Capturing the iccMAX calculatorElement: A Case Study on Format Design

Vijay H. Kothari
Prashant Anantharaman
Sean W. Smith
Dartmouth College
Hanover, NH 03755

Briland Hitaj
Prashanth Mundkur
Natarajan Shankar
SRI International
Menlo Park, CA 94025

Letitia W. Li
BAE Systems FAST Labs
Arlington, VA 22203

Iavor Diatchki
William Harris
Galois Inc.
Portland, OR 97204

***Abstract*—ICC profiles are widely used to provide faithful digital color reproduction across a variety of devices, such as monitors, printers, and cameras. In this paper, we document our efforts on reviewing and identifying security issues with the calculatorElement description from the recent iccMAX specification (ICC.2:2019), which expands upon the ICC v4 specification (ICC.1:2010). The iccMAX calculatorElement, which captures a calculator function through a stack-based computational approach, was designed with security in mind. We analyzed the iccMAX calculatorElement using a variety of approaches that utilized: the proof assistant PVS, the theorem-proving language ACL2, the data description language DaeDaLus, and tools tied to the data description language Parsley. Bringing the tools of formal data description, theorem proving, and static analysis to a non-trivial real-world specification has shed light on both the tools and the specification. This exercise has led us to discover numerous bugs within the specification, to identify specification improvements, to identify flaws with a demo implementation, and to recognize ways that we can improve our own tools. Additionally, this particular case study has broader implications for those who work with specification, data description languages, and parsers. In this paper, we document our work on this exercise and relay our key findings.**

***Keywords*-LangSec; data description languages; formal methods; static analysis; parser; specification; iccMAX;**

I. Introduction

The International Color Consortium (ICC) has worked on developing and refining the ICC profile, a color management format that provides cross-platform color consistency, since 1993 [7]. In 2019, a specification for the iccMAX profile [9], an extension to ICC v4 [8], was released. Previous iterations of the ICC profile are widely used, appearing in many modern devices, such as monitors, printers, and scanners; however, given that the iccMAX profile is still relatively new, it has yet to see significant deployment, which provides additional opportunity to review the specification before widespread adoption. Thus, under the DARPA SafeDocs project [5]—a project that aims to deliver tools that facilitate safe and verified parsing for extant and new data formats and protocols—we were tasked with reviewing the iccMAX calculatorElement description and identifying any security issues associated with that description. The calculatorElement was chosen because it has a computational interpretation as a stack machine program with security implications. Indeed, a 2020 ICC whitepaper [11] cites security and predictability as foci during the design of the iccMAX calculatorElement, and it provides notes on how to properly parse, validate, and apply a calculatorElement to achieve these aims. The purpose of the security evaluation presented in this paper was to provide suggestions on how the specification could be amended to better achieve these security and predictability goals—and it entailed: identifying potential unintended computational capabilities; ensuring the consistency of the calculatorElement description; checking for ambiguities and typos; suggesting improvements that would make the specification more amenable to secure parser implementations; and evaluating a provided demo implementation. In addition to reading the relevant parts of the specification, we used a variety of approaches and tools to capture and analyze the specification. In this paper, we document these efforts and present our findings.

Some appreciation of why this work is important can be found in the recently discovered vulnerabilities in parsers developed for ICC profiles. In 2021 alone, three CVEs were found in various implementations of ICC profile parsers. These vulnerabilities rendered several versions of Apple's iOS and Mac OSX operating systems vulnerable to crafted image attacks [6], [17], [24]. Attackers could use images containing embedded crafted ICC profiles to run arbitrary code on a target machine. These sorts of vulnerabilities often stem from security issues with the specification or mismatches between the specification and the parser implementation—the sorts of things that our work in this paper seeks to address.

This paper presents the collective work of our three teams—BAE, Galois, and SRI/Dartmouth—on analyzing the iccMAX calculatorElement. We applied an assortments of tools and techniques with the aims of identifying ambiguities and errors within the specification, suggesting ways in which the specification could be improved, and showcasing our tools. These included the proof assistant PVS [25], the theorem-proving language ACL2 [19], the data description language DaeDaLus [13], and tools tied to the Parsley data description language [23] (Parsley itself, Parsley/Rust combinators, and

Table I
KEY FINDINGS OF THIS PAPER

Key Findings	Type	Techniques used
A Resource Contract for iccMAX	Specification improvement	Parsley, DaeDaLus
Conditional Operations are Insufficiently Defined	Specification correction	DaeDaLus, Parsley/Rust
Sub-element Type Mappings Missing	Specification correction	Parsley/Rust
Operators Missing in Specification	Specification correction	Parsley, DaeDaLus, Parsley/Rust
Incorrect Sub-element Index Implementation	Specification correction	Parsley/Rust
Non-numeric Values Allow Parser Differentials	Specification improvement	DaeDaLus
Minor Errors and Issues	Specification correction	Parsley, DaeDaLus, ACL2, PVS, Parsley/Rust

a static analyzer). Table I summarizes the errors we found in the iccMAX specification and demo implementation, as well as our proposals to improve upon the specification.

Contributions. The work presented here will be of the greatest value to researchers and practitioners who focus on specifications, data description languages, and parsers. To this end, our main contributions can be summarized as follows:

- We conduct a multi-dimensional case study that utilizes formal methods and parsing tools to analyze the iccMAX specification.
- We propose augmenting the specification to provide a complete resource contract for the iccMAX calculatorElement, which specifies the required resources to invoke the calculatorElement.
- We present mitigations for specification errors of varying severity, and we provide suggestions for improving the iccMAX specification. Additionally, we demonstrate the value of applying formal approaches to verifying the correctness of specifications, and we argue for similar approaches to be employed before specifications are made public.
- Finally, this case study delivers numerous guiding principles for the design of other specifications, data description languages, and parsing methodologies. For a couple of examples, we argue that parsing should be limited via buffers and that specifications should specify operation semantics.

Organization. The remainder of this paper is structured as follows: Section II serves as a background section; we discuss the development of the ICC and the the iccMAX profile, introduce the SafeDocs project, and provide an overview of the exercise of reviewing the iccMAX calculatorElement. Section III provides a primer on the iccMAX calculatorElement, delving into the technical details. In Section IV, we document the analytical tools and approaches we used. Section V documents the security issues we found with iccMAX, as well as improvements to the Parsley language as a byproduct of this experiment. Section VI provides general takeaways and Section VII concludes. We hope the paper, in full, serves as an interesting case study for those who are interested in seeing how various tools can uncover issues in a real-world specification that was crafted with care; however, for those who want only to have a bird's-eye view of our work and to understand the takeaways, we recommend skipping Sections III and IV.

II. BACKGROUND & CONTEXT

This section provides the necessary background context to understand our work. We provide a brief history of the International Color Consortium, explain the problem that ICC profiles aim to solve, discuss the DARPA SafeDocs project, and provide an overview of our work.

A. A Brief History of the ICC and the ICC Profile

In the early 1990s, new applications involving digital color processing emerged on open computing environments, notably the web [30]. Traditional industries (e.g., broadcast television, still photography) that had to deal with color reproduction were now in a predicament [30]. On the one hand, they had to work with each other over these open computing environments, driving a need for an open standard for digital color reproduction [30]. On the other hand, dominant companies within these industries stymied early attempts at the development of an adequate standard, which were, according to Stokes [30], driven by intellectual property concerns.

Finally, in 1993, a fledgling organization emerged, one that would ultimately develop an open standard for digital color reproduction by "circumvent[ing] the standards processes" that were so heavily influenced by corporate processes [30]. This organization was the International Color Consortium (ICC); it comprised eight industry vendors—Adobe, Agfa, Apple, Kodak, Microsoft, Silicon Graphics, Sun Microsystems, and Taligent [33]—that banded together for the purpose of "creating, promoting and encouraging the standardization and evolution of an open, vendor-neutral, cross-platform color management system architecture and components" [7]. By the following year, this collaborative effort produced the first ICC profile specification, which was based on the Apple ColorSync profile [30], [33].

ICC profiles enable seamless and faithful transition of color data between device operating systems and applications, providing value to both vendors and users [7]. An ICC profile captures the color properties of a device, such as a monitor, printer, scanner, or camera [8]. It does this by creating a mapping between the device's color space and a

reference color space called the Profile Connection Space (PCS) [8]. The PCS effectively serves as the connective tissue joining source and target devices, alleviating the complexity of having to define direct color mappings between pairs of devices [8].

The ICC profile specification has continually evolved since its 1994 debut; the current iteration is the ICC v4 profile [8], which was released in 2010. More recently, in 2019, the ICC has developed the iccMAX specification [9], a significant extension to ICC v4. We reiterate that iccMAX does not replace ICC v4, but rather it *expands* upon it. Thus, capturing iccMAX's calculatorElement necessitates understanding and capturing the relevant parts of the ICC v4 profile specification. We also note that there have been corrections to the iccMAX specification since its creation, which are documented in the cumulative errata list [10].

B. The Design of the iccMAX calculatorElement

In this subsection, we touch on how iccMAX diverges from previous iterations of the ICC specification, and we present an overview of a whitepaper outlining secure implementation notes on the iccMAX calculatorElement. This section is heavily based on a 2020 ICC whitepaper on securely implementing the iccMAX calculatorElement [11].

The iccMAX specification expands upon previous ICC versions by providing profile creators a mechanism to specify sequences of processing elements for encoding color transforms. This capability is vital in complex color management scenarios where achieving reliable color reproduction requires specifying multi-channel non-linear functions. Whereas previous ICC specifications employed a lookup-table–based approach to achieve this task, iccMAX allows for accurate and precise specification of color transforms through the calculatorElement. As noted in the whitepaper, the calculatorElement was very much designed with security and predictability in mind.

The iccMAX calculatorElement contains a main function that specifies a sequence of operators to carry out. In addition to operators, the calculatorElement supports the use of multiple input, output, and temporary channels, as well as sub-element invocations, and the reading of CMM (color management module) environment variables. During application, the operations are carried out in sequence; they may put data on the stack, extract data from the stack, or otherwise modify the stack. Operations may also involve reading from input channels, writing to output channels, reading from or writing to temporary channels, reading CMM environment variables, or invoking sub-elements. A more thorough description of iccMAX and the iccMAX calculatorElement can be found in Section III.

The whitepaper provides essential security notes on the parsing, validation, and application of the calculatorElement:

- *Parsing* the calculatorElement requires ensuring that the calculatorElement structure is in conformance with the iccMAX specification—and ensuring conformity of substructures as well. Failure to ensure conformance may introduce memory corruption issues during the parsing process.
- *Validating* the calculatorElement involves ensuring that the application of the main function will be performed in a safe and secure fashion. As the calculatorElement revolves around the main function's operators, most validation efforts focus on validating the operations by: ensuring the CMM that invokes the calculatorElement supports all operations; ensuring the validity of input, output, and temporary channel accesses; ensuring valid stack manipulation (e.g., no stack underflow), and ensuring valid sub-element addressing.
- *Application* of the calculatorElement should be reliable and predictable. This is achieved in part by the aforementioned validation process. But runtime requirements such as correct initialization processes (e.g., the data stack should initially be empty) and correct operation behavior in accordance with the specification are also critical.

C. The DARPA SafeDocs Project

The DARPA SafeDocs project aims to tackle the security problems that arise from manually written parser code and from the inherent complexity of data format specifications [5]. To achieve this end, the SafeDocs project supports work on three primary fronts: (i) defining the de facto grammar of data formats used in the wild, (ii) identifying a useful but simple subset of this grammar that is conducive to the generation of secure parsers, and (iii) developing tools for creating secure parsers from grammars [5]. The ultimate goal of the SafeDocs project is to improve parser security by providing tools to industry programmers and specification creators that foster the construction of secure parsers and specifications.

Under the umbrella of the SafeDocs project, teams have been working on representing extant and nascent data format via *data description languages*. Data description languages (or *DDLs*) are high-level languages specially designed to express data formats and protocols in a manner that eliminates some of the complexity and ambiguity found in written specifications. Additionally, many DDL toolkits support conversion of the DDL representation to a parser expressed in a programming language. The hope is that well-designed, usable DDLs will enable the production of secure parsers that users can readily adopt instead of having to craft their own by hand. In addition, researchers have pursued work that extends beyond direct DDL work, such as analyzing parsers in the wild to review specifications, identifying vulnerabilties in existing specifications and parsers, and automatically

generating parsers from sources such as document object modules.

The process of reviewing the iccMAX calculatorElement strongly aligns with the aims of the SafeDocs project in that it demonstrates how an assortment of SafeDocs tools and approaches can readily be applied to improving the security of a recently developed specification description and demo implementation.

D. The Exercise

While ICC profiles are ubiquitous, the iccMAX specification is still recent, and iccMAX profiles have not yet achieved widespread deployment. As such, analyzing the specification can prove useful in discovering and resolving security issues at a relatively early stage. With this intent, we carried out the exercise of analyzing the iccMAX specification. In particular, the primary goal of the exercise was to ensure that the specification was written in a manner such that the security and predictability aims documented in the aforementioned whitepaper [11] (see Section II-B) could be met.

The work presented in this paper covers the cumulative efforts of our three teams—the BAE team, the Galois team, and the SRI/Dartmouth team—to analyze the iccMAX calculatorElement in 2021. Although there was some communication between the teams, for the most part, our teams worked separately on reviewing and analyzing the specification. The general approach involved each team applying their tools with the aim of identifying as many security issues as possible.

Our teams utilized an assortment of tools and approaches. We used data description languages, parser combinators, and a static analyzer to capture the relevant portions of the specification and to evaluate how well iccMAX files conform to the specification with the aim of comparing the de facto grammar to the actual specification grammar. We used proof checking systems to identify flaws in the specification. And, of course, we reviewed the specification directly. Collectively, our efforts have uncovered security bugs, produced proposals for specification improvements, and suggested ways to improve our own tools.

III. The iccMAX calculatorElement

This section provides the requisite information to understand the technical details of the iccMAX calculator element and the higher-level parsing of the the iccMAX profile for accessing its calculator elements.

The iccMAX profile has a sort of compositional quality. At the top level, the iccMAX profile comprises three parts: a profile header, a tag table that contains entries pointing to sub-elements, and tagged element data wherein those sub-elements live. Many sub-elements can, in similar fashion, be recursively deconstructed to expose lower-level structures. We will take you along the path from the top-level view of the iccMAX profile down to the calculatorElement, and then we will present the calculatorElement itself.

Table II
Shorthand Specification Referencing Notation

Notation	Description
(iccMAX-Clause-C)	clause C of iccMAX [9]
(iccMAX-Figure-F)	figure F of iccMAX [9]
(iccMAX-Table-T)	table T of iccMAX [9]
(ICCv4-Clause-C)	clause C of ICC v4 [8]
(ICCv4-Figure-F)	figure F of ICC v4 [8]
(ICCv4-Table-T)	table T of ICC v4 [8]

A. Preliminaries

This subsection provides preliminary notes and notation to help the reader process the upcoming presentation of iccMAX and the iccMAX calculatorElement.

We developed shorthand notation for referring to the specification. This shorthand notation is explained in Table II. The second column of the table contains the names of types used in the specification, and the third and fourth columns briefly describe the types and provide pointers to where further information about the types can be found in the specification respectively. The first column specifies the names that we use in this paper to refer to those same types.

As noted, the iccMAX specification builds upon ICC v4 [8]. The relevant types are presented in Table III.

We also note that both the ICC v4 and iccMAX specifications exclusively use the big-endian byte order.

B. The Profile Structure

The iccMAX profile structure *(iccMAX-Clause-7)* exists at the top level and comprises three parts. This structure is captured pictorially in Figure 1. Our focus is solely on validating the calculator element so the profile structure and other high-level structures are only relevant insomuch as they allow us to access calculator elements.

The first part of the profile structure, the iccMAX profile header, is made up of multi-byte fields, each of which is between 4 and 16 bytes. They specify the profile size, version information, device information, and so forth. The iccMAX profile header is out of scope for our work, which focuses primarily on the calculator element.

The second part, the tag table, begins with a 4-byte value called *tag count* that is referred to simply as n in the context of this portion of this specification. This is followed by a series of *tag entries*. These tag entries specify *tag data elements* in the third part, the tagged element data. Each tag entry contains a 4-byte *tag signature*, followed by a 4-byte *offset* to the beginning of the tag data element (encoded as a UInt32) relative to the beginning of the profile header, followed by a 4-byte *tag data element size* (encoded as a UInt32).

Table III
ICC TYPES

Our Notation	Specification Notation	Description	Specification Reference
UInt8	uInt8Number	an unsigned 8-bit integer	*(ICCv4-Clause-4.13)*
UInt16	uInt16Number	an unsigned 16-bit integer	*(ICCv4-Clause-4.10)*
UInt32	uInt32Number	an unsigned 32-bit integer	*(ICCv4-Clause-4.11)*
UInt64	uInt64Number	an unsigned 64-bit integer	*(ICCv4-Clause-4.12)*
Float32	float32Number	a 32-bit floating-point number	*(ICCv4-Clause-4.3)*
positionNumber/posNum	positionNumber	an 8-byte field containing a UInt32 offset followed by UInt32 size	*(ICCv4-Table-2)* in *(ICCv4-Clause-4.4)*

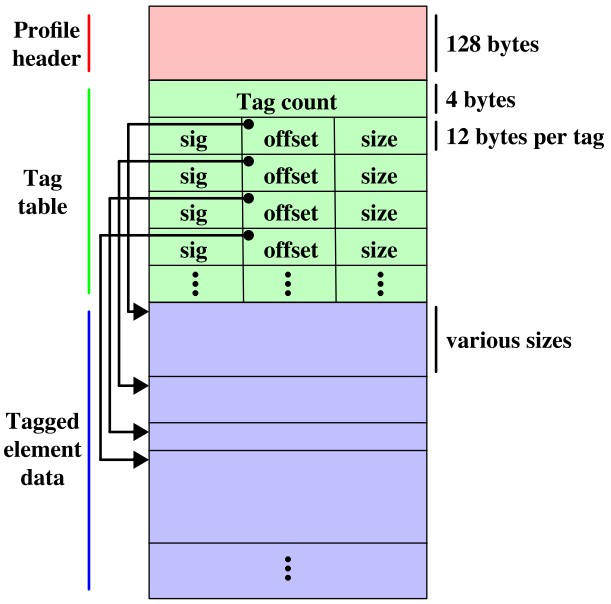

Figure 1. The profile structure, recreation of *(iccMAX-Figure-5)*.

The third and final part of the profile structure is the tagged element data. The tagged element data comprises the tag data elements to which the tag entries correspond.

The specification also notes that:

- all profile data is encoded using the big-endian byte order
- the first tagged element immediately follows the table
- all tagged element data shall be padded to reach a 4-byte boundary—and this padding must not exceed three bytes.
- all pad bytes must be the NULL character.

C. Tags & Tag Types

We restrict our focus to public tags. Each tag has a unique *tag signature*, which is used to specify the tag itself. There is also a set of permitted *tag types* that may be used for a tag. The permitted tag types for each tag are specified by *(iccMAX-Clause-9)*. For example the tag `AToB0Tag` has tag signature `A2B1`, and its permitted tag types are `lutAToBType` and `multiProcessElementsType`. Each tag data element must have a tag type that is permitted by the tag specified by the tag signature of the element's tag entry. There is more to say about tags—private tags, required tags, etc.—but these topics are out of scope.

D. The multiProcessElementsType Encoding

The only tag type that we are interested in is the MultiProcessElementsType (*MPET*) tag type, which is described in *(iccMAX-Clause-10.2.16)* as "represent[ing] a colour transform, containing a sequence of processing elements." For convenience, we have reproduced *(iccMAX-Table-63)*, which captures the MPET encoding, in Table IV.

The MPET encoding begins with four bytes, representing the 'mpet' type signature, followed by four bytes of zeros. The following two fields are of type UInt16 and correspond to the number of input and output channels respectively. The field that follows is of type UInt32, and it captures the number of processing elements contained within the structure. This is immediately followed by a process element positions table, which specifies, for each processing element, the offset of the processing element (relative to the start of the MPET tag) and the size of the processing element via a positionNumber type. Last, we have the data area, which is where the actual processing elements live.

The processing elements should be processed in the exact same order as they appear within the processing elements position table. They are processed in series, strung together by the input and output provided via input and output channels. Specifically, the output channels for a given processing element should match the input channels for the subsequent element. Additionally, the number of input channels specified in the MPET encoding should match that of the first processing element—and the number of output channels specified in the MPET encoding should match that of the last element.

There are also a number of other constraints and notes regarding the MPET encoding:

- There must be at least one processing element (i.e., N must be at least 1).
- Each processing element must end on a 4-byte boundary or be followed by up to three 00h pad bytes to

Table IV
MULTIPROCESSELEMENTSTYPE ENCODING
REPRODUCTION OF *(iccMAX-Table-63)* WITH ENTRY FOR BYTE POSITIONS COLUMN FIXED ON PENULTIMATE ROW.

Byte Positions	Field Length (Bytes)	Content	Encoded as...
0...3	4	'mpet' (6D706574h) type signature	
4...7	4	Reserved, shall be 0	
8...9	2	Number of input channels (f)	UInt16
10...11	2	Number of output channels (T)	UInt16
12...15	4	Number of processing elements (N)	UInt32
16...16+8N-1	8	Process element positions table	Array of positionNumber
16+8N...end	Data		

reach a 4-byte boundary.
- The processing elements needn't be mutually exclusive; data may be shared between them.

The multipleProcessElementsType may contain the processing elements specified in *(iccMAX-Clause-11)*. However, the only element of interest to us is the calculatorElement.

E. The calculatorElement

The iccMAX calculatorElement is captured pictorially in Figure 2 and as a table in Table V. It contains a main function, encoded in Table VI, which contains a sequence of *operations* that use Reverse Polish Notation that are processed using a stack machine.[1]

The calculatorElement supports input channels, output channels, and temporary channel storage. This temporary channel storage is effectively additional storage that may be used within the calculatorElement, though it may *not* be passed between MPET elements, including MPET sub-elements. Data may be stored within the temporary or output channels. Temporary channel data is assumed to be initialized to zero during every invocation of a calculatorElement or sub-calculator element. Similarly, output channels should be set to zero until data is stored in them by the main calculator function. The specification also states that the maximum total number of channels—the sum of input channels, output channels, and temporary channels—is 65,535.

The main function should be validated. Every operation specified in the main function should be valid. Channel indexing should be valid. And the stack should never underflow or overflow. The amount of reserved storage for the stack must also be able to hold at least 65,535 values.

1) The calculatorElement Operations: The operations of the calculator are essentially instructions for a stack machine—i.e., they involve retrieving data from the stack and placing data on the stack. In addition to real values, non-real numbers—+INF, -INF, and NaN—may be placed on the stack. Independent of the specific values on the stack and whether or not those values are real values, the number of stack values consumed and produced by a given operation must be in accordance with the description of the operation in the specification.

The iccMAX calculatorElement operations have been clearly designed for quick validation and the prevention of arbitrary control flow. As a result, there are no looping or recursive control structures, but there is an invocation mechanism similar to a function call. However, validation is made more complicated by the choice to allow different branches of conditionals to have different effects on the operand stack. This choice differs from the choice made in WebAssembly [26], which has a similar stack machine and a similar design goal for quick validation.

These stack operations are encoded using eight bytes: a four-byte signature followed by four bytes of data. The operations are grouped into the following types: floating point constant operations, channel vector operations, CMM environment variable operations, sub-element invocation operations, stack operations, matrix operations, sequence functional operations, function vector operations, and conditional operations. Below, we cover a subset of these operation classes, as well as a few actual operations.

Floating Point Operation *(iccMAX-Clause-11.2.1.2)*. The floating point constant operation (Table VII) takes a supplied Float32 and pushes it onto the stack.

Channel Vector Operations *(iccMAX-Clause-11.2.1.3)*. The channel vector operations provide a communication pathway between the different channels—input, output, and temporary channels—and the evaluation stack. Table VIII captures the encoding of a channel vector operation. Table IX describes each of these channel vector operation signatures.

The CMM Environment Variable Operation *(iccMAX-Clause-11.2.1.4)*. Recall that the PCS is a reference color space. The actual conversion of color information between the device and the PCS is done via a CMM (color management module) *(ICCv4-Clause-0.6)*. The CMM environment variable operation is used to push a CMM environment variable onto the stack.

Sub-Element Invocation Operations *(iccMAX-Clause-11.2.1.5)*. Sub-element invocation operations provide a

[1]For example, 1 2 + 3 − in Reverse Polish Notation is the same as (1 + 2) − 3 in infix notation. This expression evaluates to 0 when the integers are expressed in decimal representation. For a primer on Reverse Polish Notation, see [34].

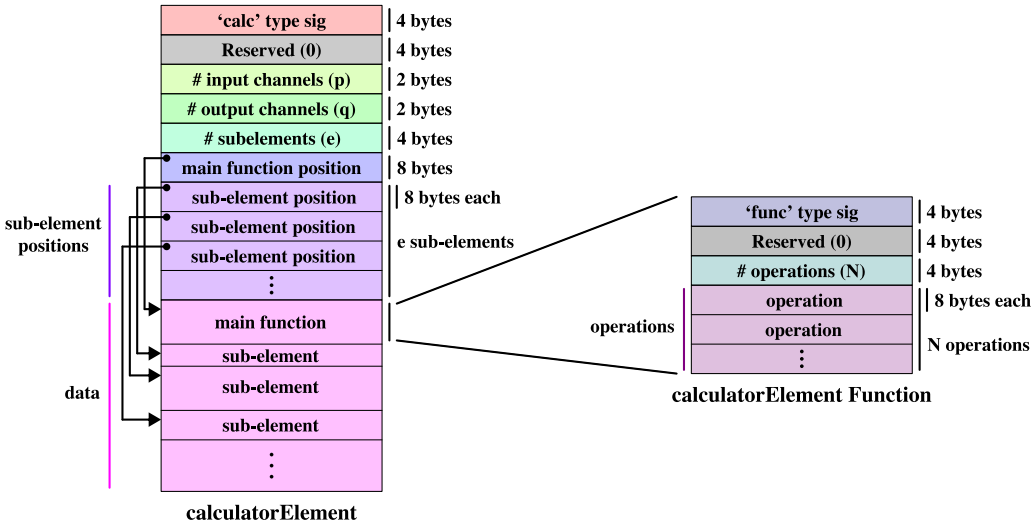

Figure 2. The calculatorElement and calculatorElement Function encodings. Corresponds to the descriptions in *(iccMAX-Clause-11.2.1.1)*).

Table V
CALCULATORELEMENT ENCODING
REPRODUCTION OF *(iccMAX-Table-85)*.

Byte Positions	Field Length (Bytes)	Content	Encoded as...
0...3	4	'calc' (63616C63h) type signature	
4...7	4	Reserved, shall be 0	
8...9	2	Number of input channels (P)	UInt16
10...11	2	Number of output channels (Q)	UInt16
12...15	4	Number of sub-elements (E)	UInt32
16...23	8	Main function position	positionNumber
24...24+8*E-1	8*E	Sub-element positions	Array of positionNumber
24+8*E...end		Data for calculator element	

Table VI
CALCULATORELEMENT FUNCTION ENCODING
REPRODUCTION OF *(iccMAX-Table-86)*.

Byte Positions	Byte Length	Content	Encoding
0...3	4	'func' (66756e63h) type signature	
4...7	4	Reserved, shall be 0	
8...11	2	Number of operations (N)	UInt32
12...12+8N-1	8N	Function operations	

Table VII
FLOATING POINT CONSTANT OPERATION ENCODING
REPRODUCTION OF *(iccMAX-Table-87)*.

Byte Positions	Byte Length	Content	Encoding
0...3	4	'data' (64617461h)	
4...7	4	# to put on stack	Float32

mechanism for calling other, separate processing elements during calculatorElement operation. This is achieved by a clever utilization of the stack as a communication medium. The input channels for the sub-element correspond to the state of the evaluation stack before the invocation, and the output channels are used to reconstruct the stack after the invocation. Table X provides the encoding for sub-element invocation operations and Table XI describes each sub-element invocation operation.

Table VIII
CHANNEL VECTOR OPERATION ENCODING
REPRODUCTION OF *(iccMAX-Table-88)*.

Byte Positions	Byte Length	Content	Encoding
0...3	4	Operation signature	
4...5	2	Starting index (S)	UInt16
6...7	2	Additional count from start (T)	UInt16

Stack Operations *(iccMAX-Clause-11.2.1.6)*.

The stack operations are limited to stack manipulation; they do not involve reading from or writing to channels. They involve operations like `copy`, which is used to duplicate stack elements, and `flip`, which is used to reverse the ordering of some of the topmost elements on the stack. The encoding for stack operations is provided in Table XII and descriptions of the operations are provided in Table XIII

Matrix Operations *(iccMAX-Clause-11.2.1.7)*.

The matrix operations involve specifying a matrix (or a matrix equation) using the topmost values on the stack,

Table IX
CHANNEL VECTOR OPERATIONS BY SIGNATURE
REPRODUCTION OF *(iccMAX-Table-89)*.

Op. Sig.	Stack Args.	Op. Defn.	Stack Results
'in ' (696e2020h)	None	Load from input pixel channel number S through S+T	in[S]...in[S+T]
'out ' (6f757420h)	A_0...A_T	Store to output pixel channel number S through S+T. Thus: out[S]=A_0, ..., out[S+T]=A_T	None
'tget' (74676574h)	None	Get temporary channels S through S+T	temp[S]...temp[T]
'tput' (74707574h)	A_0...$_T$	Put temporary channels S+T through S. Thus: temp[S]=A_0, ..., temp[S+T]=A_T	None
'tsav' (74736176h)	$_0$...$_T$	Saves arguments on stack as temporary channels S+T through S without affecting arguments on the stack. Thus: temp[S]=$_0$, ..., temp[S+T]=$_T$	$_0$,...A_T

Table X
SUB-ELEMENT INVOCATION OPERATION ENCODING
REPRODUCTION OF *(iccMAX-Table-93)*.

Byte Positions	Byte Length	Content	Encoding
0...3	4	Operation signature	
4...7	4	Element index	UInt32

performing some computation using those values, and then saving the result back onto the stack. The two matrix operations correspond to solving a matrix vector equation and transposing a matrix.

Sequence Functional Operations *(iccMAX-Clause-11.2.1.8)*.

The sequence functional operations work by specifying a sequence of values at the top of the stack and then applying a function to those values, saving the result onto the stack. The operations correspond to computing the sum, the product, the minimum value, the maximum value, the logical `and`, and the logical `or` of a sequence specified on the stack.

Functional Vector Operations *(iccMAX-Clause-11.2.1.9)*.

There are a large number of functional vector operations that use the values at the top of the stack to specify at least one vector, potentially alongside other values, and perform computation using those values; the result of the computation, which is usually a vector, is saved on the stack. In addition, there are a few operations for pushing special values onto the stack: Π, +INF, -INF, and NaN.

Table XI
SUB-ELEMENT INVOCATION OPERATIONS BY SIGNATURE
REPRODUCTION OF *(iccMAX-Table-94)*.

Op. Sig.	Stack Args.	Op. Defn.	Stack Results
'curv' (63757276h)	X_1...X_{Input}	Applies sub-element (S) as a curve set	Y_1...Y_{Output}
'mtx' (6d747820h)	X_1...X_{Input}	Applies sub-element (S) as a matrix	Y_1...Y_{Output}
'clut' (636c7574h)	X_1...X_{Input}	Applies sub-element (S) as a CLUT	Y_1...Y_{Output}
'calc' (636c7574h)	X_1...X_{Input}	Applies sub-element (S) as a calculator	Y_1...Y_{Output}
'tint' (74696e74h)	X_1...X_{Input}	Applies sub-element (S) as a tint	Y_1...Y_{Output}
'elem' (656c656dh)	X_1...X_{Input}	Applies sub-element (S)	Y_1...Y_{Output}

Table XII
STACK OPERATION ENCODING
REPRODUCTION OF *(iccMAX-Table-95)*.

Byte Positions	Byte Length	Content	Encoding
0...3	4	Operation signature	
4...5	2	Number of extra elements selector S	UInt16
6...7	2	Number of extra times selector T	UInt16

The functional vector operation encoding is provided in Table XIV and descriptions of a subset of vector operations is provided in Table XV.

Conditional Operations *(iccMAX-Clause-11.2.1.10)*.

The conditional operations—`if` and `else` (which must be combined with an `if` operation)—involve comparing the topmost stack value to 0.5 to generate a truth value (true if the value is greater and false otherwise) and then using that truth value to conduct (or skip) a specified number of operations on the associated operation stream.

Selection Operations *(iccMAX-Clause-11.2.1.11)*.

Last, the selection operations (`sel`, `case`, and `dflt`) are used in conjunction to specify a selection seqeunce that uses the topmost value on the stack to choose specify one of potentially several sequences of operations to carry out. Similar to the conditional operations, these allow for the specification of different branches of operations to perform based on the topmost stack value.

F. A Motivating Example

We use a motivating example of a calculator element represented by the hex-coded string below. Spaces and line breaks have been inserted for readability, but otherwise the input shown is a string of bytes where each byte is represented by two hexadecimal digits.

The breakdown of the format is shown in Figure 3. The

Table XIII
STACK OPERATIONS BY SIGNATURE
REPRODUCTION OF *(iccMAX-Table-94)*.

Op. Sig.	Stack Args.	Op. Defn.	Stack Results
'copy' (636f7079h)	A_0...A_S	Duplicate top S+1 elements T+1 times (stack results shown for T=0)	A_0...A_S A_0...A_S
'rotl' (726f746ch)	A_0...A_S	Rotate left top S+1 elements T+1 positions on stack (stack results shown for T=0)	A_1...A_S A_0
'rotr' (726f7472h)	A_0...A_S	Rotate right top S+1 elements T+1 positions on stack (stack results shown for T=0)	A_S A_0...A_{S-1}
'posd' (706f7364h)	A_S...A_0	Duplicate the element at the Sth position from top of stack T+1 times (stack results shown for T=0)	A_S...A_0 A_S
'flip' (666c6970h)	A_0...A_{S+1}	Reverse the top S+1 elements on the stack (T shall be zero)	A_{S+1}...A_0
'pop' (706f7020h)	A_0...A_S	Remove top S+1 elements on the stack (T shall be zero)	

Table XIV
FUNCTIONAL VECTOR OPERATION ENCODING
REPRODUCTION OF *(iccMAX-Table-101)*.

Byte Positions	Byte Length	Content	Encoding
0...3	4	Operation signature	
4...5	2	Vector index selector S	UInt16
6...7	2	Reserved, shall be 0	

string starts with a tag which is the ASCII representation of 'calc'. The 'Reserved' bytes must be zeros. The P field is the number of input channels, the Q field is the number of output channels, and the E field is the number of sub-elements. All three fields are unsigned 32-bit integers. The calculator element takes three inputs and writes three outputs. The next

```
63616C63 00000000 0003 0003
00000000 00000018 0000003c
66756e63 00000000 00000006
696e2020 00000002
64617461 3f800000
64617461 3f800000
64617461 3f800000
61646420 00020000
6f757420 00000002
```

Table XV
A SUBSET OF FUNCTIONAL VECTOR OPERATIONS BY SIGNATURE
REPRODUCTION OF *(iccMAX-Table-102)*.

Op. Sig.	Stack Args.	Op. Defn.	Stack Results
'pi' (70692020h)	None	Mathematical value Π (S shall be zero)	Π
'+INF' (2b494e46h)	None	Floating point value for positive infinity (S shall be zero)	+INF
'-INF' (2d494e46h)	None	Floating point value for negative infinity (S shall be zero)	-INF
'NaN' (4e614e20h)	None	Floating point value for "Not a Number" (S shall be zero)	NaN
'add' (61646420h)	X_0...X_S Y_0...Y_S	Z_i = X_i Y_i (for i=0...S)	Z_0...Z_S

Table 85	`63616C63` ('calc')
Reserved	`00000000`
P	`0003`
Q	`0003`
E	`00000000`
Main position/size	`000000180000003c`
Table 86	`66756e63` ('func')
Reserved	`00000000`
N	`00000006` (6)
Table 88, 89	`696e2020` ('in ')
S	`0000`
T	`0002`
Table 87	`64617461` ('data')
Datum	`3f800000`
Table 87	`64617461` ('data')
Datum	`3f800000`
Table 87	`64617461` ('data')
Datum	`3f800000`
Table 101, 102	`61646420` ('add ')
S	`0002`
Reserved	`0000`
Table 88, 89	`6f757420` ('out ')
S	`0000`
T	`0002`

Figure 3. An Example iccMAX Calculator Element

field is an unsigned 64-bit integer representing the position of the main code for the calculator element. Since there are no sub-elements, the Main position is immediately followed by the Main code which is a `func` element. It has six operations: an `in` operation that loads three values ($in[S]$, $in[S+1]$, $in[S+2]$) from the input channels onto the stack, followed by three `data` operations that each push a single 32-bit floating point constant onto the stack, followed by an `add` operaton that performs a pointwise addition of the floating point constants to each of the inputs. The final step is an `out` operation that moves the three values on the stack resulting from the `add` operation to the output channels ($out[S]$, $out[S+1]$, $out[s+2]$). The output channel values are eventually pushed to the caller's stack for further processing.

IV. Analytical Tools & Approaches

Our teams—BAE, Galois, SRI/Dartmouth—used a variety of approaches and tools to analyze the iccMAX specification, including the proof assistant PVS [25], the theorem-proving language ACL2 [19], the data description language DaeDaLus [13], and tools tied to the data description language Parsley [23] (Parsley itself, Parsley/Rust combinators, and a static analyzer). This exercise and all its parts—the diverse expertise of our teams, the complementary nature of these instruments, and the fact that applying these instruments necessitated a close read of the specification—led to the discovery of numerous findings, which are discussed in Section V. We communicated these findings to the International Color Consortium via Peter Wyatt as an intermediary. In the following subsections, we introduce these instruments and briefly explain how we used them to analyze the iccMAX calculatorElement.

A. PVS

SRI's Prototype Verification System (PVS) [25] is an interactive proof assistant for mathematical and computational formalization. It has been used extensively for large-scale formal verification projects spanning hardware processor correctness, compilers, distributed systems, and air-traffic collision detection and resolution algorithms. PVS is primarily used for defining and analyzing mathematical models in higher-order logic. In many cases, these models are executable as functional programs, and PVS can be used to generate efficient code directly from the models. We used PVS to formalize a recursive descent parser/analyzer for the iccMAX calculatorElement. The exercise revealed a number of issues in the iccMAX specification.

The parser is written in terms of three functions: `parsecalc`, `parsefunc`, and `parseOperation`. The `parsecalc` operation parses a window `w` within string `s` for a calculator element *(iccMAX-Clause-85)* to return either `undefined`, signaling a parse failure, or a channel signature consisting of a pair of input and output channel sizes. The `parsecalc` opertion is invoked recursively on the sub-elements. The window consists of a pair of unsigned 32-bit integers `startpos` and `endpos`. The size of the window `endpos` − `startpos` must be at least 24 bytes to accomodate

1) The 2-byte tag
2) The 4-byte reserved word
3) The 2-byte fields P and Q representing the number of input and output channels, respectively
4) The 4-byte E field representing the number of sub-elements, and
5) The 8-byte integer `M` representing the position and size of the main function element.

The `parsefunc` operation parses the main function component *(iccMAX-Clause-86)* of the calculator element within a window `w` of the input string `s` with respect to a sub-element signature table mapping sub-element positions to the input/output channel signature, i.e., the number of input and output channels. The `parsefunc` operation reads the tag, the reserved unsigned 32-bit parameter S, and the unsigned 32-bit number N representing the number of operations. The sub-element signature is a map from possible offset positions in the calculator element to the sub-element definitions. The `parsefunc` operation applies `parseOperation` over the window consisting of the start position at the current start position offset by 12 bytes, and the end position offset by $8N$ bytes from the start of the window. The PVS definition of `parsefunc` is shown in Figure 4. The definitions of `parsecalc` and `parseOperation` are similar, but longer.

The `parseOperation` operation is invoked on the body of a function to parse the instructions or operations in the body. In addition to the input string, window, and the sub-element signature, this operation takes two additional arguments representing the minimum and maximum sizes of the incoming stack. These are used to ensure that an operation does not underflow or overflow the stack. The classes of operations parsed include data *(iccMAX-Clause-87)*, channel vector operations *(iccMAX-Clause-88,89)*, environment variable operation *(iccMAX-Clause-90,91)*, sub-element invocations *(iccMAX-Clause-93,94)*, stack operations *(iccMAX-Clause-95,96)*, matrix operations *(iccMAX-Clause-97,98)*, sequence functional operations *(iccMAX-Clause-99,100)*, functional vector operations *(iccMAX-Clause-101,102)*, and conditional operations *(iccMAX-Clause-103,104)*. The selection operation *(iccMAX-Clause-105,106)* was omitted from the specification. For each operation, `parseOperation` checks for the absence of stack underflow and overflow, and computes the outgoing minimum/maximum stack size to be used by the succeeding operation. The primary reason for minimum/maximum bounds on the stack size is the asymmetry in the conditional expressions where one branch can consume and produce a different number of stack elements than the other branch. We did not check the validity of channel accesses, though this could easily be added to the specification.

The PVS specification was checked for numeric overflow/underflow, termination, and type correctness. Since the parsing operations are themselves specifications, no specific properties were verified on these specifications. Executable parsing code was generated from the PVS specification in both Common Lisp and C. Once the calculator element is given an operational semantics, it will be possible to verify that the minimum/maximum stack size bounds are in fact valid. The formalization of the specification in PVS highlighted several issues with the standard, most of which were typos and missing checks. We validated all the sub-element

positions and sizes extensively to ensure that references and buffers were valid. This led to over 200 proof obligations, most of which are trivially proved.

```
parsefunc((stackLimit: uint64 | 65535 <= stackLimit),
          s: bytestring,
          subelemsig: [uint32 -> signature],
          w: window(s`length)
          ) : goodresult(stackLimit) =
 (LET endpos = w`endpos,
      startpos = w`startpos
     IN
     (IF endpos < 12
      THEN error(UnexpectedEOF, startpos, 0)
      ELSIF endpos - 12 <= startpos
      THEN error(UnexpectedEOF, startpos, 0)
      ELSE
      (LET cur = startpos,
           tag = readU32(s, cur),
           S = readU32(s, cur + 4),
           N = readU32(s, cur + 8),
           B: uint32 = startpos + 12
       IN IF N > u32div(endpos - B, 8)
          THEN error(UnexpectedEOF, startpos, 0)
          ELSE
          LET E = B + (8 * N)
          IN IF tag = 0x66756e63
             THEN IF S = 0
                  THEN
                   IF  E <= endpos
                   THEN
                   parseOperation(stackLimit,
                                  s,
                                  subelemsig,
                                  0, 0,
                                  w WITH [`endpos := E,
                                          `startpos := B]
                                 )
                   ELSE error(UnexpectedEOF, startpos, 0)
                                  ENDIF
                  ELSE error(nonZero, startpos, 0)
                  ENDIF
             ELSE error(badTag, startpos, 0)
             ENDIF
          ENDIF)
      ENDIF))
```

Figure 4. The Function Element parser in PVS is defined as a function that reads the input string `s`, checks the function header (tag, reserved field `S`, number of operations `N`, and the beginning position `B`), and parses the individual operations within the window `[B, E]` using `parseOperations`. The definition is checked for the absence of underflows/overflows, and out-of-bounds accesses by generating and proving the corresponding proof obligations.

B. Formalization of Calculator Functions for ACL2 Analysis

After a profile is parsed, we can perform semantic analysis of calculatorElement operators, checking that no function or stack requirements described in Section IV-A are violated. ACL2 (A Computational Logic for Applicative Common Lisp) is a theorem-proving language based off a subset of Common Lisp [19], where the user writes functions and then proofs about those functions, which the ACL2 theorem prover checks are valid. ACL2 theorems can be considered "contracts," where the user provides input and output conditions of a function that the theorem prover verifies, where theorems about the output are guaranteed only if the input conditions are met.

We previously analyzed semantic requirements of PDF with ACL2 and the developed Tower metalanguage [20]. To analyze the diverse set of PDF objects and functions, we automatically generated ACL2 functions and theorems (to reduce human error and save time instead of writing out repetitive code) from a machine-readable DOM developed by Peter Wyatt of PDF Association [35]. The Arlington PDF DOM summarizes the rules in tabular format with parseable expressions, which can be used as input for the function generator. Our work on analyzing the iccMAX calculatorElement effectively uses the same approach, wherein we automatically generate ACL2 functions from a tabular, machine-readable representation of calculatorElement operators. Our work on analyzing the iccMAX calculatorElement effectively uses the same approach, where we automatically generate ACL2 functions and theorems from a similar tabular, machine-readable representation of calculatorElement operators.

For iccMAX, we focused on checking a series of calculatorElement operations, which we note fall into specific patterns. Therefore, the operations can be summarized into a machine-readable tabular format, and then generate ACL2 functions and proofs from them. The operations should satisfy the stack requirements of no overflow or underflow, and there should be enough arguments on the stack to perform each operation. For example, as previously mentioned, the $out(S,T)$ operation takes T elements from the stack and puts them into the output channels, which should not be allowed if there are less than T elements on the stack. There has been discussion on whether the standard should require that the missing elements be filled in as "0" or if the profile should be classified invalid.

To perform this semantic analysis, we define invariants on stack size preventing overflow or underflow, preconditions before we can apply each function, and then apply the series of operations to an empty stack. To generate the functions, we summarized the iccMAX operator descriptions, which were partially natural language, into formalized stack preconditions and stack effects.

Table XVI shows an excerpt of the iccMAX calculatorElement Operation XML.[2] For functions with identical stack requirements and effect, we combined them into a single line in the table. We use the same argument naming as the standard where the first argument of an operation is "s"

[2]The iccMAX report prepared by Peter Wyatt was internal to the SAFEDOCS program and the ICC working group and has not been made public as of yet [36].

and second argument is “t”. Based on the convention in the Arlington PDF DOM, variables are preceded by “@”.

Table XVI
CALCULATORELEMENT OPERATIONS FORMALIZED IN TABULAR FORM

Func group	# of Args	Stack Effect	Precondition	Funcs
data	1	@stacksize+1		data
in	2	@stacksize+@t		in
out	2	@stacksize - @t	@stacksize $\geq$ t	out
copy	2	@stacksize + @t*@s	@stacksize $\geq$ s	copy
rearrange	2		@stacksize $\geq$ s	rotl, rotr, flip

We then defined a stack type in ACL2 with a stack size parameter, constrained to be between 0 and 65,535. Next, we defined ACL2 functions which operate on the stack and return the new stack with a potentially updated size.

One ACL2 function is defined per aggregate function, with an example shown in Figure 5.

```
(defun stack-<NAME> (function stack)
  (if (and
        (stack-obj-p stack)
        (function-obj-p <function type>)
        <Precondition>
      )
     (stack-obj (<Operation> (stack-obj
      ->size stack) <stack size change>))
      NIL
  )
)
```

Figure 5. ACL2 Template Function Definition for Stack Operations

The stack operation function takes an instance of the function with arguments entered, as well as a stack, and checks that the input arguments are of the correct type; it then checks a precondition, if present, on the number of stack arguments. If the requirements are satisfied, the function returns a new stack object with the updated stack size. Otherwise, the function returns NIL.

Using the table, we filled in the template. For example, the copy function in ACL2 is shown in Figure 6.

Based on the XML formalization, we generated ACL2 functions with stack effects based on the ICC table by filling in a template. We then checked supplied calculator profiles [12] along with fuzzed profiles with known flaws. We set the stack size to be between 0 and 65,535, so when the stack size requirement is violated, ACL2 throws an error. If one of the stack preconditions is violated, the function call returns NIL instead of a stack.

For example, we passed the following operations into the ACL2 simulation:

```
(defun stack-copy(copy-function stack)
  (if (and
       (stack-obj-p stack)
       (function-obj-p copy-function)
         (>=
           (stack-obj->size stack)
           (function-obj->argument1 copy-function)
         )
       )
       (stack-obj
           (stack-obj->size  stack)
           (*
               (function-obj->argument1 copy-function)
               (function-obj->argument2 copy-function)
            )
           )
       )
    NIL
  )
)
```

Figure 6. ACL2 Copy Function Code Snippet

```
in [0,3]
0 0 0
mul[3]
out [0,4]
```

As there are only 3 elements on the stack when we attempt to pop 4 elements off of it, we can expect an error. As expected, we see the text below, confirming that ACL2 can be used for semantic analysis of the calculator element operations.

```
ACL2 !>VALUE (apply-icc-functions
*empty-stack* *func-list2*)

ACL2 Error in TOP-LEVEL:
 The guard for the function call
 (STACK-OBJ SIZE)
  ...
  is violated by the arguments
  in the call (STACK-OBJ -1).
```

If additional functions are added to the iccMAX calculator element operators, they can be quickly added or even automatically generated without knowledge of ACL2, which is the main advantage of the Tower metalanguage and machine-readable DOM.

C. The DaeDaLus Data Description Language

DaeDaLus [13] is an experimental data description language, designed to enable format experts to define practical data formats clearly, completely, and precisely, and to generate safe and efficient parsers of formats from their definitions. Over the course of its early development, it has been used to define representative subsets of a variety of practical data formats, including both formats related to enterprise documents (e.g., PDF and JPEG, in addition to iccMAX color profiles), as well as messages for embedded systems (e.g.,

MAVLink and NITF). It is supported by a range of parsing algorithms that are implemented as direct interpreters and code generators that target the C++ or Haskell programming languages.

To define practical formats, including iccMAX profiles, DaeDaLus supports a set of features not found in conventional data description languages, such as:

- grammars parameterized on values and other grammars, used, e.g., to define a reusable parser for parsing n function operators;
- data-dependent grammars, used, e.g., to define a number of curve segments given in the profile;
- constructs for capturing the current input and parsing with respect to a captured input, used, e.g., to parse grammars at an offset value included in the profile; and
- distinct classes of computation for parsing from an input and performing computation on a semantic value, used, e.g., to perform an execution time analysis of a calculator profile represented as a semantic value (Section V-A3) and ensure that semantics of all operations are well-defined in an interpreter (Section V-B5).

In order to formalize iccMAX in DaeDaLus, we composed a DaeDaLus specification that formalizes the core structure of color profiles and a small but representative set of tags, including tags used to identify calculator elements. We implemented a parser for calculator elements themselves by following the working standardization effort given in prose, interacting with format experts and the reference specification to define components that seemed to be either inconsistent or surprising. We defined a validator for calculator elements that attempts to validate the properties of valid calculator elements described in the working standard; as a notable variation of the standard, we validated that alternative control branches in a calculator have the same effect on their stack because it simplified the definition of the validator and because we suspected that practical calculator elements would be designed with the intent of satisfying this property. The entire validator is implemented as a DaeDaLus semantic action, returning a result in the parser monad in order to succinctly fail with an error message upon detecting unexpected content.

D. Parsley-Based Approaches

We also analyzed the iccMAX calculatorElement by:

- Directly capturing the iccMAX specification in the Parsley language, which necessitated a deeper understanding of specification than reading it alone,
- Using Parsley/Rust combinators to create a parser for the iccMAX calculatorElement, and
- Developing a static analyzer based on the syntax tree generated by the parser.

Below, we explain how we utilized each approach.

1) The Parsley Data Definition Language: We used the Parsley language to capture the iccMAX calculatorElement with the intent of uncovering issues with the specification. Additionally, the exercise provided us an opportunity to test the language and identify ways to improve it.

Parsley [23] is an attribute grammar system extended with a functional sublanguage to specify any computation that may be required during parsing. The attribute system includes inherited attributes to communicate contextual information to a parse and synthesized attributes to return structured output information from a parse. The parsing combinators are based on those in PEG grammars, and specifically employ an ordered choice combinator to enforce deterministic parsing. The functional language can be used to compute and check constraints, and to compute return values. Parsley treats parsing buffers as first class objects called *views*, and it includes combinators to restrict parsing to within the bounds of given views.

The development of the views feature is motivated by a need to support frequent adjustment of the cursor—the current input position being parsed—which is required for parsing many file formats. For one example use case, there is often a need to skip over bytes until a magic string is reached, e.g., the file contents of a PDF file lie between the `%PDF-` (or `%PDF-x.y`) and `%%EOF` tags. For another, many file formats specify element addresses via offsets; parsing often entails temporarily suspending parsing at the current address to move the cursor to the byte offset of the element so that it may be parsed—and then returning to the initial address once that element has been parsed. Cursor control is especially important for the iccMAX specification as the iccMAX format specifies many things via offsets and sizes. Like the PDF format, the iccMAX format contains a tag table with a list of tag entries, each of which contains a tag, an offset, and a size. Similarly, the calculatorElement uses positionNumbers—effectively offsets and sizes—to specify the location of the main function and all sub-elements of a given calculator element. For each of these offset-and-size structures, we define a view—a fragment of the complete parsing buffer. Parsing is done with respect to a view—and the cursor must lie within the bounds specified by that view. The current view can be changed on the fly, but in many applications, it is useful to invoke a function that parses a non-terminal or regular expression using a supplied view. We believe support for views, alongside synthesized and inherited attributes, make the Parsley toolkit suitable for creating parsers and generating abstract syntax trees for iccMAX profiles, among many other real-world formats.

The exercise of capturing the iccMAX calculatorElement in Parsley not only provided us a deeper understanding of the relevant portion of the specification; it also allowed us to test and improve the Parsley language itself. Parsley is, by design, restrained in its computational capabilities. We

have taken the approach of only implementing features when a strong need arises. As a consequence of this exercise, we implemented two new Parsley features, as described in Section V-C.

2) Parsley/Rust Combinators: We built the Parsley/Rust parsing library to capture features in the PDF specification. Subsequently, we have successfully captured the syntax of the DNS, Mavlink, and RTPS protocols. The library provides several parsing primitives to capture magic strings, characters, and integer ranges. In addition, we provide several combinators to perform complex parsing operations such as prioritized choice, sequences, alternate sequences, and star and plus operations.

Parsley/Rust combinators implements the views feature presented earlier in Section IV-D1. Creating a view using the Parsley/Rust combinators involves two steps. First, we create the TransformView structure to specify a transformation of a view. It follows the syntax `TransformView::new(position, size)`. We then call the transform function with an existing view as an argument.

Internally, views do not create new copies of the entire buffer. Instead, our Parsley library implements an API to set the bounds for a view and ensure that the cursor cannot go beyond these bounds. For every parsing operation, these checks are enforced on the parsing buffer and the view.

We used this Parsley/Rust toolkit to implement the iccMAX specification. Closely following Figure 1, we implemented checks for fields in the iccMAX header and then implemented a parser for the tag table. Next, for each tag entry in this tag table, we created a view for the tagged element data using the offset and size provided in the table.

We built a parser for a generic tagged element and the MPET element. The MPET element parser can be used to extract the embedded calculator element. The calculatorElement parser, in turn, uses views to extract the main function and the specific subelements into separate buffers. Finally, the calculatorElement parser extracts the operations and operands in the main function and the subelements. We use this data from the calculator element to statically analyze an iccMAX profile.

3) Parsley-Based Static Analyzer: To further analyze the iccMAX calculatorElement, we implemented and used a static analyzer in conjunction with the Parsley/Rust parser. The static analyzer uses the syntax tree produced by the parser to check the syntax and stack effects of operations.

The iccMAX specification defines a minimum stack size of 65,535, which must be supported by every iccMAX implementation. calculatorElement operations may push to or pop from the stack, yielding concomitant stack constraints that must be met for the operation to be considered valid. Additionally, stack overflows and underflows along multiple paths of calculatorElement functions must be validated. Underflows occur when we try to pop values from the stack when the stack does not have sufficient elements. Similarly, overflows occur when the stack size crosses the stack size used. Since the prescribed minimum size is 65,535, any implementation crossing this value can cause overflows on implementations. The static analyzer performs the requisite checks to ensure that these specification-imposed stack constraints are met.

Each calculator element contains a main function with a set of operations and sub-elements. Each operation is a 4-byte operation signature followed by a 4-byte argument. Depending on the operation, the argument may be split into 2-byte arguments or may be completely ignored.

Sub-elements invocations fall under six predefined operation signatures. A given calculator element (and the main function) holds a four-byte index value to help identify the sub-element. Sub-element operations take this four-byte index value as an argument. The type defined in the sub-element definition and the operation signature must match in four of the six sub-element types.

A given calculator element and each of its sub-elements define how many input and output channels they use. When a sub-element is invoked from the main function, we check the number of input channels used by the sub-element and pop those many elements from the stack. Similarly, we add the output channels back to the stack when the sub-element completes execution.

The iccMAX specification defines several other operations that manipulate the stack. These operations vary from stack operations to matrix operations and various mathematical operations to conditionals. However, most of these operations are deterministic in the effect they leave on the stack, and we can compute the size of the stack after each operation.

Conditionals complicate this process by producing branches. `if` conditions can take two paths, whereas `sel` statements can take any number of paths. We cannot know which branch the program must take without inspecting the inputs given to the calculator element. Tracking every possible branch, however, is resource-intensive and inefficient.

We treat `if`, `if-else`, and `sel` operations differently. Instead of tracking all the possible stack sizes after each branch, we only store the minimum and maximum stack sizes to check for underflows and overflows. If an `if` operation does not have a corresponding `else` operation, we store the previous stack size along with the stack size after the execution of the operations under `if`.

Similarly, suppose a `sel` statement does not hold a default condition. In that case, we compare the stack size before the

`sel` statement to all the possible stack sizes after various cases. Eventually, we only track this list's minimum and maximum stack sizes.

We used our static analyzer to find several stack overflow and underflows in iccMAX files produced using the DemoIccMAX toolkit. These files were produced by the ICC working group with malformations to help future iccMAX implementors test the correctness of their parsers. We were able to detect the malformations in the given files successfully.

V. Findings & Proposals

In the previous section, we discussed different approaches and tools we used to analyze the iccMAX calculatorElement. In this section, we present our findings from said analysis, as summarized in Table I. In particular, we present our proposal for an iccMAX resource contract, we note some errors and implementation challenges we discovered in the iccMAX profile specification, and we document ways that this exercise has led to improvements in our tools.

A. A Resource Contract for iccMAX

The current `multiProcessElementTypes`, and the `calculatorElement` and its sub-elements, all declare their expected number of input and output channels directly in their entry tables or element encodings. Our impression is that these I/O channel requirements seem to be treated in the specification as primarily a *processing signature*, with a view to ensuring that it is possible to statically check that there is no stack underflow or overflow and that these elements can be appropriately sequenced together such that the output channels of one element can be plugged into the input channels of the next.

We take the view that the I/O channel signature can also be viewed as a part of a *resource contract* of the element: the element declares the specified number of input and output channel resources required to perform its computation. Our core argument in this report is that this *resource contract*, although valuable in its current form, is only *partial and incomplete* for a given calculator element. We recommend adjusting the format to have a declaration of a *complete* resource contract within the calculatorElement encoding.

Our recommendation may be taken to be in the same vein as the LangSec approach to data processing, which suggests using a grammar with the minimum computability power or expressivity to completely capture the data format under consideration. We recommend the minimal resource contract that completely captures all the resources that are used by a calculator element.

Completing the Resource Contract

The element signature when viewed through a resource contract lens is incomplete with respect to the following resources:

1) temporary channels,
2) data stack size, and
3) computation effort, or alternatively, execution cost.

We argue that completing the resource contract imposes minimal additional processing requirements, given the validation step already required by the specification. Indeed, it provides some valuable benefits.

1) Temporary Channels: Although temporary channels are subject to the same maximum size limits (65,535) as input and output channels *(iccMAX-Clause-11.2.1.1 - pg. 100)*, they do not appear in the declared resource contract. The temporary channel resources that are actually used by a calculatorElement can be determined only by examining the streams of operations invoked by its main function. Although this examination is easily done as part of the stack usage validation required by the specification for a calculatorElement, it would be more uniform to include this resource amount in the processing signature along with the input/output channels.

Since it is possible that the operations actually included in a calculator's function exceed the element's declared input/output channel bounds, the specification requires I/O channel index bounds checking to be performed as part of the calculatorElement validation. It would be more uniform to treat temporary channels as just another *declared* channel resource to validate. This argues in favor of including a temporary channel usage declaration as part of the calculatorElement encoding specified in *(iccMAX-Table-85)*.

2) Data Stack Size: The specified bound for the data stack size of a calculatorElement is a *minimum*:

> The reserved storage for the data stack shall be for at least 65 535 values. *(iccMAX-Clause-11.2.1.1)*

Although requiring a specified *minimum* bound has some advantages when emitting calculator operations that perform small computations, it has also some disadvantages:

- A calculator element performing complex computations cannot directly declare its requirement for a data stack *larger* than the specified minimum. Instead, an implementation needs to traverse the operation stream of the calculator function to estimate this stack size.
- Similarly, a calculator element performing very simple small calculations cannot directly declare that it needs a data stack *smaller* than the required minimum. This may lead to wasted stack resources, especially given that data stacks are required to be separate for each calculator element, with no sharing allowed.

Indeed, there is an unusual asymmetry in the power of the various calculatorElement operations with respect to channel and stack resources. Any given calculator element cannot access more than 2*0xFFFF (S+T from *(iccMAX-Clause-89)*) channels, regardless of the number of operations in

the element, due to encoding restrictions of the channel vector operations. Even so, the resource bound for channels is specified as a *maximum*. When it comes to their power with respect to the stack resource, however, the operations are quite powerful. For example, the `copy` operation could increase the stack by large multiples of 0xFFFF *in a single operation*, giving even small programs the power to rapidly grow the stack. This seems incompatible with the specification of the stack size as a required *minimum*, leaving the maximum bound entirely implementation-defined.

3) Computational Effort or Execution Cost: Programs in more expressive and general-purpose programming languages compute intermediate values and inspect them to determine which sequence of instructions to execute. In typical programming languages; they also perform iterative or recursive computations by using looping control flow and recursively defined functions. Aside from enabling recursive computations, functions also enable a program to include a single, abstract definition of a sub-computation that can be reused in multiple contexts.

iccMAX calculator elements provide a nuanced selection of such control structures: they include branching constructs and functions, but they do not include looping constructs—and functions cannot be defined recursively. The major consequence of this design feature is that each execution of a calculator element must eventually complete (which is typically not true of programs in general-purpose languages).

Assumed Security Goal In many contexts, data (the collection of values used to carry out a computation) is not a system's only sensitive resource. The computing time (the requisite time to carry out the computation) and the available storage (the space provided for doing the computation) are sensitive as well. Security attacks which do not necessarily leak or corrupt the system's data but do attempt to consume its available computing time and space (i.e., *denial-of-service (DoS) attacks*) are a well-known class of attacks, and there are many practical and damaging instances of them [4], [14]–[16], [28], [29]. We assume that the calculatorElement control operators were designed as described above in order to ensure that a system that executes a calculator element is not susceptible to a DoS attack in which the calculator element exhausts an infinite amount of computing time on its host system.

Threat to Achievable Security Our read of the specification is that it achieves the intended security goal, *narrowly defined*. However, while it is impossible for a calculator element to use an infinite amount of time, the specification still allows for calculator elements that would take a finite but impractically long amount of time to execute. This goal seems inconsistent both with the great care that seems to have been taken to prohibit infinite executions and with the concrete bound imposed on the amount of space that a correct implementation must allocate (as discussed in *(iccMAX-Clause-11.2.1.1)*). Moreover, there are valid calculator elements whose code is much smaller than the time required to execute them, as elaborated below.

Potential Revisions to the Specification Two broad types of adjustments to the specification seem possible:

1) Include an explicit warning that, while all calculator elements terminate on all executions, users should beware that they could use an enormous amount of time to execute.
2) Extend the specification to include a bound on execution time, along with the provided bound on stack space.

Under the second option, the open issue is to determine the units of time in which the bound is to be expressed, in addition to the bound's magnitude. A natural option might seem to be to measure execution time in terms of the size of the program. However, such bounds would be able to impose little in practice, due to the fact that calculator elements can execute in time exponential in the size of the calculator element itself. Consider the function template below. A function following this template will execute in a number of calling contexts that are exponential in the program's size, i.e., its execution will yield an exponential number of call stacks.

```
f[n]:
  call f[n-1];
  call f[n-1];
  f[n-1]:
    ...
    f[1]:
      call f[0];
      call f[0];
      f[0]:
        x:=sqrt(5);
        output(x);
    ...
```

For any given value of n, the above program executes 2^n `sqrt` instructions.

A more effective option is to define the time bound in terms of the number of operations executed during computation. One variant of this option may involve using the number of calculatorElement operations executed; this variant is straightforward to define, although the costs of individual operations may vary considerably due to the inclusion of vectorized operations, in which the amount of computation to be performed is partly determined by the operations' arguments. Another variant is to define the bound in terms of individual floating-point operations. In both cases, it is important to note that even though the time taken to *execute* the program is potentially *exponential*, measuring the program's execution time can always be done in *linear* time

by implementing standard techniques from interprocedural program analysis [27].

Using our resource contract lens, it would make sense to include such a computational resource limit as part of the declared resource contract for the calculatorElement. Indeed, including this limit as a declared resource would provide a *complete* resource contract as part of the processing signature encoded in the format for every element. The design of the calculatorElement does make it feasible to statically compute such a resource bound by associating a cost metric to each computation step. The static validation step would then ensure that the declared bounds for *every* resource are not exceeded.

4) Benefits of Declaring a Complete Resource Contract: We have argued that a few minor changes[3] to the iccMAX format enable the declaration of a *complete resource contract* for the `calculatorElement`. The operations in a given calculator element can be validated to obey this declared resource contract in linear time, as part of the validation step that the iccMAX specification already requires an implementation to perform.

The benefits of incorporating such a complete resource contract are:

- Resource-constrained implementations can benefit from faster and more precise decisions on whether they have the resources to invoke a given calculator element, without needing to resort to a validation step to ascertain these bounds. That is, such implementations can pay the cost of validation for *only* those elements that declare resource limits within the resource capacity of the implementation. This is a big improvement from the current situation, where such an implementation *always* has to perform the validation step merely to ascertain the stack bound for every calculator element.
- Compartmentalized software [3], [31] and hardware [21], [32] architectures are increasingly common to address security concerns. A precise and complete resource contract enables an application to set up precisely bounded compartments[4] for untrusted computational units.
  In particular, platforms equipped with hardware support for resource compartmentalization could entirely skip the resource validation step (implemented by possibly buggy application software) in favor of relying on (likely less buggy) hardware resource bound enforcement.

[3]This is assuming a well-defined performance cost-model and a corresponding unit for computation or execution time.

[4]Execution timers can be used by a runtime executive in the application to enforce execution time bounds for such compartments.

B. iccMAX Errors & Implementation Challenges

We have uncovered several errors in the specification and the DemoIccMAX implementation. Additionally, we have reported several minor typos to the ICC working group. This section discusses more complex errors and implementation challenges pertaining to the iccMAX specification and the DemoIccMAX implementation, as well as a representative sample of the smaller errors we found.

1) Conditional Operations are Insufficiently Defined: Conditional operations such as `if` and `sel` conditions were not sufficiently described in the specification. These operations could have nested structures, where a `sel` operation could have an `if` condition or a `sel` operation embedded in it. For example, let us consider the following sequence of operations.

```
if  5
if  4
pi 0
pi 0
NaN 0
+INF 0
```

The above operations are syntactically valid. The operations `pi`, `NaN`, and `+INF` do not take arguments, and these values are set to 0. The `if` operation specifies how many operations are a part of the `if` block. It was not evident in the specification that the outer `if` accounts for every condition inside, even if it includes nested conditional operations.

We considered an entire if-else block to be just one instruction in an earlier implementation. However, upon further discussions with iccMAX experts, we were informed that the correct interpretation was to consider each operation within a block to be an instruction. If the inner condition fails, we eventually only execute one operation, not five. We proposed better language to the ICC working group to alleviate this confusion.

2) Sub-element Type Mappings Missing: The iccMAX specification *(iccMAX-Clause-11.2.1.5)* uses the following language: “The curv, mtx, and clut operators require that the indexed sub-element has the appropriate type.”. The sub-element types for each of these operators are not specified in this text. We proposed adding a mapping for these operators to sub-element types. Additionally, the `calc` operator was missing from the list, even though it should have been included. The ICC working group has acknowledged these changes.

3) Operators Missing in Specification: Since the iccMAX specification and DemoIccMAX implementations are relatively new (released in 2019), we do not have many open-source implementations and sample data available. The iccMAX profiles we used to test our implementations were available as part of the DemoIccMAX implementation.

However, when we ran our parser on these iccMAX profiles,

we found that several of these files used operations that were not defined in the specification. So we reached out to the working group to gather descriptions for all these operations and alert them to the missing specifications.

These operations were the following: `fJab`, `tJab`, `fLab`, and `not`. Our Parsley-based static analyzer now models the stack effects for all of these operations.

4) Incorrect Sub-element Index Implementation: In section IV-D3, we described that sub-element indices are a four-byte value. We can invoke any sub-element by referencing the index number and the correct operator type.

When we inspected various iccMAX profiles as part of the DemoIccMAX implementation, we found that several of these files were malformed. Instead of using a four-byte index as an argument to the sub-element operations, the DemoIccMAX implementation converted this argument to two two-byte arguments where the second argument is always zero. Any reasonably crafted calculator must not need more than 65,535 ($2^{16} - 1$) sub-elements. However, this DemoIccMAX implementation—an implementation that treats a four-byte value as two separate two-byte values, one of which is zero—clearly violates the specification.

Given that extant data has already been produced and several vendors are building on top of the DemoIccMAX implementation, the ICC working group has two options:

- First, alter the specification in multiple places. For example, make sure that the sub-element indices are always two-byte. The extant files use a four-byte value in one location and a two-byte value in another, causing confusion. This change would mean the data files already produced would violate this specification version.
- Alternatively, second, support both interpretations. We could try both interpretations of the specification as disjunctions. If we cannot find the right sub-element using the correct interpretation, we can try the incorrect but already shipped interpretation to validate the file. To the best of our knowledge, the ICC working group is yet to finalize an approach to address this issue.

5) Non-numeric Values Allow Parser Differentials: Individual operations in iccMAX calculator programs compute over floating-point values, whose definition uses the IEEE 754 [18] standard. While floating-point values are largely machine representations of the real numbers, they also include *exceptional* values, such as positive and negative infinity (+INF and -INF) and "Not a Number" (NaN), which arise from exceptional conditions in computation that cannot be defined over only the reals (e.g., dividing a number by zero, or attempting to compute the square root of a negative number). The IEEE 754 standard [18] precisely defines which exceptional values may arise from computation and which ones result from computation over non-reals. While the calculatorElement definition uses the set of exceptional values from IEEE 754, it leaves the result of computation on exceptional values to be defined by implementations.

Regarding exceptional values, we believe the intent was to strike a balance between expressivity and simplicity. By including exceptional values, it gives an implementation the freedom to potentially include sensible implementations of exceptional computations over reals; by leaving the exact definitions to the implementation, it avoids complicating the specification itself.

We believe another intended security goal was to allow implementations to produce different exceptional values as results on output channels, while ensuring that differences in final results due to implementation choices are limited strictly to exceptional values. For example, for a given calculator element and input, one implementation may produce an output channel $[1.0, 2.0,$ +INF$]$, while another produces $[1.0, 2.0,$ -INF$]$. However, it should not be possible for one implementation to produce $[1.0, 2.0, 3.0]$ while another produces $[4.0, 5.0, 6.0]$.

Threat to Achievable Security Regarding computation over exceptional values, we believe the current specification introduces two possible threats to security. Both threats could result in the deployment of calculator processors that produce different results when given the same calculator, by virtue of either implementation mistakes invited by the standard or by unintended permissiveness in the standard itself.

First, by borrowing most but not all of the exceptional values from IEEE 754 and not necessarily adopting the standard's specifications of operations over such values, the current specification invites errors by calculatorElement programmers who may note a casual remark regarding IEEE 754 in the specification and incorrectly assume that they can rely on a faithful adoption of IEEE 754 throughout.

Second, it is possible for execution of a calculator element to result in output channels that differ in both exceptional and numeric values when the calculator element is executed on different implementations. This can arise in part when branching on a non-numeric value produced by an exceptional computation. E.g., when executing the following pseudo-calculator,

```
x:=-1.0/0.0;
if x then output(1.0)
else output(2.0)
```

an implementation that evaluates `-1.0/0.0` to NaN would output 1.0, but an implementation that evaluates `-1.0/0.0` to -INF (as specified by IEEE 754) would output 2.0.

Potential Revisions to the Specification. One apparent possible revision that would both remove surprising behavior for experienced floating-point programmers and seemingly

remove divergences due to implementation details would be to specify that floating-point arithmetic shall fully implement IEEE 754. By referencing IEEE 754 as an external document, the specification of the calculatorElement itself would not be further burdened and complicated.

If this revision is not acceptable, perhaps because it is so strict that it prohibits some desired fast implementation, we would recommend revising the standard to still prohibit divergent final results due to implementation choices. This would also be feasible, though seemingly more complicated than adopting IEEE 754 directly.

6) Minor Errors and Issues: Our analysis uncovered a numerous minor but noteworthy errors and issues in the iccMAX specification. Here, we present a representative sample of them.

Flip Function

In developing the iccMAX calculator operation XML IV-B, we identified a typo/inconsistency in the iccMAX standard. The `flip` operator, as summarized in the standard *(iccMAX-Table-96)* and reproduced in Table XIII is described to act on $S + 1$ elements, but the stack arguments and stack results show that `flip` is operating on $S + 2$ elements. The formalizations of the calculatorElement rules in ACL2 and PVS could not tolerate ambiguity, as the stack effect expression had to be written as a mathematical formula, and therefore we were able to catch an inconsistency that was allowed to be present in the natural language standard.

calculatorElement Function Encoding

The fourth field in the calculatorElement function encoding corresponds to a sequence of function operations. The specified field length of 8 is inaccurate; it should be $8N$ where N is the number of operations specified in the preceding field.

Correct but Inconsistent Presentation

While not technically an error, there are places where it would be reasonable for a reader to expect a norm in presentation, but that expectation is not met. For one example, in the table comprising a list of the stack operations *(iccMAX-Table-96)* (reproduced in Table XIII), the symbols used to express the stack arguments change for no apparent reason. While this is not in an error in itself, it very well could be the source of a specification error or an implementation error due to a specification designer or parser implementer having a flawed mental model due to such a description.

C. Parsley Improvements

This exercise has led to the improvement of the Parsley language.

One goal in developing the Parsley language has been—and continues to be—to constrain its computational capability while still being useful. Thus, after developing a small core set of features, we have been evaluating the necessity of new features on a case-by-case basis. This exercise has led to the development of two features.

Our first feature involves the augmentation of the map-views combinator, which allows for the application of regular expressions and non-terminals to a list of views. To parse the iccMAX profile structure, we created a view for every tag entry in the tag table to process the tagged data elements. However, we quickly ran into a problem. While we had a map-views combinator to apply a regular expression or non-terminal to a list of views, it did not allow us to specify individual inherited attributes for each view. Thus, we developed a more flexible variant of the combinator to address this need.

The second feature of adding support for mutual recursive functions arose from the approach we were using to validate some of the calculatorElement operations, e.g., ensuring stack constraints are met. The general approach involved tracking how each operation invocation affected existing resources, e.g., elements added to the stack. By doing this, we could validate the operation constraints in one sweep after the initial parsing. However, given the existing conditional operations, we realized that we needed to support mutually recursive functions to achieve this aim.

VI. Takeaways

Our case study on the iccMAX calculatorElement, alongside our experiences with other specifications, have provided numerous takeaways that are more broadly applicable to individuals who design specifications, individuals who design and implement data description languages, and individuals who develop parsers. This section presents those takeaways.

Format Design Should Incorporate Formal Methods & Parsing Tools.

The final stages of specification design should incorporate formal methods and parsing tools to uncover inconsistencies, bugs, and other issues with the specification prior to release. The process of applying formal methods (such as PVS [25], Coq [1], and ACL2 [19]) requires the crystallization of assumptions made regarding corner cases, bounds checking, and invariants that are needed to argue about the validity of data. Data description languages and parser combinator toolkits provide a machine-readable version of a specification with all the necessary validity constraints. As these tools require a human in the loop, users of these tools must be well-versed with both the specification and the tool(s) they are applying.

Specifications Should Explicitly Define Resource Contracts.

Format specifications should provide a mechanism to specify resource contracts deemed useful by the application domain, and, moreover, specifications should require that

specification-compliant files clearly specify these contracts. For example, specifying memory requirements or computational requirements may be useful in embedded systems. Such specifications can also reduce parsing overhead and help in determining, a priori, whether it even makes sense to attempt the parse, although it may add overhead during the production the initial file.

Intents Should Be Clear.

Although specifications are definitive, they often contain mechanisms whose definition is so precise and subtle that its underlying intent is not obvious. Stating such intents, instead of requiring them to be taken as assumptions, aids the specification's readability and allows them to be checked for internal consistency.

Parsing Should Be Limited Via Buffers.

Many file format specifications specify how to disaggregate a complete file into smaller chunks, such as sections or elements. These chunks lie within well-defined boundaries specified by offsets from the beginning of the file and sizes. A critical security property that should be enforced in implementations when parsing these chunks is to constrain the parsing to these well-defined boundaries via the use of appropriately delimited parsing buffers.

Operational Semantics Should Be Specified.

The operational semantics associated with constructs such as the iccMAX calculatorElement should be specified in the specification or accompanying standards documents. Doing so ensures that the program is well-typed and reduces the likelihood of performing unintended computation. Specifications with nonexistent or inadequately defined operational semantics can result in parser differentials. In addition, doing so might help simplify the chosen semantics. For example, the formal semantics of WebAssembly uses stack signatures for its branching operations that are simpler to validate and give more precise bounds on stack usage than those chosen in iccMAX.

Data Should be Orderly Sequenced.

Data should be orderly sequenced in a manner that minimizes parsing complexity and reduces the potential for misassumptions on the parts of specification designers and parser implementers. For example, many file formats contain tables with entries that specify positions and sizes of elements. But if the elements are not ordered in the same sequence as they are ordered in the table, if the elements overlap, or if there are gaps between elements, this not only makes the parsing process more complex but it makes it harder for people to understand and reason about the specification. This can result in errors within the specification, specification-parser differentials, potential for polyglots and data exfiltration, and ultimately real-world vulnerabilities [2], [22].

Size Restrictions and Operations Should be Carefully Designed to Minimize Unnecessary Overhead.

Unnecessary storage and computation overhead should be avoided as a design principle. For example, in the iccMAX specification, the stack operator encoding specifies selector values S and T as UInt16's. However, the stack operations themselves involve the manipulation on S+1 and T+1 elements, for no apparent purpose. This adds unnecessary complexity in the code to parse and implement these operations, since they now need to guard against integer overflows.

VII. Conclusion

In this paper, we documented our work on analyzing the iccMAX specification [9], specifically the calculatorElement, as part of the DARPA SafeDocs project. Our efforts, which involved using a variety of tools and techniques—theorem-proving tools, data description languages, parser combinators, etc.—led to the the proposal of a resource contract for iccMAX, as well as the discovery of both important errors and smaller bugs with the specification and the demo implementation. Additionally, our work has revealed a number of valuable insights and suggested best practices that we believe would serve useful in a broader context pertaining to creating specifications, designing DDLs, and implementing parsers. For one example, it is imperative for specifications to specify resource contracts that meet the practical demands of the application domain. For another example, ensuring that intended computation is carried out requires the specification of operational semantics by a trusted standards-producing body.

Our research using the specification and the demo implementation showed that the demo implementation and some commercial applications have already deviated from the relatively new specification. One way new specifications can reduce these specification-parser differentials can be to provide a machine-readable specification of the data format. Data description languages provide an avenue for future specification writers to represent specifications in a less ambiguous machine-readable format. DDLs supporting complex operations such as offsets, constraints, and resource constraints can capture and describe extensions such as the calculator element.

We believe practical security problems are best addressed at the early stages:

- The specification should be thoughtfully designed, avoid security pitfalls, and lend itself to secure parsing.
- The specification should be vetted using a varied assortment of approaches and tools.
- Usable tools should be provided to practitioners to facilitate the development of secure and verified parsing.

It is our hope that this paper will provide some guidance on achieving these aims.

Acknowledgments

We would like to thank Peter Wyatt at the PDF Association for facilitating this work and acting as a friendly liaison between the ICC folks and ourselves. We would also like to thank the reviewers and Erik Poll for their valuable feedback, which significantly improved this paper.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0075. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

[1] The Coq proof assistant. http://coq.inria.fr.

[2] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *30th IEEE Symposium on Security and Privacy*, pages 360–371, 2009. DOI 10.1109/SP.2009.3.

[3] Adam Barth and Charles Reis. The Security Architecture of the Chromium Browser. In *Technical report*. Stanford University, 2008.

[4] Mitko Bogdanoski, Tomislav Suminoski, and Aleksandar Risteski. Analysis of the SYN flood DoS attack. *International Journal of Computer Network and Information Security (IJCNIS)*, 5(8) pages 1–11, 2013. DOI 10.5815/ijcnis.2013.08.01.

[5] Sergey Bratus. Safe Documents (SafeDocs) — DARPA. https://www.darpa.mil/program/safe-documents.

[6] Jeremy Brown. Processing a maliciously crafted image may lead to arbitrary code execution. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-30926, 2021. [Online; accessed 23-December-2021].

[7] International Color Consortium. About ICC. *International Color Consortium*. https://color.org/abouticc.xalter.

[8] International Color Consortium. Specification ICC.1:2010 (Profile version 4.3.0.0) Image technology colour management — Architecture, profile format, and data structure. 2019. https://color.org/specification/ICC1v43_2010-12.pdf.

[9] International Color Consortium. Specification ICC.2:2019 (Profile version 5.0.0 - iccMAX) Image technology colour management — Extensions to architecture, profile format and data structure. 2019. https://www.color.org/specification/ICC.2-2019.pdf.

[10] International Color Consortium. Specification ICC.2:2019 (Profile version 5.0.0.0) Image technology colour management — Architecture, profile format, and data structure - Cumulative Errata List. 2019. https://color.org/iccmax/ICC.2-2019_Cumulative_Errata_List_2021-09-09.pdf.

[11] International Color Consortium. White Paper #52: iccMAX calculatorElement Security Implementation Notes: A guide for implementing secure calculator element processing. *ICC White Papers*, June 2020. https://www.color.org/whitepapers/ICC_White_Paper_52_calculatorElement_security_implementation_notes.pdf.

[12] International Color Consortium. DemoIccMAX. https://github.com/InternationalColorConsortium/DemoIccMAX, 2021.

[13] Galois Inc. GaloisInc/daedalus. https://github.com/GaloisInc/daedalus/, 2022. [Online; accessed 2022 Jan 10].

[14] Red Hat Inc. OpenTTD Infinite Loop and CPU consumption vulnerability triggered by a crafted packet. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2534, 2010. [Online; accessed 2022 Jan 10].

[15] Red Hat Inc. Avahi Daemon Denial-of-service Vulnerability triggered by empty UDP packet. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1002, 2011. [Online; accessed 2022 Jan 10].

[16] Red Hat Inc. CGit Denial of Service Attack triggered by a crafted packet. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1027, 2011. [Online; accessed 2022 Jan 10].

[17] Mateusz Jurczyk. Processing a maliciously crafted image may lead to arbitrary code execution. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-30942, 2021. [Online; accessed 23-December-2021].

[18] William Kahan. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776) page 11, 1996.

[19] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96*, pages 23–34. IEEE, 1996. DOI 10.1109/CMPASS.1996.507872.

[20] Letitia W. Li, Greg Eakman, Elias J. M. Garcia, and Sam Atman. Accessible Formal Methods for Verified Parser Development. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 142–151, 2021. DOI 10.1109/SPW53761.2021.00028.

[21] Arm Limited. *Arm Architecture Reference Manual Supplement: Morello for A-profile Architecture*. Arm Limited, 2020. Online at https://documentation-service.arm.com/static/5f8da6fef86e16515cdb861e.

[22] Jonas Magazinius, Billy K Rios, and Andrei Sabelfeld. Polyglots: crossing origins by crossing formats. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 753–764, 2013. DOI 10.1145/2508859.2516685.

[23] Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anantharaman, Sameed Ali, Zephyr Lucas, and Sean Smith. The Parsley Data Format Definition Language. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 300–307. IEEE, 2020. DOI 10.1109/SPW50608.2020.00064.

[24] Alexandru-Vlad Niculae and Mateusz Jurczyk. A memory corruption issue existed in the processing of ICC profiles. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-30917, 2021. [Online; accessed 23-December-2021].

[25] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2) pages 107–125, 1995. DOI 10.1109/32.345827.

[26] Andreas Rossberg. WebAssembly Core Specification. 2019. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf, Online at https://www.w3.org/TR/wasm-core-1/.

[27] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences, 1978.

[28] Huzaifa Sidhpurwala. Wireshark Denial of Service vulnerability triggered by crafted ASN.1 data. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1142, 2011. [Online; accessed 2022 Jan 10].

[29] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. SoK: XML parser vulnerabilities. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, 2016.

[30] Michael Stokes. The History of the ICC. In *Proceedings of the 5th Color Imaging Conference*, pages 266–269. Society for Imaging Science and Technology, 1997. DOI 10.1016/S0920-5489(99)92103-7.

[31] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10. USENIX Association, 2010. Online at https://www.cl.cam.ac.uk/research/security/capsicum/papers/2010usenix-security-capsicum-website.pdf.

[32] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020. Online at https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf.

[33] Wikipedia contributors. Icc profile — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=ICC_profile&oldid=1060134567, 2021. [Online; accessed 20-December-2021].

[34] Wikipedia contributors. Reverse polish notation — Wikipedia, the free encyclopedia, 2021. [Online; accessed 13-January-2022], Online at https://en.wikipedia.org/w/index.php?title=Reverse_Polish_notation&oldid=1063046133.

[35] Peter Wyatt. Arlington PDF DOM. https://github.com/pdf-association/arlington-pdf-model, 2021.

[36] Peter Wyatt. iccMAX final report, 2022.