ which is defined in the CompCert C semantics. For complex types, it recursively checks that each element of the struct or array value matches the corresponding memory location.

Second, for each type A we choose a C type $rty(A)$, and define a refinement relation R_A between specification values and extended C values. (Fig. 4) The relation can equivalently be considered as a map from specification values to extended C values, and a domain on which the relation is valid. There can usefully be more than one such relation for the same underlying C datatype. For example, DeepSEA provides two integer types which are both implemented by unsigned C int:

- For `int` the generated specifications use mathematical unbounded integers, which is good for counting things, so arithmetic like $x + y$ in the DeepSEA program creates specifications

$$\frac{\Gamma \vdash_{\mathsf{L}} e_1 : A \rightsquigarrow e_1' \qquad \Gamma \vdash_{\mathsf{R}} e_2 : A \rightsquigarrow e_2' \qquad \texttt{access\_mode}(rty(A)) = \texttt{By\_value}}{I \mid \Gamma \vdash_{\mathsf{c}} e_1 \leftarrow e_2 : \texttt{unit} \overset{z}{\rightsquigarrow} e_1' \leftarrow e_2'}$$

$$\frac{}{I \mid \Gamma \vdash_{\mathsf{c}} \texttt{assert}\ c : \texttt{unit} \overset{z}{\rightsquigarrow} \texttt{skip}}$$

$$\frac{I \mid \Gamma \vdash_{\mathsf{c}} c_1 : \texttt{unit} \overset{z}{\rightsquigarrow} s_1 \qquad I \mid \Gamma \vdash_{\mathsf{c}} c_2 : A \overset{z}{\rightsquigarrow} s_2}{I \mid \Gamma \vdash_{\mathsf{c}} c_1; c_2 : A \overset{z}{\rightsquigarrow} s_1; s_2}$$

$$\frac{I \mid \Gamma \vdash_{\mathsf{c}} c_1 : A \overset{x}{\rightsquigarrow} s_1 \qquad I \mid \Gamma, x : A \vdash_{\mathsf{c}} c_2 : B \overset{z}{\rightsquigarrow} s_2}{I \mid \Gamma \vdash_{\mathsf{c}} \texttt{let}\ x = c_1\ \texttt{in}\ c_2 : B \overset{z}{\rightsquigarrow} s_1; s_2}$$

$$\frac{\Gamma \vdash_{\mathsf{R}} e_i : \texttt{int} \rightsquigarrow e_i' \qquad I \mid \Gamma, x : \texttt{int} \vdash_{\mathsf{c}} c : \texttt{unit} \overset{z}{\rightsquigarrow} s}{\begin{array}{l} I \mid \Gamma \vdash_{\mathsf{c}} \texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{do}\ c : \texttt{unit} \overset{z}{\rightsquigarrow} \\ \qquad\qquad x := e_1'; \texttt{while}\ (x < e_2')\ \{s; x := x + 1\} \end{array}}$$

Fig. 5. Command synthesis (selected rules)

that use mathematical addition, but we generate a verification condition (See Sec 4.3 below) to prove that the operations do not overflow.

- The type `uint` is also implemented as unsigned C ints, but the specifications use nonnegative integers modulo 2^{32}, which is good for bitwise operations and does not generate any verification condition. The relation is defined inductively by the cases shown in the figure.

For user-defined types, the refinement relation and operations are defined by following the structure of the DeepSEA type and composing a set of typeclass combinators for "array of T", "struct of" etc. The typeclass instances automatically generate specifications of the effect of C operations such as adding numbers, so these operators are proved correct by construction. An additional set of operations are provided for loading/assigning (variables), projecting out part (for pairs or structs), or indexing (for arrays). For user-defined algebraic datatypes, the hints on the datatype constructor (Sec. 2.2) are turned into a function which maps specification values to C structs or C integers.

Finally, the individual relations are put together to create the refinement relation $\mathfrak{R}$ for a complete layer. In the C semantics we are targeting, the program state is a pair (m, a) of a C memory and an "abstract value" a (which is used to model trusted primitives). In the generated specification we have a record value t, with one field v_i with a specification value for each object field in the DeepSEA program, while in the generated program we have one C global variable identifier p_i for each object field. The layer refinement relation is defined pointwise for each new field (i.e. each field that was defined in the current layer), by first applying R_{A_i} to each newly defined field, and then composing with the match relation.

Note that $\mathfrak{R}$ defines a refinement relation for an entire layer, while R defines refinement for each field. The rule of $R_{A \times B}$ relates a Coq pair to a C struct with two fields. Rule $R_{\{s_i \mapsto A_i\}}$ means that a Coq constructor with multiple fields can be related to a C struct with corresponding fields. The rule $R_{\texttt{array[}N\texttt{]}A}$ relates a Coq finite map $[a_i]^*$ to a C array $[cv_i]^*$.

4.3 Compiling Expressions and Commands

The compiler translates DeepSEA expressions and commands to C expressions and commands, using a set of fairly unsurprising rules (excerpted in Fig. 5). Because object fields behave differently when loading and assigning, there are separate translation rules for expressions used as L- and R-values.

$$
\begin{array}{ll}
\mathcal{SC}(v, t) & = \texttt{True} \\
\mathcal{SC}(e_1 \leftarrow e_2, t) & = \texttt{True} \\
\mathcal{SC}(\texttt{assert}\ c, t) & = (spec(c)\ t = \texttt{true}) \\
\mathcal{SC}(\texttt{let}\ x = c_1\ \texttt{in}\ c_2, t) & = \mathcal{SC}(c_1, t) \\
\qquad \wedge\, (\mathcal{SC}(c_1, t) \Rightarrow \texttt{let}\ (x, t') = spec(c_1)\ t\ \texttt{in}\ \mathcal{SC}(c_2, t')) & \\
\mathcal{SC}(\texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{do}\ c, t) & = \forall x.\ e_1 \leq x < e_2 \Rightarrow \mathcal{SC}(c, t)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{VC}(v, t) & = \mathcal{LC}(spec(v)\ t) \\
\mathcal{VC}(e_1 \leftarrow e_2, t) & = \\
\qquad\qquad \mathcal{LC}(spec(e_1)\ t) \wedge ((spec(e_2)\ t) \in dom(R_A) & \\
\mathcal{VC}(\texttt{assert}\ c, t) & = \texttt{True} \\
\mathcal{VC}(\texttt{let}\ x = c_1\ \texttt{in}\ c_2, t) & = \mathcal{VC}(c_1, t) \\
\qquad \wedge\, (\mathcal{SC}(c_1, t) \Rightarrow \texttt{let}\ (x, t') = spec(c_1)\ t\ \texttt{in}\ \mathcal{VC}(c_2, t')) & \\
\mathcal{VC}(\texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{do}\ c, t) = & \\
\mathcal{VC}(c, \texttt{Z.iter}\ (e_2 - e_1)\ (\lambda(x, t').\ (x + 1, spec(c)\ t'))\ (e_1, t)) &
\end{array}
$$

Fig. 6. Safety condition and verification condition (selected rules)

The `let`...`in` statements become assignment to local variables, `assert` is omitted, and `first`-loops become a while loop that breaks when the condition is met. The translation is parameterized by a local variable z, which is a place to store the "return value" of a command (z may be unused later, compare the rules for `let` and for $c_1; c_2$).

The bulk of the implementation of DeepSEA consists of a theorem, proven in Coq, that the command translation is correct with respect to the desugaring.

Data abstraction verification conditions. However, this theorem is not unconditionally true; e.g. programs are only correctly compiled if all data values remain in range. Therefore, we also define a predicate $\mathcal{VC}(c, t)$, the *verification condition*, which states that the command c can be correctly executed from state t. The DeepSEA compiler writes to a separate file a set of lemmas stating that $\mathcal{VC}$ holds for all methods in every state that satisfies the layer invariant. This design lets us separate the correctness of compilation (proven as a single theorem in the DeepSEA library) from the $\mathcal{VC}$ (proven by Ltac tactics for each compiled DeepSEA program).

The definition of $\mathcal{VC}$ is given in Fig. 6. The interesting base cases are loading object fields (v) and assigning to them ($e_1 \leftarrow e_2$). In both cases, if there is an array access, DeepSEA will generate a $\mathcal{VC}$ to do bound checking (denoted as $\mathcal{LC}$). Any time something is written to an object field, a $\mathcal{VC}$ will be generated to ensure the value of expression e_2 is in the range that can be represented by the underlying C type ($(spec(e_2)\ t) \in dom(R_A)$, e.g., that there are no integer overflows). The predicate $\mathcal{LC}$ is defined for each datatype as part of the typeclass machinery described in Sec. 4.2.

For a complex method c, we produce a $\mathcal{VC}(c, t)$ which is a conjunction of all the $\mathcal{LC}$ conditions. The interesting part here is the handling of assertions. For example, in the `mem_init` function (Sec. 2.2) array accesses can only be proved valid because of `assert (n < maxpage)`. This is handled by an auxiliary predicate $\mathcal{SC}(c, t)$, which states that c can be executed in state t without any failing asserts. This predicate is referred to in the definition of $\mathcal{VC}$ (see the case for `let`) to add an additional helpful assumption to the VC.

Apart from the definition of $\mathcal{VC}$, the compiler correctness theorem (proven in Coq) is a standard simulation between specification and C operational semantics:

Theorem 1 (Command downward simulation). *Given a command c, its C implementation impl(c) and specification spec(c): if spec(c) starts executing from a state t where $\mathcal{VC}(c, t)$ is satisfied, and it can terminate successfully in a state t', then for a memory m that has above refinement relation $\mathfrak{R}\ t\ (m, a)$, there exists another memory m' where executing impl(c) from m leads to it and has refinement relation $\mathfrak{R}\ t'\ (m', a')$.*

Using such a simulation result for each function, we can prove a contextual refinement [Gu et al. 2015] theorem for an entire layer (see Sec. 4.6).

4.4 Solving the Verification Conditions

For each DeepSEA program, the compiler then needs to solve $\mathcal{VC}$s for all the methods. In general, proving such $\mathcal{VC}$ is an undecidable problem because the statements include arbitrary functional specifications, for example a program might compute a value *n* using a complicated computation and then use *n* as an array index.

However, DeepSEA uses a set of tactics written in Coq's Ltac language to analyze the condition to solve many cases and minimize our user's work. Actually this Ltac is robust enough that it can solve all $\mathcal{VC}$ which appear in our case studies (Sec. 5) automatically. The Ltac gradually decomposes $\mathcal{VC}$ and introduces $\mathcal{SC}$ into context. The $\mathcal{SC}$ express assumptions that are available at a given point in the execution. For instance, after executing `let x = c1`, the current state should be almost the same as before except `x` being mapped to `c1`. For each loop, DeepSEA automatically generates a simple loop invariant stating which fields in the abstract states remain the same after execution of the loop body, then proves that loop invariant holds during entire loop and uses that to update the assumption over the state. This design approach minimizes what user needs to specify in source code. And even if such a loop invariant is insufficient to solve $\mathcal{VC}$, users can look into the generated $\mathcal{VC}$ to compose more suitable loop invariants or even do the entire proof themselves.

In the end, the $\mathcal{VC}$ will be decomposed into multiple requirements over states in various execution points, such as an array's index staying within bound and variables not overflowing. In each subgoal, the context contains certain assumptions introduced by the above Ltac tactics. DeepSEA will also add more assumptions by instantiating layer invariants with current state. Then it tries to solve them by applying `auto` and `omega` (a Coq tactic to prove arithmetic inequalities).

These tactics are sufficient for the programs in our case studies, but if they fail for some input program the DeepSEA user can extend the set of tactics, or fall back on writing a manual proof in the generated file.

4.5 Compiling Objects and Layers

Each DeepSEA object depends on an underlay signature, declares some number of fields, and defines methods. When being compiled to Coq, it is decomposed to only methods, but each method is parameterized on the "object context" (using Coq's "sections" feature). An object context contains the specification of each method in the underlay, as well as the type of program state and "lens" functions for each object fields (the lenses include the datatype operations described in Sec. 4.2).

When compiling an object, we go through the object methods one by one, and compile the method body in isolation. Finally, we assemble these definitions into layer interfaces and implementations as done manually by Gu et al. [2015].

4.6 Layer Refinement Proofs

We follow the idea and terminology of Gu et al. [2015] to break a system into multiple layers. A bottom layer has some trusted methods, and every other layer is built on top of another one. Each layer implements some new methods or refines its underlying layer, but all the methods called from

```
static int COUNTER = 0;
static int STACK[MAX_COUNTER];

int incr_counter() {
  COUNTER = COUNTER + 1;
  return COUNTER;
}

void push(int x) {
  unsigned int idx = get_counter();
  // An error if stack is full:
  incr_counter();
  STACK[idx] = x;
}
```

```
object Counter : CounterSig {
  ...
object Stack (counter : CounterSig) : StackSig {
  let STACK : array[MAX_COUNTER] int := array_init
  let push x =
    let idx = counter.get_counter() in
    counter.incr_counter();
    STACK[idx] := x
  ...
logical object AStack (_:StackSig) : AStackSig {
  let stack : list int := int_nil
  let push x =
    let s = stack in
    let len = int_length(s) in
    assert len < MAX_COUNTER-1;
    stack := int_cons(x,s)
  ...
layer FINAL : [{}]STACKSig =
  ASTACK :> (STACK @ COUNTER) with "RefineStack"
```

Fig. 7. Excerpts from the CCAL tutorial (left) and the DeepSEA reimplementation (right)

such a layer must satisfy the underlying layer's specification. We combine the specifications of a layer's exposed methods to form this layer's interface I. If a layer with interface I is built on top of another layer with interface I', then we say it has type $[I']I$. To verify a big system built in such manner, we only need to prove downward simulation (Theorem 2) for each layer, then apply the soundness theorem from Gu et al. [2015] to get the refinement between the entire implementation and the specification.

Finally, the generated files contain calls to layer library combinators to piece together the method definitions into a layer interface definition—and crucially, use Theorem 1 for each method body to prove a refinement theorem for the entire layer:

Theorem 2 (Layer downward simulation). *Given a layer l of type $[I']I$ with interface L and an underlying layer l' of type $[\emptyset]I'$ that has a layer interface L', if the verification conditions hold for every method defined in layer l then the layer interface L is a refinement of the layer implementation running over L'.*

5 CASE STUDIES

To check that the language is expressive enough to write real software, we selected three existing certified system software projects and re-wrote parts of them in DeepSEA: a minimal stack manager, a tiny implementation of the SHA hash function, and a memory manager for an operating system.

5.1 Stack Example from the CCAL Tutorial

The CertiKOS team used a stack of integers to explain the certified concurrent abstraction layer (CCAL) methodology.[1] The program is trivial (Fig 7), but comparing the original proofs from the tutorial with our reimplementation provides a qualitative understanding of how DeepSEA reduces proof effort. The original development consists of 1074 lines of Coq code (645 lines definitions

[1] Presented at the 2017 DeepSpec Summer School, https://deepspec.org/event/dsss17/lecture_shao.html.

and 429 lines proofs). It structures the verification of the stack methods into two refinement steps: C-to-low and low-to-abstract. Starting with source code in Clight, it first refines the stack operations into low-level specifications that work on CompCert types for memories. Then it performs a low-to-abstract refinement that represents the stack as a Coq `list`. A large fraction of the development is not proof scripts, but rather the manually written specifications and definitions of each layer.

In the DeepSEA version (Fig. 7, right), much of this is automated. The compiler correctness theorem (Sec. 4.3) handles all reasoning about operational semantics, subsuming the C-to-low step. Most of the low-to-abstract step (defining a refinement relation between C values and abstract values) is handled by the generic constructions for datatypes (Sec. 4.2), and the data abstraction verification conditions are simple enough that the automated tactics (Sec. 4.4) can handle them. In this way we automatically verify that the `Stack` object satisfies its generated specification.

Sometimes that may be enough, and we could stop there. However, the generated specification for push/pop uses the generic datatype machinery, which represents the array by a finite map from integers (`ZMap`) in Coq. The hand-written tutorial instead represented the state of the stack by a list, which makes sense since there are no random array accesses. DeepSEA can not automatically generate such a specification, but we write a purely logical specification `AStack` using the Coq list type, and state a pure refinement between that and the implementation in the same way we described in Sec. 2.5. The user must write down a suitable refinement relation in Coq:

```
Definition RefineStack (d1 d2: global_abstract_data_type) :=
  COUNTER d2 = length (stack d1)
  /\ forall n:, (n < length (stack d1)) ->
       nth_error (stack d1) (length (stack d1) - 1 - n) = Some (ZMap.get n (STACK d2)).
```

DeepSEA generates the statement of a downwards refinement lemma and asks us to prove it:

```
Lemma  ASTACK_AStack_push_exists : forall a0 d1 d2 d1',
  RefineStack d1 d2 ->
  high_level_invariant d1 -> high_level_invariant d2 ->
  execStateT (AStack_push_opt a0) d1 = ret d1'->
  exists d2', execStateT (Stack_push_opt a0) d2 = ret d2' /\ RefineStack d1' d2'.
```

This is the same type of downward refinement statements that CCAL users prove manually. There is no particular short-cut for writing this proof, and it took us about 110 lines of Coq. That's still much less than the original CCAL tutorial, since we only reason about functional data structures `ZMap` and `list` (not C), and we only have to prove the key lemmas for the method specifications themselves (not the scaffolding to link together the layers).

5.2 The SHA-256 Cryptographic Hash Function

The first realistic project we examine is Appel's verification of the SHA-256 hash function [Appel 2015]. Appel starts with two existing artifacts: on the one hand the open-source OpenSSL implementation (about 235 lines of C), and on the other hand the FIPS 180-4 Secure Hash Standard (about 16 pages of English text and math notation).

To prove that these match, he first translates the mathematical functions in the standard into 169 lines of Coq (these are six functions $Ch(x, y, z)$, $Maj(x, y, x)$, $\Sigma_0^{\{256\}}$, $\Sigma_1^{\{256\}}$, $\sigma_0^{\{256\}}$, $\sigma_1^{\{256\}}$ which operates on 32-bit words, the message schedule function W_t and round function $H^{(i)}$ which operate on message blocks, and the hash function H which ties them all together).

Although these are well-defined functions, it is not possible to execute them efficiently, because the standard defines W_t in a "naive" recursive way, which would take exponential time to run. So in order to experiment with the specification, Appel also defines a functional programming implementation in Coq, and proves (in 2424 lines of Coq) that it computes the same result. This can be run (but requires a full runtime system with garbage collection etc). The functional programming implementation does not play any role in the rest of the development, but provides a way to exercise the specification before embarking on the full proof.

The proof of the C code itself is 6539 lines of Coq developed using the VST program logic [Appel 2011]. The correctness theorem for the hash function is a Hoare triple which states a refinement square: if the concrete starting state of the C program (defined in terms of pointers etc) is related (via a manually defined Coq relation) to a text m (a mathematical Coq type), then after calling the hash function the concrete C variables are related to the hash value $H(m)$.

Our implementation. A basic tradeoff of our approach is that the user must write programs in DeepSEA rather than verifying existing C programs. So instead of starting with an existing C implementation, we write a new program, in about 400 lines of DeepSEA. Fig. 8 shows an illustrative excerpt. The function `fresh` tabulates W_t for a message block, while `block_data_order` implements $H^{(i)}$.

```
let fresh block_id =
  if block_id < 0x10000000 then
  begin
  for i = 0 to 16 do
      let t = fbr.read(block_id, i) in
      buffer[i] := t
  end;
  for i = 16 to 64 do
      let d0 = buffer[i-16] in
      let d1 = buffer[i-7] in
      let t0 = buffer[i-15] in
      let s0 = bitop.sigma0(t0) in
      let t1 = buffer[i-2] in
      let s1 = bitop.sigma1(t1) in
      buffer[i] := (s1 + d1) + (s0 + d0)
let block_data_order block_id =
  sha256_word.fresh(block_id);
  sha256_word.load_regs();
  begin for i = 0 to 64 do
    sha256_word.rnd(i)
  end;
  sha256_word.add_regs()
```

Fig. 8. Excerpt from our SHA implementation.

Limitations. To save effort we take a few short-cuts when we re-implement the code, so our final artifact has a little less functionality than the OpenSSL version. First, we decided not to implement the code to pad the message, and assume it is given as an array of 64-byte blocks. Also, we assume that the entire message is available at once, while the OpenSSL implementation allows adding more data incrementally. These could be implemented, with some more developer effort.

A deeper limitation is that the OpenSSL function interface takes a pointer to the data, while DeepSEA generally assumes that objects encapsulate all their data in field variables (we believe it will be easier to prove theorems about programs if clients use methods to query other objects, instead of a pointer to their innards, so that it is clear who maintains the object invariants). One can emulate pointers to shared data by having a dedicated object containing a buffer, and instead of pointers passing around numbers which are used as indexes into that buffer, but for this hash example we just assume a lowest layer which contains the message data and exposes it through a "`get`" method.

Proof. Does our new implementation compute the correct hash value? Yes. We know this because we prove that it satisfies *exactly* the mathematical specification developed by Appel, copied verbatim. However, in our development we never have to reason about C semantics; for each function we prove a refinement between the desugared specification generated by the DeepSEA compiler, and the mathematical function defined by Appel.

This illustrates the typical way DeepSEA fits into a larger development. We first generate low-level functional specs for each method in the program, and then we load those specs into Coq to interactively prove some theorem about them. In this case, the theorem is a refinement result, but for a different project it might be some other kind of correctness property.

In total there are about 664 lines of manual Coq proof. However, as mentioned above our program is more limited than OpenSSL SHA, and for this reason we also do not prove a refinement for the bottom layer containing the data array (instead we add a hypothesis which states that the values it returns correspond to a padded message). For a more fair comparison, we can consider just the C function `block_data_order` which processes a single block (and the `DeepSEA` helper functions `fresh` and `rnd`). Our Coq proofs are about 308 lines of function refinement, while Appel's proof (in files `verif_sha_bdo*.v`) are about 1788 lines.

Notably, proving that our implementation satisfies the mathematical specification appears no harder than to prove that Appel's *functional programming* implementation does so. We attained our goal: we support proofs by equational reasoning as well as in a functional language, while still generating low-level code.

Performance. The full OpenSSL implementation does more sophisticated copying of data (and testing whether it needs padding) than our `DeepSEA` version. We omit the padding step and let the lowest layer in our test harness provide bytes directly from a buffer with no copying, and comment out the corresponding code in the OpenSSL version. When compiled with the same compiler (`gcc -O3`), the `DeepSEA` version is slightly faster than OpenSSL (hashing 100 MB/s vs 90 MB/s), although OpenSSL is in any case not extremely optimized and a faster implementation [Gay 2005] can do 180 MB/s. Although programming directly in C offers more control over performance, for this example `DeepSEA` is fast enough.

5.3 The CertiKOS memory manager

The other project we evaluate is the memory manager for the CertiKOS OS kernel by Gu et al. [Gu et al. 2015]. While the SHA example shows how `DeepSEA` reduces the burden of individual proofs (the lines of proof/lines of code ratio), the memory manager example is interesting because the original code is split into multiple modules and has non-trivial state, so it shows that the `DeepSEA` feature set can handle nontrivial programs, and also help organize them.

CertiKOS is organized into a stack of layers (which inspired `DeepSEA`), and at the bottom of it are three families of functionalities: quota management "container," physical memory management "allocation table," and virtual memory management "page table." They are implemented in one, three, and seven layers, which each consist of manually written C code, Coq specifications, and Coq refinement proofs. We have re-implemented these 11 layers in `DeepSEA`, as we described in Sec. 2. The generated code is essentially the same as the original CertiKOS implementation, so there is no difference in performance.

In this case study, we do not go on to prove any high-level correctness theorem about the generated spec. This is because the memory manager is only a part of the original CertiKOS development, and in this part, the CertiKOS team *only* proved the correspondence between code and functional specification. Higher layers built on and included these specifications, and then eventually proved correctness theorems [Costanzo et al. 2016] about the linked artifact. Thus, using `DeepSEA` the only task is to write the program itself: our generated specifications are at the same level of abstraction as hand-written specifications that the CertiKOS team produced.

Implementation effort. Because it automatically proves the correspondence between C code and Coq specifications, using `DeepSEA` drastically reduces the amount of code that needs to be written. The following table shows the number of lines it takes to define various parts in both settings.

The MBoot layer implements quota management, MContainer layer defines `at_get_c` and `at_set_c` that get and set the reference count of each allocation table entry, MALInit layer initializes memory management such as computing number of pages and initializing allocation table, MALOp layer defines palloc and pfree.

On the CertiKOS side, the spec column shows the number of lines of Coq that define the specifications of methods, the Clight column shows the number of lines the abstract syntax tree takes up, and the correctness proof column shows the number of lines of proof script for the simulation proof between Coq specification and Clight implementation. All these are written manually, which is 8235 lines in total.

On the DeepSEA side, all the user needs to do is writing down the DeepSEA source program (810 lines). Although we mentioned that users might need to prove the verification condition $\mathcal{VC}$ in complicated cases, here our Ltac can prove all of them automatically.

Layer	DeepSEA	CertiKOS		
	Source Spec	Spec	Clight	Correctness
MBoot	248	1601	80	569
MContainer	358	257	649	2132
MALInit	87	204	329	791
MALOp	117	286	409	928
Total	810		8235	

Note that most operations defined in MBoot are trusted, that is, they only have specifications, but the implementation will be supplied separately and the programmer must ensure that the implementation matches the spec. This is why the original CertiKOS Clight code and correctness proofs for MBoot are relatively small.

6 RELATED WORK

Languages for systems programming. There has been much work on the development of systems programming languages [Apple 2015; Bershad et al. 1995; Gosling et al. 1996; Grossman et al. 2002; Hunt and Larus 2007; Microsoft Corp., *et al.* 2001; Mitchell et al. 1979; Nelson 1991; Odersky et al. 2005; The Rust Team 2015], including the use of advanced type systems to rule out various forms of run-time errors [Apple 2015; Odersky et al. 2005; The Rust Team 2015]. But these languages are still not suitable for building certified code: they lack formal (layered, functional) specifications and direct equational reasoning. Full functional correctness proofs themselves rule out all run-time errors (e.g., our compiler correctness theorem implies that if the specification does not return None, the generated C code does not encounter undefined behavior), so we are content with a simply-typed type system.

Functional languages with reasoning support. Instead of ascribing a functional specification to a program written in an imperative language, one can use a verified compiler for an existing functional language, such as CakeML [Tan et al. 2016] or CertiCoq [Anand et al. 2017], and then embed and reason about the programs in a proof assistant [Guéneau et al. 2017]. However, these languages use a large runtime system with a garbage collector, so they are not suited for low-level systems programming. DeepSEA carefully picks a small enough subset of features that it can be directly translated into C, yet used for realistic programs.

Cogent. The work most closely related to our aims is the Cogent language [Amani et al. 2016; O'Connor et al. 2016], which also aims to automatically generate C code and proof assistant specifications, and is motivated by the experience of the seL4 verified kernel. However, the detailed language design is completely different. Cogent deals with straight-line code, and the main focus is on a functional treatment of malloc/free. For DeepSEA we are interested in detailed user control over datatypes, loops, and in large-scale modular structure. Our choice of features were based on the examples in the CertiKOS project [Gu et al. 2015, 2016, 2018], which did not use malloc/free, but made very heavy use of refinement layers. DeepSEA provides such layers as a basic construct.

Hoare-style reasoning about low-level code. We made a detailed comparison with the Verified Software Toolchain (VST) [Appel 2011] in the SHA case study (Sec. 5.2). One big design difference is that VST uses Hoare logic instead of equational reasoning, i.e. VST formalizes the syntax of C programs and provides rules for proving pre- and post-conditions of program fragments, while DeepSEA represents programs as Coq functions that are reasoned about using the native Coq logic. The automatically generated DeepSEA specifications still look quite "imperative," but the difference becomes more important when using pure layer refinement since then the user can replace parts of the specifications with arbitrary functional programs. Low$*$ [Protzenko et al. 2017] and Bedrock [Chlipala 2011] also offer Hoare-style reasoning about a C subset or C-like language.

The C semantics are quite low-level, so typically a user writes a pure functional specification and then uses the Hoare rules to prove equivalence of the spec and the C program. By contrast, DeepSEA's generated specs are already at the level of abstraction of handwritten CertiKOS functional specs (Sec. 5.3).

Simpl [Schirmer 2006] translates a C subset into an imperative language and provides a Hoare logic, but C expressions are translated directly into Isabelle/HOL expressions—so it uses the Hoare approach for statements and the DeepSEA approach for expressions. Autocorres [Greenaway et al. 2012, 2014] further translates a Simpl program into a fully monadic HOL function, and tries to abstract from bounded ints into mathematical integers. The end result therefore looks similar to DeepSEA, but while the DeepSEA user writes "monad-ish" program which gets translated to C, Autocorres translates a subset of C into monads; and while DeepSEA tries to automatically prove integer overflows (but the user may need to add asserts), Autocorres first puts asserts everywhere and then tries to remove as many asserts as possible using rewrite rules. Therefore, Autocorres relies heavily on Isabelle/HOL's support for automatic rewriting. DeepSEA's verification condition generator, which creates a large conjunction of formulas, is a better fit for Coq's proof automation, which works by pattern matching the form of the formula to be proven.

VST provides access to all features of C, while DeepSEA, Low* and Simpl target a limited subset of C. DeepSEA, VST, and Bedrock give end-to-end machine-checked correctness proofs down to assembly, Simpl/Autocorres gives a machine-checked proof to C, while the Low* translation to C was only proved correct on paper. None of these languages address large-scale proof structure like DeepSEA's layers.

Frama-C/WP. Frama-C [Kirchner et al. 2015] is one of the most established tools for verification of C programs. It supports many approaches to verification—the most related one is the "WP" deductive verification module, which lets programmers annotate a C program with pre- and post-conditions written in a specification language called ACSL, which then generate goals for either an automatic or interactive theorem prover. It has been applied to verify parts of hypervisors/microkernels [Blanchard et al. 2015, 2018; Mangano et al. 2016]. The emphasis of ACSL is on logic formulas: it includes first- and higher-order quantifiers, separation-logic connectives, inductive definitions, built-in sets and lists, and recursive definitions. Accordingly, most use-cases verify a single data structure, e.g. a list or mapping, that can be easily specified by a first-order logic formula. DeepSEA's functional specifications are written in a full-featured programming language (Coq), which we believe will scale better to large systems.

Certified program synthesis. Software synthesis [Delaware et al. 2015; Kuncak et al. 2012; Manna and Waldinger 1971; Solar-Lezama 2008; Srivastava et al. 2010; The Kestrel Institute 2015; Torlak and Bodik 2014] aims to apply powerful decision procedures and theorem provers to automatically generate programs from specifications. Because large specifications are difficult to write, existing synthesis work often applies sophisticated algorithms to produce small pieces of code. DeepSEA offers a complementary perspective: its goal is to let programmers decompose a system into many simpler objects, to specify each object, and define interfaces between them. With layered specification and refinement, each layer can incorporate different local synthesis tools with varying degrees of abstraction and automation. We anticipate that synthesis will play a major role in DeepSEA-like languages in the future.

7 CONCLUSION

Certified systems programming is a unique challenge for language design: operating systems are inherently low-level and effectful, while software verification requires high-level abstractions and pure functions. In DeepSEA, we bridge this chasm *automatically*—from a single input program we derive the relation between ADTs and bytes, and between functional specification and implementation.

ACKNOWLEDGMENTS

We would like to thank anonymous referees for helpful feedback that improved this paper significantly. This research is based on work supported in part by NSF grants 1521523, 1715154, and 1763399 and DARPA grant FA8750-15-C-0082. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. 175–188.

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *CoqPL 2017: The Third International Workshop on Coq for Programming Languages*.

Andrew Appel. 2011. Verified Software Toolchain. In *ESOP'11: European Symposium on Programming*, Gilles Barthe (Ed.). LNCS, Vol. 6602. Springer, 1–17.

Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2, Article 7 (April 2015), 31 pages. https://doi.org/10.1145/2701415

Apple. 2013–2015. The Swift Programming Language. http://developer.apple.com/swift.

Brian N. Bershad et al. 1995. Extensibility, Safety and Performance in the SPIN Operating System. In *15th ACM Symposium on Operating System Principles*. 267–284.

Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue. 2015. A case study on formal verification of the anaxagoros hypervisor paging system with frama-C. In *FMICS 2015 - Formal Methods for Industrial Critical Systems (Lecture Notes in Computer Science - LNCS)*, Nunez M. Gudemann M. (Ed.), Vol. 9128. Springer Verlag, Oslo, Norway, 15–30. https://doi.org/10.1007/978-3-319-19458-5_2

Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. 2018. Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. In *NASA Formal Methods*, Aaron Dutle, César Muñoz, and Anthony Narkawicz (Eds.). Springer International Publishing, Cham, 37–53. https://doi.org/10.1007/978-3-319-77935-5_3

Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *J. Automated Reasoning* 43, 3 (2009), 263–288.

Adam Chlipala. 2011. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. In *Proc. 2011 ACM Conference on Programming Language Design and Implementation*. 234–245.

David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 648–664. https://doi.org/10.1145/2908080.2908100

Benjamin Delaware, Clement Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in Proof Assistant. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*. 689–700.

Olivier Gay. 2005. Software implementation in C of the FIPS 198 Keyed-Hash Message Authentication Code HMAC for SHA2. https://github.com/ogay/hmac

James Gosling, Bill Joy, and Guy Steele. 1996. *The Java Language Specification*. Addison-Wesley.

David Greenaway, June Andronick, and Gerwin Klein. 2012. Bridging the Gap: Automatic Verified Abstraction of C. In *International Conference on Interactive Theorem Proving*, Lennart Beringer and Amy Felty (Ed.). Springer, Princeton, New Jersey, USA, 99–115. https://doi.org/10.1007/978-3-642-32347-8_8

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2014. Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Edinburgh, UK, 429–439. https://doi.org/10.1145/2594291.2594296

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proc. 2002 ACM Conference on Programming Language Design and Implementation*. ACM Press, 282–293.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan(Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*. 595–608.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 653–669. http://dl.acm.org/citation.cfm?id=3026877.3026928

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jeremie Koenig, Vilhelm Sjober, Hao Chen, David Costanzo, and Tahnia Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proc. 2018 ACM Conference on Programming Language Design and Implementation*. ACM, New York, 646–661.

Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *Programming Languages and Systems*. Springer Berlin Heidelberg, 584–610. https://doi.org/10.1007/978-3-662-54434-1_22

Tony Hoare. 1974. Hints on programming language design. In *Computer Systems Reliability, State of the Art Report*, C. Bunyan (Ed.), Vol. 20. Pergamon/Infotech, 505–534.

Galen C. Hunt and James R. Larus. 2007. Singularity: rethinking the software stack. *Operating Systems Review* 41, 2 (2007), 37–49.

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (Jan. 2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70.

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2012. Software Synthesis Procedures. *Commun. ACM* 55, 2 (February 2012), 103–111.

Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculi and Applications*, Jean-Yves Girard (Ed.). Lecture Notes in Computer Science, Vol. 1581. Springer Berlin Heidelberg, 228–243. https://doi.org/10.1007/3-540-48959-2_17

David MacQueen. 1984. Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*. ACM, New York, NY, USA, 198–207. https://doi.org/10.1145/800055.802036

Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. 2016. Formal Verification of a Memory Allocation Module of Contiki with Frama-C: a Case Study. In *CRiSIS 2016 - 11th International Conference on Risks and Security of Internet and Systems*. Roscoff, France. https://hal.inria.fr/hal-01351142

Zohar Manna and Richard J. Waldinger. 1971. Automatic Program Synthesis. *Commun. ACM* 14, 3 (March 1971), 151–165.

Microsoft Corp., *et al.* 2001. C# language specification. (2001). Drafts of the ECMA TC39/TG3 standardization process. http://msdn.microsoft.com/net/ecma/.

James G. Mitchell, William Maybury, and Richard Sweet. 1979. *Mesa Language Manual*. Technical Report CSL-79-3. Xerox PARC, Palo Alto, CA.

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of Symposium on Logic in Computer Science*. IEEE, 14–23.

Greg Nelson. 1991. *Systems Programming with Modula-3.* Prentice Hall.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement Through Restraint: Bringing Down the Cost of Verification. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016).* ACM, New York, NY, USA, 89–102. https://doi.org/10.1145/2951913.2951940

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2005. *An Overview of the Scala Programming Language.* Technical Report IC/2004/64. Ecole Polytechnique Federale de Lausanne.

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *PACMPL* 1, ICFP (Sept. 2017), 17:1–17:29. https://doi.org/10.1145/3110261

Norbert Schirmer. 2006. *Verification of sequential imperative programs in Isabelle-HOL.* Ph.D. Dissertation. Technical University Munich, Germany.

Armando Solar-Lezama. 2008. *Programming Synthesis by Sketching.* Ph.D. Dissertation. University of California, Berkeley.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *Proc. 37th ACM Symposium on Principles of Programming Languages.* 313–326.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016).* ACM, New York, NY, USA, 60–73. https://doi.org/10.1145/2951913.2951924

The Coq development team. 1999 – 2014. The Coq proof assistant. http://coq.inria.fr.

The Kestrel Institute. 2015. The SpecWare System. www.kestrel.edu/home/prototypes/specware.html.

The Rust Team. 2011–2015. The Rust Programming Language. http://www.rust-lang.org.

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proc. 2014 ACM Conference on Programming Language Design and Implementation.* 530–541.