Synthesis of Verified Architectural Components for Critical Systems Hosted on a Verified Microkernel

David Hardin
Collins Aerospace, USA
david.hardin@collins.com

Konrad Slind
Collins Aerospace, USA
konrad.slind@collins.com

Johannes Åman Pohjola
Data61/CSIRO, Australia
Johannes.Amanpohjola@data61.csiro.au

Michael Sproul
Data61/CSIRO, Australia
Michael.Sproul@data61.csiro.au

Abstract

We describe a method and tools for the creation of formally verified components that run on the verified seL4 microkernel. This synthesis and verification environment provides a basis to create safe and secure critical systems. The mathematically proved space and time separation properties of seL4 are particularly well-suited for the miniaturised electronics of smaller, lower-cost Unmanned Aerial Vehicles (UAVs), as multiple, independent UAV applications can be hosted on a single CPU with high assurance. We illustrate our method and tools with an example that implements security-improving transformations on system architectures captured in the Architecture Analysis and Design Language (AADL). We show how input validation filter components can be synthesised from regular expressions, and verified to meet arithmetic constraints extracted from the AADL model. Such filters comprise efficient guards on messages to/from the autonomous system. The correctness proofs for filters are automatically lifted to proofs of the corresponding properties on the lazy streams that model the communications of the generated seL4 threads. Finally, we guarantee that the intent of the autonomy application logic is accurately reflected in the application binary code hosted on seL4 through the use of the verified CakeML compiler.

1. Introduction

The creation of accreditable, safey- and security-critical autonomous systems will demand the highest degree of design assurance.[1] We are thus researching highly automated synthesis techniques for high-assurance components implementing formally verified security-enhancing architectural transformations for critical systems. In particular, we provide evidence, in the form of formal proofs, that these transformations actually do improve key security, as well as safety, properties. This work is part of the Defense Advanced Research Projects Agency (DARPA) Cyber-Assured Systems Engineering (CASE) research program, which is tasked with providing tools and techniques that allow system designers to proactively account for resiliency against cyber-attacks during the design phase, rather than by discovering and patching these flaws in an already implemented system in *ad hoc* fashion.[2]

By way of a motivating example, consider the (simplified) model of a mission control system for an unmanned aerial vehicle (UAV) shown in in Figure 1. This model is based on the UxAS UAV developed at the U.S. Air Force Research Laboratory [1], and uses legacy components from that system. The two main components of the system are a ground station (GS) and the UAV, which has as subcomponents a radio (RADIO), a flight planner (FPLN), a waypoint manager (WPM), and a flight control computer (FCC). General mission parameters flow from GS to FPLN, by way of the radio link. The flight planner generates the full flight plan—a sequence of waypoints to follow—and sends it to the waypoint manager. The WPM processes a fixed-size window of the next few waypoints to be dealt with by the flight control computer. The FCC is a separate computer which is in charge of actually flying the vehicle, interpreting the waypoints and incoming sensor data and sending directives to control motors, etc. As the FCC makes progress, it tells the WPM to advance the window, and also sends back various sensor data to the FPLN, WPM, and potentially the GS via the RADIO.

A first step towards securing such a system is *spatial isolation*: when an untrusted legacy component is compromised or otherwise malfunctions, we do not want it to be able to interfere with the correct execution of other components, or to read privileged data from other components through unintended side channels. To achieve this, we want the connections between

[1]DISTRIBUTION STATEMENT A. Approved for Public Release.

[2]The views expressed are those of the authors and do not reflect the official policy or position of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

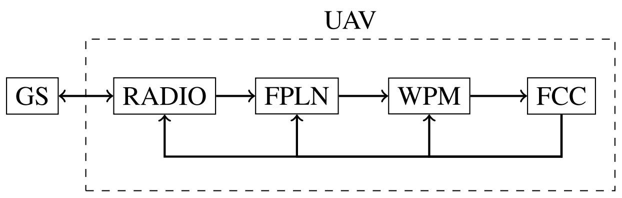

Figure 1. Simplified model of a UAV flight controller.

the components that are explicitly present in design diagrams such as Figure 1 to be the only communication channels present in the running system. While such isolation can be achieved by running each component on its own physically isolated hardware, we reduce cost, gain flexibility, and enable miniaturization by using the same general-purpose hardware to run several distinct components. In order to recover spatial isolation in this setting, we run components on seL4 [2], a verified capability-based microkernel accompanied by formal proof of spatial isolation properties down to the binary level.

While isolation defends against attacks through *unintended* channels, it does nothing to guard against attacks through the *intended* channels. For example, a compromised ground station or radio driver could be used to feed malformed messages that the legacy flight planner is not equipped to handle. This could be exploited to induce crashes or privilege escalation through buffer overflow, etc.

Our goal in the present paper is that the system designer, having noticed the potential for such a vulnerability, should have at her disposal a toolbox of *security-enhancing architectural transformations* for inserting countermeasures into the architecture. These countermeasures should be non-intrusive, reusable and highly trustworthy. They should not change the (potentially brittle) legacy components — instead, they place security enhancements *around* them. They should be easy to configure and deploy for the needs of different systems. Trustworthiness, the main focus of this paper, is very important: when we introduce more components into a system, they too become part of the attack surface. Thus, each component inserted by a transformation is accompanied by a formal proof connecting the component's intended behaviour with the behaviour of its implementation down to the binary level. The transformations in our toolbox include isolation (realised, for example, by the introduction of the verified seL4 separation kernel), filter insertion, data transformer insertion, safety monitor insertion, and measurement/attestation.

The concrete transformation we consider in this paper is *filter insertion*, which is a pattern to prevent attacks that rely on malformed messages. This transformation is illustrated on the UAV mission control example in Figure 2. The filter controls the flow of messages on the communication channel, ensuring that only messages that are well-formed get forwarded from the radio driver to the flight controller, and conversely, that every well-formed message gets forwarded. The precise definition of "well-formed" will of course vary from situation to situation; hence, it is a parameter that the system designer can instantiate. In this paper, we consider filters where well-formedness can be decided by regular expressions. Our toolchain will then:

1. compile the regular expression to a table-driven deterministic finite-state automaton (DFA) by instantiating a general regexp compiler theorem provided in HOL4;
2. synthesise CakeML code that implements matching of strings against the DFA from the preceding step using proof-producing synthesis [3];
3. generate an I/O wrapper around the synthesised code that allows it to be inserted on any Remote Procedure Call (RPC) connection between two components of the CAmkES [4] component architecture for seL4;
4. produce a proof, covering both liveness and safety, that this program implements the expected behaviour of a filter component: it lets through exactly the messages described by the regular expression; and
5. generate a binary filter implementation using the verified CakeML compiler.

These steps are carried out in the HOL4 theorem prover, which we integrate into the CAmkES build system: thus, strong evidence for the trustworthiness of transformations are built along with the system image wherein they are used.

Since regular expressions tend to be somewhat opaque, it is also important to have some independent assurance that a given regular expression really does correspond to the intended meaning of "well-formed". To address this, we require proofs of the formal relationship between a given regular expression and a high-level logic specification extracted from the Architecture Analysis and Design Language (AADL) model of the UAV system.

Notation This paper uses notation that closely follows that of the underlying HOL4 formalisation, with some modifications for readability. We use this font for HOL4

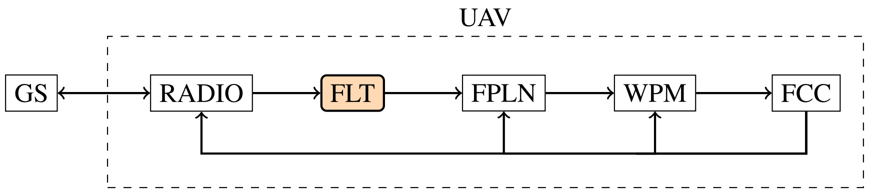

Figure 2. Simplified model of a UAV flight controller with inserted filter.

constant names and type constructors, *this font* for (free or bound) HOL4 variables, and `this font` for HOL4 types and CakeML code. We write deeply embedded CakeML code in concrete syntax for readability.

We write $f\ \ x$ or $f(x)$ for the application of the function f to the argument x. T and F denote truth and falsity. SOME and NONE are the type constructors of the `option` type, denoting respectively a value or the absence of a value. The function the extracts the value of an `option` that denotes a value, so the(SOME x) $= x$.

We overload [] to denote both the empty (inductive) list and the empty (coinductive) lazy list; similarly $\frown$, written in infix, denotes concatenation for both lists and lazy lists. Finite list "cons" is written infix (::). length, map, filter, take, zip, flatten and drop denote the usual library functions on lists; when prefixed by an l as in llength, they denote the corresponding functions on lazy lists. llength returns `num option` rather than `num`: llength $l =$ NONE means that l is infinitely long. replicate $n\ e$ denotes a list consisting of n copies of e. We write l_n for the n:th element of the list l (if it exists), and lgenlist f for the list $[f(0), f(1), f(2), \ldots]$. Equality on lazy lists coincides with list bisimilarity in HOL4, so $l_1 = l_2$ is true iff no difference between l_1 and l_2 can be observed by repeated application of the destructors lhd (lazy list head) and ltl (lazy list tail).

2. Architectural modelling and transformation

AADL [5] is an industrial system architecture description language. We have been using AADL on projects aimed at performing analysis and verification of system-level designs for cyber-physical systems, such as UAVs [6]. AADL provides the constructs needed to model embedded systems such as processes, threads, processors, devices, buses, and memory. An AADL model captures the whole system and can thus can serve as a central place for system-level analysis, reasoning, scheduling, generation of executables, etc. We are investigating architecture-to-architecture transformations that provably improve the security of a system. The Collins team has built AADL-based reasoning tools [7] [8] in previous work, and we are currently engaged in enhancing these tools with the technology developed in the present paper.

2.1. AGREE contract checking and the Resolute assurance case tool

The AGREE tool [7] supports component-based hierarchical reasoning based on assume-guarantee behavioural specifications (contracts) placed on elements of an AADL architecture. AGREE contracts are expressed with past-time LTL formulas, and model-checking is accomplished using the JKind k-induction model-checker [9]. For our class of transformations, in which a new filter component is added to meet a security requirement, we wish to show that the filter indeed provides the specified well-formedness check on its inputs. This has twin aspects: first, the "empty" filter component is added to the architecture, and well-formedness of messages is expressed as an AGREE assertion on its output. The assertion becomes a leaf-level statement in the reasoning performed by the AGREE tool; thus, it must be proved separately. Second, the empty component is filled in: a regular expression is given and the corresponding DFA-based implementation is generated and compiled. The well-formedness property becomes a specification on the generated implementation, and the specification is (automatically) propagated to the infinitary behaviour of the filter component, by means of the proofs in Section 4.2. The creation of the filter component creates multiple pieces of verification evidence from the AGREE and HOL4 tools. Managing and combining the results of multiple reasoners justifying architectural transformations is the subject of current extensions we are making to Resolute, an assurance case tool with deep connections to AADL models [8].

2.2. Example: UAV Coordinates

Consider an AADL model corresponding to the filter-transformed architecture depicted informally in Figure 2, a portion of which is shown in Figure 3. One of the data elements to be transferred between the Ground Station and the UAV are Location coordinates. By way of example, consider a location coordinate represented by the following record structure in AADL:

```
data implementation Coordinate.Impl
  subcomponents
    latitude  : data Base_Types::Integer;
    longitude : data Base_Types::Integer;
    altitude  : data Base_Types::Integer;
end Coordinate.Impl;
```

We only wish to accept well-formed coordinates from the ground, as ill-formed coordinates could be used by a cyber attacker to inject malware, cause computational errors, direct the UAV into hostile territory, etc. We thus define a set of coordinate constraints in the AGREE contract language; one such set of constraints is presented in conventional mathematical notation as follows:

$$\begin{array}{lcl} \mathsf{wf\_coordinate}(c) & \Leftrightarrow & -90 \leq c.latitude \leq 90 \wedge \\ & & -180 \leq c.longitude \leq 180 \wedge \\ & & 0 \leq c.altitude \leq 15000 \end{array}$$

Thus the assertion to be added on the output out of the filter component is $\mathsf{wf\_coordinate}\ out$. Note that such high-level specification of messages ignores important aspects of the wire format such as field order, endianess, and packing. Our toolchain provides support for such "metadata", which we exploit in order to obtain declarative specifications of message representations.

3. CakeML on seL4

In this section, we describe our setup for building CakeML programs that can run as components on seL4. The impact is a low-cost way of building highly trustworthy systems: we run components verified to the binary level on a microkernel verified to the binary level.

seL4 is a capability-based microkernel which aims to be both performant and secure. It is accompanied by a formal specification in Isabelle/HOL and an end-to-end proof that its C implementation and compiled binary refine this specification [2, 10, 11]. High-level confidentiality and information flow properties are proved over the abstract specification, which then apply to the binary by refinement.

As a general-purpose microkernel, seL4 provides only basic facilities: threads, virtual memory and messaging-passing inter-process communication. To build useful systems atop seL4 therefore requires significant design and implementation work, which should ideally be accompanied by further formal verification to maintain confidence in the correctness of the whole system. The seL4 ecosystem includes a component system called CAmkES [4] for building systems, which until now has primarily supported components written in C, for which formal verification is costly. This work describes the first implementation of a CAmkES component in a high-level language for which a complete formal semantics is available, namely CakeML.

CakeML is a a dialect of Standard ML with a verified compiler, implemented and verified in the HOL4 theorem prover. CakeML programs written in concrete syntax or generated via translation from HOL4 [3] can be compiled in a semantics-preserving way to machine code for a variety of architectures, including ARMv6, ARMv8 and x86-64.

Figure 4 is an excerpt of the filter's CakeML code, which implements the interaction with seL4 as an event loop. In this loop, the `#(accept_call)` and `#(emit_string)` calls are *Foreign Function Interface* (FFI) function calls which act to receive and send data via seL4's RPC mechanism, implemented in C. Intuitively, `#(foo) s b` calls the foreign function `foo` with the string argument `s` and a byte array `b` on which data that should be passed back to CakeML can be written (`#(emit_string)` does not return anything and is therefore called with a zero-length array). The filter converts each array of bytes received into a null-terminated string, passes it to the verified regular expression matcher, and if it matches, sends it out via `#(emit_string)`. In our example system, there is a CAmkES component called the producer which sends strings to the filter, and another component called the consumer which receives the matching strings.

The filter program is written as deeply embedded CakeML code inside the HOL4 prover. HOL4 is invoked by the CAmkES build system to construct the filter and its accompanying correctness proofs (described in Section 4.2) alongside the seL4 system image on which it runs. To extract executable code from the HOL4 deep embedding, a number of options are available. Our current version of the tool pretty-prints the deeply embedded program as an S-expression, which we then compile outside the logic using a bootstrapped version of the verified CakeML compiler. If stronger assurance is desired, it is possible to do in-logic evaluation of the CakeML compiler, then print the resulting bytes to a file. We currently don't do this because it leads to much longer build times (measured in hours rather than seconds).

4. Filters specified by regular expressions

In this section, we describe the formal justification underlying our filter transformations. In earlier work [12] we reported on a verified compiler from extended regular expressions to table-driven DFAs, using Brzozowski's "derivative" approach; this compiler is available in the

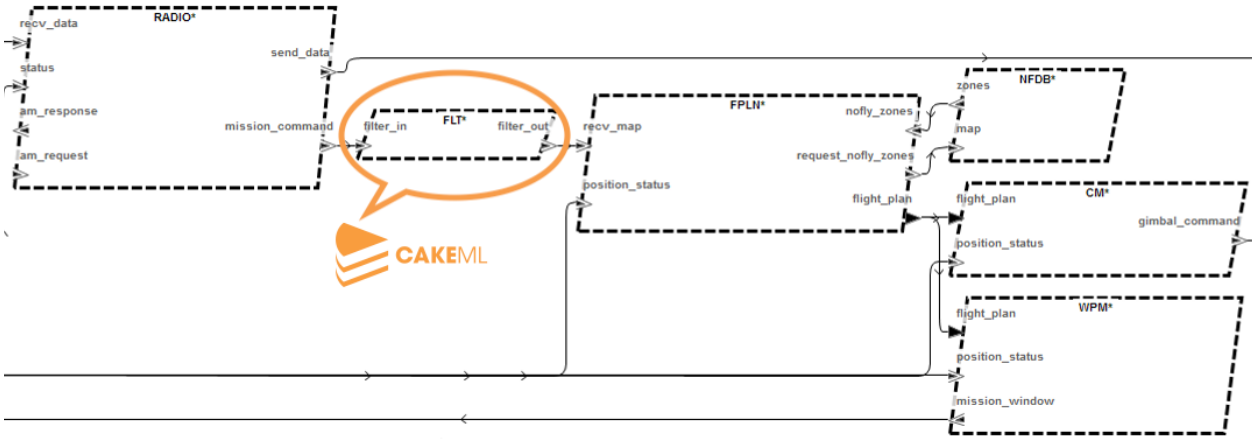

Figure 3. Partial UAV Software Architecture in AADL, with inserted CakeML filter.

```
fun forward_loop() =
 (#(accept_call) "" buffer;
  let
   val s1 = Word8Array.substring
              inputarr 0 256;
   val s2 = cut_at_null ln;
  in
   if match_string s2 then
    #(emit_string) s2 dummy
   else ()
  end;
  forward_loop())
```

```
fun forward_matching_lines() =
 let
  val buffer =
   Word8Array.array
    256 (Word8.fromInt 0);
  val dummy =
   Word8Array.array
    0 (Word8.fromInt 0);
 in
  forward_loop()
 end
```

Figure 4. An excerpt of the filter's CakeML code.

HOL4 distribution.[3] We make use of this compiler to create verified regexp-based filters. We have built a translation of AGREE arithmetical constraints to regexps and provide proof tools verifying semantical properties of the generated regexps. Final steps generate and verify CakeML code implementing the CAmkES regular expression filter component.

4.1. Semantic properties of formal languages

In our setting, a filter *specification* takes the form of logical constraints on a record type (supplemented with layout and well-formedness information) modelling network messages. The filter *implementation* is then generated by translating the logical constraints to an equivalent regular expression. This gives rise to a class of verification problems: namely, when does a regular expression r exactly capture well-formedness constraints on the fields of a record structure. We will write $r \models \mathsf{SPEC}$ for the following relation between regexp r and logic specification $\mathsf{SPEC} : \mathsf{recd} \rightarrow \mathsf{bool}$.

$$\forall recd.\ \mathsf{SPEC}(recd) \Leftrightarrow \mathsf{encode}(recd) \in \mathcal{L}(r) \qquad (1)$$

where encode maps elements of the given record type to strings, and $\mathcal{L}(r)$ is the regular language generated by regexp r. We call this setting SPLAT (*Semantic Properties for Language and Automata Theory*), since it combines formal language theory with specifications on the interpretation of the flat strings comprising message formats.

Theorems having the form of (1) bridge between the requirements needed for AGREE architectural reasoning and the correctness of regexp compilation. However, there are related theorems that are also required, e.g., invertibility and injectivity of encoding:

$$\begin{array}{l} \forall x.\ \mathsf{decode}(\mathsf{encode}\ x) = x \\ \forall x\ y.\ (\mathsf{encode}\ x = \mathsf{encode}\ y) \Rightarrow x = y \end{array} \qquad (2)$$

These can be required in the proof of (1), and are further evidence that the filter will work properly. Tan and Morrisett [13] explore some of the subtleties in formal proofs of Instruction Set Architecture (ISA)-level encoders/decoders.

Example 1. *Consider the well-formed coordinate filtering example of Example 2.2. The theorem proved is*

$$\mathcal{R} \models \mathsf{wf\_coordinate},$$

or

$$\forall r.\ \mathsf{wf\_coordinate}(r) \Leftrightarrow \mathsf{encode}(r) \in \mathcal{L}(\mathcal{R})$$

where

$$\begin{array}{l} \mathsf{encode}(r) = \mathsf{enc}\ 1\ r.latitude \frown \mathsf{enc}\ 2\ r.longitude \frown \\ \qquad\qquad \mathsf{enc}\ 2\ r.altitude \end{array}$$

This theorem is proved automatically using the HOL4-based SPLAT infrastructure. SPLAT utilizes a suite of pre-proved rewrite rules, including invertibility of the component encoders, as well as specialised provers for membership in the appropriate character sets.

Expressiveness We think that specifying filters with regexps is a promising direction, combining a declarative approach to behaviour with a strong proof basis in formal language theory. Many common message formats are naturally expressed with regular expressions extended with intervals. Messages with fixed repetitions of data elements, e.g., arrays, can be expressed easily. Also, messages with *indeterminate* repetitions of data elements can be expressed with Kleene star. There are, of course, limitations: regexps are not able to handle data where wellformedness is a context-free (or beyond) language. For example, determining the well-formedness of JSON or XML-formatted messages would be out of scope, as would message formats where the wellformedness of one field depends on a value in another field. We are currently developing a verified lexing/parsing capability to extend our reach to more complex message formats.

4.2. CakeML filter verification

Next we extend the correctness theorem about the regular expression compiler to the I/O behaviour of the filter component's CakeML implementation. The theorem can be composed with the CakeML compiler correctness theorem to obtain a corresponding theorem about the binary code that runs in the filter component. Taken together, these theorems imply a number of desirable properties:

> *Safety:* The filter's I/O behaviour is consistent with its specification, which is to forward to the consumer precisely those messages from the producer that match the regular expression.
>
> *Liveness:* The filter will keep executing indefinitely; it does not crash or otherwise terminate, given that seL4 continues to provide the necessary execution environment. As long as the producer keeps sending messages to the filter, the filter will always eventually process the next message. The filter cannot get stuck in an infinite silent loop, e.g. while processing any one message.

[3] In `examples/formal-languages/regular`.

In order to state the theorems with more precision, we need to recapitulate the semantics of two pertinent language features: divergence and foreign function calls. We will take the liberty of simplifying or eliding technical details that do not pertain to the matters at hand; those interested in the details are referred to the HOL4 sources for CakeML.[4]

Semantics of divergence and FFI calls The CakeML semantics is defined in two layers, which we shall call the *evaluator* and the *top-level semantics*. The evaluator is defined using *functional big-step semantics* [14] — or in other words, it is a function $\mathsf{eval} \in state \times program \Rightarrow state \times result$ which defines the (unique) outcome of executing a program from a given initial state. eval is recursively defined in the style of an interpreter, as is common in these sorts of formalisations.

In order to make sure that eval terminates even if the program does not, the state contains a semantic clock, which is a natural number that gets decremented whenever eval visits a program construct that might induce non-termination, such as recursive function calls. When the clock runs out, eval immediately returns with the result Timeout. As we shall see, a divergent program is one that times out for every clock value.

CakeML programs that communicate with the outside world via the FFI have semantics that are parameterised by a *foreign function oracle*, which is a part of the program state that models the outside world. The oracle describes the effects of foreign function calls in terms of their impact on the outside world, and what data they pass from the outside world to CakeML.

Henceforth, we will write $st_{k,o,t}$ for a state st with clock value k, FFI-oracle o and a list t of the foreign function calls that the program has emitted so far. We elide the clock value when it is irrelevant. We also write $O(st)$ to denote this list of foreign function calls, so $O(st_{k,o,t}) = k$. We lift O to $state \times result$ pairs as $O((s,r)) = O(s)$.

Finally, the top-level semantics of a program is defined as follows:

```
semantics($st_{o,[]}, prog$) =
  Terminate if $\exists k, st', v.\mathsf{eval}(st_{k,o,[]}, prog) =$
    $(st', \mathsf{Value}\ v)$ and $O(st') = t$
  Diverge($t$) if $\forall k.\ \exists st'.\mathsf{eval}(st_{k,o,t}) =$
    $(st', \mathsf{Timeout})$ and
      $t = \sup\{O(\mathsf{eval}(st_{k,o,t}, prog)) : k \in \mathbb{N}\}$
  Failure otherwise
```

Note that the observable behaviour of a divergent program is the supremum (ordered by list length) of the program's FFI traces for every possible clock value. This supremum might be an infinite, and hence coinductive, list.

Verification CakeML-level verification was carried out using the CF (characteristic formulae) program logic [15], which supports sound reasoning about impure CakeML programs through an abstraction that models programs as higher-order logic formulae relating preconditions to postconditions. Unlike other tools to ease the verification of CakeML code [3, 16], all of which are restricted to terminating programs, CF was recently extended to accommodate liveness proofs for diverging programs [17]. This greatly eases verification because we are spared many low-level details considered in CakeML's big-step semantics; we need never explicitly massage low-level details such as namespace lookups and semantic clocks. Because CF is sound wrt. the CakeML semantics presented in the previous section, its use does not increase the trusted computing base. The resulting HOL4 proof script, covering the toolchain from invocation of the regexp compiler to generation, verification and compilation of the resulting CakeML implementation, is around 1500 lines of definitions and proofs and is available online. [5] All theorem statements that follow are formally proven therein.

To characterise the expected I/O behaviour of the filter in a way that does not refer back to the filter implementation, we define the function next_events, which describes the I/O events produced by one iteration of the filter loop.

```
next_events $f\ (i, l) =$
let $h = \mathsf{take}\ 256\ i \frown \mathsf{drop}\ (\mathsf{length}\ i)\ l$;
in
  [IO_event "$accept\_call$" [] $(zip\ (l, h))$] $\frown$
  if $f(\mathsf{cut\_at\_null}(i))$ then
    [IO_event "$emit\_string$" (cut_at_null $h$) []]
  else[]
```

We parameterise next_events on an arbitrary *filter predicate* f. Later, we will instantiate f to be the membership predicate for a regular language, but it is worth stressing that our approach is applicable to filters that are not regexp-based. We abstract the internal state as a tuple (i, l), where i is the next input to be consumed, and l is the current contents of the message buffer.

A specification of the filter's limit behaviour can then be obtained by the (possibly infinite) concatenation of the next_events at every loop iteration.

This gives a coinductive list that describes the sequence of observable events that the filter program will

[4] `http://code.cakeml.org/`

[5] At `code.cakeml.org`, in `examples/filterProgScript`. The version available there presents only the filter correctness proofs, and has been decoupled from the build system presented in this paper.

reach throughout its execution. This is a very low-level specification: apart from being a HOL4 function rather than a CakeML program, it is at about the same level of abstraction as the implementation itself. Its main use is to construct a bisimulation witness in the proof of the following theorem:

Theorem 1. *If every message in o.*input *is null-terminated and at most 256 bytes long (including the null terminator), then there is a trace t such that*

$$\begin{aligned}&\mathsf{semantics}(st_{o,[]}, \texttt{forward\_matching\_lines}()) = \\ &\quad \mathsf{Diverge}(t)\end{aligned}$$

and

$$\begin{aligned}&\mathsf{lfilter}\ \mathit{is\_emit}\ (t) = \\ &\mathsf{lmap}\ \mathsf{cut\_at\_null}(\mathsf{lfilter}\ (\mathcal{L}(r) \circ \mathsf{cut\_at\_null})\ o.\mathsf{inputs})\end{aligned}$$

where r is the regular expression under consideration for this particular filter.

Proof. The proof is by Hoare logic-style reasoning using the CF proof rule for divergence [17].

In the inductive case, we prove by standard weakest precondition-style reasoning that the n:th iteration always terminates, producing as I/O the n:th next_events.

In the limit case, we prove that the following is a list bisimulation:

$$\begin{aligned}&\{(l_1, l_2) : \exists i, l. \\ &\quad l_1 = \mathsf{lfilter\ is\_emit}(\mathsf{lflatten}(\mathsf{lgenlist} \\ &\qquad\qquad (\lambda n. \mathsf{next\_events} \\ &\qquad\qquad\qquad (\mathcal{L}(r) \circ \mathsf{cut\_at\_null}) \\ &\qquad\qquad\qquad (i_n, l_n) \\ &\qquad\qquad) \\ &\qquad)) \\ &\quad \wedge\ l_2 = \mathsf{lmap\ cut\_at\_null}(\mathsf{lfilter}\ (\mathcal{L}(r) \circ \mathsf{cut\_at\_null})\ l) \\ &\quad \wedge\ \mathsf{every} \\ &\qquad (\lambda l.\ \mathsf{length}\ l \leq 256 \wedge \mathsf{null\_terminated}\ l)\ i, l\}\end{aligned}$$

□

This shows that the behaviour our filter, which can be explicitly characterised by filter_events, corresponds to our intuition about filter operation — namely, that the list of output events is equal to the list of input events filtered by the regular expression's language. This final characterisation makes no reference to the filter's internal state.

These "good filter" proof results are recorded as evidence in the Resolute Assurance case tool for a given filter instance.

5. Related Work

In [18], Bohrer et al. study the synthesis of verified control monitors for cyber-physical systems that ultimately are implemented by synthesised CakeML code. Control monitors are similar to filters in that they are architectural components that are not meant to terminate. While we explicitly consider the limit behaviour of such non-terminating executions, Bohrer et al. avoid having to formally treat divergence by introducing a *stop oracle* which decides when the control loop should stop executing. Thus their verification is restricted to safety properties, whereas ours also addresses liveness.

Férée et al. [19] use CakeML to develop a verified implementation of the `grep` command-line tool, building on an earlier version of the regular expression compiler presented in this paper. While we perform DFA compilation at compile-time, `grep` requires the DFA to be generated at runtime based on the command-line arguments. Hence Férée et al.'s correctness theorem for `grep` must assume the unproven conjecture that Brzozowski derivation with smart constructors terminates — hence, the verification is limited to partial correctness of a (hopefully) terminating program. Since we perform this derivation at compile-time, our liveness property is not predicated on any such termination assumption.

The CompCert verified C compiler proofs have been extended to cover semantics preservation for divergent programs [20]; however, the only instance we are aware of where this capacity has been used is in the context of a compiler for the dataflow language Lustre by Bourke et al. [21], which uses CompCert as a backend. The compiler correctness theorem preserves guarantees similar to ours: every node in a Lustre program has semantics that characterise the infinite stream of the inputs it receives and the outputs it emits, and the paper establishes that the generated assembly code exhibits a bisimilar trace of read/write operations to volatile memory. Our verified filters are similar to the kinds of signal processing nodes considered in dataflow programming, but do not neatly fit the setting: while Lustre nodes produce an output for every input, our filters produce output only for well-formed inputs, and drop all other inputs.

The idea of relating arithmetic constraints to regular expressions and automata goes back at least to Büchi [22] and has been robustly mechanised in Mona [23]. It will be interesting to see if implementation techniques from that work can be used in ours. Recent papers by D'Antoni [24] and Suriyakarn et al. [25] focus on correctness and synthesis of decode/encode function pairs, but do not address the use of such building blocks in verifying higher-level properties.

6. Conclusion and Future Work

We have detailed a method and toolchain for the creation of formally verified security-enhancing autonomy components that run on a verified microkernel. We demonstrated how this toolchain can be used to implement security-improving transformations on system architectures specified in AADL, with implementations automatically synthesised using the verified CakeML compiler. Such transformations, e.g. input validation filters based on regular expressions, can be used to improve resiliency against cyber attack. We have proved safety and liveness properties that connect the filter's regular expression specification to the I/O behaviour of the running filter component in the context of the verified seL4 microkernel. Moreover, we have seen how SPLAT proofs connect the regular expression to higher-level designer-specified properties, captured in AADL.

This paper presents several directions for future work, many of which we are currently pursuing. One research thrust is to make verification of liveness properties for non-terminating CakeML programs more scalable. The CakeML compiler correctness theorem can transport our theorems about the filter implementation to the binary level, with one important caveat: it preserves semantics up to the fact that the binary is allowed to abort at any time if it runs out of memory. Hence liveness becomes "liveness unless we run out of memory". The filter *should not* run out of memory — the live memory does not grow between loop iterations — but it would be good to be able to state and prove unconditional liveness for the binary. The missing puzzle piece for this is a space-cost semantics for CakeML and a proof that the compiler preserves space-cost.

We will be using the approach described in this paper to support other verified architectural transformations. We are currently pursuing the synthesis of more complex filter components utilizing verified lexing and/or parsing, data transformation components, as well as verified safety monitors. Finally this paper does not address the entirety of the grand challenge of whole-system verification; that is, we do not as yet formally connect the components' correctness proofs with the seL4 correctness proofs. One major verification gap in our toolchain is that between the AADL architecture and its corresponding expression in CAmkES. In previous work [26] a tool known as *Trusted Build* handled this transformation; however, this was an informally developed tool with no explicit proof story. As part of current research, we are working to close this gap.

Acknowledgments

We thank the anonymous referees for their insightful comments. This work was sponsored in part by the U.S. Defense Advanced Research Projects Agency (DARPA).

References

[1] S. Rasmussen, D. Kingston, and L. Humphrey, "A brief introduction to unmanned systems autonomy services (UxAS)," in *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 257–268, June 2018.

[2] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an operating-system kernel," *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, 2010.

[3] M. Myreen and S. Owens, "Proof-producing translation of higher-order logic into pure and stateful ML," *Journal of Functional Programming*, vol. 24, no. 2–3, pp. 284–315, 2014.

[4] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "CAmkES: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2007.

[5] P. Feiler and D. Gluch, *Model-based engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language*. Addison-Wesley, 2012.

[6] G. Klein, J. Andronick, M. Fernandez, I. Kuz, T. Murray, and G. Heiser, "Formally verified software in the real world," *Communications of the ACM*, vol. 61, pp. 68–77, October 2018.

[7] D. Cofer, A. Gacek, S. Miller, M. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *Fourth NASA Formal Methods Symposium (NFM 2012)*, pp. 126–140, 2012.

[8] A. Gacek, J. Backes, D. Cofer, K. Slind, and M. Whalen, "Resolute: an assurance case language for architecture models," in *HILT 2014*, pp. 19–28, ACM, 2014.

[9] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani, "The JKind model checker," in *CAV 2018*, pp. 20–27, 2018.

[10] T. Sewell, M. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *PLDI 2013*, pp. 471–482, ACM, 2013.

[11] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: From general purpose to a proof of information flow enforcement," in *2013 IEEE Symposium on Security and Privacy*, pp. 415–429, IEEE, 2013.

[12] D. Hardin, K. Slind, M. Bortz, J. Potts, and S. Owens, "A high-assurance, high-performance hardware-based cross-domain system," in *SAFECOMP 2016*, vol. 9922 of *LNCS*, pp. 102–113, Springer, 2016.

[13] G. Tan and G. Morrisett, "Bidirectional grammars for machine-code decoding and encoding," *Journal of Automated Reasoning*, vol. 60, no. 3, pp. 257–277, 2018.

[14] S. Owens, M. Myreen, R. Kumar, and Y. Tan, "Functional big-step semantics," in *ESOP 2016*, vol. 9632 of *LNCS*, pp. 589–615, Springer, 2016.

[15] A. Guéneau, M. Myreen, R. Kumar, and M. Norrish, “Verified characteristic formulae for CakeML,” in *ESOP 2017*, pp. 584–610, 2017.

[16] S. Ho, O. Abrahamsson, R. Kumar, M. Myreen, Y. Tan, and M. Norrish, “Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions,” in *IJCAR 2018*, vol. 10900 of *LNCS*, pp. 646–662, Springer, 2018.

[17] J. Åman Pohjola, H. Rostedt, and M. O. Myreen, “Characteristic formulae for liveness properties of non-terminating CakeML programs,” in *Proceedings of the 10th International Conference on Interactive Theorem Proving ITP* (J. Harrison, J. O’Leary, and A. Tolmach, eds.), vol. 141, pp. 32:1–32:19, LIPIcs, 2019.

[18] B. Bohrer, Y. K. Tan, S. Mitsch, M. Myreen, and A. Platzer, “Veriphy: verified controller executables from verified cyber-physical system models,” in *PLDI 2018*, pp. 617–630, ACM, 2018.

[19] H. Férée, J. Åman Pohjola, R. Kumar, S. Owens, M. Myreen, and S. Ho, “Program verification in the presence of I/O: Semantics, verified library routines, and verified applications,” in *VSTTE 2018*, pp. 88–111, Springer, 2018.

[20] X. Leroy and H. Grall, “Coinductive big-step operational semantics,” *Information and Computation*, vol. 207, no. 2, pp. 284–304, 2009.

[21] T. Bourke, L. Brun, P. Dagand, X. Leroy, M. Pouzet, and L. Rieg, “A formally verified compiler for Lustre,” in *PLDI 2017*, pp. 586–601, 2017.

[22] J. R. Büchi, “Weak second-order arithmetic and finite automata,” *Zeitschrift Math. Logik und Grundlagen der Mathematik*, no. 6, pp. 66–92, 1960.

[23] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm, “Mona: Monadic second-order logic in practice,” in *TACAS 1995*, pp. 89–110, 1995.

[24] L. D’Antoni and M. Veanes, “Static analysis of string encoders and decoders,” in *VMCAI 2013*, pp. 209–228, 2013.

[25] S. Suriyakarn, C. Pit-Claudel, B. Delaware, and A. Chlipala, “Narcissus: Deriving correct-by-construction decoders and encoders from binary formats,” *Computing Research Repository (CoRR)*, vol. abs/1803.04870, 2018.

[26] D. Cofer, J. Backes, A. Gacek, D. DaCosta, M. Whalen, I. Kuz, G. Klein, G. Heiser, L. Pike, A. Foltzer, M. Podhradsky, D. Stuart, J. Grahan, and B. Wilson, “Secure Mathematically-Assured Composition of Control Models,” tech. rep., Defense Technical Information Center, September 2017. http://www.dtic.mil/dtic/tr/fulltext/u2/1039782.pdf.