Formal Synthesis of Filter Components for Use in Security-Enhancing Architectural Transformations

David S. Hardin
Trusted Systems Group
Collins Aerospace
Cedar Rapids, IA USA
Email: david.hardin@collins.com

Konrad L. Slind
Trusted Systems Group
Collins Aerospace
Bloomington, MN USA
Email: konrad.slind@collins.com

***Abstract*—Safety- and security-critical developers have long recognized the importance of applying a high degree of scrutiny to a system's (or subsystem's) I/O messages. However, lack of care in the development of message-handling components can lead to an *increase*, rather than a *decrease*, in the attack surface. On the DARPA Cyber-Assured Systems Engineering (CASE) program, we have focused our research effort on *identifying* cyber vulnerabilities early in system development, in particular at the Architecture development phase, and then automatically *synthesizing* components that mitigate against the identified vulnerabilities from high-level specifications. This approach is highly compatible with the goals of the LangSec community. Advances in formal methods have allowed us to produce hardware/software implementations that are both performant and guaranteed correct. With these tools, we can synthesize high-assurance "building blocks" that can be composed automatically with high confidence to create trustworthy systems, using a method we call *Security-Enhancing Architectural Transformations*. Our synthesis-focused approach provides a higher-leverage insertion point for formal methods than is possible with *post facto* analytic methods, as the formal methods tools directly contribute to the implementation of the system, without requiring developers to become formal methods experts. Our techniques encompass Systems, Hardware, and Software Development, as well as Hardware/Software Co-Design/Co-Assurance. We illustrate our method and tools with an example that implements security-improving transformations on system architectures expressed using the Architecture Analysis and Design Language (AADL). We show how message-handling components can be synthesized from high-level regular or context-free language specifications, as well as a novel specification language for self-describing messages called *Contiguity Types*, and verified to meet arithmetic constraints extracted from the AADL model. Finally, we guarantee that the intent of the message processing logic is accurately reflected in the application binary code through the use of the verified CakeML compiler, in the case of software, or the Restricted Algorithmic C toolchain with ACL2-based formal verification, in the case of hardware/software co-design.**

I. Introduction

Experienced safety-critical and security-critical system architects have long emphasized the importance of applying the highest degree of scrutiny to a system's I/O messages.[1] From a safety perspective, message validation has long been a "best practice." For security-critical architecture and design, identification of the attack surface has emerged as an important analysis technique. One of our key research focus areas on the DARPA Cyber-Assured Systems Engineering (CASE) program[2] concerns the identification of and mitigation against message-based attacks, using the highest-assurance techniques and tools available.

Advances in formal methods have allowed us to produce hardware/software implementations that are both performant and guaranteed correct. With these tools, we can synthesize security-enhancing "building blocks" that can be composed automatically with high confidence to create trustworthy systems, using a method we call *Security-Enhancing Architectural Transformations*. Our synthesis-focused approach is compatible with the goals of the LangSec initiative [1], and provides a higher-leverage insertion point for formal methods than is possible with *post facto* analytic methods, as the formal methods tools directly contribute to the implementation of the system, without requiring developers to become formal methods experts. Our techniques encompass Systems, Hardware, and Software Development, as well as Hardware/Software Co-Design/Co-Assurance, as we detail in the sections that follow.

We have identified several security-enhancing architectural transformations, including the introduction of filters, monitors, attestation/measurement managers, data transformers, as well as the provision of a verified operating system, such as the seL4 microkernel [2], with mathematically proven space and time isolation properties.

The particular transformation we consider in this paper is *filter insertion*, which is a pattern to prevent attacks that rely on malformed messages to flow through a system. The filter ensures that only messages that are well-formed are further processed by the system, and that no well-formed messages are dropped. The precise definition of "well-formed" will, of course, vary from situation to situation; hence, it is a

[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.

parameter that the system designer can instantiate. In this paper, we consider filters where well-formedness can be expressed in terms of regular or context-free languages. Our toolchain then:

1) compiles the well-formedness specification to table-driven deterministic finite-state automaton (DFA) based implementations by instantiating a general compilation theorem utilizing a theorem prover, such as HOL4;
2) synthesizes code, preferably using a verified compiler (*e.g.*, CakeML), that implements matching of input messages against the DFA from the preceding step using proof-producing synthesis [3];
3) generates an I/O wrapper around the synthesized code, allowing it to be automatically inserted on an interprocess communication channel, utilizing, *e.g.*, CAmkES [4] for seL4;
4) produces a proof, covering both liveness and safety, that the synthesized component implements the expected behavior: namely, it lets through exactly the messages described by the high-level well-formedness specification; and
5) generates a filter implementation, as binary code, RTL, or a combination of the two, preferably using a verified compiler.

In many cases, these steps can be carried out in a theorem prover, such as HOL4, which we can integrate into the CAmkES build system: thus, strong evidence for the trustworthiness of transformations is built along with the system image wherein they are used.

Since high-level specifications such as regular expressions tend to be somewhat opaque, it is also important to obtain independent assurance that a given specification really does correspond to the intended meaning of "well-formed". To address this, we incorporate automated test-case generation into our environment, and encourage users to validate the synthesized code using a combination of automated and manually-constructed test cases.

II. Architectural Model Processing

On DARPA CASE, we process architectural models for cyber-physical systems expressed in the Architecture Analysis and Design Language (AADL) [5], a standard architectural modelling language that we have successfully utilized on several cyber-physical systems programs [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 locus for system-level analysis, reasoning, scheduling, generation of executables, *etc*. A primary goal of our CASE effort is to develop architecture-to-architecture transformations that provably improve the security of a system, as detailed below.

A. AGREE contract checker and Resolute assurance case tool

AGREE is an AADL annex and tool [7] that supports component-based hierarchical reasoning based on assume-guarantee behavioral specifications (contracts) placed on elements of an AADL architecture. AGREE contracts are expressed using past-time LTL formulas, which are checked using the JKind k-induction model-checker [8]. For our class of security-enhancing architectural transformations, in which new AADL component components are inserted into a model in order to satisfy a security requirement, we wish to ensure that the added component does indeed provide the specified capability. This has twin aspects: first, the "empty" 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 assume/guarantee reasoning performed by the AGREE tool; thus, it must be proved separately. Second, the empty component is filled in: the DFA-based implementation corresponding to the well-formedness requirement is generated and compiled. The well-formedness property becomes a specification on the generated implementation, and the specification is (automatically) propagated to the infinitary behavior of the filter component, by means of the proofs described in [9]. The creation of the message-handling component creates multiple pieces of verification evidence from AGREE, as well as theorem proving tools such as HOL4. Managing and combining the results of multiple reasoners justifying architectural transformations is performed by Resolute, an assurance case tool with deep connections to AADL models [10].

B. CASE Workflow

A common first step towards securing a cyber-physical system is establishing *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 the components that are explicitly present in representations 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 a verified operating system such as 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

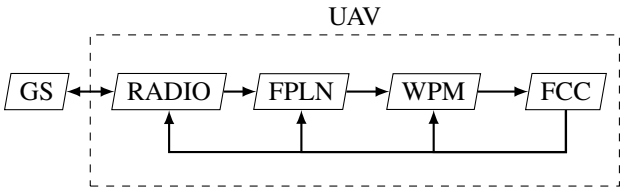

Figure 1. Simplified architectural model of a UAV flight controller.

ground station or radio driver could be used to feed malformed messages to a legacy UAV flight planner that the flight planner is not equipped to handle. This could then be exploited by an attacker to induce crashes, cause privilege escalation through buffer overflow, *etc*.

Our vision for CASE is that the system developer, having had the potential for such a vulnerability brought to her attention (the CASE program is developing cyber requirements analysis tools for this purpose), 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. These protective components should be easy to configure and deploy for the needs of different systems. Component trustworthiness is of utmost importance: when we introduce more components into a system, they too can become part of the attack surface. Thus, each component inserted by a transformation is accompanied by a formal proof connecting the component's intended behavior with the behavior of its implementation down to the binary level.

Security-enhancing architectural transformations are conducted with the aid of "wizards" that our team has added to the open source AADL development tool OSATE [11]. Automated synthesis of an executable system from the AADL model, targeting either a JVM instance, Linux, or seL4, is then produced by the HAMR tool, also a product of our research team [12].

In the sections that follow, we examine several example system realizations constructed using our high-assurance workflow and toolchain.

III. A Motivating Example

By way of a motivating example, consider the (simplified) architectural 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 [13], and utilizes legacy elements 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). 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. WPM processes a fixed-size window of the next few waypoints to be dealt with by the flight control computer. 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, actuators, *etc*. As FCC makes progress, it tells WPM to advance the window, and also sends back various sensor data to FPLN, WPM, and GS via the RADIO.

Recognizing the need for wellformedness checking of messages received by the radio, we perform an architectural transformation on the UAV model to insert a high-assurance component, FLT, between RADIO and FPLN, resulting in the transformed model of Figure 2.

In the sections that follow, we describe synthesis methods for the inserted component FLT for a number of similar, but distinct, UAV coordinate specifications, utilizing several high-assurance, formal methods-based techniques that we have developed during the course of the CASE program.

IV. Regular Expression-Based Message Specifications

In this section, we discuss an implementation of FLT, specified using regular expressions, and synthesized using formal methods tools. In earlier work [14] we reported on a verified compiler from extended regular expressions to table-driven DFAs, using Brzozowski's "derivative" approach [15]; this compiler is available in the HOL4 distribution [3]. We make use of this compiler to create verified regexp-based message processors. 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 message-handling component [9].

A. Semantic properties of formal languages

A message *specification* takes the form of logical constraints on a record type (supplemented with layout and well-formedness information) modelling network messages. The message processing *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 $\mathsf{SPEC} : \mathsf{recd} \rightarrow \mathsf{bool}$.

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

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

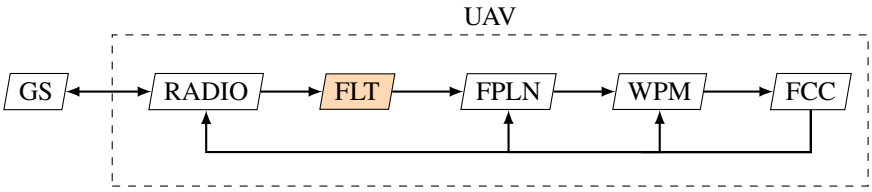

Figure 2. Simplified architectural model of a UAV flight controller with inserted message wellformedness checker.

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:

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

These can be required in the proof of (1), and are further evidence that the component will work properly.

Example 1: Consider the well-formed coordinate checker of Section IV. 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

$$\mathsf{encode}(r) = \mathsf{enc}\ 1\ r.latitude \frown \mathsf{enc}\ 2\ r.longitude \frown \mathsf{enc}\ 2\ r.altitude$$

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 specialized provers for membership in the appropriate character sets.

B. Example: UAV Coordinate Constraints

Consider an AADL model corresponding to the transformed architecture depicted informally in Figure 2. One type of data to be transferred between the Ground Station and the UAV is a Location coordinate, represented by the AADL record structure of Figure 3.

We wish to only 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;

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

Figure 3. AADL specification for a UAV coordinate.

one such set of constraints is presented in conventional mathematical notation as follows:

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

Thus the assertion to be added on the output out of the inserted 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 "meta-data", which we exploit in order to obtain declarative specifications of message representations.

Expressiveness: We have found that specifying message formats with regexps is a promising approach, combining a declarative approach to behavior 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.

V. Messages Specified by Context-Free Grammars

Let us now imagine that the position information for the UAV is expressed using a common standard interchange format, namely JSON. JSON (JavaScript Object Notation) is a popular lightweight interchange format for structured data [16]. JSON is text-based, and is relatively simple to generate and parse. JSON data payloads are built from two basic primitives: a collection of name-value pairs, and

an ordered list of values. In our use case, a UAV air-ground communications system employs JSON to encode certain messages sent between the UAV and its ground-based control station. For example, a UAV coordinate could be encoded in JSON as:

```
{"lat":42.08, "long":-91.64, "alt":5000}
```

To aid in thwarting cyber attacks, we need to construct a high-assurance component that checks whether a given air-ground message is legal JSON, and rejects any malformed messages. In keeping with the CASE vision, we need to design said component in the highest assurance manner possible. In order to create such a JSON wellformedness checker, we need to perform both lexical and syntactic level analysis on any candidate JSON message received[4].

A. Lexical Analysis

For lexical level analysis, we have constructed verified tools that generate a verified lexer based on Deterministic Finite-state Automata (DFAs), where the individual lexeme specifications are given as regular expressions (as in [14] and [9], and discussed in Section IV). These individual regexps are combined to form a "maximal-munch" lexer, following standard practice. A relatively novel aspect of our work is the use of the HOL4 theorem prover and the CakeML compiler to produce the lexer binary.

B. Syntactic Analysis

For syntactic-level analysis, we employ the Vermillion verified LL(1) parser generator due to Lasser, *et. al.* [17], coupled with a parse table traverser hand-written in CakeML. The parse table traverser is a bit more complex than that for the lexer, in that it requires a rule stack; for this, we employ a verified stack component. We also record the rules used as parsing proceeds in a list, allowing us to reconstruct the parse tree later on.

Our LL(1) grammar rules for JSON are presented in Figure 4. An initial capital letter indicates a nonterminal symbol; ε designates the empty string; and the rest are terminals. The terminal symbols fls (false), flt (float), int (integer), nul (null), str (string), tru (true), open brace, close brace, open bracket, close bracket, colon, and comma constitute the JSON lexemes produced by the lexer.
The overall structure of the JSON filter, including both lexical and syntactic analysis toolchains, is displayed in Figure 5.

C. Results

We were able to successfully realize a JSON lexical analyzer and syntactic analyzer in RAC for a significant subset of JSON (we chose not to deal with the complexities of, *e.g.*,

[4]Note that we assume that there is some system-defined maximum message size, such that the message to be checked can be held all at once in memory.

$$
\begin{array}{lcl@{\qquad}lcl}
Value & \rightarrow & \{\ Pairs\ \} & Pairs & \rightarrow & Pair\ PairsTl \\
Value & \rightarrow & [\ Elts\ ] & PairsTl & \rightarrow & \varepsilon \\
Value & \rightarrow & \text{str} & PairsTl & \rightarrow & ,\ Pair\ PairsTl \\
Value & \rightarrow & \text{int} & Pair & \rightarrow & \text{str}\ :\ Value \\
Value & \rightarrow & \text{flt} & Elts & \rightarrow & \varepsilon \\
Value & \rightarrow & \text{tru} & Elts & \rightarrow & Value\ EltsTl \\
Value & \rightarrow & \text{fls} & EltsTl & \rightarrow & \varepsilon \\
Value & \rightarrow & \text{nul} & EltsTl & \rightarrow & ,\ Value\ EltsTl \\
Pairs & \rightarrow & \varepsilon & & & \\
\end{array}
$$

Figure 4. LL(1) grammar for JSON.

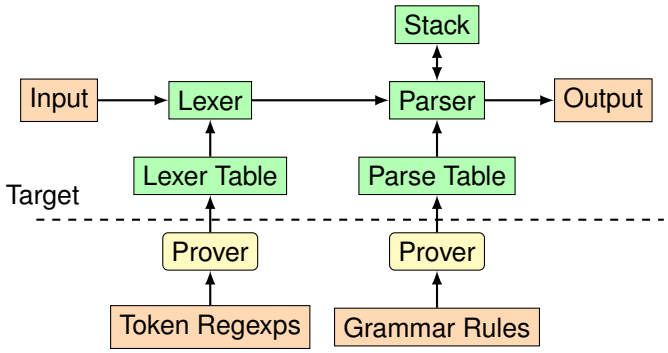

Figure 5. A JSON filter built using verified lexer/parser generation tools.

Unicode in this first experiment). In our implementation, an input message (sequence of bytes) is presented to the lexical analyzer, and the tokens generated by the lexer are then directly fed to the syntactic analyzer. If the lexical and syntactic analyses both succeed, an input-bytes-to-token map, as well as a parse tree, are generated, and the message is allowed to be further processed by the UAV software; otherwise the input message is rejected. We also successfully validated our CakeML- based lexer/parser for JSON by presenting it with a test suite consisting of both valid and invalid JSON messages.

Finally, we note that by constructing a table-based lexer and parser, we provide easy transition of "verified data" from theorem provers running on a host system to an embedded target. This approach also enables hardware-based implementation, as discussed in Section VII.

VI. Messages Specified by Contiguity Types

We now consider a new specification technique for the processing of *self-describing* messages: those where information embedded in fields of the message determines the message structure. The characteristic property of such messages is *contiguity*: all the elements of the message are laid out side-by-side in a byte array (or string). We assume that a message is the *result* of a map from structured data, and we employ a collection of programming language types to capture that structure, in a specification language we call *Contiguity Types*.

$$
\begin{array}{rcl}
base & = & \mathsf{bool} \mid \mathsf{char} \mid \mathsf{u8} \mid \mathsf{u16} \mid \mathsf{u32} \mid \mathsf{u64} \mid \mathsf{i16} \mid \mathsf{i32} \\
& \mid & \mathsf{i64} \mid \mathsf{float} \mid \mathsf{double} \\
\tau & = & base \\
& \mid & \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n) \\
& \mid & \mathsf{Array}\ \tau\ exp \\
& \mid & \mathsf{Alt}\ bexp\ \tau_1\ \tau_2
\end{array}
$$

Figure 6. Contiguity types.

Difficulties in self-describing message formats include variable-length arrays and unions. A *variable-length array* is a field where the number of elements in the field depends on the value of some already-seen field (or, more generally, a computation involving previously-seen information in the message). The length is therefore a value determined at runtime. A *union* is deployed when some information held in a message is used to determine the structure of later portions of the message. For example, unions can be used to support versioning where version i has n fields, and version $i+1$ has $n+1$. In settings where both versions need to be supported in a single format, it makes sense to encode the version handling inside the message, and unions accomplish this.

Another problematic aspect is that self-describing data formats fall outside the realm of common formal language techniques; *e.g.*, variable-length fields clearly aren't able to be described by regular or context-free languages. (These language classes encompass repetitions of a fixed or unbounded size, but not repetitions of a size determined by parts of the input string.) It seems that context-sensitive grammars can, in principle, specify such information, but there are few tools supporting context sensitive languages. Another possibility would be to use *parser combinators*; it seems likely that the combinators can be instrumented to gather and propagate contextual information. However, we are seeking a high level of formal specification and automation, while still being rooted in formal languages, with their emphasis on sets of strings.

A. Contiguity Types

Contiguity types [18] (Figure 6) start with common base types (booleans, characters, signed and unsigned integers, *etc*.) and are closed under the construction of records, arrays, and unions. Notice that τ (we will use the terms *contiguity type*, contig, and τ interchangeably) is defined in terms of a type of arithmetic expressions exp and also $bexp$, boolean expressions built from exp. Now consider

$$\mathsf{Array}\ \tau\ exp\ .$$

For this to specify a varying length array dependent on other fields of the message, its dimension exp should be able to refer to the *values* of those fields. The challenge is just how to express the concept of "other fields", *i.e.*, we need a notation to describe the *location* in the message buffer where the value of a field can be accessed. Our core insight is that this is similar to a problem that programming language designers had in the 1960s and 1970s, resolved by the notions of *L-value* and *R-value*. The idea is originally due to Christopher Strachey in CPL [19] and developed subsequently, for example by Dennis Ritchie for the C programming language [20].

Before getting into formal details, we discuss a few examples. We will use familiar notation: records are lists of $name : \tau$ elements enclosed by braces; an array field $\mathsf{Array}\ c\ dim$ is written $c\,[dim]$; and $\mathsf{Alt}\ b\ \tau_1\ \tau_2$ is written '`if` b `then` τ_1 `else` τ_2'. 'Cascaded' Alts may be written in Lisp 'cond' style, *i.e.*, as

$$
\begin{array}{ll}
\mathsf{Alt} & b_1 \longrightarrow \tau_1 \ldots \\
& b_n \longrightarrow \tau_n \\
& \text{otherwise} \longrightarrow \tau_{n+1}
\end{array}
$$

1) The following is a record with no self-describing aspects: each field is of a statically known size.

```
{ A : u8
  B : {name : char [13]
       cell : i32}
  C : bool }
```

   The `A` field is specified to be an unsigned int of width 8 bits, the `B` field is a record, the first element of which is a character array of size 13, the second element a 32 bit integer, and the last field a boolean.

2) Variable-sized strings are a classic self-describing aspect. In this example the contents of the `len` field determines the number of elements in the `elts` field.

```
{ len : u16
  elts : char [len] }
```

3) The following examples shows the Alt construct being used to support multiple versions in a single format. Messages with the value of field `versionID` being less than 7 have three fields in the message, while all others have two.

```
{ versionID : u8
  versions : if versionID < 7 then
                {A : i32, B : u16}
             else {Vec : char [8]} }
```

B. Expressions, L-values, and R-values

In programming languages, an *L-value* is an expression that can occur on the left-hand side of an assignment statement. Similarly, *R-value* designates expressions occurring on the right-hand side of assignments.

Figure 7 presents the formal syntax for L-values, R-values, and the boolean expressions we will use. An L-value can be a variable, an array index, or a record field access. R-values are arithmetic expressions that can contain L-values (we will use exp interchangeably with R-value).

$$
\begin{array}{rcl}
lval & = & varname \mid lval\,[exp] \mid lval.fieldname \\
exp & = & \mathsf{Loc}\ lval \mid \mathsf{nLit}\ \mathsf{nat} \mid constname \mid exp + exp \\
 & \mid & exp * exp \\
bexp & = & \mathsf{bLit}\ bool \mid \neg bexp \mid bexp \vee bexp \mid bexp \wedge bexp \\
 & \mid & exp = exp \mid exp < exp
\end{array}
$$

Figure 7. L-values, expressions, and boolean expressions.

```
AltitudeType = AGL | MSL

Location3D = {
  Latitude : double,
  Longitude : double,
  Altitude : float,
  AltitudeType : AltitudeType,
  Wellformed : Assert (
    -90.0 <= Latitude <= 90.0 and
    -180.0 <= Longitude <= 180.0 and
    0.0 <= Altitude <= 15000.0) }
```

Figure 8. Contiguity type specification for a UAV location.

An L-value denotes an *offset* from the beginning of a data structure, plus a *width*. In an R-value, an occurrence of an L-value is mapped to the value of the patch of memory between $offset$ and $offset + width$. It may not be obvious that a notation supporting assignment in imperative languages can help, but there is indeed a form of assignment lurking.

C. Wellformedness Checking using Contiguity Types

An example GPS coordinate checker using contiguity types is depicted in Figure 8. Note the use of the `Wellformed` assertion to guarantee that all received GPS coordinate values are within the expected ranges. This contiguity type specification is compiled to executable code using the facilities of the verified CakeML compiler, along with proofs of soundness and functional correctness. For details, please consult [18].

VII. Hardware/Software Co-Assurance

Thus far, we have only discussed message-handling components that can be implemented as software. Many advantages would accrue in terms of speed, as well as basic integrity of the generated artifact itself, if the core algorithm could be implemented in hardware. More broadly, the ability to selectively implement components of a system architecture in hardware, while maintaining other elements as software, all with very high assurance, would provide system developers with maximum flexibility.

Formal Methods have been successfully applied to systems that are software-only, or hardware-only. Relatively little attention has been paid to hardware/software systems, specifically to the assurance of hardware/software co-designs. The need for hardware/software co-assurance techniques is increasing, notably for autonomous and semi-autonomous platforms for land, sea, air, and space, as well as other complex, connected systems.

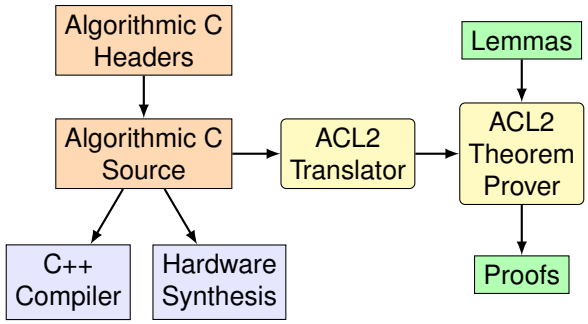

Figure 9. Restricted Algorithmic C (RAC) toolchain.

In order to investigate hardware/software co-assurance, we have conducted several experiments employing a state-of-the-art toolchain, due to Russinoff and O'Leary, and originally designed for use in floating-point hardware verification [21], to determine its suitability for the creation of mission-critical applications in various domains. Note that this toolchain has already demonstrated the ability to scale to industrial hardware designs at both Intel and Arm.

Algorithmic C [22] is a High-Level Synthesis (HLS) language that defines C++ header files to enable compilation to both hardware and software platforms, including support for the peculiar bit widths employed, for example, in floating-point hardware design. The Russinoff-O'Leary Restricted Algorithmic C (RAC) toolchain, depicted in Figure 9, translates a subset of Algorithmic C source to the Common Lisp subset supported by the ACL2 theorem prover, as augmented by Russinoff's Register Transfer Logic (RTL) theorem libraries ("books," in ACL2 parlance).

The ACL2 Translator component of Figure 9 converts imperative RAC code to functional ACL2 code. Loops are translated into tail-recursive functions. Structs and arrays are converted into functional ACL2 records. The combination of modular arithmetic and bit-vector operations of RAC source code is faithfully translated to functions supported by Russinoff's RTL books. The RTL books are quite capable of reasoning about a combination of non-linear arithmetic and bit-vector operations, which is a very difficult feat for most automated solvers.

A. Experiments in Hardware/Software Co-Assurance

In a first hardware/software co-assurance experiment, we created a number of algebraic data types of fixed maximum size, suitable for both hardware and software implementation, in RAC, including lists, stacks, queues, deques, and

binary trees. This effort was inspired by our previous work on fixed-size formally verified algebraic data types [23].

In a second experiment, we created an Instruction Set Architecture (ISA) simulator for a representative 64-bit RISC ISA in the RAC C++ subset. We used the RAC tool to translate the ISA simulator code to ACL2, produced small binary programs for the ISA used to validate the simulator, and then utilized the ACL2 Codewalker decompilation-into-logic facility to prove those test programs correct [24].

In a third effort, we implemented a high-assurance checker for JSON-formatted data used in an Unmanned Air Vehicle (UAV) application, described in Section V, using the RAC toolchain [25] instead of CakeML. As described previously, the JSON checker component was built using a table-driven lexer/parser, supported by mathematically-proven lexer and parser table generation technology, as well as a verified stack data structure (reused from our earlier RAC data structure work). The implementation was validated by comparing test cases produced by C++ compiled RAC code with that produced in ACL2. We also gathered performance data, indicating that the RAC-generated (software-only) lexer/parser for JSON was very competitive in speed to JSON parsers generated by non-verified parser generators.

The aforementioned efforts have focused on software implementations of RAC applications. Recently, we have begun work on targeting FPGA hardware, in collaboration with colleagues at Kansas State University. A goal of this work is to generate high-assurance hardware and/or software from high-level architectural specifications expressed in AADL.

VIII. Related Work

The LangSec initiative advocates for many of the same goals that we pursue in our research. The LangSec community "regards the Internet insecurity epidemic as a consequence of ad hoc programming of input handling at all layers," and posits that "the only path to trustworthy software that takes untrusted inputs is treating all valid or expected inputs as a formal language." [1].

One way in which our work differs somewhat from many LangSec efforts is our emphasis on providing standalone high-assurance message filter and monitor components. This is due to the fact that part of our mission statement on DARPA CASE is to provide security-enhancing components for existing systems, whose legacy components may not be able to be readily changed to introduce a LangSec-based parser as part of its codebase.

The best-known LangSec tool Hammer "tackles the problem of writing correct parsers" by implementing combinators taken from formal language theory [26]. Hammer is written in C and provides C developers with a familiar syntax for expressing their input language specifications. Our work shares the approach of creating correct parsers by building on language-theoretic foundations, but we perform our work primarily in theorem proving environments in order to provide additional assurance. We do not attempt to tightly integrate our message format specifications with a particular development language, but we appreciate the value in so doing from a developer adoption perspective.

A LangSec tool that is particularly related to our efforts on DARPA CASE is Parsley [27]. As in our work, Parsley focuses on data format parsing grounded in formal language theory, as well as formal verification of the resulting parser. Parsley takes on a much more difficult problem than our work currently addresses, emphasizing context-sensitive parsing, nested grammars, as well as very complex formats such as PDF. Our work, on the other hand, distinguishes itself by its integration with systems engineering, focus on "self-describing" message formats commonly used in mission-critical systems, verified compilation all the way to binary, as well as our emphasis on hardware/software co-design and co-assurance. The latter also drives us more towards table-driven lexer/parser implementations.

Much of the work on system-level formal verification has been performed in the context of higher-level Model-Based System Engineering languages such as AADL [6], or Simulink/Stateflow (*e.g.*, [28], [29]). While hardware/software co-design is enabled by these system architecture and design tools, little work has been done on true hardware/software co-assurance. Notable work has been done on High-Assurance Domain-Specific languages targeting hardware/software implementation (*e.g.*, [30], [31]).

Floating-point hardware verification utilizing theorem proving technology has a notable history (*e.g.*, [32], [33], [21]). Many of these efforts have focused on designs expressed using traditional Hardware Description Languages, such as Verilog; Russinoff's work using Algorithmic C is a notable exception.

Many tools exist for C code verification. Tools that take a similar approach to ours include Appel's Hoare Logic for CompCert C [34], which is derived from the operational semantics of the CompCert verified C compiler in Coq [35]. AutoCorres [36] arose from the `seL4.verified` operating system verification effort; it translates ASTs from a parser for the restricted C dialect used in seL4 to Schirmer's SIMPL theory in Isabelle/HOL [37].

IX. Conclusion and Future Work

We have detailed a method and toolchain for the creation of formally verified components for critical systems. We have demonstrated how this toolchain can be used to implement security-enhancing transformations on system architectures specified in AADL, with implementations automatically synthesized using the verified CakeML compiler. Such transformations, *e.g.*, message handlers based on regular languages or context-free languages, can be used to improve resiliency against cyber attack. We have documented examples of the use of regular expressions, context-free languages, as well as a novel *contiguity type* specification language, to

describe message formats. We have also described formal synthesis techniques for wellformedness checkers, carried out in the context of the DARPA CASE high-assurance system development workflow.

We have also described methods and tools for enhancing the safety and security of critical systems using a hardware/software co-assurance approach for systems implemented in a High-Level Synthesis (HLS) language. We have employed one such state-of-the-art toolchain, Restricted Algorithmic (RAC), due to Russinoff and O'Leary, to develop high-assurance architectural transformations that can be realized as hardware, software, or a combination of the two. We have utilized the RAC toolchain on a number of development examples, including a JSON lexer/parser application featuring algebraic data types, verified table-driven lexing, as well as verified table-driven parsing.

In future work, we will enhance our verified synthesis tools for high-level engineering specification languages, and pursue end-to-end validation of the toolchain for increasingly complex systems, demonstrating formal composition of components generated using the toolchain. We will continue to evolve the contiguity type specification language, developing a compiler for contiguity type specifications, and bridging the gap with conventional parsing technology based on context-free grammars.

The CakeML compiler correctness theorem can transport our theorems about an application to the binary level, with one important caveat: it preserves semantics up to the fact that the binary can abort if it runs out of memory. Hence liveness becomes "liveness unless we run out of memory". Our components as designed *should not* run out of memory, but we should be able to state and prove unconditional liveness for the binary. This proof will build on recent work on the space-cost semantics for CakeML [38].

On the hardware/software co-assurance front, we are keen to implement a verified version of the RAC-to-ACL2 translator using the verified lexer/parser technology used to create the JSON lexer/parser; however, we first need to develop a means for high-assurance invocation of "action code". Colleagues at Kansas State University will continue development of a transpiler from a modern functional/imperative language to HLS code, building on the existing Scala-subset-to-C transpiler featured in the HAMR tool [12], and produce a working proof-of-concept application on a common FPGA. Finally, we will investigate hardware/software applications, using the languages and tools described in this paper, with correctness proofs that span hardware/software boundaries.

X. Acknowledgments

Many thanks to Sam Lasser of Tufts University for his assistance with the Vermillion verified LL(1) parser generator code; to David Russinoff of Arm for answering questions about the RAC toolchain; and to Robby and Matthew Weis of Kansas State University for their ongoing work to further our hardware/software co-assurance efforts. This work was sponsored in part by the Defense Advanced Research Projects Agency (DARPA).

References

[1] *LangSec: Language-theoretic Security*, LangSec Organization, 2021. [Online]. Available: http://langsec.org

[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," *Comm. ACM*, vol. 53, no. 6, pp. 107–115, 2010. [Online]. Available: http://doi.acm.org/10.1145/1743546.1743574

[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. [Online]. Available: https://doi.org/10.1017/S0956796813000282

[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. [Online]. Available: https://doi.org/10.1016/j.jss.2006.08.039

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

[6] 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," Air Force Research Laboratory Information Directorate, Tech. Rep., September 2017. [Online]. Available: https://apps.dtic.mil/dtic/tr/fulltext/u2/1039782.pdf

[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)*, 2012, pp. 126–140.

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

[9] D. S. Hardin, K. L. Slind, J. Å. Pohjola, and M. Sproul, "Synthesis of verified architectural components for autonomy hosted on a verified microkernel," in *Proceedings of the 53rd Hawaii International Conference on System Sciences*, January 2020, pp. 6365–6374.

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

[11] *OSATE: Open Source AADL Tool Environment*, Software Engineering Institute, 2021. [Online]. Available: https://osate.org/

[12] *Sireum HAMR: High Assurance Modeling and Rapid Engineering for Embedded Systems*, Kansas State University, 2021. [Online]. Available: http://hamr.sireum.org/

[13] 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)*, June 2018, pp. 257–268. [Online]. Available: https://doi.org/10.1109/SP.2013.35

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

[15] J. Brzozowski, "Derivatives of Regular Expressions," *Journal of the ACM*, vol. 11, no. 4, pp. 481–494, October 1964.

[16] *The JSON Data Interchange Syntax Standard (ECMA-404)*, ECMA International, 2017. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[17] S. Lasser, C. Casinghino, K. Fisher, and C. Roux, "A Verified LL(1) Parser Generator," in *10th International Conference on Interactive Theorem Proving (ITP 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. Harrison, J. O'Leary, and A. Tolmach, Eds., vol. 141, 2019, pp. 24:1–24:18. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/11079

[18] K. L. Slind, "Specifying message formats with Contiguity Types," in *Proceedings of the Twelfth International Conference on Interactive Theorem Proving (ITP 2021)*, June 2021, to appear.

[19] C. Strachey, "Fundamental concepts in programming languages," in *Higher-Order and Symbolic Computation*, vol. 13, 2000, pp. 11–49.

[20] D. Ritchie, *The Development of the C Language*, 2003. [Online]. Available: https://www.bell-labs.com/usr/dmr/www/chist.html

[21] D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer, 2018.

[22] *Algorithmic C (AC) Datatypes*, Mentor Graphics Corporation, 2016. [Online]. Available: https://www.mentor.com/hls-lp/downloads/ac-datatypes

[23] D. S. Hardin and K. L. Slind, "Using ACL2 in the design of efficient, verifiable data structures for high-assurance systems," in *Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, S. Goel and M. Kaufmann, Eds., vol. 280, 2018, pp. 61–76.

[24] D. S. Hardin, "Instruction set architecture specification, verification, and validation using Algorithmic C and ACL2," in *Workshop on Instruction Set Architecture Specification (SpISA19)*, September 2019. [Online]. Available: https://www.cl.cam.ac.uk/~jrh13/spisa19/paper_09.pdf

[25] ——, "Verified hardware/software co-assurance: Enhancing safety and security for critical systems," in *Proceedings of the 2020 IEEE Systems Conference*, 2020.

[26] P. Anantharaman, M. C. Millian, S. Bratus, and M. L. Patterson, "Input handling done right: Building hardened parsers using language-theoretic security," in *IEEE Cybersecurity Development, SecDev 2017*. IEEE Computer Society, 2017, pp. 4–5. [Online]. Available: https://doi.org/10.1109/SecDev.2017.12

[27] P. Mundkur, L. Briesemeister, N. Shankar, P. Anantharaman, S. Ali, Z. Lucas, and S. Smith, "The Parsley data format definition language," in *2020 IEEE LangSec Workshop*, 2020, pp. 300–307.

[28] M. Whalen, D. Cofer, S. Miller, B. Krogh, and W. Storm, "Integration of formal analysis into a model-based software development process," in *FMICS*, 2007.

[29] D. S. Hardin, T. D. Hiratzka, D. R. Johnson, L. G. Wagner, and M. W. Whalen, "Development of security software: A high assurance methodology," in *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM'09)*, K. Breitman and A. Cavalcanti, Eds. Springer, 2009, pp. 266–285.

[30] S. Browning and P. Weaver, "Designing tunable, verifiable cryptographic hardware using Cryptol," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, D. S. Hardin, Ed. Springer, 2010, pp. 89–143.

[31] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing Kansas Lava," in *Implementation and Application of Functional Languages*, ser. LNCS, vol. 6041. Springer, 2009, pp. 18–35.

[32] J. Harrison, "Floating-point verification using theorem proving," in *Formal Methods for Hardware Verification*, M. Bernardo and A. Cimatti, Eds. Springer Berlin Heidelberg, 2006, pp. 211–242.

[33] W. A. Hunt, S. Swords, J. Davis, and A. Slobodova, "Use of formal verification at Centaur Technology," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, D. S. Hardin, Ed. Springer, 2010, pp. 65–88.

[34] A. W. Appel, *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[35] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[36] D. Greenaway, J. Lim, J. Andronick, and G. Klein, "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*, ser. PLDI '14. ACM, 2014, pp. 429–439. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594296

[37] N. Schirmer, "Verification of sequential imperative programs in Isabelle/HOL," Ph.D. dissertation, TU Munich, 2006.

[38] A. Gómez-Londoño, J. Å. Pohjola, H. T. Syeda, M. O. Myreen, and Y. K. Tan, "Do you have space for dessert? a verified space cost semantics for CakeML programs," *Proc. ACM Program. Lang. (OOPSLA)*, vol. 4, pp. 204:1–204:29, 2020. [Online]. Available: https://cakeml.org/oopsla20.pdf