Spoq: Scaling Machine-Checkable Systems Verification in Coq

Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh, *Columbia University*

https://www.usenix.org/conference/osdi23/presentation/li-xupeng

This paper is included in the Proceedings of the
17th USENIX Symposium on Operating Systems
Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the
17th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Spoq: Scaling Machine-Checkable Systems Verification in Coq

Xupeng Li
Columbia University

Xuheng Li
Columbia University

Wei Qiang
Columbia University

Ronghui Gu
Columbia University

Jason Nieh
Columbia University

Abstract

System software is often large and complex, resulting in many vulnerabilities that can potentially be exploited to compromise the security of a system. Formal verification offers a potential solution to creating bug-free software, but a key impediment to its adoption remains proof cost. We present Spoq, a highly automated verification framework to construct machine-checkable proofs in Coq for system software with much less proof cost. Spoq introduces a novel program structure reconstruction technique to leverage LLVM to translate C code into Coq, supporting full C semantics, including C macros, inline assembly, and compiler directives, so that source code no longer has to be manually modified to be verified. Spoq leverages a layering proof strategy and introduces novel Coq tactics and transformation rules to automatically generate layer specifications and refinement proofs to simplify verification of concurrent system software. Spoq also supports easy integration of manually written layer specifications and refinement proofs. We use Spoq to verify a multiprocessor KVM hypervisor implementation. Verification using Spoq required 70% less proof effort than the manually written specifications and proofs to verify an older implementation. Furthermore, the proofs using Spoq hold for the unmodified implementation that is directly compiled and executed.

1 Introduction

System software such as operating systems and hypervisors [7] forms the software foundations of our computing infrastructure. However, modern system software is large, complex, and imperfect, with vulnerabilities that can be exploited to compromise the security of a system. Formal verification offers a potential solution to this problem by mathematically proving that system software can provide critical security guarantees. This typically involves verifying that the software implementation satisfies a formal high-level specification of its behavior, then proving that the specification guarantees the desired security properties.

The former, referred to as functional correctness, is generally the most challenging part to do, given the complexity of system software implementations. Implementations are commonly written in C, which has complex semantics and language features, many unsupported by verification tools. Verification tools powerful enough to verify real-world system software are difficult and tedious to use to write specifications and proofs. Furthermore, a high-level specification that is useful for verifying higher-level properties such as security often has a significant semantic gap from the implementation, requiring substantial manual proof effort to bridge this gap. However, without functional correctness to ensure that the proofs hold on the actual implementation, formally verified guarantees can be meaningless in practice.

We introduce Spoq (Scaling Proofs in Coq), a new verification framework to reduce proof costs for machine-checkable verification of system software. Spoq focuses on simplifying formal verification of functional correctness to reduce proof costs while ensuring that all proofs are machine-checkable by a theorem prover and verified down to the actual software implementation. It operates on widely used unmodified C code and leverages the Coq proof assistant [55] to enable machine-checkable verification of complex systems. Its key feature is making Coq easier to use by automating many aspects of writing Coq specifications and proofs. This reduces the amount of Coq code that needs to be manually written, which significantly reduces the time to conduct machine-checkable verification.

Spoq is the first system that can automatically translate unmodified C systems code, such as found in the Linux kernel, into a Coq representation so that it can be verified. Previous approaches such as CompCert's ClightGen [35] only support a subset of the C language. Systems that use ClightGen such as CertiKOS [18, 20] require significant manual effort to retrofit the systems implementation before it can be verified, extra effort to develop and maintain the retrofitted version, and still cannot provide any verified guarantees on the actual running version. Spoq address this problem by leveraging the widely used Clang compiler front end to parse C code into

LLVM's language-independent intermediate representation (IR). Because LLVM IR represents functions as control flow graphs, Spoq introduces a novel program reconstruction technique that translates control flow graphs back into a Coq representation using program-style functions with if-then-else and loop statements that is more amenable to verification. This approach enables Spoq to support full C language semantics, including GNU C-specific extensions and inline assembly code, yet work with an IR with clean semantics designed for automated translation into another representation.

Spoq then leverages a layering proof strategy based on Concurrent Certified Abstraction Layers (CCAL) [19, 21] to modularize and decompose verification into smaller steps to make each verification step easier. This involves defining the layer structure of the implementation, where each layer consists of a group of functions that define the layer's interface. Higher layers can call the functions exposed by a lower layer's interface, but not the other way around. The top layer is a high-level specification of the behavior of the entire implementation, while the bottom layer is a machine model whose interface is designed to support LLVM IR semantics. Verification involves proving that the layers compositionally refine the top layer specification of the entire implementation. While layering makes each verification step easier to accomplish, if done manually, it has the disadvantage of requiring a user to construct additional layer specifications, including both low-level and high-level specifications, and refinement proofs for each layer, which can involve tediously writing thousands of lines of additional Coq code. That code then has to be manually rewritten each time the program implementation is updated, imposing significant, time-consuming proof costs. Spoq instead takes advantage of layering and the easier verification steps it affords to make it possible to automatically generate the Coq layer specifications and mechanized refinement proofs from the layer structure definition. It is the first system that can automate the generation of layered specifications and proofs in Coq for concurrent system software.

Spoq constructs a machine-checkable proof object for each layer showing its implementation built on top of a lower layer interface refines its own layer interface. It decomposes the proof for a layer into two tasks. The first task is to prove that the layer's implementation, namely its Coq abstract syntax tree (AST) representation, refines a low-level specification that is closer to the source code and independent of the state of the machine model. The second task is to prove that the low-level specification, built on top of a lower layer interface, refines a high-level specification that defines the layer's interface and is self-contained. By self-contained, we mean that the specification does not contain any calls to functions in any other layer other than the bottom layer machine model. Making the high-level specification self-contained simplifies verification because refinement proofs of any layers built on top of this layer can effectively ignore any layers below it.

Spoq introduces a library of Coq tactics to automatically generate low-level specifications and refinement proofs between the implementation and low-level specification. Functions with loops are synthesized into Coq recursive specifications, then refined to their specifications using an induction proof template. To generate the specification for a function with loops, a ranking function is provided for each loop, which is monotonically decreasing and non-negative during loop iterations. Spoq leverages the ranking functions to generate loop termination proofs.

Spoq introduces transformation rules to automatically generate high-level specifications and refinement proofs between low-level and high-level specifications. Transformation rules include unfolding function definitions, syntactically reorganizing program structures, eliminating pre-determined branches and assertions, and performing mathematical simplification. Refinement proofs are done by introducing automatically generated annotations to track how transformations are applied, then using Coq tactics to prove the sequence of transformations preserves specification semantics. Automatic generation of specifications and proofs is only done for high-level specifications that do not introduce data abstractions to hide low-level data representation details, such as abstracting an array into a Coq `Map`. High-level specifications that introduce data abstractions or have very complex functions require manual assistance from the user to complete the specifications and proofs. Our experience indicates that the vast majority of functions can be automatically specified and refined without manual effort.

Spoq reduces the trusted computing base (TCB) for performing source code-level mechanized verification. There is no need to trust Spoq for generating specifications or proofs. Incorrect specifications will be rejected during refinement proofs, and incorrect proofs will be rejected by the Coq proof checker. Although Spoq relies on Clang which is not verified, most system software already needs to trust either widely used Clang or unverified alternatives such as the GNU C compiler to generate the executable code that actually runs. Using a verified compiler such as CompCert [35] is not viable in practice since it cannot even compile C code such as Linux kernel code. The only part of Spoq that is unverified yet needs to be trusted is its translator from LLVM IR to Coq, which is minimal by design. This TCB is much smaller than CompCert's ClightGen, which is larger and more complex since it has to directly parse and translate C code, a more difficult and involved process.

We have implemented Spoq and evaluated its effectiveness on commodity system software. We show that Spoq automatically translates over 99% of functions in unmodified C systems code into Coq representations, including the source code for the Linux kernel, while ClightGen fails to translate the vast majority of functions, including almost complete failure on the Linux kernel. We use Spoq to verify a multiprocessor KVM hypervisor implementation. Although an older version of the hypervisor was previously verified in Coq without Spoq, the proofs no longer work with the

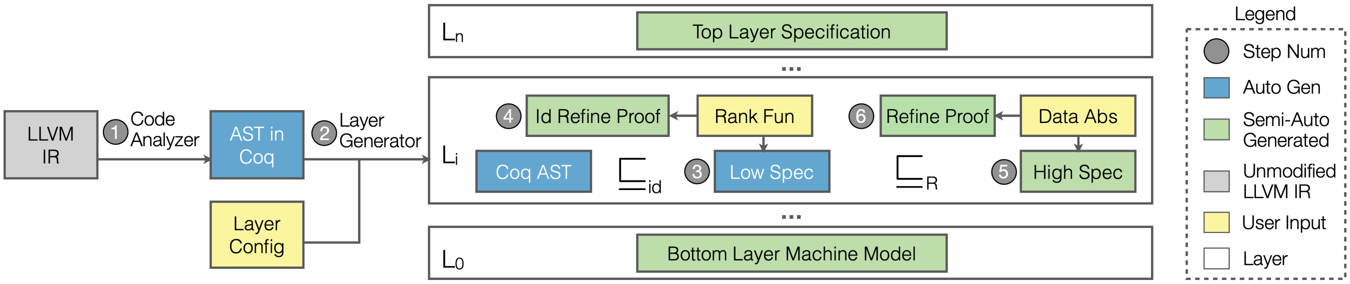


Figure 1: Spoq workflow.

updated version that supports additional hardware platforms. Previously, using ClightGen to translate the C implementation into a Coq representation required modifications to the source code, creating a gap between the verified and running code. Verifying the updated hypervisor using Spoq required much less proof effort, reducing the amount of manually written Coq code by over 70% compared to the verification of the older implementation. The proofs using Spoq are done on the unmodified source code of the hypervisor that is directly compiled and executed. Spoq even automatically generates the top layer specification, which we then use to verify the overall security properties of the hypervisor hold on the actual running software implementation.

2 Spoq Usage Model

To use Spoq, a user compiles the source code into LLVM IR and writes a layer configuration file defining the layer structure for the proof. The layer structure is defined to modularize the proof, with the additional constraint that a layer can only call functions in lower layers. For example, if the source code has three functions A, B and C such that A calls B and B calls C, at least three layers must be used. The configuration file specifies the name of each layer, the name of each function in each layer, the path to the source IR code, and the path to the Coq project. The configuration file should include the bottom layer abstract machine model, including its machine state definition. Spoq then generates the Coq project, including all specifications and proofs for each layer. If the source code or layer structure are changed, the user can rerun Spoq to update the Coq project. Spoq will regenerate the specifications and proofs for the parts affected by the changes, while other parts will remain unchanged.

Spoq guarantees that all generated specifications have exactly the same behavior as their source code implementations, but some generated high-level specifications may be too complex to be useful, and some refinement proofs may fail. Spoq makes it easy to integrate manually written specifications and proofs, which are simply annotated in the layer configuration file so that Spoq uses the provided specifications or proofs instead of generating them directly. If Spoq generates a high-level specification for a layer that is not concise enough, especially in how it updates the machine state, the user can manually write the specification and rerun Spoq with the provided high-level specification. If Spoq fails in generating refinement proofs for a layer, the user will see the resulting compilation errors of the generated Coq project identifying the specific functions with errors. If the error occurs for a generated specification, it is most likely due to a failed loop termination proof. The user can manually write the loop termination proof that failed and rerun Spoq with the provided termination proof. If the error occurs for a manually written specification, the user can check if there is an error in the specification or if the refinement proof also needs to be manually written, then rerun Spoq again.

Spoq is useful for both verifying functional correctness as well as higher-level system properties such as security. In verifying functional correctness, Spoq can generate the top-level specification, which will be guaranteed to have exactly the same behavior as the source code implementation. This notion of functional correctness ensures that the implementation satisfies the specification, but not necessarily that the code has no bugs. If the code is buggy, the generated top-level specification will still have the same behavior, including any buggy behavior. To provide a stronger notion of correctness, a user can use the generated top-level specification to verify higher-level properties such as security, which will identify bugs in the specification. Alternatively, a user can manually write the top-level specification and leverage Spoq to generate intermediate layer specifications and refinement proofs to verify that the implementation is functionally correct with respect to a manually written specification, though such a specification can also have bugs. The key benefit of Spoq is ensuring that whatever verification is done holds not just for a specification, but all the way down to the source code implementation.

3 Spoq Workflow

Figure 1 shows the workflow of Spoq. We use the example in Figure 2 to explain each step in the workflow and show how Spoq scales machine-checkable verification for systems code. This example contains a simplified C function `alloc` to allocate a free page by scanning the array of page descriptors `page`. The main computation is implemented as a statement expression in a macro definition `ALLOC`, in which we use a loop to iterate all elements of `page` and set the page status of

```
// Layer interface L1
uint page[MAX_PAGE];
uint get_page (uint i) { return page[i] }
void set_page (uint i, uint s) { page[i] = s; }
// Layer interface L2
#define ALLOC() ({                       \
  uint i;                                \
  for (i = 0; i < MAX_PAGE; i++){        \
    if (get_page(i) == 0) {              \
      set_page(i, 1);                    \
      break;                             \
    }                                    \
  }                                      \
  i;})
uint alloc() { return ALLOC(); }
```

Figure 2: A running example to allocate a free page.

the first free page to `1`. The accesses to `page` are encapsulated into functions `get_page` and `set_page`. This coding style is quite common in systems software such as Linux kernel code.

Generating Coq representations. To conduct mechanized verification, the first step is to translate the implementation into a representation in theorem provers, which is challenging even for simple and common C systems code like `alloc`; ClightGen cannot parse this simple example. Spoq leverages the Clang compiler front end to parse C into LLVM IR, and provides a code analyzer to parse LLVM IR code into an AST representation defined in Coq (Step 1 in Figure 1). We use LLVM IR because it is language- and machine-independent, supports full C language semantics and most extensions of C, can be easily integrated with assembly code semantics, and is much simpler and more rigorously defined than C. However, LLVM IR does not keep program structures, such as if-then-else and loop statements, making it hard to conduct proofs in a structural and inductive manner. Spoq resolves this issue by analyzing the control flow graphs of the LLVM IR code and reconstructing program structures. For example, Spoq reconstructs the loop, branch, and break statements in the Coq representation for the LLVM IR generated from the `alloc` function in Figure 2:

```
Definition f_alloc :=
  {| fname := "alloc"; rettype := ...; fargs := ...;
     fbody := ...  ::
        (ILoop (... :: (IIf ... IBreak) :: ...))... |}.
```

Spoq also models the semantics of Armv8 instructions [4] and parses assembly code into a list of assembly instructions in their Coq representations.

Defining layer structure. Spoq takes as input a layer configuration file which it uses to scale constructing mechanized proofs using CCALs. Using CCALs, we can construct a machine-checkable proof object "$M@L \sqsubseteq_R L'$," showing that the implementation M, built on top of a lower layer interface L, *refines* the interface L' with the refinement relation R. The file defines the layers and at which layer each function should be verified (Step 2 in Figure 1). For example, the layer configuration for the running example in Figure 2 defines that `get/set_page` should be verified on top of layer L_0, while `alloc` should be verified on top of layer L_1.

The layer structure presumes a bottom layer machine model, which Spoq automatically generates in part by identifying each global memory object in the source code and generating a corresponding machine state in Coq. Spoq also generates memory load/store primitives for each element in the state. The primitives take a memory pointer as an argument and calculate based on offset the array indices and structure elements to be accessed. Index boundary and data range checks are also included. The initial generated machine model does not include concurrency-related structures, such as an event log and oracle [40], which need to be manually added to complete the model to support CPU-local concurrency reasoning.

Given the layer configuration file, Spoq will automate generating the CCALs. It will build a CCAL "$M_{\texttt{page}}@L_0 \sqsubseteq_{R_1} L_1$" to abstract the `page` array into a Coq `Map` object from natural numbers to integers, such that its elements can only be accessed through getter and setter methods, `get_page` and `set_page`, respectively, rather than arbitrary memory operations which may lead to unexpected behavior. The refinement relation R_1 defines how the `page` array is abstracted into the `Map` object. It will then build a CCAL "$M_{\texttt{alloc}}@L_1 \sqsubseteq_{id} L_2$" to verify the `alloc` function on top of L_1 using the `Map` object without the need to worry about concrete implementation details of `page`. Here, *id* is an identical refinement relation since no data abstraction is needed when verifying `alloc`.

To make building CCALs easier, Spoq decomposes the required proofs into an *identical refinement* and a *lifting refinement*. The identical refinement refines M to a low-level specification S_{low} that is closer to the code and does not introduce any data abstraction, i.e., "$M@L \sqsubseteq_{id} S_{low}$." The lifting refinement refines the low-level specification to a high-level specification L', i.e., "$S_{low} \sqsubseteq_R L'$." The high-level specification is self-contained and may introduce abstractions to some data in lower layers.

Synthesizing identical refinements. Spoq generates low-level specifications and identical refinement proofs for each layer. The low-level specification of a function aggregates the small-step transition of each instruction in the function into a big-step transition of the entire function while preserving the semantics. For assembly code and C code without loops, generating the specifications and proofs is straightforward (Step 3 in Figure 1). Spoq provides a Coq tactic library to generate the identical refinement proofs; a tactic is a pre-defined decision procedure to generate proof scripts in Coq. Neither the specification generator nor tactic library needs to be trusted, since incorrect low-level specifications will be rejected by refinement proofs, and incorrect proofs will be rejected by the Coq proof checker.

For C code with loops, Spoq requires the user to provide a ranking function for each loop, which is non-negative and monotonically decreasing during the loop iterations. This is necessary because a termination proof is needed for each loop to prove refinement, and automating such termination proofs without user input is generally undecidable. With the input

```
Definition rank (i: nat) := MAX_PAGE - i.(*user input*)
Fixpoint alloc_loop_low (r i: nat) (st: ST) :=
  match r with
   | O => Some (MAX_PAGE, st)
   | S r' =>
     match get_page_high i st with (* spec from L1 *)
     | Some 0 => match set_page_high i 1 st with
                 | Some st' => Some (i, st')
                 | None => None
                 end
     | Some _ => alloc_loop_low r' (i+1) st
     | _ => None
     end
  end.
Definition alloc_low (st: ST) :=
  let r := rank 0 in alloc_loop_low r 0 st.
```

Figure 3: Low-level specification for the alloc function.

ranking function, Spoq automatically synthesizes a recursive function as the low-level specification using the Fixpoint construction in Coq, which requires an argument that decreases for each recursive call. For example, Figure 3 shows a recursive function alloc_loop_low synthesized for the loop in the alloc function with a user-provided ranking function (MAX_PAGE-i) as the decreasing argument for the Fixpoint construction. Note that the low-level specification of alloc is not self-contained and depends on functions get_page_high and set_page_high provided by the high-level specification at a lower layer. Spoq generates the refinement proof using a uniform induction-proof template (Step 4 in Figure 1).

Synthesizing lifting refinements. Spoq generates high-level specifications and lifting refinement proofs for each layer. This is done automatically when data abstractions are not used to hide low-level data representation details to simplify proofs at higher layers. If data abstractions are needed, users need to formulate the refinement relations, define abstract operations, and conduct the refinement proofs manually.

For example, the layer L_1 abstracts the array page into a Coq Map st.page, and transforms the memory operations load_mem and store_mem—offered by the bottom layer L_0's machine model—into Map operations (st.page#i and st.page#i<-s) with boundary checks:

```
(* Low-level specifications *)
Definition get_page_low (i: nat) (st: ST) :=
  load_mem st ("page", i * 4) u32.
Definition set_page_low (i s: nat) (st: ST) :=
  store_mem st ("page", i * 4) s u32.
(* High-level specifications in L1 *)
Definition get_page_high (i: nat) (st: ST) :=
  if 0 <=i< MAX_PAGE then Some st.page#i else None.
Definition set_page_high (i s: nat) (st: ST) :=
  if 0 <=i< MAX_PAGE then Some st.page#i<-s else None.
```

Because of the data abstraction, the lifting refinement proof for layer L_1 is not automated and has to be provided manually.

On the other hand, the layer L_2 does not use data abstractions. For layer L_2, Spoq automatically generates the high-level specification of alloc from its low-level specification by applying a sequence of *transformation rules*, including unfolding definitions, merging near-duplicate sub-expressions, eliminating pre-determined branches and assertions, and performing mathematical simplification. The latter two rules are applied by using the Z3 SMT solver [16]. For alloc_loop_low in Figure 3, Spoq first unfolds the definitions provided by L_1 and simplifies the representation as shown below:

```
Fixpoint alloc_loop_low' (r i: nat) (st: ST) :=
  match r with
   | O => Some (MAX_PAGE, st)
   | S r' =>
     if 0 <= i < MAX_PAGE then (* <- always true *)
       if st.page#i =? 0 then
         if 0 <= i < MAX_PAGE then (* <- redundant *)
           Some (i, st.page#i<-1)
         else None
       else alloc_loop_low' r' (i + 1) st'
     else None
  end.
```

Spoq then applies rules to eliminate an inner if statement which is redundant and eliminate the outer if statement by inferring that i is always within the range, resulting in the high-level specification in L_2 shown in Figure 4. Unlike the low-level specification, the high-level specification in L_2 is self-contained and does not refer to anything from L_1. Thus, any modules depending on L_2 can be reasoned about using L_2 alone without the need to look at lower layers. Otherwise, after building dozens of layers, the specification at a higher layer may wrap many levels of definitions from various lower layers, making the verification non-modular and much harder.

```
Fixpoint alloc_loop_high (r i: nat) (st: ST) :=
  match r with
   | O => (MAX_PAGE, st) (* no need of Some anymore *)
   | S r' => if st.page#i =? 0 then (i, st.page#i<-1)
             else alloc_loop_high r' (i + 1) st'
  end.
```

Figure 4: High-level specification for the alloc function.

Spoq automatically generates refinement proofs to verify the transformations that are applied to transform low-level into high-level specifications (Steps 5-6 in Figure 1). Since all specifications are guarded by machine-checkable proofs in Coq, there is no need to trust Spoq's specification generation algorithms or any Z3 results.

4 Generating Coq Representations

Spoq uses Clang to compile C code to LLVM IR, enabling it to support full C semantics and various extensions, including arbitrary type casting, integer-pointer conversion, inline assembly code, C macros that use GNU C extensions, and GNU C compiler directives. Spoq then translates LLVM IR code into an AST defined in Coq. IR code consists of structs, global variables, and functions. Spoq literally translates IR structs, similar to C structs, and global variables into their Coq representations, but does additional program reconstruction for IR functions. An IR function can be viewed as a control flow graph (CFG) over a set of basic blocks with an entry point. All instructions in a basic block are sequentially executed, and the last instruction either jumps to another block or returns from the function. Since systems code may contain goto statements and IR code is compiled with optimizations enabled, the CFG

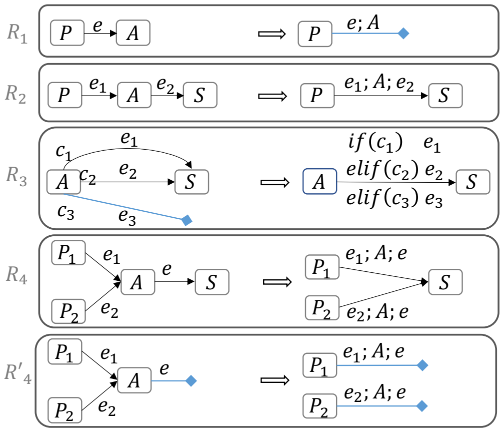


Figure 5: Rewrite rules for program CFGs without loops.

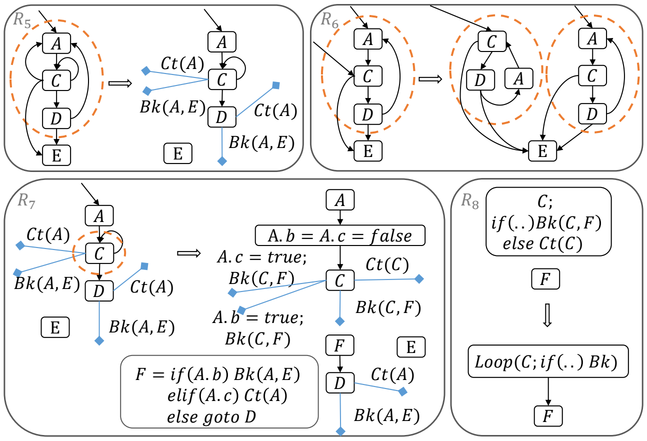


Figure 6: Rewrite rules for program CFGs with loops.

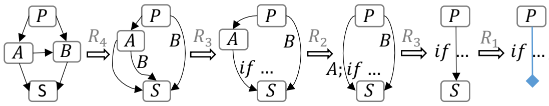
can be very complex and hard to reason about directly.

Spoq introduces a novel algorithm to merge each function's CFG of basic blocks into one code block and reconstruct program structure using if-then-else, loop, continue, break, and return statements. Spoq only uses these statements to construct a program structure that is amenable to proof decomposition, which may not be the same as the program structure of the original source code. For example, any goto statements in the original source code will be eliminated. The algorithm reconstructs program structure by repeatedly applying a set of rewrite rules to reduce the size of the CFG by merging blocks and deleting edges. Spoq performs the reconstruction in IR. No attempt is made to reconstruct the original C code, which would bloat an otherwise minimal implementation.

Reconstructing programs without loops. For programs without loops, Spoq uses four rewrite rules to reconstruct programs from CFGs, shown in Figure 5. Each node denotes a code block and each edge denotes a change in control flow. A, P, and S in the nodes denote the instructions inside the respective blocks. c_1, c_2, and c_3 at the beginning of edges denote the conditions to jump through the respective edges. Unlike regular CFGs, e, e_1, and e_2 denote instructions attached to edges which will be executed when jumping through the respective edges. A blue edge ending with a rhombus denotes an edge without a destination, whose attached instructions must end with a continue, break, or return statement.

The CFG of a function without loops has no cycles, so Spoq can repeatedly apply the rewrite rules to reduce the graph to a single node. Rule R_1 deletes a *dangling* node, a node with only one incoming edge e and no outgoing edge, and moves its instructions A to its incoming edge, which becomes an edge without a destination and has instructions "e;A." Rule R_2 deletes a *bridge* node A, a node with exactly one incoming edge e_1 and one outgoing edge e_2, and redirects the incoming edge from its predecessor node P to its successor node S with instructions "e_1;A;e_2." If all the outgoing edges of a node A either point to the same node S or do not have destinations, rule R_3 merges all the edges into one edge with branch statements. Since only the last instruction in a node changes the control flow, when a node has more than one outgoing edge, each edge must have a condition c. If a node has multiple incoming edges but only one outgoing edge, rule R_4 deletes the node and redirects all incoming edges to its successor node S with aggregated instructions. Rule R'_4 is logically the same as R_4, but shows the case when the only outgoing edge does not have a destination.

The reconstruction algorithm prioritizes applying the first three rules and only applies R_4 to the farthest valid node from the entry point if no other rules are applicable. We prove that this algorithm can rewrite any CFGs without loops into a single code block. The following example shows a sequence of rewrites to reconstruct the program structure from its CFG:



Reconstructing programs with loops. Loops introduce cycles into CFGs. For CFGs with cycles, Spoq computes the strongly connected components (SCCs). An SCC is the largest set of nodes in which every node is reachable from every other node. One node with self-pointed edges can also be an SCC. Spoq then uses four additional rewrite rules shown in Figure 6 to convert SCCs (marked by dotted orange circles) into loop-related statements.

Rule R_5 breaks cycles in an SCC which only has one incoming edge (pointing to node A in the SCC), and all its outgoing edges point to the same destination (node E outside SCC). It redirects any edge to A in the SCC to having no destination, and appends `Ct(A)` (a continue statement for the loop A) to the edge. It also redirects any edge to E in the SCC to having no destination, and appends `Bk(A,E)` (a break statement from the loop A to E) to the edge. After the rewrite, there is no longer a cycle back to node A and the size of the SCC becomes smaller. When an SCC has incoming edges from more than one node, rule R_6 duplicates the SCC for each node with incoming edges so that each SCC has only one incoming edge. For nested loops in which the inner loop may directly jump out of the outer one, rule R_7 converts such an SCC into one in which the jump target remains within the outer loop. Rule R_7 inserts a new node F, and all outgoing edges from the inner loop are redirected via break statements to F. Flags are also appended to the outgoing edges. Node F contains instructions to jump to different destinations depending on the flag. Flag $A.b$ means breaking the outer loop A, $A.c$ means going back to the beginning of the outer loop A, and no flag means breaking the inner loop. Once cycles are removed, rule R_8 converts a node's instructions into a single `Loop` statement, and re-establishes the edge from the loop node to its successor indicated by the break statement.

Assembly code. Spoq also handles assembly code, representing assembly instructions as parameterized inductive types in Coq. Each instruction corresponds to one construct with the operand as the parameter. Since assembly is not a structured language, Spoq simply translates each assembly procedure or inline assembly statement into a list of assembly instructions in their Coq representation. For inline assembly, LLVM IR already encapsulates it as a function. Spoq extracts the assembly code into a separate assembly procedure, and replaces the original function body with a call to the assembly procedure, decoupling the inline assembly from the LLVM IR in the Coq representation. The current implementation only handles Armv8 assembly code.

Semantics of Coq representations. Once LLVM IR and assembly code is translated to its Coq representation, it can then be verified. This requires defining the semantics of LLVM IR and assembly instructions in Coq, to specify the behavior of the Coq representation. Semantics are defined with respect to a layer interface for a bottom layer machine model. The interface contains a machine state `st` and getter and setter methods that access objects in the machine state through object pointers. An object pointer is a pair `(base, ofs)`, where `base` specifies the object and `ofs` specifies the field or offset within the object. In other words, the semantics of LLVM IR and assembly instructions define how those instructions use the getter and setter methods and how they update the underlying machine state. The machine state contains memory blocks and registers, as discussed below.

LLVM IR semantics only depend on memory objects, each of which is a set of disjoint memory blocks that can be accessed using `load_mem` and `store_mem` methods through object pointers with boundary checks. A memory block is contiguous and its size is defined by the type of the respective structure or global variable. For example, the `page` array in Figure 2 is a memory block with (`MAX_PAGE` × 4) bytes and can be accessed using an object pointer `("page", i)`, where $0 \leq$ `i` $<$ `MAX_PAGE` $\times 4$. The layer interface contains a variable environment providing a one-to-one mapping of variable names to corresponding addresses in memory.

For assembly code, Spoq models the semantics of the Armv8 instructions based on not only memory block objects, but also register objects. For example, the register objects model that clearing the `VM` bit in `HCR_EL2` register will disable the stage-2 translation for EL1 and EL0. Since an assembly procedure is just a list of assembly instructions, the semantics of an assembly procedure is defined as applying the semantics for each assembly instruction in the list one after the other.

Based on CCALs, Spoq uses CPU-local reasoning and distinguishes memory objects as CPU-private memory, lock-synchronized memory, and lock-free memory. Each CPU-private memory object belongs to and can only be accessed by a particular CPU. Each lock-synchronized memory object is associated with a lock. When accessing a lock-synchronized memory object, Spoq checks that the corresponding lock is held by the local CPU. Accessing a lock-free memory object generates an event appended to a global log, and an event oracle is queried to simulate other CPUs' behavior before generating each event. Correct concurrent behavior is guaranteed in the same way as previous work using CCALs [38,40]. This event-based machine model assumes sequential consistency (SC). To propagate proof results for a system to Arm's relaxed memory hardware, users can follow the methods introduced by VRM [54] to verify that the system satisfies six weak-data-race-free conditions. This implies that the system exhibits no more behaviors when running on Arm relaxed memory hardware versus an SC model. Thus, any guarantees proven using the SC model still hold on Arm's relaxed memory hardware.

5 Synthesizing Identical Refinements

Low-level specifications without loops. Spoq recursively aggregates the small-step semantics of every IR statement in a function and generates a Coq definition to reflect the entire transition as the low-level specification of the function. Leveraging the reconstructed program structure, Spoq simply scans through the Coq AST representation, conducts case analysis starting with the first statement, and generates the corresponding Coq definition as a string based on the defined LLVM IR semantics. A small piece of Python pseudocode for assignment and branch statements is shown below:

```python
def spec_gen (ast, spec):
  for n in range(len(ast)):
    i = ast[n]
    if isinstance(i, IAssign): # Assignment case
      s = f"let {coq_name(i.asg)} := {val(i.v)} in"
      spec.append(s)
    elif isinstance(i, IIf):   # Branch case
      spec.append(f"if {coq_name(i.cond)} then")
      spec_gen(i.true_body + ast[n+1:], spec)
      spec.append(f"else")
      spec_gen(i.false_body + ast[n+1:], spec)
    ...
```

For an `IAssign` statement, which assigns a value to a temporary variable, Spoq generates a `let` binding in Coq. For an `IIf` statement, Spoq recursively invokes its specification generator `spec_gen` for each branch in the code and concatenates the branch body with the rest of the AST.

Identical refinements without loops. Spoq automatically generates identical refinement proofs by using a Coq tactic lrefine. The idea is to do case analysis for each conditional by recursively decomposing each conditional into two sub-proofs, one for when the conditional is true and another for when it is false. Once a branch body is reached with no further conditionals, the proof can simply show that if the low-level specification transforms the machine state from st to st’, then the small-step semantics of the Coq AST also transforms the machine state from st to st’. Spoq aggregates the sub-proofs for all the branch cases to form the overall refinement proof. Take the following pseudo-specification generated from an if statement as an example:

```
Definition foo_low (st: ST) :=
  if cond then foo_true_low st else foo_false_low st
```

The lrefine tactic will conduct case analysis over cond, which generates two sub-proof goals. The first goal is to prove that the AST transfers st to “foo_true_low st” with an additional hypothesis “H0: cond = true.” The lrefine tactic then executes the semantics of AST for one step by showing that the branch condition will be evaluated to true when H0 holds and finally invokes lrefine recursively to prove that the first branch implementation will transfer st to “foo_true_low st,” a specification generated using the first branch. The second goal can be proved similarly.

Low-level specifications for loops. Spoq generates low-level specifications for loops using a recursive Fixpoint construction in Coq. A Fixpoint definition requires a decreasing argument, which has the type nat and decreases for each recursive call of the function. Spoq requires the user to provide a ranking function for each loop as the decreasing argument. It then generates low-level specifications for loops by filling in the parts marked with {{ }} in the template below:

```
1  Fixpoint _loop (n: nat) (bk rt: bool) {{Vi Vo}} st:=
2    match n with
3    | O => Some (bk, rt, {{Vo}}, st)
4    | S n’ =>
5      match _loop n’ bk rt {{Vi Vo}} st with
6      | Some (bk’, rt’, {{Vo’}}, st’) =>
7        if bk’ then Some (bk’, rt’, {{Vo’}}, st’)
8        else if rt’ then Some (bk’, rt’, {{Vo’}}, st’)
9        else {{low-level spec of the loop body}}
10     | _ => None
11     end
12   end.
13 Definition _low {{args}} (st: ST):=
14   {{low-level spec before the loop}}
15   let n := {{rank i_Vi}} in
16   match _loop n false false {{i_Vi i_Vo}} st with
17   | Some (bk, rt, {{Vo}}, st’) =>
18     if rt then Some ({{Vo}}, st’)
19     else {{low-level spec after the loop}}
20   | _ => None
21   end.
```

For the loop, Spoq generates a Fixpoint construction such that one recursive call of the Fixpoint construction corresponds to one iteration of the loop, so its body is the low-level specification of the loop body (line 9). Five Fixpoint arguments track the state of the loop (line 1). Vi are the input variables initialized before the loop and accessed by the loop body; they have initial values i_Vi. Vo are the output variables accessed after the loop that were also accessed in the loop body; they have initial values i_Vo. For example, the loop in alloc in Figure 2 simply has i for both Vi and Vo, with initial values 0 and MAX_PAGE, respectively. Spoq determines input and output variables and their initial values from syntactic analysis of the IR code. n is the decreasing argument, which is a natural number that is determined by the user-provided ranking function, which takes as input all the input variables of the loop. n is initialized using the ranking function over the initial value of input variables i_Vi, which sets the maximum number of “loop iterations” (line 15), and decreases by one for each “loop iteration” (line 4). Flags bk and rt indicate whether the loop has already been terminated by a break or return statement. The loop body (line 9) sets bk to true when executing a break statement or exiting when the loop condition becomes false, and sets rt to true when executing a return statement. Fixpoint will not make further changes once bk or rt is set to true (lines 7 and 8).

For the function containing the loop, Spoq generates low-level specifications for the code before the loop (line 14); invokes the Fixpoint with the initial values of the ranking function, flags, and variables (line 16); skips the rest of the function if rt is true (line 18); and generates low-level specifications for the code after the loop if not returned (line 19). Spoq will syntactically analyze the IR code and produce Vi, Vo, and their initial values i_Vi and i_Vo. Note that Figure 3 shows a simplified low-level specification that omits the bk and rt flags and uses a tail recursion style.

Identical refinement proofs for loops. Spoq proves identical refinements for loops using induction. The base case is trivial because the input machine states are the same. Spoq only needs to prove that the initial ranking function is non-negative. This is automated using a tactic xlia, extended from Coq’s tactic lia, a decision procedure for arithmetic. The induction step is to show that when the input machine states for the low-level specification and Coq AST are the same after the i-th iteration and both bk and rt are false, the output machine states are still the same after the $(i+1)$-st iteration. The $(i+1)$-st iteration may have one of three outcomes: 1) continue to the next iteration, 2) break the loop due to a break statement or the loop condition becoming false), and 3) return from the function. For all three outcomes, Spoq first proves that the loop body and Fixpoint body have the same semantics by recursively invoking lrefine. Spoq then proves additional properties for each outcome. For the first outcome, Spoq proves that the ranking function decreases by at least one and is still greater than zero using xlia. This guarantees that the loop must terminate after at most the number of iterations indicated by the initial ranking function. For the second outcome, Spoq proves that bk is true after the iteration, and the ranking function is still non-negative when the loop condition becomes false using xlia. For the third outcome, Spoq proves that rt is true after the iteration. Note that the Fixpoint function continues the iteration after bk or rt is true but will not make any changes to the state.

Spoq automatically generates the identical refinement proof for a loop if the loop is not contained within a conditional in the function. However, if the loop is contained within a conditional, or a series of conditionals, this results in the loop being used in multiple branches of execution, which Spoq currently does not automatically handle. In this case, the user will see that the loop termination proof failed in one or more branches, and needs to copy and paste the induction proof template into the other branches of execution with possible minor modifications; this is generally straightforward to do.

Assembly code. Spoq generates low-level specifications for assembly code by evaluating the assembly instruction list. The current implementation only supports automatic generation of low-level specifications for assembly code without jumps. Spoq simply evaluates instructions sequentially and outputs the machine state of the last instruction. If the destination of a call instruction is a C function, Spoq uses registers according to the Procedure Call Standard for the Arm 64-bit Architecture (AAPCS64) [5]. Spoq sets the arguments to the values in the argument registers according to AAPCS64. After the function call, Spoq checks the linker register of the machine state and evaluates the assembly instruction from where the linker register points. After returning from the function call to assembly code, Spoq sets the value of the caller-saved registers to UNKNOWN because the caller cannot assume any value in the caller-saved registers according to AAPCS64. Spoq disallows reads from any register with value UNKNOWN; assembly code must write to the caller-saved register first before it can be read. This helps prevent unexpected information leakage from registers.

By using the AAPCS64 calling conventions for assembly code functions so that arguments and return values are treated the same as C code functions, Spoq provides a unified approach to generating low-level specifications for assembly and C code. This includes using the same type value used in the IR semantics for assembly code. This unified approach makes it possible to link the proofs for assembly and C code.

Spoq generates low-level specifications for inline assembly in the same manner as other assembly code, since it already extracts the inline assembly into a separate assembly code procedure. However, Spoq requires that the operands used in inline assembly are C variables specified in the input or output operand list, system registers, and constants. Directly reading or writing general-purpose registers is disallowed to ensure proof correctness when linking inline assembly and C code, as the compiler may use them for temporary variables [40].

Spoq automatically generates identical refinement proofs for assembly code, which is straightforward without jumps as there are also no loops. The proof simply shows that the low-level specification and assembly instruction list transform the machine state in the same way.

6 Synthesizing Lifting Refinements

High-level specifications. Spoq generates high-level specifications by applying a set of transformation rules to low-level specifications to make them self-contained and simple. Spoq uses 12 transformation rules shown in Figure 7, though additional rules can easily be added. Spoq uses the Z3 SMT solver to apply rules involving symbolic execution or mathematical simplification. The goal of the transformation rules is to simplify the required control flow and eliminate as much as possible unnecessary operations.

T_1 unfolds a function's definition in an expression. Functions defined in lower layers that are called in the low-level specification are generally unfolded as part of the high-level specification to make it self-contained. Unfolding may also provide opportunities to apply other transformation rules to eliminate unnecessary operations to further simplify the specification. T_2 eliminates a let assignment by substituting the variable with its value, which helps find opportunities for simplifying expressions. T_3 eliminates an if branch if both branches are the same. T_4 eliminates a match statement by syntactically determining which pattern matches the source value. T_5 eliminates a match statement if both the source and return values are of Option type, and if the source value is None, the return value is None. It eliminates the match by making body the return value for all source values that are not None. T_6 transforms a match statement in which the source value matches the pattern and is used in the return value by substituting the pattern in the return value. This can provide more opportunities for simplification since patterns are more specific. T_7 moves the control flow of the source value to the outside of the match statement. Spoq tries to simplify the source value of match statements to make it easier to determine matching patterns. T_8 moves the control flow within an expression to the outside of the expression to aggregate computations within the expression, which helps find opportunities for simplifying expressions. T_9 does various simplifications for getter and setter methods. Here i and j indicate different fields. Whether i equals j can be determined syntactically (if they are structure names), or by Z3 (if they are integer indices). T_{10} performs symbolic execution using Z3 to identify whether the assertion of a rely is valid or invalid. If the assertion is always true, then rely is redundant and can be removed. If the assertion is always false, the statement can simply return None. T_{10} will do nothing if Z3 cannot decide if the assertion is true or false. T_{11} performs symbolic execution using Z3 to simplify if statements. T_{12} simplifies math expressions using Z3. For example, Spoq applies T_1, T_2, and T_{11} to generate the high-level specification in Figure 4 from its low-level specification.

While the transformation rules can be applied in different orders to yield the same result, the order in which the rules are applied can have a significant impact on the execution time required. Spoq reduces execution time by applying the rules in stages. In the first stage, it applies rules T_2 - T_8 and the T_9

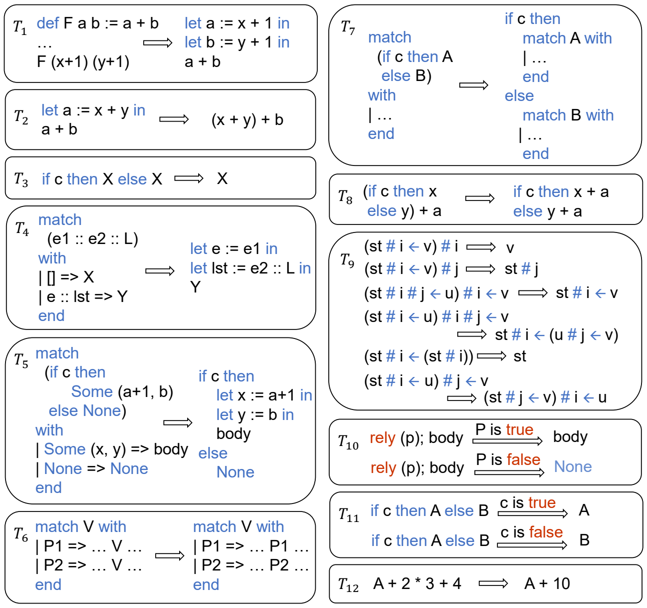

T1 def F a b := a + b … F (x+1) (y+1) ⟹ let a := x + 1 in let b := y + 1 in a + b
T2 let a := x + y in a + b ⟹ (x + y) + b
T3 if c then X else X ⟹ X
T4 match (e1 :: e2 :: L) with | [] => X | e :: lst => Y end ⟹ let e := e1 in let lst := e2 :: L in Y
T5 match (if c then Some (a+1, b) else None) with | Some (x, y) => body | None => None end ⟹ if c then let x := a+1 in let y := b in body else None
T6 match V with | P1 => … V … | P2 => … V … end ⟹ match V with | P1 => … P1 … | P2 => … P2 … end
T7 match (if c then A else B) with | … end ⟹ if c then match A with | … end else match B with | … end
T8 (if c then x else y) + a ⟹ if c then x + a else y + a
T9 (st # i ← v) # i ⟹ v
(st # i ← v) # j ⟹ st # j
(st # i # j ← u) # i ← v ⟹ st # i ← v
(st # i ← u) # i # j ← v ⟹ st # i ← (u # j ← v)
(st # i ← (st # i)) ⟹ st
(st # i ← u) # j ← v ⟹ (st # j ← v) # i ← u
T10 rely (p); body P is true ⟹ body
rely (p); body P is false ⟹ None
T11 if c then A else B c is true ⟹ A
if c then A else B c is false ⟹ B
T12 A + 2 * 3 + 4 ⟹ A + 10

Figure 7: Transformation rules for high-level specifications.

syntactic transformations. In the second stage, it applies rule T_1, then repeats applying the rules from the first stage. Spoq unfolds only one function, and only when no other syntactic rules can apply, because unfolding multiple functions too early can cause extra work. In the extreme case, unfolding all functions first will cause the size of the specification to explode and result in many unnecessary tests on each expression in each unfolded function body. In the third stage, it applies rules that use Z3, specifically rules T_9 - T_{12}, then repeats applying the rules from the first and second stages. Spoq applies syntactic rules first to simplify the specification as much as possible before applying Z3 rules because Z3 rules take much longer to process. To avoid long Z3 processing times, Spoq enforces a short timeout on Z3 operations, which is set to half a second by default. Essentially, Spoq repeatedly applies all rules until the high-level specification converges, meaning the rules no longer change the specification.

Using transformation rules to make the high-level specification of each layer self-contained generally results in the high-level specification being of larger size than its corresponding low-level specification. However, this size increase is outweighed by the ability to use the self-contained specification to simplify reasoning for higher layers, especially with regard to reasoning about higher-level properties based on the top layer high-level specification.

Lifting refinement proofs. Spoq automatically generates lifting refinement proofs to prove that the low-level specification refines the high-level specification generated by the transformation rules. This will necessarily be the case for transformation rules done in Coq, so the task reduces to reconstructing the proofs in Coq for all transformations done by Z3; there is no need to trust any results from Z3. Spoq uses a Coq tactic library to enable the proof automation.

Spoq simplifies the construction of refinement proofs by introducing annotated high-level specifications, which are the same as high-level specifications except that they have additional annotations that encapsulate the results of all of the Z3 transformations applied. For example, if T_{11} is applied, there will be an annotation showing that $A+2*3+4 = A+10$, which serves as a hint for constructing proofs. Spoq generates the annotations as it is generating the high-level specification. Spoq then uses the annotations to tell Coq what step-by-step syntactic substitutions it should perform to prove the low-level specification refines the annotated high-level specification. Because the annotations tell Spoq what transformations to do, it only has to validate them in Coq, which is much easier than automatically discovering the transformations in Coq; that would be difficult without Z3. Spoq finally trivially proves that the annotated high-level specification refines the high-level specification by showing that removing the annotations does not change the machine behavior. The two-part refinement proof shows that the low-level specification refines the high-level specification.

Spoq introduces a Coq tactic `hrefine` to automate the core part of the proof, namely proving that the low-level specification is equivalent to the annotated high-level specification. The strategy of `hrefine` is similar to the one for `lrefine` used for the identical refinement proof discussed in Section 5. The `hrefine` tactic analyzes the structure of the annotated high-level specification, decomposes it into all possible branches of state transitions, and conducts the proof for each branch. For each branch, all `match`, `if`, and `rely` are eliminated because the branch corresponds to a specific set of values for their conditions. Each branch therefore has a list of conditions and annotations. Spoq uses those conditions and annotations to simplify the low-level specification and prove that the low-level specification has the same behavior as the high-level one for that branch. It then repeats this process to prove the refinement for each branch.

Section 7 shows that Spoq was able to automatically generate all lifting refinement proofs involving Z3 transformations in verifying a multiprocessor KVM hypervisor. However, it is theoretically possible for there to be Z3 transformations for which Spoq is not able to generate lifting refinement proofs, in which case the user needs to manually complete those proofs.

Spoq also uses Coq tactics to automatically generate lifting refinement proofs for `Fixpoint` constructions, which are used in high-level and low-level specifications for functions with loops. The proofs use induction and are straightforward to generate because they only involve `Fixpoint` constructions, which are guaranteed to terminate. The hard part of refining loops to `Fixpoint` constructions and completing termination proofs has already been done in the low-level specifications.

Using Spoq provides significant advantages in terms of proof modularity over previous approaches that required users to manually write high-level specifications and proofs [20, 38, 40]. Because creating a self-contained high-level specification often involves unfolding function definitions from lower layers, any change to an implementation at a lower layer can require rewriting the high-level specifications for all higher layers, which also requires rewriting their refinement proofs. This makes it difficult to port specifications and proofs as a software implementation evolves over time if high-level specifications and refinement proofs are manually written, as many of them may have to be manually rewritten. With Spoq, the impact of an implementation change can be localized to its respective layer, even if that layer requires writing high-level specifications or proofs manually, since high-level specifications and proofs for higher layers can be automatically generated. This makes it much easier to port specifications and proofs across software updates.

Assembly code. Spoq generates high-level specifications and lifting refinement proofs for assembly code without jumps in the same manner as for C code. The current implementation leaves it to the user to write specifications and refinement proofs for assembly code with jumps.

7 Evaluation

We have implemented a Spoq prototype, which consists of three components: the translator from systems code into Coq, the specification and proof generator, and the Coq libraries for LLVM IR and assembly semantics and tactics. The three components are implemented using 4K lines of code (LoC) in C++ and Python, 6K LoC in Python, and 5K LoC in Coq, respectively. We evaluated Spoq's effectiveness in translating C systems code into Coq for various widely used open-source software, and verifying a KVM hypervisor implementation.

7.1 Translating system software into Coq

Since the first step in verification is to translate systems code into Coq, we evaluated Spoq's ability to do so for the applications, libraries, and Linux kernel version listed in Figure 8. We used the Makefile for the source code tree of each application, library, and kernel to build the source code using the default configuration, but output LLVM IR (.ll) files, in some cases by modifying the Makefiles by replacing the `-o` compilation option to output an executable with the `-S -emit-llvm` option to output LLVM IR files, which are then read by Spoq to translate them into Coq. The Linux kernel uses a more complex KBuild system [29], but no modifications were needed since it already accepts the `-S -emit-llvm` option to output LLVM IR files.

For comparison, we also tried to use ClightGen to translate the systems code into Coq. This required much more effort to the build source code trees because many of the compiler flags are not accepted by ClightGen. Instead, for most cases, we ran the existing Makefiles to get the compilation commands executed and saved them to a file, then used a script to filter options not supported by ClightGen, then reran the filtered compilation commands using ClightGen instead.

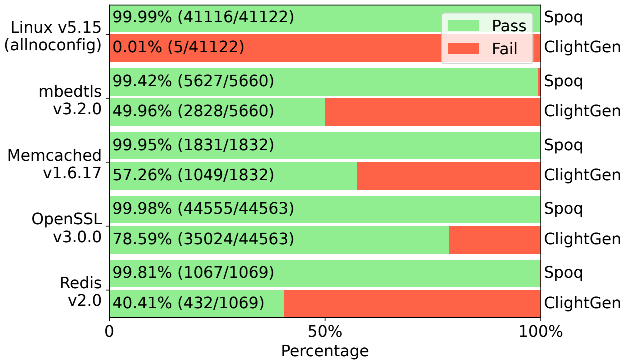


Figure 8: Translating C code into Coq. Each bar shows how many of the total number of C functions are successfully translated.

Figure 8 shows the results for translating C systems code into Coq using Spoq versus CompCert's ClightGen. Across all of the applications, libraries, and the Linux kernel, Spoq successfully translates over 99% of the functions in the source code into their Coq representations. The failures were caused by currently unsupported LLVM instructions, mainly advanced branching instructions (e.g. `callbr`, `invoke`, `resume`). Support for them is left for future work.

Spoq performs significantly better than ClightGen, which fails almost entirely on the Linux kernel and only translates roughly 50% of the functions in the source code into their Coq representations for most cases. Its best performance is on OpenSSL, for which it is still able to only translate less than 80% of the functions in the source code into Coq representations. ClightGen fails due to numerous unsupported C features, including variable-sized arrays, function parameters or return values with `union`/`struct`, additional keywords, C statements, and other unsupported inline assembly features. Furthermore, for the Linux kernel, GNU C directives are ubiquitous in almost all header files included by source code files and prevent ClightGen from translating the kernel source code into Coq.

Not only does Spoq perform far better than ClightGen in translating systems code into Coq representation, but it has a much smaller implementation. The module in Spoq responsible for translating systems code into Coq consists of 2.7K LoC in Python and 1.3K LoC in C++, the latter to make use of the official LLVM library to parse LLVM IR files. Its minimal implementation avoids bloating the TCB. In contrast, ClightGen is enormous, consisting of at least tens of thousands of lines of unverified OCaml code. ClightGen performs worse than Spoq and increases the TCB size much more significantly than Spoq as well.

7.2 Verifying a KVM hypervisor

We evaluated Spoq's ability to reduce proof costs by verifying SeKVM, a retrofitted version of the KVM/Arm

```
#define __hyp_text __section(.hyp.text) notrace
u32 ~~__hyp_text~~ mem_region_search(u64 addr)
```

(a) Unsupported compiler directive.

```
/* Orignal source code:
 * inline assembly and macro of a C statement */
u32 __raw_readl(const volatile void __iomem *addr){
  u32 val;
  asm volatile("ldr %w0, [%1]\r\nldar %w0, [%1]",)
  : "=r" (val) : "r" (addr));
  return val;}
#define readl_relaxed(c)                          \
  ({ u32 __r =                                    \
     le32_to_cpu((__force __le32)__raw_readl(c));  \
     __r;})
```

```
/* Verified source code:
 * original source code replaced with only C function
 * declaration so it can be parsed by ClightGen. */
u32 readl_relaxed(u64 addr);
```

```
(* Specification modeling the behavior of
 * readl_relaxed; implementation unverified. *)
Definition readl_relaxed_spec (addr: Z) (st: ST) :=
  (ZMap.get st.(mem) addr, st).
```

(b) Unsupported GNU Inline Assembly and C statement.

Figure 9: Example SeKVM changes required to use ClightGen.

hypervisor [13–15, 37] that was previously verified in Coq [38, 39, 54]; only its trusted core needed to be verified to guarantee the security properties of the entire multiprocessor hypervisor. We updated SeKVM to run on additional hardware, specifically the Raspberry Pi 4, which involved modest changes to its previously verified codebase. However, this required updating the proofs, so we used Spoq to verify the updated version, and compare the proof effort to the manually written Coq proofs for the earlier version of SeKVM.

Generating Coq representations. We first used Spoq to automatically translate the source code of the trusted core of the updated hypervisor version into Coq. Spoq successfully translated all of the 3.8K LoC of C and Arm assembly code into Coq. The same code that is compiled to execute is used for verification; there is no difference, ensuring that the proofs hold at the source code level for the code that is executed. This is in contrast to the previous work to verify SeKVM, which used ClightGen to translate its implementation into Coq. This required further retrofitting of the source code because of its use of many features unsupported by ClightGen, including removing all header files with versions that were amenable to translation by ClightGen.

Figure 9 shows examples of the retrofitting required to use ClightGen. Figure 9a shows a GNU C compiler directive `__section` which tells the linker to link the function into a special text section that SeKVM later isolates and protects from the rest of the kernel. ClightGen does not support such GNU C compiler directives, which are heavily used in systems code to control compilation and linking behavior. To use ClightGen, we first need to remove those GNU C compiler directives from all functions. Figure 9b shows a C macro `readl_relaxed` with inline assembly. ClightGen does not support such C macros or inline assembly. To use ClightGen, we need to either rewrite all such macros into standard C functions, or model them as abstract functions whose implementations are not verified and must be included in the TCB. Figure 9b shows the latter approach. The macro is replaced with just a function declaration so it can be translated by ClightGen, and a specification is written for the function, but the function implementation cannot be verified. There are over a hundred such functions in the original source code. These required changes result in a gap between the code that is verified versus the code that is compiled and executed. Unfortunately, without supporting features such as GNU C compiler directives, the verified code cannot be directly compiled and executed.

Generating specifications and proofs. We then used Spoq to generate the top-level specification for SeKVM, including all layer specifications and refinement proofs. Table 1 shows the manual proof effort required to verify SeKVM's functional correctness using Spoq, as measured by the LoC in Coq that still needed to be manually written to complete the verification. We also propagated the proofs to Arm's relaxed memory hardware, but omit details as it is similar to VRM's proof [54].

We wrote less than 100 LoC to provide the layer structure in a layer configuration file consisting of the same 34 layers as the original proofs for SeKVM; the changes in the updated version of SeKVM were minor enough that no changes in the layer structure were needed. We wrote 0.5K LoC for the bottom layer machine model for concurrency-related structures.

For C code without loops and Arm assembly code without jumps, Spoq automatically generated all low-level specifications and identical refinement proofs. For C code with loops, Spoq automatically generated all low-level specifications given a ranking function for each loop, each requiring 2 LoC. For C code with loops within conditionals, we wrote 0.8K LoC for identical refinement proofs that could not be automated by the current Spoq prototype, much of which involved copying and pasting of Coq code for termination proofs when multiple conditional branches used the same loop.

For C code and Arm assembly code without jumps, Spoq automatically generated all high-level specifications and lifting refinement proofs that do not use data abstractions. No manual proofs were required to verify Z3 transformations. For assembly code with jumps, we wrote 0.3K LoC for specifications and 0.1K LoC for refinement proofs, without decomposing specifications and proofs into low-level and high-level ones. For layers using data abstractions, one for locks and three for page tables, we manually wrote high-level specifications and lifting refinement proofs. For high-level specifications, we wrote 1.0K LoC for layers using data abstractions. For lifting refinement proofs, we wrote 0.8K LoC for locks, 2.5K LoC to show multi-level page tables refine a single-level page mapping, and 0.9K LoC to show data structures tracking ownership of physical pages refine an abstract map.

Reducing manual proof effort. Table 1 compares the proof effort to verify SeKVM using Spoq versus the manually writ-

LoC in Coq	Original	Spoq	Reduction
Layer configuration	—	0.1K	—
Machine model	1.8K	0.5K	72%
Low-level specifications for C	5.6K	0	100%
Ranking function	—	26	—
High-level specifications for C	5.5K	1.0K	82%
Specifications for Asm	0.5K	0.3K	40%
Identical refinement proofs for C	3.6K	0.8K	78%
Lifting refinement proofs for C	14.7K	4.2K	71%
Refinement proofs for Asm	1.8K	0.1K	94%
Security proof	4.8K	3.0K	38%
Total for functional correctness	**33.5K**	**7.0K**	**79%**
Total w/security	**38.3K**	**10.0K**	**74%**

Table 1: Manual proof effort to verify SeKVM.

ten proofs for the original version of SeKVM. The original manual proof effort required writing more than 3 times as many lines of specification and 5 times as many lines of proof as verified source code. Spoq only required writing a third as many lines of specification and roughly 1.4 times as many lines of proof as verified source code. In terms of LoC, Spoq reduced the overall manual proof effort by more than 70% compared to the original manually written proofs. The largest reductions in proof effort were for writing the specifications themselves. Spoq reduced manual effort for writing specifications by more than 90% overall, including eliminating the cost for specifications without data abstractions. Spoq reduced manual effort for refinement proofs by more than 70% overall, including eliminating the cost for C code without loops or data abstractions. Spoq reduced manual effort for refinement proofs for assembly code by more than 90% and linked them together with the proofs for C code, in contrast to the original assembly code proofs for SeKVM. Spoq largely eliminated the cost of using intermediate layers to modularize proofs, a substantial cost in the original manually written proofs, as the vast majority of those layer specifications and refinement proofs were automatically generated by Spoq.

Spoq also reduced the manual effort in defining the bottom layer machine model by roughly 70% due to three reasons. First, Spoq automatically derived many aspects of the abstract machine model from the source code. In contrast, the machine model for the original manually written proofs did not have such a correspondence with the source code and had to be manually written. Second, Spoq can use a simpler machine model because it does not need data oracles [38], which were introduced in the original manually written proofs to verify security properties. We discuss below how we verify security properties in a different manner, making data oracles unnecessary. Finally, Spoq does not need to include various getter and setter functions in the bottom layer, which were required in the original manually written proofs. These getters and setters, written using various Linux macros, previously had to be manually specified as part of the bottom layer specification because they could not be translated by ClightGen into Coq and hence could not be verified. In contrast, Spoq automatically translated these getters and setters into Coq and verified them, eliminating them from the bottom layer specification.

We compared the Coq code generated by Spoq versus the original manually written proofs for SeKVM to provide a measure of the quality of the generated specifications and proofs versus what would be produced by humans. Spoq generated 2.5K, 6.6K, 4.2K, 6.9K, and 17.5K LoC in Coq for the machine model, low-level specifications for C code, high-level specifications for C code, identical refinement proofs for C code, and lifting refinement proofs for C code, respectively. In most cases, the generated Coq code was only modestly larger than what was produced by a human writing hand-tuned Coq specifications and proofs. In fact, Spoq generated tighter high-level intermediate layer specifications than the original manually written specifications. The top-level specification generated by Spoq was 1.6K LoC in Coq. This is essentially the same size as the original manually written top-level specification, though it is quite different as it is based on a different machine model derived from the source code for the bottom layer. The quality and complexity of the top-level specification is especially important since it should be simple enough that it can be used to prove higher-level properties of the system.

Proving security properties. To demonstrate the usefulness and correctness of the top-level specification generated by Spoq, we used it to verify the security properties of SeKVM, specifically that it protects the confidentiality and integrity of virtual machine (VM) data. The original manually written proofs used noninterference to prove the security properties along with data oracles for declassification. We instead leverage the ideal/real paradigm to prove security properties, introduced in our recent work on verifying the firmware for the Arm Confidential Compute Architecture [40]. We define an ideal machine model that guarantees the security of each VM's private data regardless of the behavior of the hypervisor. The ideal machine defines for each VM a logically isolated memory space and register set, and directs all memory and register accesses from VMs to the logical state unless data declassification is defined. To account for data declassification in SeKVM in which a VM can make requests to dynamically start and stop sharing a piece of memory with the hypervisor, the ideal machine moves data from the VM's logical memory to shared memory and vice versa. The VM accesses its private data from its logical memory space, and accesses the shared data from the shared physical memory. By definition, the per-VM isolated state is only accessible by the VM itself, so the confidentiality and integrity of VM data is naturally guaranteed in the ideal machine. We then prove the top-level specification refines the ideal machine, which verifies that SeKVM indeed protects the confidentiality and integrity of VM data.

The security proof provides three key advantages compared to the original security proofs based on noninterference. First, it does not require incorporating data oracles in the machine model and in various layer specifications, decoupling the security proof from verifying functional correctness. Second,

Name	Description
Apache	Apache server v2.4.41 handling 100 concurrent requests via TLS/SSL from remote ApacheBench [2] v2.3 client, serving index.html of the GCC 7.5 manual.
Kernbench	Compilation of the Linux kernel v4.18 using allnoconfig for Arm with GCC 9.3.0.
Memcached	Memcached v1.5.22 handling requests from a remote memtier [49] v1.2.11 client with default parameters.
MongoDB	MongoDB server v3.6.8 handling requests from a remote YCSB [10] v0.17.0 client running workload A with 16 concurrent threads and operationcount=500000.
MySQL	MySQL v8.0.31 running sysbench v1.0.11 with 32 concurrent threads and TLS encryption.

Table 2: Application benchmarks.

the proof itself is simpler, reducing manual proof effort. Table 1 shows the manual proof effort for the security proof using this approach is 38% less than the original security proof using noninterference, though this reduction in proof effort is unrelated to using Spoq. Finally, and most importantly, the security proof only needs to trust the specification of a small idealized secure machine model, which is roughly 200 LoC; the much larger specification of the real system does not need to be trusted. The trusted specification defines how VMs load and store private data to their logical isolated space and specifies the data declassification policy for moving data between the logical isolated and shared machine states.

Performance of verified implementation. We directly compiled the Spoq-verified SeKVM source code into a binary image, and executed it on a Raspberry Pi 4B with 8 GB RAM, 64 GB SanDisk SD card, and a built-in 1 Gbps NIC. We measured its performance by running the application workloads listed in Table 2 in a VM using SeKVM. For comparison, we also ran the workloads in a VM using vanilla KVM and natively on the hardware. Each VM was configured with 2 vCPUs and 4 GB RAM. vCPUs were pinned to individual physical cores, VHOST networking was used, and virtual block storage devices were configured with `cache=none` [12, 24, 33, 52]. When running natively, we restricted the workloads to use 2 CPUs and 4 GB RAM to provide a fair comparison. VMs used a vanilla Linux v5.4 kernel as their guest OS. The VM on SeKVM included modified virtio drivers in its guest OS to support SeKVM. The Raspberry Pi ran a proprietary Linux v5.4.55 kernel [45]. It lacks support for virtio front-end drivers so could not be used as a guest OS. For client-server applications, clients ran on an x86 machine with 10-core Intel Xeon CPU E5-2640 2.4 Ghz CPU, 48 GB RAM and a NetXtreme BCM5719 1 Gbps NIC, connected to the Raspberry Pi via a Netgear GS308 1 Gbps switch.

Figure 10 shows application workload performance when using VMs with vanilla KVM and SeKVM. Performance was normalized to native execution; lower is better. The performance results are consistent with those previously reported for SeKVM [54], with worst case overhead being less than 15% compared to vanilla KVM. I/O intensive application workloads incurred higher overhead because the hypervisor cannot access VM memory unless the virtio front-end driver makes explicit hypercalls to request memory pages used for I/O be temporarily accessible to the hypervisor to pass the I/O data to the back-end driver in the host.

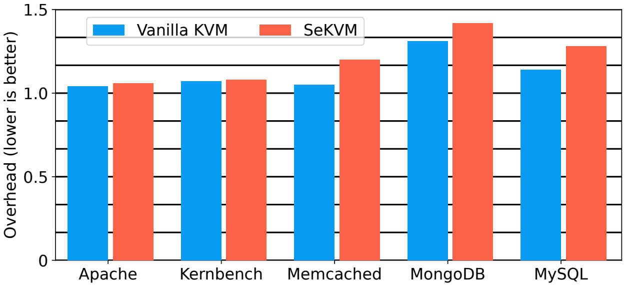


Figure 10: Application benchmark performance.

8 Limitations

Spoq's TCB includes the Clang and LLVM toolchains, Spoq's translator, and Spoq's semantic definitions for LLVM IR and assembly. The translator is currently unverified and supports a subset of LLVR IR and Arm assembly, so it may fail to translate some source code into Coq. Spoq's specification and proof generator are not part of its TCB. Enhancing their support for assembly code with jumps is an area of future work.

The Z3 solver is currently the bottleneck in Spoq's runtime performance. Synthesizing high-level specifications for relatively large functions can take over 30 minutes because it may involve thousands of Z3 queries. Nevertheless, automatically generating specifications and proofs for SeKVM only takes two hours on an AWS machine with an 8-core 2.3 GHz Intel Xeon CPU E5-2686 v4 and 32 GB RAM, an insignificant amount of time compared to the time it takes to manually write specifications and proofs.

Spoq currently relies on users to complete all data abstraction proofs. Developing a library of commonly used data abstraction proofs for proof automation is an area of future work.

9 Related Work

Verified systems in C. seL4 [31] presents the first machine-checked functional correctness proof of an OS kernel. It used an unverified parser to translate C into Isabelle/HOL, and is manually proved with simplified C semantics. For example, pointers to local variables are disallowed by the simplified C semantics. Assembly code is also unverified. AtomFS [61] used a verification framework [56] that does not support assembly code or full C semantics. Many verified systems [3, 8, 11, 20, 30, 32, 38–41, 54] used ClightGen. For code that can be parsed by ClightGen and compiled by CompCert, the CompCert toolchain can guarantee proofs hold at the assembly level. However, CompCert cannot make any guarantees regarding concurrent code even if it can compile, and our results show that ClightGen cannot support verification of most real-world unmodified systems code.

Modeling and verifying LLVM IR. VeLLVM [59] includes formal semantics and tools to verify LLVM IR code in Coq. VeLLVM adopts CompCert's sequential machine memory model, so it cannot verify concurrent systems. It directly models small-step semantics of IR instructions in CFGs, making it problematic to use for systems with complex control flows. CreLLVM [28] extends VeLLVM to verify compiler optimization passes, but shares the same limitations of VeLLVM. VIR [48] also models small-step semantics of IR instructions in CFGs, suffering the same problems as VeLLVM. K-LLVM [36] defines LLVM IR operational semantics in the K framework, but cannot be used for deductive reasoning. SeaHorn [22] statically checks assertions in C programs by translating them to LLVM IR, then using the IR with an SMT solver and abstraction interpretation. Such automated verification tools cannot verify the functional correctness of a complex system. It is difficult to define program specifications using just assertions, and SMT solvers and abstract interpretation cannot prove complex proof goals.

Automating systems verification. AutoCorres [17] synthesizes specifications from C programs based on a fixed and simplistic machine model, which cannot be used to verify concurrent systems. It only supports an even more limited subset of C than ClightGen and does not support assembly code. The specifications generated are low level yet machine dependent, making them difficult to use to verify higher-level properties.

Push-button verification is a fully automated verification technique that has been used to verifying a file system [50], compiler [53], and OS kernel [43, 44, 51]. Users only need to write specifications in addition to the system implementation, and the verification framework automatically completes the proofs. However, implementations have restrictive constraints, such as uniprocessor only and constant bounds for loops so SMT solvers can be used. Unlike Spoq, verification requires manually defined specifications, does not hold for concurrent systems, and lacks machine-checkable proofs, as the unverified SMT solver provides no proof of its answer or any way to express a proof that can be machine checked.

Verification-aware programming languages such as Dafny [34] and F* [47] have been used to implement verified storage systems [25, 26] and crypto libraries [46, 60]. Developers write code, specifications, and proofs all together in the same language. A compiler validates users' proofs, in part using an SMT solver, and generates source code in familiar programming languages, which can in turn be further compiled and executed. Building on Dafny, Armada [42] uses a set of pre-built proof strategies to generate refinement proofs between levels of intermediate specifications, which users are expected to write to bridge the semantic gap between an implementation and its high-level specification. Unlike Spoq, layers are not supported and systems written in C and assembly code cannot be verified without being rewritten.

Decompilation. Decompilation techniques recover a program's source code given only its binary [1, 6, 9, 23, 27, 57, 58], though the recovered and original source code generally do not match. Some techniques do not reconstruct program structure [1, 58], some do so with goto statements [6], and some only do so with various restrictions on program CFGs [57]. More recent work can reconstruct program structure for arbitrary CFGs [23] without using goto statements, but requires a much more complex algorithm than used by Spoq. In contrast, Spoq employs a simpler algorithm to reconstruct program structure for arbitrary CFGs, and keeps the original LLVM IR instructions, which are much simpler and more rigorously defined than C, instead of trying to recover source code. To support proof decomposition and simplify specification synthesis, Spoq intentionally does not employ a richer variety of source code primitives such as goto or switch statements. Its resulting representation is more amenable to verification. Its design and implementation is far simpler than previous approaches to keep its TCB small, which is important for verification.

10 Conclusions

Spoq is the first system that can automate the generation of Coq representations, specifications, and proofs for C systems code to enable machine-checkable verification of concurrent system software. Spoq translates C systems code compiled into LLVM IR directly into Coq, converting IR control flow graphs into structured program functions to simplify verification while supporting full C semantics, including GNU C extensions and inline assembly. Using a layering proof strategy, Spoq introduces novel Coq tactics and transformation rules to automatically synthesize layer specifications and refinement proofs, even for functions with loops. Users can interact with Spoq to further refine the generated specifications and proofs at any layer. We used Spoq on commodity system software, such as the Linux kernel, to translate over 99% of their source code directly into Coq for verification. We also used Spoq to verify a multiprocessor KVM hypervisor implementation, showing that it reduces manual proof effort by over 70% while ensuring that the proofs hold for the unmodified implementation that is compiled and executed.

11 Acknowledgments

Jay Lorch provided helpful and meticulous feedback on earlier drafts. This work was supported in part by three Amazon Research Awards, a Guggenheim Fellowship, a VMware Systems Research Award, an NSF CAREER Award, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080. Ronghui Gu is the founder of and has an equity interest in CertiK.

References

[1] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 2016)*, pages 583–600, Austin, TX, August 2016.

[2] Apache Software Foundation. ab - Apache HTTP Server Benchmarking Tool. `http://httpd.apache.org/docs/2.4/programs/ab.html`. Accessed on December 13, 2022.

[3] Andrew W. Appel. Verified Software Toolchain. In *Proceedings of the 20th European Symposium on Programming (ESOP 2011)*, pages 1–17, Saarbrücken, Germany, March 2011.

[4] ARM Ltd. ARM Architecture Reference Manual ARMv8-A DDI0487A.a, September 2013.

[5] ARM Ltd. Procedure Call Standard for the Arm® 64-bit Architecture (AArch64). `https://github.com/ARM-software/abi-aa/releases/download/2022Q1/aapcs64.pdf`, April 2022.

[6] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 2013)*, pages 353–368, Washington, D.C., August 2013.

[7] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, February 2017.

[8] Hao Chen, Xiongnan Newman Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 431–447, June 2016.

[9] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010)*, pages 143–154, Indianapolis, IN, June 2010.

[11] David Costanzo, Zhong Shao, and Ronghui Gu. 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)*, pages 648–664, June 2016.

[12] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, pages 304–316, Seoul, South Korea, June 2016.

[13] Christoffer Dall and Jason Nieh. KVM/ARM: Experiences Building the Linux ARM Hypervisor. Technical Report CUCS-010-13, Department of Computer Science, Columbia University, June 2013.

[14] Christoffer Dall and Jason Nieh. Supporting KVM on the ARM Architecture. *LWN Weekly Edition*, pages 18–22, July 2013.

[15] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, pages 333–347, Salt Lake City, UT, March 2014.

[16] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2008)*, pages 337–340, Budapest, Hungary, March 2008.

[17] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't Sweat the Small Stuff: Formal Verification of C Code without the Pain. *ACM SIGPLAN Notices*, 49(6):429–439, June 2014.

[18] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, and Haozhong Zhang. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL 2015)*, pages 595–608, Mumbai, India, January 2015.

[19] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan Newman Wu, Vilhelm Sjöberg, and David Costanzo. Building Certified Concurrent OS Kernels. *Communications of the ACM*, 62(10):89–99, October 2019.

[20] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI 2016), pages 6530–669, Savannah, GA, November 2016.

[21] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 646–661, Philadelphia, PA, June 2018.

[22] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The SeaHorn Verification Framework. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015)*, pages 343–361, San Francisco, CA, July 2015.

[23] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A Comb for Decompiled C Code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS 2020)*, pages 637–651, Taipei, Taiwan, October 2020.

[24] Stefan Hajnoczi. An Updated Overview of the QEMU Storage Stack. In *LinuxCon Japan 2011*, Yokohama, Japan, June 2011.

[25] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage Systems are Distributed Systems (So Verify Them That Way!). In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, pages 99–115, November 2020.

[26] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, pages 165–181, Broomfield, CO, October 2014.

[27] R. Nigel Horspool and Nenad Marovac. An Approach to the Problem of Detranslation of Computer Programs. *The Computer Journal*, 23(3):223–229, August 1980.

[28] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, et al. Crellvm: Verified Credible Compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 631–645, Philadelphia, PA, June 2018.

[29] Kbuild - The Linux Kernel Documentation. https://docs.kernel.org/kbuild/kbuild.html. Accessed on December 13, 2022.

[30] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and Liveness of MCS Lock—Layer by Layer. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems (APLAS 2017)*, pages 273–297, November 2017.

[31] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, pages 207–220, Big Sky, MT, October 2009.

[32] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C Pierce, and Steve Zdancewic. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2019)*, pages 234–248, Cascais, Portugal, January 2019.

[33] KVM Contributors. Tuning KVM. https://www.linux-kvm.org/index.php?title=Tuning_KVM&oldid=173911, June 2018. Accessed on December 13, 2022.

[34] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2010)*, pages 348–370, Dakar, Senegal, April 2010.

[35] Xavier Leroy. The CompCert C Verified Compiler: Documentation and User's Manual. https://compcert.org/man/, November 2022. Accessed on December 13, 2022.

[36] Liyi Li and Elsa L Gunter. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP 2020)*, pages 7:1–7:29, Dagstuhl, Germany, November 2020.

[37] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1357–1374, Santa Clara, CA, August 2019.

[38] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (IEEE S&P 2021)*, pages 1782–1799, San Francisco, CA, May 2021.

[39] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, pages 3953–3970, Vancouver, BC Canada, August 2021.

[40] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 465–484, Carlsbad, CA, July 2022.

[41] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Manki Yoon. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL 2020)*, January 2020.

[42] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Haojun Ma, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Automated Verification of Concurrent Code with Sound Semantic Extensibility. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44:1 – 39, May 2022.

[43] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 225–242, October 2019.

[44] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, pages 252–269, Shanghai, China, October 2017.

[45] Raspberry Pi. Kernel Source Tree for Raspberry Pi-provided Kernel Builds. https://github.com/raspberrypi/linux/tree/rpi-5.4.y. Accessed on December 13, 2022.

[46] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *Proceedings of 2020 IEEE Symposium on Security and Privacy (IEEE S&P 2020)*, pages 983–1002, San Francisco, CA, May 2020.

[47] 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. Verified Low-Level Programming Embedded in F*. In *Proceedings of the ACM on Programming Languages*, volume 1, pages 1–29, August 2017.

[48] Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*, pages 106–113, Vienna, Austria, July 2014.

[49] Redis Labs. Memtier Benchmark. https://github.com/RedisLabs/memtier_benchmark. Accessed on December 13, 2022.

[50] Helgi Sigurbjarnarson, James Bornholt, Nicolas Christin, and Lorrie Faith Cranor. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI 2016)*, pages 1–16, Savannah, GA, November 2016.

[51] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, pages 287–305, Carlsbad, CA, October 2018.

[52] SUSE. Performance Implications of Cache Modes. https://documentation.suse.com/sles/12-SP5/html/SLES-all/cha-cachemodes.html. Accessed on December 13, 2022.

[53] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Fred Chong, and Ronghui Gu. Giallar: Push-Button Verification for the Qiskit Quantum Compiler. *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*, pages 641–656, June 2022.

[54] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP 2021)*, pages 866–881, Virtual Event, Germany, October 2021.

[55] The Coq development team. The Coq Proof Assistant. http://coq.inria.fr. Accessed on December 13, 2022.

[56] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A Practical Verification Framework for Preemptive OS Kernels. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016)*, pages 59–79, Toronto, ON, Canada, July 2016.

[57] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS 2015)*, San Diego, CA, February 2015.

[58] Alon Zakai. Emscripten: an LLVM-to-JavaScript Compiler. In *Proceedings of the 26th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications Companion (Wavefront 2011)*, pages 301–312, Portland, Oregon, October 2011.

[59] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 427–440, New York, NY, January 2012.

[60] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 1789–1806, October 2017.

[61] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 259–274, Huntsville, ON Canada, October 2019.