Automated Verification of RPC Stub Code

Matthew Fernandez, June Andronick, Gerwin Klein, Ihor Kuz

NICTA and UNSW
Sydney, Australia
{matthew.fernandez,june.andronick,gerwin.klein,ihor.kuz}@nicta.com.au

Abstract. Formal verification has been successfully applied to provide strong correctness guarantees of software systems, but its application to large code bases remains an open challenge. The technique of component-based software development, traditionally employed for engineering benefit, also aids reasoning about such systems. While there exist compositional verification techniques that leverage the separation implied by a component system architecture, they implicitly rely on the component platform correctly implementing the isolation and composition semantics they assume. Any property proven using these techniques is vulnerable to being invalidated by a bug in the code of the platform itself. In this paper, we show how this assumption can be eliminated by automatically generating machine-checked proofs of the correctness of a component platform's generated Remote Procedure Call (RPC) code. We demonstrate how these generated proofs can be composed with hand-written proofs to yield a system-level property with equivalent assurance to an entirely hand-written proof. This technique forms the basis of a scalable approach to formal verification of large software systems.

1 Introduction

In the design of safety- and security-critical software, it is desirable to provide the high levels of assurance that can be achieved by formal verification. State of the art code-level verification currently scales to tens of thousands of lines of code [13, 16], while high assurance software can often exceed one million lines of code. For such large systems, pervasive code-level verification still is infeasible and new techniques are required.

Component-based software engineering facilitates the design and implementation of large software systems [25]. This methodology involves specifying a system as a collection of isolated software elements that communicate via explicit connections, expressed in an architectural description. An example would be a simple system with two separate components, a client `c` and a server `s`, with a communication connection between them, allowing `c` to invoke a function implemented in `s`. The component platform would generate so-called *glue code* to perform argument marshalling and unmarshalling, and use the underlying operating system's communication mechanisms to transfer the data between components. By decomposing the problem of system verification along component boundaries, assurance of larger systems becomes tractable. A proof of system correctness chains together individual correctness proofs of the underlying operating system, component platform code, and user-provided component code.

Compositional verification of component systems is not a new concept, with existing techniques such as [2, 4] aiming to increase scalability through decomposition.

However, all existing techniques we are aware of assume that an underlying component platform correctly implements the isolation and composition semantics they rely upon. This assumption encompasses the glue code, generated by the component platform. A defect anywhere in the glue code generation logic can falsify the implicit assumptions of a compositional reasoning framework and thereby invalidate derived properties [7].

To preserve the abstraction of a component system architecture and to aid reasoning about component systems, we aim to automate the production of functional correctness proofs for platform glue code. In this paper, we focus on generated code for *Remote Procedure Calls* (RPC) in particular. RPC is a common communication abstraction in component platforms. We refer to such generated code as RPC stubs in the context of this paper. Our generated proofs are machine-checked by the interactive theorem prover Isabelle/HOL [19], and the resulting proofs are designed to be manually composed with hand-written proofs of user-provided code. Together, the generated and hand-written proofs can yield a proof of functional correctness of a whole system. Though relying on generated proof script, the final proof carries the same level of assurance as a manually-constructed, machine-checked proof.

The main challenge of this work is in generating an Isabelle/HOL proof script that corresponds to a generated implementation. Since the implementation is derived from a system architecture description, and the existing code generator is largely string- and template-based, it is infeasible to provide a single proof for the correctness of any possible glue code (that is, to verify the generator itself). Instead, we use a translation validation approach and build on our previous work that demonstrated the absence of undefined behaviour in component platform glue code [6].

We use an existing component platform, CAmkES [14], and focus on the verification of its C backend targeting the seL4 microkernel [13]. In future work, we intend to leverage the functional correctness proof of seL4, though for now we implicitly assume the semantics of its system calls in our execution model.

We make the following novel contributions: (i) We demonstrate a technique for automatically generating functional correctness proofs of generated RPC code, removing the assumption of correct RPC stubs present in existing compositional reasoning frameworks and (ii) we present a strategy for composition of automated and manual proofs that does not require trusting the proof generator.

After describing, in Sect. 2, the runtime environment and the C verification framework we use, we elaborate the proof generation process in Sect. 3 and describe what precisely the generated proofs show. Being result verification, the approach is best demonstrated working through an example instance, which we do in Sect. 4. We discuss the trustworthiness of the resulting property and limitations of our approach in Sect. 5.

2 Background

2.1 seL4

To date, seL4 is the only general purpose operating system kernel with a code-level proof of functional correctness [13]. It contains around 9700 lines of C that have been proven to implement an abstract specification of the kernel's behaviour. The verification of seL4 extends to access control and information flow guarantees [17, 24].

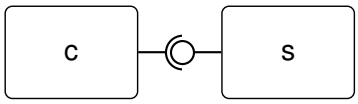

Fig. 1. Example CAmkES system architecture

The microkernel provides a minimal number of services to userspace processes, with abstractions for processor time (threads) and virtual memory. All authority in seL4 is provided through capabilities, kernel-administered access tokens for resources. Communication is achieved by having a capability to an endpoint object and invoking this capability. Capabilities have associated rights, with a *write* capability required to send on an endpoint and *read* capability to receive on an endpoint. The kernel provides a synchronous Inter-Process Communication (IPC) mechanism that allows a sender to transfer up to 484 bytes to a cooperative receiver from a fixed window of their address space known as their IPC buffer. The IPC buffer is accessed by a pair of utility functions, `seL4_GetMR` and `seL4_SetMR`, that are provided for reading and writing individual words at offsets into the buffer. For further information about the seL4 primitives, the reader is referred to the programmer's reference [26].

2.2 CAmkES

CAmkES is a component platform for implementing microkernel-based embedded systems [14]. A user provides a high-level architectural description of her system, and code implementing the logic of each component in the system. At compile time, CAmkES generates glue code to establish and enforce communication channels between the user's component instances, as described in her architectural description. For each component instance, its user-provided code and generated code are compiled and linked together to form an executable image. We focus on CAmkES' C backend for seL4 in this work, as representative of an environment for building high assurance systems. The correctness guarantees of seL4 present a strong foundation and C requires no implicit assumption of a correct language runtime.

CAmkES architectures are limited to static systems and all components are instantiated on system start. Three communication abstractions are available for system design: dataports, events and procedures. Procedures, which we focus on in this work, are used for representing communication in the style of synchronous function calls. Their semantics follows the well known Remote Procedure Call abstraction, so we use RPC terminology to refer to them. At compile time, CAmkES generates RPC stubs to perform argument marshalling, argument unmarshalling, and kernel invocations to transfer control and data between components. Because all communication channels are established statically and are local to a processor, communication runtime failures in CAmkES systems can only occur by defects in the underlying kernel or generated glue code. In this work, we prove the absence of defects in the generated glue code and use a formally verified kernel as substrate.

Component architectures are often represented diagrammatically as a set of boxes for the component instances and arrows for the connections between them. Fig. 1 provides an example, showing two connected component instances. Here *s* implements a single procedural interface, containing one or more exposed methods. An interface of

the same type is expected by c. The two interfaces are connected, such that s provides the interface c is using. Though Fig. 1 only shows basic functionality, the full feature set of CAmkES is sufficient to build complex systems such as network routers and file systems. We will elaborate the example from Fig. 1 in Sect. 4.

2.3 Verification Framework

To reason about C programs, we first translate them into Isabelle/HOL. We initially use the C-to-Simpl parser [27, 29] to translate source code to a representation in the generic imperative language Simpl [23] in Isabelle/HOL. This translation is designed to be as straightforward as possible and to match the semantics of a large subset of C99 [12]. Following this, we use the AutoCorres tool [10, 11] to perform further automatically verified transformations within Isabelle/HOL that abstract the Simpl representation, resulting in a monadic functional specification that more closely resembles the programmer's intuition and facilitates reasoning on top. The C-to-Simpl parser's initial translation step can be independently validated using binary verification [24].

To reason about abstracted C programs, we adopt a variant of Hoare triples for stating the pre- and post-conditions of a potentially nondeterministic monadic function [5], which, in addition to transforming the state, may return a value or may fail. The following notation states that, if the pre-condition P holds and the function f terminates normally, then the post-condition Q will hold after f has executed.

$$\{\!|P|\!\}\ f\ \{\!|Q|\!\}$$

P is a predicate over the initial state (memory, global structures, etc.), whereas Q is a predicate over two parameters: the return value of f and the final state.

The above statement allows for the possibility that f fails or does not terminate, for example, by performing an operation with undefined behaviour in C. To express total correctness of f, we use the following variant that requires termination and absence of failure, including absence of guard violations.

$$\{\!|P|\!\}\ f\ \{\!|Q|\!\}!$$

In the remainder of this paper we use the following notation to refer to the initial state in the post-condition, for example, to express that the effect of a given C function f is to modify the state according to a specification function g, ignoring f's return value.

$$\{\!|\lambda s.\ s = s0|\!\}\ f\ \{\!|\lambda\_\ s.\ s = g\ s0|\!\}!$$

We refer to [5] for further details of this Hoare calculus for monadic functions.

3 Generating Correct RPC Stubs

Our anticipated process for verifying a component-based system is depicted in Fig. 2, where solid borders surround user-provided artefacts and dashed borders surround generated artefacts. The CAmkES platform generates a generic theorem of the correctness of RPC stub code for each procedure, alongside the generated code itself. The formal representations of the RPC stub functions in these theorems are derived by the C-to-Simpl parser directly from the generated code. The user can then instantiate the generic theorems for specific correctness properties.

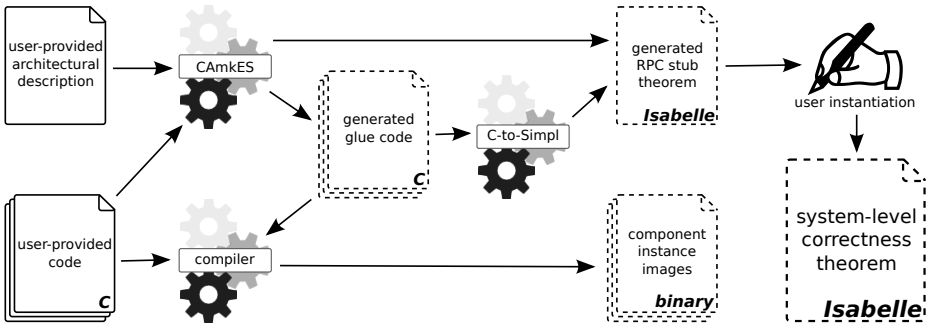

Fig. 2. Workflow for producing a proof of system correctness

In the context of RPC stubs, it is not immediately clear what "correctness" means for generated code. In this section, we explain our criteria for correct RPC stubs and elaborate on what precisely is proven in the Isabelle/HOL theories we generate.

Let `f` be the RPC stub code that provides a conduit for invoking a remote function `g`. Then, intuitively, invocation of `f` should be somehow equivalent to a direct invocation of `g`, were it colocated with the caller.

Since, by design, it is expected that the RPC code perform observable actions, we cannot expect full equivalence. However, we do expect to be able to lift suitable correctness specifications in the form of Hoare triples about `g` from the remote context into the local context. We state the correctness of glue code by specifying this lifting, and the generated glue code proofs establish that this specification lifting is indeed achieved. The generality of the specifications we allow for `g`, implies not only what the glue code must do, but also what it must *not* do (e.g. interfere with `g`'s private state or the caller's private state). We expect this form of statement and mechanism to readily generalise to the other transport mechanisms component platforms provide.

To be useful, the proofs we generate must be composable with (almost) arbitrary user-provided proofs, both of functions `g`, and of the contexts where `g` and `f` are used. We allow the proof engineer to state and prove the correctness criteria of her own functions *after* generation of the proofs of RPC stub correctness. To accomplish this, we parameterise the generated proofs within an Isabelle locale. Isabelle locales are named contexts containing fixed parameters, assumptions and definitions [28]. We show an informal version of the template of the locale we generate in Fig. 3, and refer to lines from it in the following discussion.

For each user-provided function `g`, we assume locale parameters $\mathtt{P}_g$ and $\mathtt{Q}_g$ that capture the pre-condition and post-condition of `g` (lines 2-4). The generated proofs and specification for `f` then describe under which circumstances $\mathtt{P}_g$ and $\mathtt{Q}_g$ specify the behaviour of the function `f`, including the RPC stubs for `g`.

Not all $\mathtt{P}_g$ and $\mathtt{Q}_g$ are suitable. In particular, they must not make statements about glue code variables and memory. We refer to this as wellformedness of the pre- and post-condition. An example of this is shown in lines (5-7) of Fig. 3, which require that $\mathtt{P}_g$ and $\mathtt{Q}_g$ do not depend on the contents of the IPC buffer. Ideally, this would already be

```
1  locale rpcstubs =
2   fixes P_g :: lifted_globals ⇒ ... ⇒ bool
3   fixes Q_g :: lifted_globals ⇒ lifted_globals ⇒ ... ⇒ bool
4   assumes g_wp: {|λs. s = s0 ∧ inv s ∧ P_g s ...|} g {|λr s. inv s ∧ Q_g s0 s ...|}!
5   assumes g_stable_setmr1: ∀ s i x. P_g (setMR s i x) = P_g s
6   assumes g_stable_setmr2: ∀ s i x. Q_g (setMR s i x) = Q_g s
7   assumes g_stable_setmr3: ∀ s0 s i x. Q_g s0 (setMR s i x) = Q_g s0 s
8
9  theorem f_wp:
10  {|λs. s = s0 ∧ inv s ∧ ... ∧ P_g s ...|}
11   do f_marshal ...;
12     g_internal;
13     f_unmarshal ...
14   od
15  {|λr s. inv s ∧ Q_g s0 s ...|}!
```

Fig. 3. RPC locale template

achieved by language scoping mechanisms, but C provides no guarantees and Isabelle no scoping. Instead we phrase these conditions as explicit locale assumptions that the proof engineer must discharge when making use of the generated proofs.

In addition to restrictions on $\mathtt{P}_g$ and $\mathtt{Q}_g$, we also require restrictions on the behaviour of `g`: it must not modify glue-code internal state. We refer to this as the user function being well behaved. This is included in the explicit locale assumption in line 4 of Fig. 3 as *inv s* and must be discharged later by the proof engineer.

The template for the parameterised correctness theorem that is produced appears in lines 9-15 of Fig. 3. This theorem lifts the Hoare triple over `g`, to a Hoare triple over `f`, where `f` is represented by a specific sequence of glue code operations, expressed in a syntax similar to Haskell's `do` notation. When `f` is called, it marshals arguments into the sender's IPC buffer (line 11), then performs an seL4 system call to transfer this data to the waiting receiver's stub. On the receiver's side (line 12), arguments are unmarshalled and the user's implementation, `g`, is invoked. When `g` returns, the receiver's stub marshals return arguments and performs another system call to transfer data back to the sender. Finally, `f` unmarshals return arguments and returns to the caller (line 13). The seL4 system calls do not occur in this theorem; this means that for now we axiomatise the semantics of these seL4 system calls as a direct function call from `f` to the receiver's stub code.

To utilise the generated proofs, for instance in the context of hand-written proofs of further user-provided code, the proof engineer instantiates (*interprets* in Isabelle parlance) the generated locale by providing specific pre- and post-conditions for `g` and discharging the locale assumptions. The manual inputs are the specific pre- and post-conditions for `g`, a proof that `g` satisfies these, a proof that $\mathtt{P}_g$ and $\mathtt{Q}_g$ are wellformed, and a proof that `g` itself is well behaved. The result is a specific Hoare triple about `f` that can be used in further proofs.

Neither proving wellformedness of $\mathtt{P}_g$ and $\mathtt{Q}_g$ nor proving `g` is well behaved is onerous. They can mostly be discharged automatically once the user-code and specification

```
1  procedure Swapper {
2    unsigned int swap(inout int a, inout int b);
3  }
4
5  component Client { control; uses Swapper cs; }
6
7  component Service {  provides Swapper ss;  }
8
9  assembly {
10   composition {
11     component Client c;
12     component Service s;
13     connection seL4RPCSimple conn(from c.cs, to s.ss);
14   }
15 }
```

Fig. 4. Example system specification in CAmkES

are provided. The work for proving something about an RPC function `f` is reduced to proving almost the same property about `g`, pretending that it runs locally.

Sect. 4 provides a worked example of how exactly to achieve this.

4 Methodology Demonstrated on Example System

4.1 Component Architecture

The simplified example depicted previously in Fig. 1, describes an RPC interface provided by *s* and used by *c*. Fig. 4 shows the textual description of this system in CAmkES. It starts with the procedure interface definition `Swapper` (lines 1-3), comprising one method `swap`. The method takes two integer parameters that are used as both inputs and outputs, and also returns an unsigned integer value. The specification proceeds to define two component *types*, *Client* (line 5) and *Service* (line 7). *Client* has an active thread of execution, denoted by the *control* keyword and expects an instance of the *Swapper* interface under the name *cs*. *Service* is reactive, indicated by the lack of the *control* keyword, and implements an instance of the *Swapper* interface under the name *ss*.

The final block describes the architecture of the composed system. There is a single instance of each component type (lines 11-12) and the outgoing interface, expected by *c*, is provided by *s* via a connection *conn* of type *seL4RPCSimple*. The connection type determines the underlying transport for communication at runtime. Here, *seL4RPCSimple* is a type for RPC communication in the C language that uses the component instances' IPC buffers and an seL4 endpoint to pass arguments and return values.

The system we have just described contains a component instance *s* that exports a method for swapping the values of two integer parameters. The semantics we give to the `swap` method is not described in the architecture description. It is instead defined by user-provided code discussed in Sect. 4.2.

Component architectures would typically have many more procedures, interfaces, components, and connections. Though we have used this approach on larger systems, for simplicity of presentation we keep the system small in this example. There are few

```
1  static unsigned int counter;
2
3  unsigned int
4  ss_swap(int *a, int *b){
5      int temp = *a;
6      *a = *b; *b = temp;
7      counter++;
8      return counter;
9  }
```

Fig. 5. User-provided source code of *Service*

```
1  int run(void) {
2      int x = 3;
3      int y = 5;
4      unsigned int i;
5      i = cs_swap(&x, &y);
6      return 0;
7  }
```

Fig. 6. User-provided source code of *Client*

surprises and minimal manual work involved in moving to larger systems, chiefly because the lemma statements and generated proofs we go on to describe work for arbitrary numbers of components and connections, and the glue code proofs are pair-wise independent, so the proof size scales linearly in the size of the architecture.

4.2 User and Generated Code

The engineer developing a component system provides code for each component type in the system. Conceptually, she provides the contents of each of the boxes of an architectural diagram such as Fig. 1.

Fig. 5 and Fig. 6 give the C code for the components *Service* and *Client*, respectively. The code for *Service* exports a function, `ss_swap`, that swaps the value of two integer pointers and increments a global counter, returning the new value of the counter. The code for *Client* comprises a function, `run`, that acts as the component's entry point. It calls the function `cs_swap` with two local values it wishes to exchange. No code needs to be provided for `cs_swap` as this is what the component platform generates. The example demonstrates that our framework can handle bidirectional parameters, return values and manipulation of component-global state.

Note that this code is provided once per component *type* and then re-used for every instance of that type. In our example, there is only one instance of each type (*c* of type *Client* and *s* of type *Service*) so each piece of user code is only used once.

Fig. 7 and Fig. 8 show the RPC stubs automatically produced by CAmkES for this system. The first of these receives the user's call to `cs_swap`, marshals function arguments into *c*'s IPC buffer and then invokes a capability to an seL4 endpoint to transfer the data to *s*. After receiving *s*'s reply, it unmarshals the response and returns to the user. The second RPC stub operating in *s*'s address space, receives an incoming call from *c* as an invocation of the function `ss_swap_internal`. It unmarshals the call arguments, calls the user's implementation from Fig. 5, marshals the user's return values into *s*'s IPC buffer and then sends this reply message to *c*.

The stub for *s* is more complex than the stub for *c* as it has to deal with pointers that are part of the stub's private state. These pointers are accessed via the functions `get_swap_a` and `get_swap_b`, which are only expected to be called from the stub code.

Though the user designing a component system may think of her architecture as depicted in Fig. 1, the system at runtime is shown more precisely in Fig. 9. Each component instance is comprised of the code the user has provided (shown in solid boxes)

```
1  static unsigned int cs_swap_marshal(int a, int b) {
2    unsigned int index = 0;
3    seL4_SetMR(index, 0);  index++;
4    seL4_SetMR(index, (seL4_Word)a);  index++;
5    seL4_SetMR(index, (seL4_Word)b);  index++;
6    return index;
7  }
8
9  static void cs_swap_call(unsigned int length) {
10   seL4_MessageInfo_t info =
11     seL4_MessageInfo_new(0, 0, 0, length);
12   (void)seL4_Call(6, info); /* Call the seL4 endpoint */
13 }
14
15 static unsigned int cs_swap_unmarshal(int *a, int *b) {
16   unsigned int index = 0;
17   unsigned int ret = (unsigned int)seL4_GetMR(index);  index++;
18   *a = (int)seL4_GetMR(index);  index++;
19   *b = (int)seL4_GetMR(index);  index++;
20   return ret;
21 }
22
23 unsigned int cs_swap(int *a, int *b) {
24   unsigned int length = cs_swap_marshal(*a, *b);
25   cs_swap_call(length);
26   unsigned int ret = cs_swap_unmarshal(a, b);
27   return ret;
28 }
```

Fig. 7. Generated stub for *c*

```
1  /* User-provided implementation. */
2  extern unsigned int ss_swap(int *a, int *b);
3
4  static void ss_swap_unmarshal(int *a, int *b) {
5    unsigned int index = 1;
6    *a = seL4_GetMR(index);  index++;
7    *b = seL4_GetMR(index);  index++;
8  }
9
10 static unsigned int ss_swap_invoke(int *a, int *b) {
11   return ss_swap(a, b);
12 }
13
14 static unsigned int ss_swap_marshal(unsigned int ret, int a, int b) {
15   unsigned int index = 0;
16   seL4_SetMR(index, (seL4_Word)ret);  index++;
17   seL4_SetMR(index, (seL4_Word)a);    index++;
18   seL4_SetMR(index, (seL4_Word)b);    index++;
19   return index;
20 }
21
22 unsigned int ss_swap_internal(void) {
23   int *a = get_swap_a();
24   int *b = get_swap_b();
25   ss_swap_unmarshal(a, b);
26   unsigned int ret = ss_swap_invoke(a, b);
27   unsigned int length = ss_swap_marshal(ret, *a, *b);
28   return length;
29 }
```

Fig. 8. Generated stub for *s*

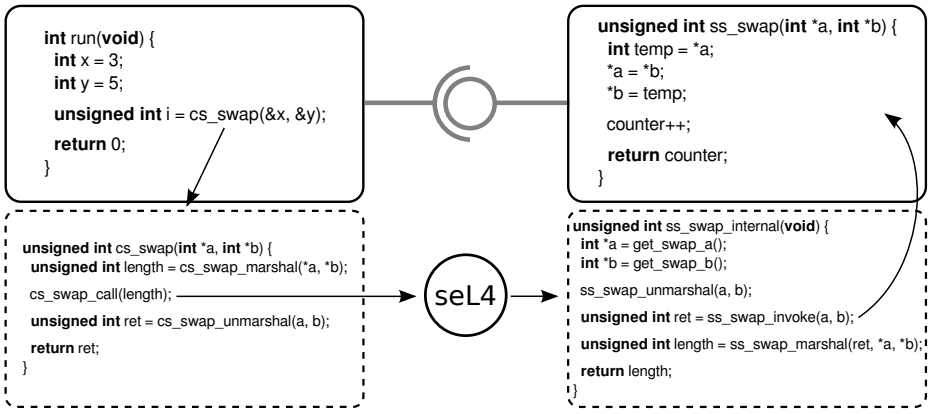

Fig. 9. Example system at runtime

```
1   {|λs. s = s0 ∧ ptr_valid_s32 s x ∧ ptr_valid_s32 s y|}
2    ss_swap x y
3   {|λr s. r = counter s0 + 1 ∧ counter s = r ∧
4         heap_w32 s (ptr_coerce x) = heap_w32 s0 (ptr_coerce y) ∧
5         heap_w32 s (ptr_coerce y) = heap_w32 s0 (ptr_coerce x)|}!
```

Fig. 10. A 'natural' correctness property if `ss_swap` were a local function

and the generated stubs (shown in dashed boxes). When c invokes `cs_swap`, which the user thinks of as an RPC to s, c's RPC stub runs and performs an seL4 system call. The seL4 kernel invokes the RPC stub of s which then calls the user's `ss_swap` function, making it seem as if this function call came directly from c. On return from `ss_swap`, the RPC stub returns the response via seL4 to c's RPC stub, which in turn delivers this to `cs_swap`. This control flow has the effect of allowing the user to design her system and reason about it in terms of abstract RPC operations, while CAmkES and seL4 implement the underlying communication mechanism.

4.3 Generated Proofs

The aim of the generated proofs is to guarantee the correctness of the RPC code produced by the component platform, intuitively showing that using the component platform gives the same assurance as running the function locally. The verification effort required from the proof engineer should therefore be comparable to the effort that would have been required to show the correctness of a local call.

For our example, Fig. 10 shows a 'natural' correctness property for `ss_swap` that might be used if it were just a local function. The pre-condition (line 1) requires that x and y are valid pointers. More precisely, the expression *ptr_valid_s32 st p* requires that the pointer p is a valid reference to a signed 32-bit value in state *st*.

```
1  swap a b ≡ do cs_swap_marshal (*a) (*b);
2              ss_swap_internal;
3              cs_swap_unmarshal a b
4           od
```

Fig. 11. A convenience abbreviation for the RPC invocation of `ss_swap`

```
1  {|λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s p0_out ∧ ptr_valid_s32 s p1_out ∧
2      distinct [p0_out, p1_out] ∧
3      (∀ s1 s2 v. Q_ss_swap s1 (update_s32 s2 p0_out v) = Q_ss_swap s1 s2) ∧
4      (∀ s1 s2 v. Q_ss_swap s1 (update_s32 s2 p1_out v) = Q_ss_swap s1 s2) ∧
5      P_ss_swap s p0 p1|}
6    do cs_swap_marshal p0 p1;
7      ss_swap_internal;
8      cs_swap_unmarshal p0_out p1_out
9    od
10 {|λr s. inv s ∧ Q_ss_swap s0 s r p0 (ucast (heap_w32 s (ptr_coerce p0_out)))
11                  p1 (ucast (heap_w32 s (ptr_coerce p1_out)))|}!
```

Fig. 12. Generated RPC stub equivalence lemma

The post-condition states that the value of the global *counter* will be updated and returned (line 3) and that the values at the pointers *x* and *y* will be swapped (lines 4-5). The function *ptr_coerce* is analogous to type casting in C and the expression *heap_w32 st p* returns the value pointed to by *p* in the state *st*. It is straightforward to prove this property using the existing translation tools and manual reasoning. It would also be straightforward to then use the resulting lemma in proofs for functions that call `ss_swap` directly.

In order to reason about execution in our example component-based system, we wish to claim that invoking *c*'s stub, `cs_swap`, is equivalent to invoking `ss_swap` directly. In this work, we are focussing on the generated glue code. This means we do not yet connect to the formal seL4 specification, but instead axiomatise the `seL4_Call` as a direct invocation of `ss_swap_internal`, which is the effect of this system call. We intend to replace this axiomatisation with the seL4 specification in future work as discussed in Sect. 5.2. For now, the resulting sequence of operations that we wish to claim is equivalent to `ss_swap` is shown in Fig. 11. The steps are: marshalling in `c`, then the entire execution in `s`, including unmarshalling, executing `ss_swap`, and marshalling results, and finally unmarshalling of the result in `c` again.

The pre-condition and post-condition of `ss_swap` that the generated proofs are parameterised with are referred to as P_{ss_swap} and Q_{ss_swap}, respectively. The pre-condition is a predicate over the initial state and values of the input arguments to `ss_swap`. The post-condition is a predicate over the initial state, final state, return value, input arguments and output arguments to `ss_swap`. The types of these parameters may seem unnecessarily verbose, but they provide the user with the flexibility to state any correctness property that is expressible of a direct invocation of `ss_swap`.

Fig. 12 shows the correctness statement that the generated proof for the RPC stub provides. Though this appears larger and more dense than that in Fig. 10, it is not much

more complicated. The expression *inv s*, present in both the pre- and post-conditions, captures the assumptions on user code mentioned in Sect. 3: in particular that user code does not violate the stub code's invariant which usually is easiest achieved by showing that it does not access glue code private state at all. The *ptr_valid_s32* pre-conditions (line 1) are familiar from the previous lemma. The *distinct* pre-condition (line 2) requires that the two pointers involved are not equal. While not strictly necessary in this case, the generated proofs conservatively require absence of aliasing between any user-provided pointers. This requirement, which is not an assumption of the implementation itself, is a convenience to ease the proof which we intend to remove in future.

The next two conjuncts (lines 3-4) state that the user's post-condition must not access the values of the pointer arguments through its final state parameter. This seems counter-intuitive, but we use it to allow internal stub code variables to substitute for the user's pointers when marshalling and unmarshalling arguments. That is, the user's post-condition can depend on the *values* of the arguments, but cannot depend on their *addresses*. The final conjunct (line 5) states that the user's pre-condition must hold prior to execution.

The post-condition (lines 10-11) is simpler, merely stating that the user's post-condition holds in addition to the glue code invariants. Here, *ucast* converts an unsigned 32-bit value to a signed 32-bit value.

For readability, we have omitted the proofs of the generated lemmas in this section, which themselves are also generated. However, the full CAmkES specification, user-provided code and framework for generating and validating the proofs we have described in this section are available online.[1]

4.4 User Instantiation

With the generated lemma from Fig. 12, all that remains is for the user to instantiate the locale with her specific pre- and post-condition and provide proofs for the locale assumptions, in particular the correctness of her `ss_swap` function. The natural pre-condition for this function is that from Fig. 10, but this is already subsumed by the generated pre-condition in Fig. 12. Therefore the user may instantiate her pre-condition to just $\lambda\ s\ a\ b.\ True$.

The natural post-condition, shown in Fig. 13, differs slightly from that of Fig. 10 as well. The first two conjuncts (line 1) are familiar from Fig. 10 and state the modification to, and return of, the global `counter`. The next (line 2) states that the initial state and the final state have identical sets of valid 32-bit pointers; that is, `ss_swap` does not invalidate any `int` pointers. This condition is relied upon by the generated proofs in assuming that internal pointers used for marshalling and unmarshalling are not invalidated by running user code. The final two conjuncts (line 3) are the equivalent of the final two from Fig. 10, though note that the user can now more conveniently express the property in terms of values, rather than pointer dereferences.

Having shown that this pre- and post-condition are wellformed and that the user-provided code is well behaved, the generated lemma is instantiated as shown in Fig. 14.

[1] https://github.com/seL4/camkes-manifest/tree/FM2015

```
1   Q_ss_swap s0 s r a b a_out b_out ≡ r = counter s0 + 1 ∧ counter s = r ∧
2                 is_valid_w32 s0 = is_valid_w32 s ∧
3                 a_out = b ∧ b_out = a
```

Fig. 13. User-instantiated post-condition

```
1   {|λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s p0_out ∧ ptr_valid_s32 s p1_out ∧
2       distinct [p0_out, p1_out]|}
3    do cs_swap_marshal p0 p1;
4      ss_swap_internal;
5      cs_swap_unmarshal p0_out p1_out
6    od
7   {|λr s. inv s ∧ r = counter s0 + 1 ∧ counter s = r ∧
8               is_valid_w32 s0 = is_valid_w32 s ∧
9               ucast (heap_w32 s (ptr_coerce p0_out)) = p1 ∧
10              ucast (heap_w32 s (ptr_coerce p1_out)) = p0|}!
```

Fig. 14. Instantiated generated correctness lemma

Note that this is simpler than the generic lemma of Fig. 12 because some of the pre-conditions can be automatically discharged by simplification. The idea is that this will always be the case for well-behaved functions.

To demonstrate that this lemma can be used in further hand-written proofs, we consider a sample property of the `ss_swap` function, that swapping two pointers twice returns the pointers to their original value. This property is shown in Fig. 15 and has a straightforward proof stemming from the lemma in Fig. 14. Again, as intended, this final lemma is much simpler than the intermediate generated forms, requiring only the stub code invariant, the user's properties of the pointer arguments and inequality of the two pointers.

5 Discussion

5.1 Trusting Generated Proofs

Having proven a system property by composing manual proofs with generated proofs, it is reasonable to ask what elements of the proof infrastructure need to be trusted. Isabelle/HOL is an LCF-style theorem prover [9], meaning that any proof within it relies only on the correctness of a small proof kernel. While we retain the assumption on the correctness of the Isabelle/HOL kernel, we would like to avoid requiring the user to trust additional tools.

We have not proven correctness of the code and proof generator itself. However, the proofs it produces are checked by Isabelle/HOL against the representation of the generated code presented by the C-to-Simpl translation. That means the proof generator does not need to be trusted, but the C-to-Simpl translation does. As mentioned in Sect. 2.3, there is separate work that reduces this trust even further and can be used to connect the Isabelle C semantics directly to binary code [24].

```
1  {|λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s x ∧ ptr_valid_s32 s y ∧ x ≠ y|}
2   do swap x y;
3     swap y x
4   od
5  {|λ_ s. inv s ∧
6        ucast (heap_w32 s (ptr_coerce y)) = ucast (heap_w32 s0 (ptr_coerce y)) ∧
7        ucast (heap_w32 s (ptr_coerce x)) = ucast (heap_w32 s0 (ptr_coerce x))|}!
```

Fig. 15. Applying generated proofs: swapping pointers twice returns them to their original state

A fully manual proof using the same C verification infrastructure would end up with the same level of trustworthiness in the resulting property. The automation we provide saves the user the time and effort on tedious proofs, without increasing her assumptions.

5.2 Assumptions, Limitations and Future Work

Our generated proofs have limitations that we intend to lift in future work. In this section we make these explicit and discuss how they may be removed.

CAmkES supports a wide range of data types for RPC parameters, including language independent types such as `int` and `string`, C-specific types such as `uint64_t` and more general arbitrary types that are represented by a `typedef` in C. Additionally, arrays of any type from these categories are supported. The generated proofs currently only handle RPC interfaces using C-specific integer types and language independent types excluding `string`. This limitation is driven by pragmatics and is not fundamental to the system design. A future iteration of the tool will support all CAmkES data types.

The semantics of the seL4 system calls that we use is currently implicitly assumed in the generated proofs. In particular, we assume that the IPC primitives transfer the sender's IPC buffer to the receiver. This is the case in seL4. This implicit assumption could be eliminated by connecting to the existing seL4 specifications for these system calls and composing them with the RPC stub proofs. This connection to seL4 would also solve the following two limitations.

The current structure of the generated proofs would permit heap accesses that cross component boundaries. This most closely models colocating two components in a single address space. While the generated proofs do not assume or rely on this property, the framework currently does not prevent the proof engineer from making use of it. A user-provided proof written for a context where a global variable of component A is accessible in component B would be unsound in the case where the components are isolated from each other. In other words, connecting the model to the full seL4 specification with a setup where address spaces are not shared, would fail. Future versions of the framework could enforce this separation of component heaps from the outset using separation logic [22] and thereby ensure that user-provided proofs will compose correctly in a final system instance.

As a final limitation, the execution model used in our proofs relies on the seL4 kernel to be configured correctly. In particular, the proofs in this work describe the

correctness of communication code in a CAmkES system. This communication is effected by operating on seL4 capabilities, unforgeable access tokens that are distributed to components on start up. Their presence is necessary to ensure the expected semantics of the system calls we assume. Furthermore, the *absence* of additional capabilities to component-private memory will guarantee isolation between components. An approach for removing this limitation would be to target the existing seL4 initialisation framework that has been verified to correctly configure userspace systems [3]. This proof can compose with our framework, but we do not yet show that the input CAmkES delivers to this initialiser implements the user's architecture description.

As far as we are aware none of these limitations are fundamental problems of the approach. The aim of this paper is to show the feasibility of automatically generating correctness proofs for glue code. The instantiation of these proofs to the seL4 execution environment can be achieved separately in future work.

6 Related Work

The proofs of generated code we have presented in this work are produced by the same tool [14] that generates the RPC stubs themselves. We do not rely on the correctness of the generator, or any implicit correspondence between the generated code and proofs as they are checked by Isabelle/HOL. In this sense, our approach is inspired by translation validation [20] and proof-carrying code [18].

Many verification frameworks have been proposed in the past for dealing with component-based systems, for example [1, 2, 8]. Our framework provides similar functionality. The work on which we report is not specifically aimed at increasing the ease of compositional reasoning. Instead, where our work differs is that we do not implicitly assume the correctness of generated RPC stubs, and instead provide an accompanying formal proof. To our knowledge, no current component-based verification framework provides such an automated code-level proof of generated platform code.

With respect to correct code generation and chained proofs, our work shares aspects with compiler verification. The CompCert verified compiler [15] and recent extensions to apply verification across translation units [21] have similarities, but work in a more controlled environment which enables more automated techniques. Our focus is on providing a compositional environment for interactive theorem proving that integrates with a larger interactive proof about the behaviour of the system, allowing the user a high degree of expressivity and control over the correctness properties they prove.

7 Conclusions

As the amount of code in high assurance systems increases, the only feasible approach to software verification is the application of compositional techniques. Existing frameworks for the verification of component-based systems all assume the correctness of the generated code of the component platform. In this work, we have demonstrated a technique for removing this assumption in the case of RPC stubs. We have shown how to compose generated RPC stub proofs with hand-written proofs to eventually yield a system-level correctness guarantee. By reducing the assumptions in component-based

reasoning, we increase the reach of formal verification and raise the bar for assurance of large software systems.

Acknowledgements

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S. Government.

Bibliography

[1] Adamek, J.: Static analysis of component systems using behavior protocols. In: OOPSLA, Anaheim, CA, US (Oct 2003) 116–117

[2] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. Softw. **28**(3) (2011) 41–48

[3] Boyton, A., Andronick, J., Bannister, C., Fernandez, M., Gao, X., Greenaway, D., Klein, G., Lewis, C., Sewell, T.: Formally verified system initialisation. In: 15th ICFEM, Queenstown, New Zealand (Oct 2013) 70–85

[4] Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: 9th TACAS, Warsaw, Poland (2003) 331–346

[5] Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: 21st TPHOLs, Montreal, Canada (Aug 2008) 167–182

[6] Fernandez, M., Kuz, I., Klein, G., Andronick, J.: Towards a verified component platform. In: PLOS, Farmington, PA, USA (Nov 2013) 6

[7] Fisler, K., Krishnamurthi, S.: Decomposing verification around end-user features. In: VSTTE 2005. (Oct 2005) 74–81

[8] Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Assumption generation for software component verification. In: 17th ASE, Edinburgh, UK (Sep 2002) 3–12

[9] Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF. Volume 78 of LNCS. (1979)

[10] Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: 3rd ITP. Volume 7406 of LNCS., Princeton, New Jersey (Aug 2012) 99–115

[11] Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: Formal verification of C code without the pain. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, UK (Jun 2014) 429–439

[12] ISO/IEC: Programming languages — C. Technical Report 9899:TC2, ISO/IEC JTC1/SC22/WG14 (May 2005)

[13] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: SOSP, Big Sky, MT, USA (Oct 2009) 207–220

[14] Kuz, I., Liu, Y., Gorton, I., Heiser, G.: CAmkES: A component model for secure microkernel-based embedded systems. Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems **80**(5) (May 2007) 687–699

[15] Leroy, X.: Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In: 33rd POPL, Charleston, SC, USA (2006) 42–54

[16] Leroy, X.: A formally verified compiler back-end. JAR **43**(4) (2009) 363–446

[17] Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: seL4: from general purpose to a proof of information flow

enforcement. In: IEEE Symp. Security & Privacy, San Francisco, CA (May 2013) 415–429
[18] Necula, G.C., Lee, P.: Safe kernel extensions without run-time checking. In: 2nd OSDI, Seattle, WA, US (Oct 1996) 229–243
[19] Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. (2002)
[20] Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: 4th TACAS, Lisbon, Portugal (Mar 1998) 151–166
[21] Ramananandro, T., Shao, Z., Weng, S.C., Koenig, J., Fu, Y.: A compositional semantics for verified separate compilation and linking. In: 4th CPP, Mumbai, India (Jan 2015) 3–14
[22] Reynolds, J.C.: Separation logic: A logic for mutable data structures, Copenhagen, Denmark (Jul 2002)
[23] Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
[24] Sewell, T., Myreen, M., Klein, G.: Translation validation for a verified OS kernel. In: PLDI, Seattle, Washington, USA (Jun 2013) 471–481
[25] Szyperski, C.: Component Software: Beyond Object-Oriented Programming, Essex, England (1997)
[26] Trustworthy Systems Team: seL4 v1.03, release 2014-08-10 (Aug 2014)
[27] Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: 34th POPL, Nice, France (Jan 2007) 97–108
[28] Wenzel, M.: The Isabelle/Isar Reference Manual. (Aug 2014)
[29] Winwood, S., Klein, G., Sewell, T., Andronick, J., Cock, D., Norrish, M.: Mind the gap: A verification framework for low-level C. In: 22nd TPHOLs. Volume 5674 of LNCS., Munich, Germany (Aug 2009) 500–515