Spring 5-3-2022

Protecting Systems From Exploits Using Language-Theoretic Security

Prashant Anantharaman
Prashant.Anantharaman.GR@Dartmouth.edu

PROTECTING SYSTEMS FROM EXPLOITS USING LANGUAGE-THEORETIC SECURITY

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of

Doctor of Philosophy

in

Computer Science

by

Prashant Anantharaman

Guarini School of Graduate and Advanced Studies
Dartmouth College
Hanover, New Hampshire

May 2022

Examining Committee:

Sean W. Smith, Ph.D., Chair

Sergey L. Bratus, Ph.D.

Shagufta Mehnaz, Ph.D.

Natarajan Shankar, Ph.D.

F. Jon Kull, Ph.D.
Dean of the Guarini School of Graduate and Advanced Studies

Abstract

Any computer program processing input from the user or network must validate the input. Input-handling vulnerabilities occur in programs when the software component responsible for filtering malicious input—the parser—does not perform validation adequately. Consequently, parsers are among the most targeted components since they defend the rest of the program from malicious input. This thesis adopts the Language-Theoretic Security (LangSec) principle to understand what tools and research are needed to prevent exploits that target parsers.

LangSec proposes specifying the syntactic structure of the input format as a formal grammar. We then build a recognizer for this formal grammar to validate any input before the rest of the program acts on it. To ensure that these recognizers represent the data format, programmers often rely on parser generators or parser combinators tools to build the parsers.

This thesis propels several sub-fields in LangSec by proposing new techniques to find bugs in implementations, novel categorizations of vulnerabilities, and new parsing algorithms and tools to handle practical data formats. To this end, this thesis comprises five parts that tackle various tenets of LangSec.

- First, I categorize various input-handling vulnerabilities and exploits using two frameworks. First, I use the mismorphisms framework to reason about vulnerabilities. This framework helps us reason about the root causes leading to

various vulnerabilities. Next, we built a categorization framework using various LangSec anti-patterns, such as parser differentials and insufficient input validation. Finally, we built a catalog of more than 30 popular vulnerabilities to demonstrate the categorization frameworks.

- Second, I built parsers for various Internet of Things and power grid network protocols and the iccMAX file format using parser combinator libraries. The parsers I built for power grid protocols were deployed and tested on power grid substation networks as an intrusion detection tool. The parser I built for the iccMAX file format led to several corrections and modifications to the iccMAX specifications and reference implementations.

- Third, I present SPARTA, a novel tool I built that generates Rust code that type checks Portable Data Format (PDF) files. The type checker I helped build strictly enforces the constraints in the PDF specification to find deviations. Our checker has contributed to at least four significant clarifications and corrections to the PDF 2.0 specification and various open-source PDF tools. In addition to our checker, we also built a practical tool, PDFFixer, to dynamically patch type errors in PDF files.

- Fourth, I present ParseSmith, a tool to build verified parsers for real-world data formats. Most parsing tools available for data formats are insufficient to handle practical formats or have not been verified for their correctness. I built a verified parsing tool in Dafny that builds on ideas from attribute grammars, data-dependent grammars, and parsing expression grammars to tackle various constructs commonly seen in network formats. I prove that our parsers run in linear time and always terminate for well-formed grammars.

- Finally, I provide the earliest systematic comparison of various data description

languages (DDLs) and their parser generation tools. DDLs are used to describe and parse commonly used data formats, such as image formats. I conducted an expert elicitation qualitative study to derive various metrics that I use to compare the DDLs. Next, I also systematically compare these DDLs based on sample data descriptions available with the DDLs—checking for correctness and resilience.

Acknowledgments

This thesis is a culmination of several years of struggle and toil. The COVID-19 pandemic changed the course of my Ph.D. in several ways. "Research doesn't stop," but I had to adapt to completely virtual meetings and isolation when the pandemic was raging. This work wouldn't have been possible without the support, encouragement, and trust of many others.

My advisor, Sean Smith, has been a pillar of support throughout graduate school. His engagements take a Socratic approach and have helped me develop research acumen. Whenever I was stuck and went to him to ask for help, his questions helped me arrive at a solution at my own pace. Sean also enriched my graduate life in two other ways. First, he provided a platform for me to collaborate with people of extremely diverse fields. These collaborations opened up several opportunities for me as I was looking for jobs in the last leg of my Ph.D. journey. He also ensured that his students were exposed to enough opportunities to travel to conferences, summer schools, and even media engagements. Sean has been a great advisor but an even better human being—especially during the challenging periods of the pandemic.

I also thank my committee members Sergey Bratus, Shagufta Mehnaz, and Natarajan Shankar for their support, suggestions, and encouragement. This thesis has dramatically benefited and improved as a result of their in-depth feedback. A special thanks to Sergey Bratus for taking me under his wing early in my graduate school life. I have had several long, insightful conversations with him—often contrast-

ing Russian and Indian thoughts and ideas—but more importantly, he introduced me to the hacker culture. My connections with the hacker community have only strengthened since then, and I have grown to appreciate the blend between hacker culture and academia.

Michael Locasto, Gabriela Ciocarlie, and Ulf Lindqvist hosted me for three internships at SRI International, each better than the last. In addition, they gave me several IoT devices to play with and explore at their newly set up IoT security and privacy center. I am deeply grateful for their sage advice and the deeply insightful discussions and debates.

I would also like to thank N. Shankar, Prashanth Mundkur, and Briland Hitaj. I have greatly enjoyed working with them on the SAFEDOCS project. Prashanth Mundkur has helped me become a better programmer in several ways, always suggesting improvements. They provided insightful suggestions and newer project ideas to explore. Sometimes these projects would not pan out as expected, but they had immense faith and confidence in my ability and gave me ownership of large portions of the Parsley project.

Bill Smith, Matt Seidel, and John Interrante have eased my exposure to the world of DFDL and FPGAs by providing easy-to-use starter codes and tutorials for me to better understand the concepts in these universes. I am deeply grateful for their patience and support.

I would also like to thank Susan Perry Cable, Isaiah Snelling, Sarah Brooks, and Gary Hutchins. They have all been extremely friendly and helpful in communicating with me. Without their management and organizational skills, I wonder how difficult things could have been.

My writing and presentations have improved by many folds in the course of my Ph.D. This is a direct result of feedback from Sean, Dave Kotz, Tom Cormen, and Vi-

jay Kothari. I would also like to thank Dave Kotz, who served on my RPE committee and exposed me to many exciting research directions in IoT Security and Privacy.

I have made several close friends as a part of my journey at Dartmouth. I thank my close friends over the years in Hanover, Adhithyan Karunakaran, Joshua Ackerman, Kirti Rathore, Mahesh Devalla, Maryam Negahbani, Prantar Ghosh, Raghavan Ravi, Reshmi Suresh, Sameed Ali, Satya Kandala, Sisira Kanhirathingal, Suman Bera, Varun Mishra, Zephyr Lucas, and several others I am sure I am forgetting. Most of them went through or are going through similar struggles as me in grad school. Thank you all for being there for me at my best and worst in this long Ph.D. journey. A special thanks to Linta Joseph for being a great listener and for the long and interesting conversations. Somehow, she always has reading resources to share about any life situation.

I would also like to thank trust labmates bx Shapiro, Ray Jenkins, Jason Reeves, Kartik Palani, Vijay Kothari, Pete Brady, and Michael Millian for working with me on so many papers and participating in numerous brainstorming sessions. Some of you have also been great travel buddies at conferences. Its been a privilege and joy to work with several undergrads and masters students: Anmol Chachra, Syed Tanveer, Shikhar Sinha, Patrick Flathers, Benjamin Cape, Linda Xiao, Celina Tala, Rafael Brantley, and many others. A special thanks to Kris Udomwongsa for helping out so much with Chapter 7. He provided good, constructive feedback on drafts of this chapter that helped increase clarity.

My lifelong friends Srravya, Vinay, Sravan, Deepak, Nishanth Nathaniel, Lakshminarayan, Pooja, Roopika, Waqar, Sandesh, Nivetha, Shriya, and Mithul have all been with me in this journey in spirit—always supportive, kind, encouraging, and listening to my long and emotional rants. I would also like to thank my late friend Annamalai Alagappan—I wish he were here to see me graduate.

A special thanks to Bill Nisen and Julie and John Gilman, who I met through the Institute for Security, Technology, and Society. They have subsequently taken the role of my family in the area. The Gilmans refer to me as their "Indian son" and have invited me to their family Christmas and Thanksgiving dinners, so I am not alone so far from family. Bill and the Gilmans have been with me through thick and thin through my years at Dartmouth.

Most of all, I am deeply grateful to my family for the love, encouragement, and laughs and for putting up with me for almost 29 years. My brother Karthik and sister-in-law Pavana have been my guiding lights, providing deep insight and sage advice on life and academia. They have always been candid and truthful, telling me things I needed to hear. My parents, Anantharaman and Jayanthi, have been supportive of every decision I have made so far and have raised me to be strong, generous, and always be there for others. And to Avi: I have only known you for a little over a year—watching you grow up and play has been an enriching experience. I hope this thesis helps make the digital world just a little bit safer for your future.

We should also take a moment to acknowledge the land on which we are gathered. For thousands of years, the land where Dartmouth is located has been the home of Abenaki people.

Funding This material is based upon work supported by Department of Energy under Award Number DE-OE0000780 (CREDC), the Defense Advanced Research Projects Agency (DARPA) under Contract Nos. HR001119C0075 (SAFEDOCS) and HR001119C0121 (GAPS), and the United States Air Force and DARPA under Contract No. FA8750-16-C-0179 (RADICS). Any opinions, findings and conclusions, or recommendations expressed in this material do not necessarily reflect the views of the United States Air Force, United States Department of Energy, or DARPA.

Preface

I first ventured into the realm of Language-Theoretic Security at an internship at SRI International. Unsurprisingly, the internship was my first exposure to IoT and Language-Theoretic security. Since then, I decided to pursue a Ph.D. to explore these areas further, and this thesis is a culmination of that five-year journey. Although this thesis document primarily discusses my research through my Ph.D., I've also written it to serve as a handbook for anyone looking to pick up research in Language-Theoretic Security (LangSec). I have not assumed any prior knowledge in Systems Security or LangSec.

Chapter 2, in particular, discusses the state of research in LangSec and some of the related fields in detail. Anyone interested in understanding how LangSec ties to these other research ideas and what open problems remain would benefit from Chapter 2. Likewise, Chapter 7 focuses on understanding Data Description Languages, their features, capabilities, and pitfalls. This chapter includes an expert elicitation study and an evaluation of existing tools to understand future research directions.

For those in a hurry, the thesis outline in Chapter 1 includes a reading guide that would point out the most significant results of this thesis without diving deeper into the background and related work.

Readers can contact me by email at `prashant@prashant.at`.

Contents

List of Tables

List of Figures

List of Code Snippets

Chapter 1

Introduction

Programs reading formatted input rely on a parser to check the input and transform it into an appropriate internal representation. Unfortunately, programmers often fail to explicitly define the set of valid input a program must accept. As a result, when a program receives unanticipated input, the program may reach a state that was not accounted for by the developers.

The parser—the code that checks user input to ensure validity and creates a structured representation for the rest of the program—is the most targeted portion of software since it must defend the rest of the program from malicious input. Unfortunately, in the year 2021, there have been over 1000 entries in the MITRE Common Vulnerabilities and Exposures (CVE) database for various parser vulnerabilities.[1] Despite several advances in memory safety and fault isolation, parser bug discovery continues unabated. Hence, we need a paradigm shift in handling input in code.

The field of language-theoretic security (LangSec) posits that defending against attacks that exploit the lack of adequate input validation requires a paradigm shift. We need to take a principled approach to validate input that uses formal language

[1] We looked at the following Common Weakness Enumeration (CWEs) to estimate the CVEs with parser vulnerabilities: (1) Improper Input Validation, (2) Deserialization of untrusted data, (3) Uncontrolled format strings, and (4) Buffer overflows `https://www.cvedetails.com/cwe-details/502/cwe.html`

theory fundamentals. At its core, we can distill the core principles of LangSec to the following:

- *Formal languages and data formats principle*: We must define the set of valid or acceptable inputs to a program using formal grammars.
- *Principle of least expressiveness*: The formal grammar used to define the input formal must not be more complex than necessary in the Chomsky Hierarchy.
- *"Full recognition before processing" principle*: We must use a parser for the formal grammar to validate all input before acting on it.
- *Principle of parser equivalence*: Different parsers deciding the same language must be functionally equivalent.

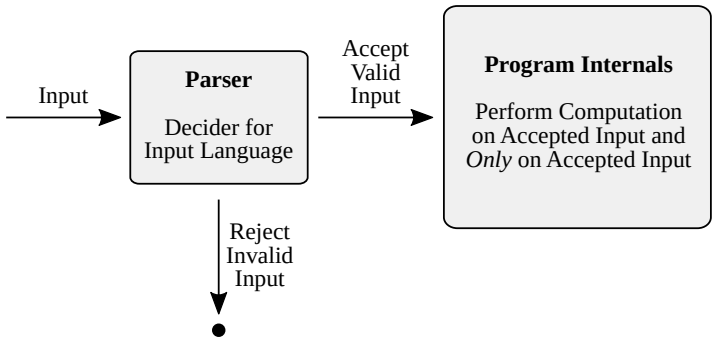


Figure 1.1: An overview of the LangSec methodology from my earlier paper [11].

Figure 1.1 demonstrates the overall LangSec methodology. We describe the input language as a formal grammar and build a decider for this input language. The program internals must only access well-typed, valid input. The parser must also be separate from the rest of the codebase to ease auditing the parser.

The Chomsky hierarchy (Figure 1.2) demonstrates various classes of formal grammars based on their computational complexity. In this hierarchy, regular grammars

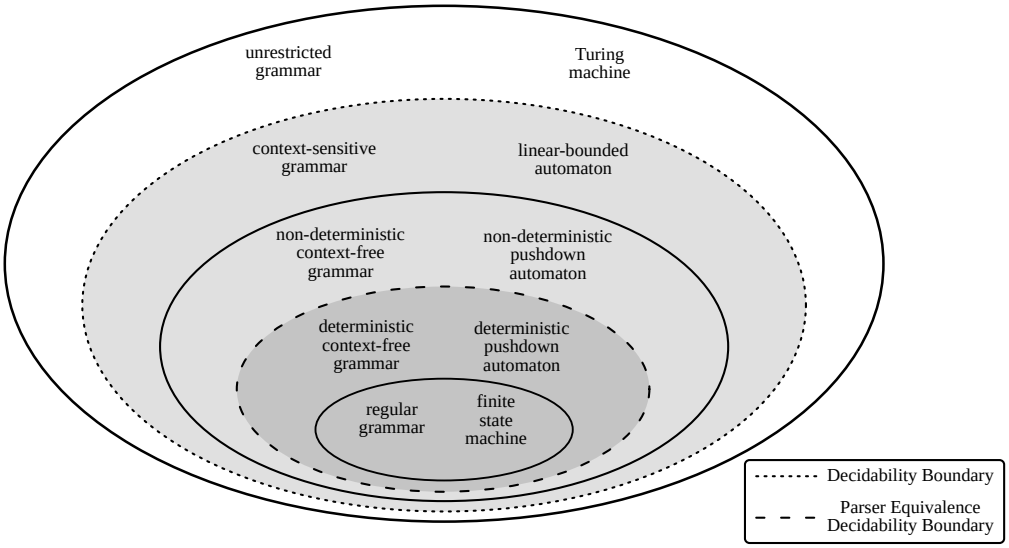


Figure 1.2: Chomsky Hierarchy Extended With LangSec Boundaries (from my earlier paper [11]).

are the simplest form and can be decided by a finite state automaton. Context-free grammars are the following level of grammars and can describe languages L taking the form $L = \{a^n b^n \mid n > 0\}$. Regular languages cannot handle grammars of this form. Deterministic context-free grammars form a strict subset of unrestricted context-free grammars. We can prove parser equivalence for deterministic context-free and regular languages and parse them in linear time.

Proving parser equivalence is necessary to ensure that two parsers that perform the exact computation are equivalent and interoperable. Equivalent parsers directly address *parser differentials*—a LangSec anti-pattern we have commonly seen in the wild. An example of such an anti-pattern is the recent iOS attack demonstrated in the Psychic paper [247]. The attack leverages the fact that iOS uses four different XML parsers in different situations, and all of them differ in subtle ways in how they handle comments in XML. As a result, an adversary can trick one of the parsers into believing that a particular data element is in a comment, whereas the other parsers may consider it data. Using the same formal grammar to recognize the XML

structures in iOS could limit this problem.

Staying within the deterministic context-free language class is not a realistic expectation, however. Most network protocols and file formats extensively use the tag-length-value construct (TLV). In this construct, we parse a tag, followed by a length. This length field specifies how long the value is going to be. The TLV format reduces the burden on the parser since the parser can process a fixed number of bytes and continue parsing. Hence, parsing tools tackling real-world formats, such as network protocols and file formats, must handle some context-sensitive constructs, such as TLV, repeat fields, and offsets, commonly seen in these formats.

Parser Combinators and Data Description Languages. Parser combinator toolkits and data description languages provide ways to enforce these LangSec principles in code. Parser combinators allow users to define grammar directly in code. For example, suppose a developer wanted to define a parser for the domain name service (DNS) networking protocol in their C code. They could use a parser-combinator library, such as Hammer, to describe this syntax. Unfortunately, robust parser-combinator toolkits may not be available for every target language.

In contrast, data description languages are domain-specific languages designed to capture data format syntax and constraints. These languages include accompanying tools that can be used to validate the descriptions, run the description on input, and, most importantly, generate code. Developers can then use this generated code with the rest of their codebase. To adopt data description languages, we need code generators for a wide range of languages.

Both these approaches have their merits and pitfalls. Parser combinators are easy to import into existing codebases and have a smaller learning curve. However, when developers write validation code in their programming language of choice, porting the parser code from one language to another while ensuring equivalence can prove chal-

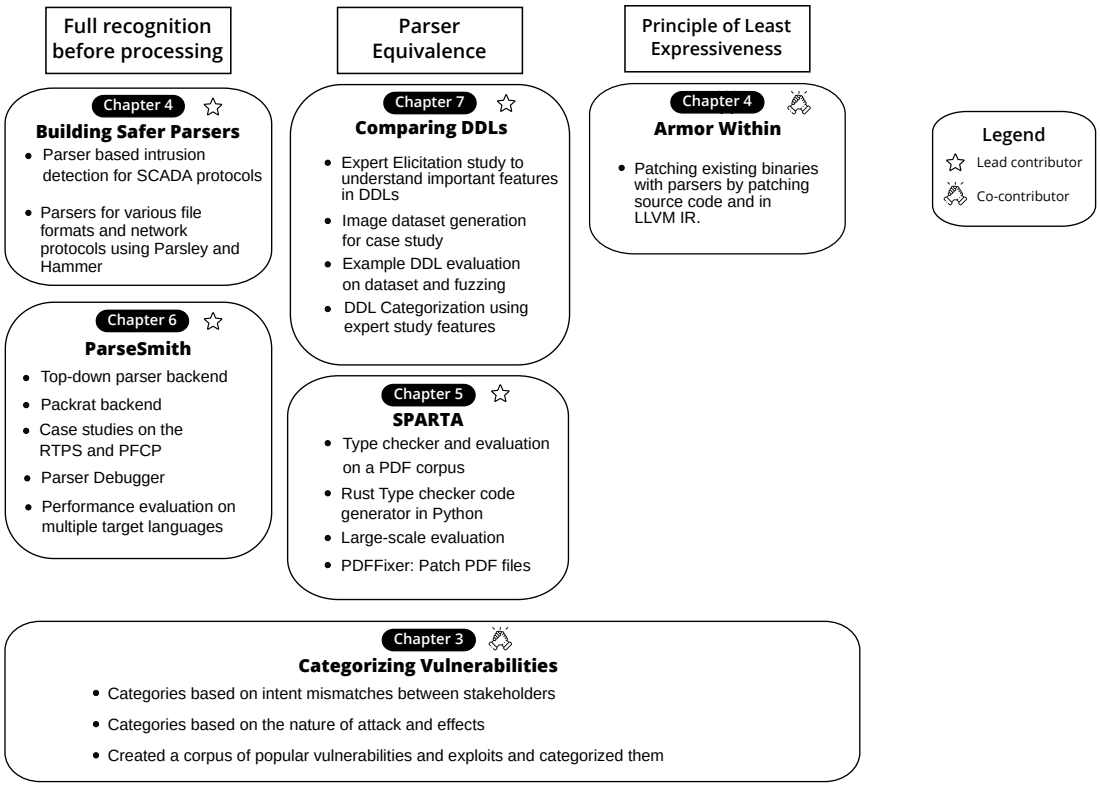


Figure 1.3: Outline of my thesis

lenging. Hammer and Nom are popular parser-combinator toolkits to tackle practical data formats [214, 66].

On the other hand, data description languages are appealing since they provide a single interface to generate code in multiple target languages. The designers of the language and the corresponding tools ensure that the various implementations are identical and interoperable. However, these domain-specific languages can prove harder to learn, and designing code generators for multiple popular programming languages is an engineering challenge. Kaitai Struct bridges this gap by providing code generators to Python, C++, Java, and many other popular languages [295].

LangSec vulnerabilities stem from violations of one or more of the core principles.

Hence, each of these principles serves as an avenue for LangSec research. Figure 1.3 provides an overview of this thesis. The thesis chapters are aligned with the principles that they enforce. First, in Chapters 4 and 6 I present work where we build parsers that can augment existing software to validate input. These parsers can be used with existing codebases to enforce the principle of "full recognition before processing."

Next, in Chapter 5 I compare PDFs generated by different software and document the differences—especially highlighting errors introduced by software. These errors indicate violations of the principle of parser equivalence. Then, in Chapter 7, I systematically compare various data description languages (DDLs) from the LangSec lens. DDLs present a way to enforce the principle of parser equivalence by defining machine-readable specifications of data formats. These DDLs are also accompanied by code generation software that can generate code in multiple target languages. Hence, DDLs are most appropriate to enforce this principle.

Next, Chapter 4.5 discusses how we inject LangSec parsers into existing code and what the minimal parser needed is to fix a vulnerability. Finally, Chapter 3 explores vulnerabilities and ties them to LangSec anti-patterns. We study several popular vulnerabilities from over the years and dissect them with respect to these LangSec principles and identify why these vulnerabilities came to be.

Thesis overview

This thesis is divided into five core parts that I summarize below.

- **Chapter 3 Understanding exploits and weird machines**: As we use more and more sophisticated fuzzing tools to find new exploits, we need to understand what common anti-patterns developers follow to lead to vulnerabilities. Researchers find and define various anti-patterns that lead to various input-handling vulnerabilities. In Chapter 3, I use the mismorphisms framework my

lab developed and which I helped with to reason about vulnerabilities. For example, we can define a *parser differential* as a mismatch between two or more developers' understanding of a specification.

- **Chapter 4 Principled parsing with parser combinators**: Parser combinators follow a different school of thought from those who propose using DDLs. Parser combinator toolkits allow developers to define grammars within the programming language of a user's choice. In addition, parser combinators make it easy for developers to include their parsers in their existing code base. We can enforce the "full recognition principle" by augmenting existing code with parsers. Chapter 4 discusses three of my previous projects where I used parser combinator libraries to recognize various network protocols and file formats.
  (1) I built a parser for the Industrial IoT protocols Message Queuing Telemetry Transport (MQTT) and Extensible Messaging and Presence Protocol (XMPP) using the Hammer parser combinator toolkit. I demonstrated that both these protocols could be parsed in the order of milliseconds on a Raspberry Pi.
  (2) I helped build a parser for the SCADA protocol IEEE C37.118 using the Hammer parser combinator toolkit. The parser was implemented as a validation filter in an ingress-based network appliance that inspects packets in the Wide Area Measurement network for potentially malformed inputs.
  (3) I built a communication validity detector (CVD) for various Supervisory Data and Control Acquisition (SCADA) protocols. These protocols are used extensively in the Power Grid. CVD comprises parsers for six SCADA protocols and various network configuration validators that alert Power Grid operators of malicious actions.
  (4) I built a parser and static analyzer for the iccMAX color format. iccMAX is a file format used to share coloring information across operating systems.

Building a parser for this format involved several challenges. This format has only been available for the last two years, and there are currently no open-source tools available to generate files in this format. This exercise has led to several bug reports in the iccMAX specification and the reference implementation.

- **Chapter 5 Understanding how different parsers deviate from specifications**: As described earlier, to prevent the risk of parser differentials, different parsers for the same input format must implement the same grammar. However, for a file format such as the Portable Document Format (PDF) that has been around since the 1990s, numerous softwares are available to generate these files [36]. Therefore, finding syntactic mismatches between files generated from different PDF creation software requires new tooling. Chapter 5 describes SPARTA, a tool I am building to generate type checker code from a machine-readable specification of the PDF format. I find the source of these parser differentials by defining a strict type checker for the PDF format. Our type checker has contributed to bug reports in PDF creation software that deviate from the specification. I have also proposed numerous corrections and clarifications to the PDF specification.

- **Chapter 6 Verified parser interpreters**: As we find new features in popular data formats, we need to extend formal languages to support them. For example, Calc-Regular Languages [172] were introduced in 2017 to add support for length fields to regular languages. We also need new algorithms to parse these new language classes as we create these new languages. In Chapter 6, I propose extensions to the Packrat and Scaffold Automata parsing algorithms to support the attribute and data-dependent grammar constructs in the Parsley Data Definition Language [198].[2] I implemented portions of our tool, Parse-

[2] I was part of the team led by Natarajan Shankar and Prashanth Mundkur that build the Parsley

Smith, and demonstrated it on the Real-Time Publish-Subscribe (RTPS) protocol. I verify ParseSmith for correctness and termination, and use the Dafny code generator to generate C++, Go, and C# code. Developers can use the ParseSmith-generated code in these programming languages to validate input.

- **Chapter 7 Building principled parsers with data description languages (DDLs)**: DDLs are used to describe network formats and file formats. Existing DDLs such as DFDL [181] and Kaitai [295] have their drawbacks. For example, DFDL does not support recursion or arbitrary constraints. Similarly, Kaitai does not include the features necessary to support a format such as PDF and relies on implementing arbitrary functions in the host programming language. Several researchers are building newer DDLs to support such complex features in formats. In Chapter 7 of this thesis, we discuss vital features DDLs require to support file formats and for broad adoption. Subsequently, we categorize these DDLs based on these features and evaluate these DDLs on a set of formats.

Real-World Evaluation and Impact

Several portions of this thesis have been deployed to real-world systems or have led to practical changes to commercial implementations and specifications. I have listed some of these impacts below.

- Our tool, CVD, outlined in Section 4.3, was deployed to real-world power grid substation networks as part of the DARPA RADICS project. As a result, we detected several real-world attack scenarios and malicious activity on power grid networks using CVD.

- My parser and static analyzer for the iccMAX file format, outlined in Sec-

language and the accompanying tools.

tion 4.4, led to several corrections to the specification document and reference implementation.

- We have proposed several changes to the PDF specification and the Arlington PDF Model (the machine-readable PDF specification). Several of these changes have already been adopted, with a few still under review as of May 3rd, 2022 (Section 5.5).

- Our expert elicitation study in Section 7.1, and the adjacent evaluation in Section 7.2 provides guidelines for future DDL creators and maintainers of current DDLs. Our image format evaluation also led to corrections to data descriptions in Kaitai and DFDL that are being reviewed by researchers.

- We identified several parser differentials and mismatches between image format specifications in DFDL and Kaitai, as we demonstrate in Section 7.4. We have subsequently been in touch with developers of both these DDLs and have proposed fixes to their format descriptions.

Section 1.1

Outline

First, Chapter 2 provides a comprehensive overview of LangSec theory and prior work. Second, in Chapter 3 I discuss various categorizations of input-handling vulnerabilities. Third, in Chapter 4, I describe my experiences building parsers for various network and file formats. Next, Chapter 5 discusses how we can find differentials in PDF creation tools. Fourth, in Chapter 6, I introduce two parsing algorithms to handle various constructs found in network protocols. Finally, Chapter 7 explores and compares state-of-the-art data description language implementations: their features, complexity power, usability, and robustness.

I anticipate three types of audiences for this thesis document. (1) For those interested in getting an overview of LangSec: the theory, related approaches, and ways to enforce LangSec principles, I suggest reading Chapters 2 and Chapters 3. (2) For those who want to understand my work and contributions to computer security, I suggest reading Chapters 4, 5, 6, and 7. Finally, (3) for those who want to understand prior LangSec work and where my work fits in, a complete reading of this thesis may be helpful.

If you are unwilling to read entire chapters and want to dive directly into the heart of my work, here is a possible order you could follow.

- Section 3 discusses taxonomies of LangSec vulnerabilities we created.
- Section 4.2 presents a communication validity detector we build for Industrial Control Systems protocols that was deployed to power grid networks.
- Section 5.3 and 5.5 present our work on generating PDF type checker code from a machine-readable specification and showcases the bugs we found in the specification and in PDFs found in the wild.
- Section 6.2 discusses the type system, inference rules, and the core properties we prove as a part of ParseSmith. Section 6.4 subsequently presents real-world case studies where we use ParseSmith.
- Section 7.1 presents a first-of-its-kind expert elicitation study to understand DDLs and their features.

Section 1.2

Prior Publications

This thesis builds on several publications I have worked on with colleagues at Dartmouth and elsewhere.

My work in Chapter 2 originally appeared in a book chapter.

- Prashant Anantharaman et al. *Intent as a Secure Design Primitive*, Modeling and Design of Secure Internet of Things, John Wiley & Sons, Inc., 2020 [11].
- Prashant Anantharaman et al. *Going Dark: A Retrospective on the North American Blackout of 2038*. Proceedings of the New Security Paradigms Workshop. 2018 [10].

Chapter 3 incorporates content from two papers. The first paper was a collaboration with various students in the Trust lab, Jim Blythe, and Ross Koppel. We built a mismorphisms framework and categorized various vulnerabilities based on this framework. Next, we surveyed popular vulnerabilities over a few months and categorized them based on their manifestation.

- Prashant Anantharaman et al. *Mismorphism: The Heart of the Weird Machine*, Cambridge International Workshop on Security Protocols, Springer, 2019 [14, 8].
- Sameed Ali, Prashant Anantharaman et al. *What we have here is failure to validate: Summer of LangSec*, IEEE Security & Privacy, 2021 [3].

Chapter 4 discusses my prior work from multiple papers. First, I describe how we build hardened IoT clients using the Hammer parser combinator and state machine libraries. Next, I describe a parser for the phasor measurement protocol. Then, I briefly discuss our methodology to patch existing binaries using parsers. Finally, I detail our approach to building an intrusion detection system and constructing a static analyzer for the iccMAX file format.

- Prashant Anantharaman et al. *Building Hardened Internet-of-Things Clients with Language-Theoretic Security*, IEEE Security and Privacy Workshops (SPW), 2017 [15].

- Prashant Anantharaman et al. *PhasorSec: Protocol Security Filters for Wide Area Measurement Systems*, IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm), 2018 [16].

- Sameed Ali, Prashant Anantharaman, and Sean W. Smith. *Armor Within: Defending against Vulnerabilities in Third-Party Libraries*, IEEE Security and Privacy Workshops (SPW), 2020 [4].

- Prashant Anantharaman et al. *A Communications Validity Detector for SCADA Networks*. Critical Infrastructure Protection XV: 15th IFIP WG 11.10 International Conference ICCIP Revised Selected Papers. Springer, Jan 2022 [12].

- Vijay Kothari, Prashant Anantharaman et al. *Capturing the iccMAX calculatorElement: A Case Study on Format Design*, IEEE Security and Privacy Workshops (SPW), 2022 [153].

Although Chapter 5 mostly includes new, unpublished work, portions of section 5.7 will appear later this year. This paper discusses our approach to dynamically fixing various malformations in PDF files.

- Prashant Anantharaman et al. *A Format-Aware Reducer for Scriptable Rewriting of PDF Files*, IEEE Security and Privacy Workshops (SPW), 2022 [13].

Chapters 6 and 7 include new work that has not appeared in any papers before this thesis.

Section 1.3

Prerequisites and Suggested Reading

A reader with a general understanding of computer science and engineering should be able to follow this thesis. I have assumed no specific background or knowledge in computer security, networking, theory of computation, or compilers while writing this document. Instead, I have attempted to ensure that I introduce fundamentals and theory as I use them to help the reader follow along.

However, several textbooks may help the reader understand concepts better—just as they've helped me through my Ph.D. For an overview of computer security and various input-handling vulnerabilities and attacks, I suggest:

- Sean Smith and John Marchesini. *The Craft of System Security.* Pearson Education, 2007 [252].
- Jon Erickson. *Hacking: The Art of Exploitation.* No starch press, 2008 [88].
- Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* John Wiley & Sons, 3rd Edition, 2020 [18].

A lot of the core LangSec concepts stem from formal language theory. I benefited greatly from the following books in understanding formal languages and parsing algorithms:

- Michael Sipser, *Introduction to the Theory of Computation.* 3rd Edition, Cengage Learning, 2013 [250].
- Ceriel J.H. Jacobs and Dick Grune, *Parsing Techniques: A Practical Guide.* 2nd Edition, Springer, 2008 [109].

The reader may also benefit from a background in networking to follow some of the concepts in Chapter 4. The following cover fundamentals and modern concepts in networking:

- William Stallings, *Computer Networking with Internet Protocols and Technology*. Pearson/Prentice Hall, 2004 [261].
- Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley Professional, 1995 [287].

Finally, type theory is another related field to LangSec and ties into formal methods and verification. I recommend the following books for a more thorough introduction to these concepts:

- Benjamin C. Pierce, *Types and Programming Languages*. MIT press, 2002 [218].
- Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT press, 2012 [133].

Chapter 2

Language-Theoretic Security

LangSec posits that we need to take a principled approach to input recognition—rooted in formal language theory—to defend against exploits that rely on unintended computation [44, 238]. The LangSec methodology requires complete recognition of input before any processing happens. Once the input is stored in a local buffer, we must validate it before performing any other processing.

Section 2.1

LangSec in a nutshell

LangSec presents a paradigm to design and implement correct and verified parsers that exhaustively validate input based on a formal language. At its core, the principles of LangSec can be boiled down to the following:

- *Full recognition before processing:* The acceptable or valid set of inputs to a program must be defined using a formal language.
- *Principle of least expressiveness:* The formal language is kept as simple as necessary (following the principle of least expressiveness [91]) in the Chomsky Hierarchy [58].

- *Principle of parser computational equivalence:* Two parsers deciding the same grammar must be functionally equivalent.

We must separate the input validation process from the rest of the program in a way that the program never acts on inadequately validated input and acts on well-typed objects only. The parser must strictly adhere to the grammar and reject any non-conforming inputs. However, extracting and writing grammars for various networking protocols and file formats is not a trivial task. Such data formats often contain *dependent fields*, *length fields*, and sometimes arbitrary *seek* operations.

Recognition Before Computation

LangSec advocates recognizing the input language—or more precisely, deciding the input language—before the program performs any (non-parser) computation on that input. That is, the program should be separated into a parser which rejects all invalid input immediately, only allowing valid input to be passed on to the program internals.

Principle of Least Expressiveness

The principle of least privilege roughly states that an entity within a system, such as a user or a process, should operate with the least privilege required to perform the task at hand [237]. LangSec advocates a similar notion for the design of protocol specifications or grammars and their parser implementations.

The Principle of Least Expressiveness states:

> *One should always use the least expressive grammar, language, or automaton that achieves the task.*

This principle has the following two implications:

- Protocols should be designed to be minimally expressive: no complex structures that may be hard to implement.
- Parsers should be no more expressive than is required to decide the languages specified by the protocols they obey.

Moreover, constraining expressiveness may assist humans in correctly conceptualizing protocol specifications and parser implementations.

Principle of Parser Computational Equivalence

Secure composition is one of the leading security challenges. As such, it is vital to the security of interfaces. In particular, given two parsers that act on the same input, those parsers must wholly agree, i.e., for every possible input, either they both accept the input or they both reject it.

The Principle of Parser Computation Equivalence states:

> *Secure composition requires any two parsers that should decide the same language, e.g., due to sharing a component-component boundary, do decide that language.*

Recall that the equivalence problem is not decidable for non-deterministic context-free languages and more expressive languages, whereas it is decidable for deterministic context-free languages. When constructing parsers involved in secure composition, one should, therefore, aim to ensure they are no more expressive than deterministic context-free.

Section 2.2

The Halting Problem, Undecidability, and the Equivalence Problems

LangSec builds upon the theory of computation. Here, I give a very brief primer of the theory of computation and its branches to provide the requisite language and machinery to understand the theoretical underpinnings of modern-day exploits. For the reader who seeks a more complete treatment, I highly recommend Sipser's acclaimed book [250]. In formal language theory, an *alphabet* is a non-empty finite set of *symbols*. A string over an alphabet is a finite sequence of symbols belonging to that alphabet. For example, consider what is colloquially considered the English alphabet: $\Sigma = \{a,\ b,\ \ldots,\ z\}$. Here, $a \in \Sigma$ is a symbol belonging to the alphabet and the word $cat \in \Sigma^*$ is a string over the alphabet. A *language* is defined in relation to an alphabet as a set of strings over that alphabet. A natural way to express a language is to give a grammar. A *grammar* specifies rules, each of which is a mapping from a variable to a sequence of variables and symbols. A grammar comprises a start symbol along with a sequence of rules. The language of the grammar is simply the language comprising all strings generated by the start symbol. These language-theoretic notions form the building blocks of automata theory and computability theory—the connection being what computation is required to "capture" what language. This in turn lays the foundation for language-theoretic security.

As suggested by its name, the central focus of automata theory is *automata*— or mathematical models of computation— and their capabilities. Core to automata is the notion of *state*, which roughly refers to a condition that may dictate a subsequent course of action. Automata maintain state, often have some form of memory, and perform computation on input in pursuit of some task, such as generating output

or determining whether the supplied input is well-formed. The task that we are concerned with in this chapter, which is perhaps the most common task, is to accept or reject input. The automaton begins in a start state. The automaton (or perhaps its operator) then repeatedly examines the state that the automaton is in; based on that state, it will read symbols from the input and/or the memory that is part of the automaton, such as a stack or a tape, it may perform an action such as writing a symbol to memory, and it will transition to the next state. When supplied with input, the automaton may either run indefinitely or it may terminate when some condition is met, such as there being no symbols left to process. If the computation terminates, the state that the automaton is in during termination determines whether the input is to be accepted or rejected. That is, the automata we're interested in take as input a string and either *accept* or *reject* the string. An automaton *recognizes* a language if the automaton accepts only those input strings that belong to that language. And it *decides* a language if it accepts those strings that belong to the language and rejects those that do not (instead of sometimes merely running forever).

Central to discussion of languages, grammars, and automata is the notion of *expressiveness*, which enables us to meaningfully differentiate classes of a given type. A class of languages $\mathcal{L}$ is more *expressive* than a class of language $\mathcal{L}'$ if $\mathcal{L}$ is a strict superset of $\mathcal{L}'$. Similar notions of expressiveness exist for classes of grammars and automata, grounded in the languages associated with these grammars and automata. Thus, this notion of expressiveness intimately links classes of languages, grammars, and automata, e.g., the grammars corresponding to the finite state automata are regular grammars and the corresponding languages are regular languages. Moreover, it enables us to develop a nested classification schemes such as the well-known Chomsky Hierarchy. We find, for example, that in this classification hierarchy, finite state machines (and the corresponding grammar and language classes of regular grammars

and regular languages) are much less expressive than Turing machines (and the corresponding grammar and language classes of unrestricted grammars and recursively enumerable languages).

A particularly well-known, expressive class of automata is Turing machines. Running a Turing machine on an input can lead to three outcomes: accept, reject, or loop. The collection of strings M accepted or *recognized* by a language L, is denoted by L_M. Turing-recognizable languages are also called *Recursively enumerable languages*. When a Turing machine rejects an input, it may reject or simply loop—not providing a definitive answer. Hence, we prefer machines that halt on all inputs, or *decides*. Such Turing machines are called *Deciders* or *decidable* languages.

All decidable languages are Turing recognizable. Certain classes of languages are not recognizable using a Turing machine and there are still more that are undecidable. The classic example of a Turing-undecidable language comes from the Halting Problem, which involves designing a Turing machine that accepts all (M, I_M) pairs where M is a Turing machine and I_M is an input for which M halts when run on M and rejects all other Turing machine, input pairs.

It turns out that while the language of the halting problem, L_{TM}, is not decidable, it is recognizable. However, Turing-unrecognizable languages also exist; one such example is the complement of L_{TM} [250]. Recursive languages are exactly those languages that are decidable; notably, the slightly less expressive context-sensitive languages are also decidable.

Undecidability is relevant to LangSec because if an input language is computationally undecidable, then it *cannot* be validated—a validation layer will always let some crafted attack inputs through. In addition, if an input language falls under the class of recursively enumerable languages but not recursive languages (or Turing recognizable but not Turing decidable), then the validation process may not terminate

on specific inputs and may end up looping. It is also relevant because of the *equivalence problem*—determining whether the language of one grammar is equivalent to the language of another. In terms of LangSec: do two input validation layers accept the same input? While this problem is undecidable in the general case and for many context-free languages, it is decidable for deterministic context-free languages [242] and regular languages.

For most complex data formats that fall beyond the deterministic context-free boundary in the Chomsky Hierarchy, we cannot reasonably argue about the equivalence of two parsers implementing the same data format. Instead, developers have usually tackled this problem by visually comparing grammars and implementing the same parsing algorithms.

Section 2.3

Language-Theoretic Security vs. Language-Based Security

Since the two names sound similar, it would be very easy for the reader to be confused between these two separate research areas. Hence, I thought it would be good to understand what Language-based security is and how the name is the only thing similar between the two areas for all practical purposes.

The core principles of language-based security [239] can be distilled down to two points:

- *Principle of least privilege*: Each software system component must be provided with the least privilege possible.
- *Minimal trusted computing base (TCB)*: When the TCB is small, it is easier to argue and reason about its security.

These principles stem from understanding how common vulnerabilities propagate and how attacks proceed. For example, since a large software system traditionally runs as a user or root, the permissions and accesses are set accordingly. However, each tinier component of the larger system may not need access to all the accessible files or data.

Language-Based Security argues the need for ways to design fine-grained security policies and having ways to enforce these policies in the kernel. These techniques restrict memory access by way of strict policies.

We can enforce policies using type-safe programming languages and theorem provers where the correctness is guaranteed by the person writing the code. Additionally, security monitors provide another way of monitoring data requests and ensuring that each data access is valid per the specified policies.

LangSec also argues the need for policies—but policies strictly pertaining to the parsers. We provide a broad, well-defined notion of a functionally correct parser. By providing clear definitions of correct parser behaviors, LangSec sits on top of the stack of these methods. Any software system accepting input must apply LangSec principles and policies for better guarantees.

LangSec policies cannot be enforced dynamically using monitors in the same sense as Language-based security—but can be enforced using type checkers and theorem provers. We can ensure security guarantees for the parser code written.

Section 2.4

Weird Machines

LangSec principles emanate from a deep understanding of exploits—what exploits are, how they are crafted, and how they stem from unintended, latent computational capabilities exposed by programs. Core to these threads of inquiry is the *weird ma-*

chine [44], the engine of exploitation fueled by meticulously crafted input, which is the focus of this section.

> "Successful exploitation is always evidence of someone's incorrect assumptions about the computational nature of the system—in hindsight, which is 20-20."

To understand the weird machine, let us start with the basics, a program. A *program* takes (potentially zero) input, performs (potentially zero) computation on that input, and produces (potentially zero) output. The computation that the program may perform is specified by the underlying *machine* upon which it executes. The machine comprises a set of *instructions*, and the programmer carefully selects a sequence of instructions, each belonging to the instruction set provided by the machine, to achieve the programming task at hand.

In many instances, the software programmer creates a program that *seems* to work correctly; when given an anticipated input, it looks like it produces the correct output. However, in practice, it is often the case that carefully selected input may drive hidden computation that the programmer never intended. For example, supplying a given input to the program may result in a register value changing on the machine on which the target program is executed. These small bits of unintended input-driven computation are referred to as *weird instructions*.

2.4.1. Programming the weird machine

Thomas Dullien discussed the implications of a weird machine [81]. Dullien described how weird machines arise when the implementation fails to simulate the abstract machine correctly.

- Complex programs, if attacked, favor the attacker since they are harder to debug and isolate.

- Even the simplest programs may contain sufficient vectors for an attacker to launch an exploit.
- Automated exploit generation is equivalent to the problem of synthesizing a program for a machine.

Based on this description, an *exploit* is essentially a program to the weird machine that leads to the violation of various security properties in place—such as the integrity of the heap and stack.

Unexploitability statements It is difficult to claim a specific machine is "unexploitable." As we saw earlier, an exploit is essentially a program to a weird machine—proving unexploitability relies on proving security properties on the machine, which can be incredibly hard. Moreover, such a machine-assisted proof would have to iterate over every possible input combination and state in the weird machine.

Modeling Vulnerabilities as Developer Errors

Since input-handling vulnerabilities are still on the rise every year, several authors have created models to capture vulnerabilities as developer errors. These models capture vulnerabilities to understand why developers make these mistakes and what we can do to reduce future occurrences of these classes of vulnerabilities.

Vulnerabilities as blind spots A security blindspot occurs when a developer does not completely understand what an API is supposed to be doing. Since developers often do not think through all possibilities for their code and hence end up cutting corners, Cappos et al. [53] propose viewing software vulnerabilities as blind spots in a developer's decision-making process. The authors converted vulnerabilities into puzzles and proposed presenting these puzzles to a developer to understand these

blindspots better. These puzzles provide a user with a series of hints to reproduce the series of steps they followed in producing the flawed vulnerability.

In follow-on work, Oliviera et al. [24] conducted a study with 109 developers to understand the root causes of vulnerabilities. They presented developers with puzzles that were a series of questions about the APIs. By modeling security vulnerabilities as blindspots, they found that programmers find it much harder to detect vulnerable code in an API blindspot. They also found that even experienced developers find it harder to detect these vulnerabilities.

A developer survey Although developers are often blamed for software vulnerabilities, not enough research has been done to understand how tightly security fits into software development lifecycles. Assal et al. [207] explored why developers do not pay sufficient attention to security by conducting an online study with 123 participants. They reported that developers thought only 19% of their time was spent on security-related tasks on average. Developers also said they used mental checklists and various programming tools to ensure secure software.

Such HCI-based studies are vital to understanding why humans introduce so many vulnerabilities. We need more large-scale studies to understand coding patterns in large open-source projects, as well as corporations. With this understanding, we could provide users with more efficient and targeted tools to curb these vulnerabilities at every stage of the software development lifecycle.

Section 2.5

Parser Combinators

The idea of parser combinators originates from functional programming [122]. A general approach in functional programming to build recursive descent parser algo-

rithms is to define each grammar construction operation such as sequences, choices, and Kleene star as higher-order functions [210, 163, 73, 132]. This approach makes composing larger parsers using these combinators a lot easier.

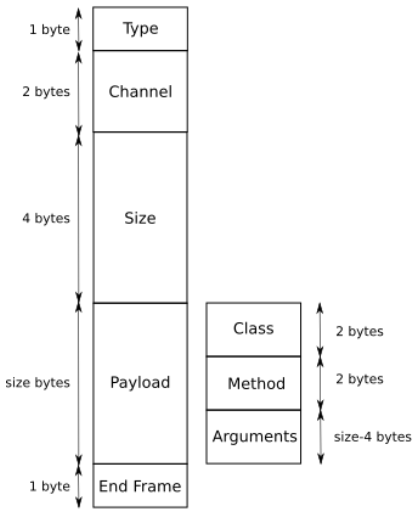
A parser-combinator tool is a library that provides these combinator functions—where each of these functions returns a parser object that can be called on input. On a successful parse, these parsers return an abstract syntax tree (AST) to provide access to these parsed objects.

Syntax	**Usage**	**Semantics**
`h_ch`	`h_ch('a')`	Matches a single specified character token.
`h_ch_range`	`h_ch_range('a','z')`	Matches a single token in the specified character range
`h_uint8`	`h_uint8()`	Matches a single integer token
`h_int_range`	`h_int_range(1,16)`	Matches a single integer token in the specified range
`h_sequence`	`h_sequence(h_ch('a'),h_ch('a')`	Performs as the concatenation combinator
`h_choice`	`h_choice(h_ch('a'),h_ch('b'))`	Performs the boolean "or" operation
`h_many`	`h_many(h_ch('a'))`	Performs as the Kleene Star operator
`h_optional`	`h_optional(h_('a'))`	Specifies that matching the inner string is optional

Table 2.1: Syntax and usage for Hammer.

Although parser combinators are standard in the functional programming world, Hammer [214] and Nom [66] were among the first parser combinators to be introduced in the systems programming universe. Meredith Patterson introduced the Hammer parser combinator as "Parser combinators for binary formats, in C." This toolkit introduces several combinators primarily applicable for binary formats. Combinators such as the ones used to define the endianness format and bit-level parsing are mostly only applicable to binary formats. Table 2.5 provides an example to the parser combinator syntax used in Hammer. Additionally, Figure 2.1 demonstrates the mapping of a data format to the parser combinator syntax.

Geoffoy Couprie took this work forward in 2015, introducing the first Rust-based parser combinator—Nom [66, 57]. They discuss how we can build parsers that are more suited to streaming protocols while focusing on eliminating memory copy instructions from their tool. They demonstrate how Nom combinators and Hammer




```
h_sequence(
    h_uint8(),     # Type
    h_uint16(),    # Channel

    h_and(h_length_value(
        h_uint32(),
        h_ch_range('\x00', '\xff'))),
    h_uint32(),     # Size
    h_sequence(
        h_uint16(), # Class
        h_uint16(), # Method
        h_optional(arguments)), # Arguments
    h_ch('\xce'),   # End Frame
    h_end_p(), NULL);
```

Figure 2.1: Mapping of a data format to parser combinator syntax (adapted from [11])

combinators can be used to build parsers for the MP4 video format. They found that Nom outperformed Hammer using less than a third of the time to parse the same MP4 files.

Section 2.6

Data Description Languages

A different paradigm from parser combinators, where we implement parsers in code, is the use of Data Description Languages (DDLs). In DDLs, we describe grammar in particular syntax and run a parser generator to generate code in any target language. Data Description Languages (DDLs) are essential to standardization efforts.

Standardization bodies can define the syntax and semantics of a data format in a DDL and rely on the parser generation tools to generate parsers for most languages. This approach can significantly reduce parser differentials.

RecordFlux defines a domain-specific language with its concrete syntax [228]. It allows users to specify dependencies in the data using invariants. Next, they generate

SPARK code to detail the behavior of various software components using contracts. They use this generated SPARK code to prove the absence of runtime error conditions and correctness in the generated code. Finally, they demonstrated their tool on various TLS protocol messages, including heartbeat.

Nail is a parser generator for binary protocols that supports user-defined functions to implement complex computations such as checksums and length fields [27]. Nail also supports offsets that DFDL and Kaitai do not support directly. In addition, nail grammars support real-world data formats such as DNS, UTF-16, and Ethernet packets. However, since Nail builds on Hammer, it does not free heap memory after use. Also, Nail generates C code, and the authors do not fuzz test the code to ensure reliability. Code Snippet 2.6 demonstrates the Nail syntax on a portion of the TCP segment.

```
segment = {
src_port uint16
dst_port uint16
seq_num uint32
ack_num uint32
}
```

Code Snippet 1: An example of the Nail language

Kaitai Struct is a data description language and a parser generation toolkit that supports runtimes in several languages [295]. It is one of the most used DDLs and supports various formats in its gallery. Kaitai relies on YAML syntax and unconventionally handles offsets using a syntax where locations and types associated with those locations are defined. Additionally, they also support a web IDE to build specifications and visualize them with data. Code Snippet 2.6 demonstrates the Kaitai Struct language on a simple TCP segment example. In this example, the endianness is set to big-endian. The *u2* and *u4* types denote unsigned integers of varying lengths.

```
meta:
  id: tcp_segment
  endian: be
seq:
  - id: src_port
    type: u2
  - id: dst_port
    type: u2
  - id: seq_num
    type: u4
  - id: ack_num
    type: u4
```

Code Snippet 2: An example of the Kaitai Struct Language

Data Format Description Language (DFDL) is an industry-standard often used to model text-based and binary data [181]. The DFDL specifications or schemas are used with the Daffodil runtime to parse data and convert the data to an information set (infoset). DFDL schemas are described in XML syntax. DFDL is unique in that it also includes a serializer to restore these infosets into binary data. However, DFDL does not support complex constructs such as offsets and checksums. The Code Snippet 2.6 provides an example similar to the TCP segment demonstrated in Snippet 2.6. This example demonstrates DFDL's XML syntax.

```
<xs:complexType name="TCP">
   <xs:sequence>
    <xs:element name='TCPHeader'>
     <xs:complexType>
      <xs:sequence>
        <xs:element name="PortSRC" type="b:bit" dfdl:length="16"/>
        <xs:element name="PortDest" type="b:bit" dfdl:length="16"/>
        <xs:element name="Seq" type="b:bit" dfdl:length="32"/>
        <xs:element name="Ack" type="b:bit" dfdl:length="32"/>
```

Code Snippet 3: An example of the DFDL schema Language

The DaeDaLus DDL [111] defines data formats in a syntax that closely resembles Nail [27]—while implementing it as a library within Haskell. Additionally, DaeDaLus also includes a parser generator for C++. At its core, DaeDaLus relies on various Haskell features, for example, offsets and other complex constructs.

The Parsley Data Definition Language [198] (hereafter, the Parsley language) provides users to input data descriptions in a BNF-like syntax while building on other parsing approaches such as data-dependent languages [138] and parsing expression grammars [98]. The Parsley language includes a compiler and an interpreter, both written in OCaml and designed to work for file formats such as PDF and network formats such as DNS.

DDLs provide a rich syntax to describe data format syntax and their constraints and dependencies. However, many user studies are still needed to understand the lack of significant adoption of DDLs in the industry. I envision that DDLs would provide a machine-readable, concise language to describe data formats instead of the long-winded text specifications available today. In addition, DDLs can play a huge role by generating parsers in various programming languages from a single machine-readable specification, bringing us closer to eliminating parser differentials.

Parser Combinators are not DDLs

In the previous section (Section 2.5), we studied parser combinators and their usage. The use case for parser combinators and DDLs are very similar. They are both used to produce parsers that can then be imported into existing applications. However, there are certain differences in their uses, and both methodologies have their positives and negatives.

Parser combinators normally do not generate code In the parser combinator approach, we commit to a programming language L and use a combinator library that supports input in the language L of our choice. DDLs, on the other hand, support code generation. We define the specification in a DDL and generate code in one of the languages supported in the code generators.

Porting parser combinator code Once we have a specification implemented using parser combinators, we need to rewrite the format using a new parser combinator syntax for the second language to port it from the programming language to another language. Unfortunately, even in parsing tools such as Hammer—where the same tool provides bindings in several languages—the syntax in different programming languages is different. This forces developers to rewrite syntax or build ways to translate the syntax from one language to another.

Correcting errors in DDLs DDL developers follow a methodology where any bug fixes are made to the original description in the DDL and not to the generated code. This means that every time a small tweak has to be made, we generate code again and import it into the application. In contrast, parser combinator developers can directly modify the parser code in the application.

Section 2.7

Parsing Algorithms

Parsing algorithms construct a parse tree based on the set of grammar productions to indicate how the input string can be produced from a given grammar. A parse tree for an input with n tokens would have n nodes belonging to the terminals and many nodes of the intermediate non-terminals. A depth-first traversal of the parse tree should produce the original input string.

Ambiguity An input that can be represented with more than one parse tree is known as *ambiguous*. Although ambiguity could be the only way to be parsed for some grammars, parsing algorithms for ambiguous grammars is highly inefficient. The Cocke-Younger-Kasami (CYK) [144, 298, 112] and Earley [83] algorithms can recognize unrestricted and ambiguous context-free grammars in $O(n^3)$ runtime, where n is the input size.

Context-Free Grammars These grammars occupy a prominent position in the research of parsing algorithms since the description of CFG rules translates well to a parsed-tree representation, and these grammars can capture natural language syntax well. Parsing algorithms are traditionally top-down or bottom-up. The Unger [274] and Earley [83] algorithms are standard top-down parsing algorithms, and the CYK algorithm presents a commonly used bottom-up algorithm for CFGs.

The LL parser is a deterministic top-down method that moves left to right on the input. In particular, the LL(1) parser is extremely popular [108]. The LR parser identifies the rightmost production instead of the leftmost production in the LL approach [164, 139]. The (1) in LL denotes how many character lookaheads are allowed. An unbounded LL(k) variant also exists. ANTLR uses an LL(*) algorithm that recognizes some context-sensitive languages [212]. LL(*) is a linear algorithm that supports a language with predicates and lookahead.

PEGs Bryan Ford proposed Parsing Expression Grammars (PEGs) [98] to resolve ambiguities in context-free grammars (CFGs) by (1) using ordered choice instead of an ambiguous, unordered choice, and (2) making the Kleene star operation greedy, instead of supporting backtracking. The Packrat algorithm generalizes the approach to parsing PEGs and uses linear time and space [97].

Jim et al. proposed YAKKER, a parsing engine that supports modern data-

processing applications [138]. They implemented a variation of Earley's parsing algorithm to parse their nondeterministic automata. Their algorithms support unrestricted context-free grammars and regular expressions on the right-hand side of nonterminal definitions.

Valiant's algorithm for unrestricted context-free grammar uses the efficient matrix multiplication algorithm to reduce the computational complexity from $O(n^3)$ to $O(n^{2.38})$. Jansson et al. formalized this algorithm in Agda [135].

Parsing with Derivatives Brzozowski derivatives were first defined for regular languages [47]. A derivative of a language is a subset of the language that has been filtered. For example, let us consider a language $\mathcal{L}$ containing the following strings $\{prash, poo, bar\}$. The derivative of this language with respect to the character p can be defined as:

$$D_p = \{rash, oo\}$$

Intuitively, the language set we get after performing the derivative, restricts the set of input to only those that start with the letter p in the above case. We then remove this initial character from the remaining set.

To determine if an input i or length n is in a language $\mathcal{L}$, we successively compute the derivatives n times. At each of the input stages, the derivative must be non-null, and the derivative must be null for the last input. Computing the derivative of a language with respect to an input token is straightforward.

However, derivatives become extremely complex when we expand this to CFGs for two reasons. First, since CFGs are recursive, the derivatives would never lead to null sets and hence would lead to non-termination. Second, derivatives only check input membership and do not produce a parse tree. Might et al. [184] present an extension to the derivatives approach to recognize CFGs. Henriksen et al. [114] also

extended the derivatives approach to recognize input in $O(n^2)$ time and produce parse trees in $O(n^3)$ time. They also showed connections between the derivatives obtained and the Earley sets. Most recently, in 2020, Edelmann et al. [84] presented a verified implementation of a linear-time algorithm to parse context-free expressions using derivatives.

Given the recent revival in work looking at using derivatives to parse formal languages, there may soon be extensions to these algorithms to look beyond context-free languages into data formats. Data format parsers usually require a parse tree generator and a recognizer. Given explorations in producing parse trees using derivatives, a data format parser may be an achievable target.

A more significant reason I believe parsing with derivatives is a useful research direction is the simplicity and explainability that comes with it. Parser debugging is a challenging problem, especially since often we are confused if the bugs exist in the parser, parser generator, or the language description. A derivative-based approach can provide the derivatives for each input stage, significantly reducing the debugging effort.

Hardware Parsing Algorithms Researchers have explored parsing context-free grammars (CFGs) in FPGAs. Most of these approaches provide ways to parallelize the traditional parsing algorithms. Ciresson et al. [61] presented an algorithm to parse unrestricted CFGs in FPGAs and demonstrated a 240x speedup to use it in natural language processing applications. Bordim et al. [39] implemented the Cocke-Kasami-Younger algorithm to parse CFGs in an FPGA and showed a 750x speedup over a software implementation. Taylor proposed generating FPGA code from parser code in C using High-Level Synthesis (HLS) tools [269]. They mostly focus their efforts on simplifying library functions in the Hammer parser toolkit to generate VHDL from it. Lucas et al. [171] provide a parallel implementation of PEGs in VHDL.

Section 2.8

Verified Parsers

Verified parsers have been explored for several years, ever since verification tools have been commonly available due to the application to building secure compilers. Such verification approaches prove several properties. For example, researchers often prove that a particular algorithm is equivalent to another naive algorithm by showing that these two algorithms produce the same output for all inputs. Researchers also use verification tools to prove specific properties, such as completeness or correctness, by providing a strict definition of these properties and a machine-assisted proof that the verified implementation satisfies these properties.

Verifying PEG parsers and interpreters has been a rich avenue for research since PEGs are unambiguous and can be parsed using a simple recursive-descent algorithm. In 2020, Blaudeau et al. [37] built a verified PEG parser in PVS. They defined what a well-formed PEG was and proved that their implementation terminates for all well-formed PEGs. TRX also formalized the notion of termination for PEGs [152]. The CakeML compiler used a PEG parser verified using HOL4 [156].

Barthwal et al. demonstrated an SLR parser generator verified in HOL4 that also produces a verifiable parse tree [29]. Ridge verified a recursive descent parser's termination and soundness properties by using parser combinators for context-free languages [230]. A lot of work has focused on building verified encoder-decoder pairs to be able to serialize and deserialize data [75, 277].

Verified Parsing of Binary Formats Unfortunately, not much work has happened in building verified parsers and parsing toolkits for binary formats. Van Geest et al. [277] described binary languages using the data types in a general-purpose dependently typed language. They demonstrated their parser and pretty-printer on

IPv4 protocol. Ramanandro et al. [226] also built a parser generator for tag-length-value languages. They generated parsers using a simple message format description and verified a parser-combinator toolkit using F* and Low*.

Additionally, in this thesis I implemented ParseSmith: a parser interpreter that can handle complex structures in binary formats (Chapter 6). Similar to the above approaches, we defined the security and safety properties vital to us and built invariants in code to prove those safety properties.

Section 2.9

Grammar Learning

There are thousands of known file formats—some open source and some proprietary. To adhere to the LangSec principles, the parsers for these data formats must conform to the standards specified. However, in practice, implementations do deviate in several ways [21]. Grammar learning approaches provide ways to learn the grammar from either a corpus of input or the protocol implementation. Higuera et al. [74] summarize and discuss decades of research in grammar inference—mostly focused on natural languages. Lee [162] summarize decades of research on learning context-free languages.

Protocol reverse engineering has its ties to the formal language problem of grammar induction or learning. We wish to learn a protocol syntax and state machine using network traces and implementation source code. Protocol syntax captures individual message formats within a network protocol—describing the various fields. A protocol state machine captures what messages are expected as a response to a message X.

Learning grammars solely from input samples is, however, not possible. Instead, algorithms such as Angluin's L* rely on an oracle to at least respond with coun-

terexamples given a learned regular grammar [19]. The algorithm will then use this counterexample to produce a more accurate grammar. Unfortunately, building such an oracle for protocols when we have no prior knowledge is difficult.

Protocol Reverse Engineering The goal of input reverse engineering is to understand the syntax of network protocol messages or a file format without specifications. Often companies use proprietary protocols and file formats to transmit data—making it challenging to build intrusion detection tools that can detect malformations in data that use these formats.

Another security application is understanding how a particular parser implementation deviates from other implementations in subtle ways. For example, Argyros et al. [21], and Poll et al. [134, 278] found that various operating systems implemented the TCP protocol state machine slightly differently—especially in the finishing states. Attackers can use these differences in parsers and state machines to fingerprint devices.

Hence, with these goals, reversing input formats is a significant problem and is of great relevance to the LangSec community. By finding these parser differentials, we can reduce mismatches between parsers and build a taxonomy of grammar drifts.

In 2007, Caballero et al. [49] proposed the idea of using dynamic analysis of program binaries to reverse engineer network protocols. They learned the syntax of five different network protocols and compared the learned syntax to the Wireshark implementation. They demonstrate that their tools can generate accurate syntax and be used for better grammar-aware fuzzing and fingerprint generation. Wondracek et al. [286] presented another technique to extract protocol grammars. Instead of using single packets to understand field boundaries, they use dynamic analysis on multiple inputs of the same type to extract a grammar specification for that message format. AutoFormat [169] monitors the call stack to gather information about fields and their

dependencies.

Cui et al. [70] presented Discoverer, a tool to use network traces to learn a format. They focus on learning the message formats and leave the protocol state machine to future work. They tokenize input grouping raw bytes or text and then cluster these tokens to discover patterns from several packets. The grammar learned by Discoverer may not be complete since it can only learn syntax that is present in the corpus.

Cui et al. later presented Tupni [71], a tool to extract file formats from an input corpus and an implementation of the format. They demonstrated their tool on complex formats such as WMV, BMP, PNG, and other network protocols such as DNS, RPC, HTTP, and FTP. They learn the boundaries to each field in the format, sequence of fields (such as long sequences of image chunks or DNS records), and constraints followed by fields. They rely on a taint tracking engine to trace x86 instructions directly. They also use Tupni to generate signatures for zero-day vulnerabilities for ShieldGen [72].

Most recently, Ye et al. [296] presented NetPlier, a technique that uses only network traces and no binary analysis to reverse a network protocol. They use a multiple sequence alignment instead of the single sequence alignment used in the past (Protocol Informatics [30], and Netzob [40]). Their key contribution is to probabilistically predict which locations are the keywords deciding what message type in the protocol it is. Normally, the first byte in the application layer is magic characters showing that the message conforms to a certain protocol (for example, the real-time publish-subscribe protocol uses the magic word "RTPS" at the beginning of a packet). Following these magic bytes, messages contain a field to specify what type of packet it is.

Protocol state machines are one level higher than message formats. Network protocols traditionally comprise multiple messages. To reverse engineer a protocol state machine using data, you need input sequences for each path in the protocol

state machine to understand all the different dependencies possible.

Wang et al. proposed Veritas, a tool to produce a probabilistic protocol state machine (P-PSM) [281]. They demonstrated their tool on several real-world protocols by generating these state machines from network traces. Prospex, on the other hand, uses program execution traces and dynamic analysis to extract the protocol state machine [63].

Krueger et al. [154] used specialized embedding and clustering techniques to reverse message formats and state machines. First, they embed a message in a vector space to understand statistically significant features. Next, each group or sequence of events is labeled as an event. Finally, they compose these events to produce a probabilistic state machine using Markov models.

Grammar Learning Algorithms Learning formal grammars has its applications. First, by learning the grammar syntax followed by input in a corpus, we can identify various common patterns seen in the input. Second, learning the natural language followed by the input also aids in speech recognition.

Feasibility One of the crucial questions in learning grammars was answered in 1967 by Mark Gold. It is not feasible to learn grammars in any of the four classes of languages in the Chomsky hierarchy using only a corpus of input samples [162, 103]. In addition, Angluin showed that even with the ability to ask equivalence queries, there is no guarantee that the grammar would be found in polynomial time [20]. We study how researchers have gotten around these two theorems by relying on dynamic taint analysis, probabilistic approaches, and other learning techniques.

Hoschele et al. [121] use language induction along with taint tracking to understand the origins of various input fragments by generating context-free grammars using dynamic analysis. First, these input fragments are inserted in an interval tree.

The children of any node in this interval tree would correspond to the entire input on the root node. For each node in the interval tree, they then compute all possible productions by creating a sequence of other terminals and non-terminals. They demonstrate their grammar induction tools on formats such as CSV, JSON, INI, and URLs. Of these formats, only JSON needs to be described using context-free grammars. Other formats can be described using regular languages.

Argyros et al. [21] presented SFADiff, a tool to learn the grammars for web application firewalls and TCP protocol state machines. In addition, they demonstrated a framework for learning symbolic-finite automata (SFA) and comparing them. Their approach relies on the Angluin L* algorithm and only applies to regular languages. However, this approach presents a way to compare protocol implementations to uncover parser differentials.

Cowger et al. built ICARUS, a grammar inference engine that uses reinforcement learning [67]. They used "merge parsing" to combine neighboring terminals and nonterminals repeatedly. The merge policy is random at first, and then the rewards are adjusted based on the expected parse trees. They eventually reach an approximate merge policy. They demonstrated their approach on context-free grammars in the JSON format but not on other practical data formats.

In summary, being able to infer grammar, data format syntax, and the protocol state machine is crucial to building better intrusion detection tools. For decades, researchers have looked at what information would be needed to predict the grammar followed by input accurately. More recently, researchers have turned to reinforcement learning and dynamic analysis to leverage parser code other than just using an input corpus. However, many traditional algorithms, such as L*, used to learn grammars require oracles or teachers that provide counterexamples if a particular grammar produced is incorrect. This is not a reasonable expectation in the case of black-box

proprietary protocols and implementations.

Section 2.10

Conclusions

This section provided an overview and survey of prior LangSec research focused on various topics. I believe that each of these topics provides an avenue for plenty of future work. In addition, a lot of these LangSec sub-areas need to progress in tandem for better outcomes. For example, as we design new language classes to capture data formats in the existing Chomsky Hierarchy, we must also design newer grammar learning approaches to learn these languages automatically and new parsing algorithms to recognize input in these languages.

Chapter 3

Motivation

Understanding how vulnerabilities occur in code and how they are exploited is key to building better software and security solutions that can tackle exploits. In previous chapters, we understood what the core principles of LangSec are and various research directions tied to LangSec. In this chapter, I will discuss some of my early work in understanding vulnerabilities and categorizing them based on various features. This chapter motivates my work in later chapters.

Although developers introduce bugs and vulnerabilities in code, exploits need to be crafted to take advantage of such artifacts. Attackers craft input to programs to create such exploits. These inputs are usually buffers read from sockets, files, or direct user input. For these exploits to succeed, the parser—the first line of defense for any program accepting input—must have a vulnerability or allow an exploit through.

Input-handling vulnerabilities can be triggered via crafted user input. Buffer overflows and buffer overreads are examples of such vulnerabilities. Exploits using input-handling vulnerabilities could cause damage to a user in various ways. (1) Attackers can gain remote access to a machine. (2) Attackers can cause a denial-of-service attack by crashing a process using a crafted input. (3) Attackers can leak private information from a host machine. I describe buffer overflows and buffer overreads

and provide various real-world examples of these attacks.

Buffer overflows and overreads

Buffer overflows occur when we try to copy a buffer into a destination buffer of a smaller size [208]. This results in the memory locations adjacent to the destination buffer being overwritten. An attacker can replace the subroutine return address on the stack using a buffer overflow and return to a subroutine in `libc` which includes useful commands such as `system` [77]. Attackers can use a command such as `system("/bin/sh")` to get access to a shell on the target machine.

The first known occurrence of a buffer overflow was in 1972 [17] when the Computer Security Technology Planning Study noted that some functions do not check the source and destination addresses, which can lead to some of the kernel memory being overwritten. However, the first known occurrence of a buffer overflow in the wild was the Morris Worm [256].

Morris Worm used the Finger command on Unix systems to replicate [241, 31]. The Finger command accepts a request about a user and provides simple information such as their name, location, and extension numbers as a response. The Finger command, however, accepts the remote request using `gets()` with a 512-byte buffer on the stack. Hence, Morris Worm uses a 536-byte message to overrun the stack and overwrite the return address.

Buffer overreads are another type of memory safety violation. An attacker can trick a program into reading data beyond its buffer boundaries. Let us consider the example of the Heartbleed bug [82, 301]. Heartbeat messages are used to keep OpenSSL encrypted connections alive. These messages follow the format shown in Figure 3.1. The heartbeat message holds a size and a payload field.

The receiving party needs to send back a message with the identical size and

Figure 3.1: Syntax of a Heartbeat message in TLS. The client sends a request and expects to receive an identical response with just the packet type changed.

payload. However, if the payload length and the size field do not match, the receiving party must silently drop the packet. Instead, the library would read the number of bytes prescribed by the size and send them back to the sender. If the attacker sends over a large size value, such as 32000, they can expect to read the private contents of memory, such as private keys. An attack such as Heartbleed does not leave logs in the operating system and can be very difficult to detect.

Similarly, there was a parser bug in popular implementations of the point-to-point protocol (PPP). PPP is widely used to establish internet connections and transfer data [248]. PPP implementations also support the Extensible Authentication Protocol (EAP) to provide additional authentication. In 2020 it was found that an attacker could send a crafted EAP packet to a vulnerable PPP server and use the unverified data in the malformed packet to corrupt the stack using a stack-based buffer overflow vulnerability [145]. Surprisingly, this bug existed in the code base for the last 17 years. Since PPP software often runs with high privileges such as root on Linux machines, attackers could use this vulnerability to execute arbitrary code.

Parser Differentials

When a new deployment of a particular protocol needs to interact with other existing deployments, most systems tend to be more permissive and accept data with various malformations and deficiencies. However, by following such a paradigm, over time, we would get several implementations of the same protocol that disagree with each other in subtle and tiny ways. Such parser implementations that deviate from the

specification and each other are called *parser differentials*.

A classic case of a parser differential was found by Dan Kaminsky, Len Sassaman, and Meredith Patterson in 2010 [142]. They found that the URLs in certificates get parsed very differently by various OpenSSL parsers. For example, when a null character `\0` is added in the middle of a URL in a certificate, not only does a certificate authority issue a certificate with this null character, different parsers read the URLs differently. Some parsers terminate the string at the null character. Some other parsers either skip the null character or leave the null character in the middle of the URL. An attacker can leverage these inconsistencies to craft certificates that get rendered differently on different browsers—leading to phishing attacks where we may think we are on a secure, HTTPS-enabled web page.

Spath et al. [258] discuss several parser differentials in various XML parsers. They crafted several XML messages to cause denial of service attacks by resource consumption. They also craft messages to gain file system access on the host machine. They found that most of these XML parsers were susceptible to some of these crafted files. However, some of the countermeasures they propose to limit and counteract these vulnerabilities included limiting the allocated resources by implementing thresholds and filtering by validating input based on allowlists and blocklists. This thesis argues that although limiting thresholds can be a good software engineering practice, we need to take a more holistic approach to input validation instead of just using lists.

Apple uses property lists or plists to store serialized data such as configurations and permissions [300]. The plists that store code signatures are called entitlements. Siguza found that iOS used at least four different XML parsers, and they have differentials in how they handle comments in XML. As a result, an attacker can craft plists containing material that some parsers ignore as comments, whereas others believe to be authentic signatures.

In 2013, Jeff Forristal revealed the Android Master Key vulnerability at the Black Hat Conference [136]. This was yet another case of a parser differential. Since Android applications are essentially a collection of compressed (ZIP) files, this attack leverages the fact that if there are multiple versions of a file within a ZIP archive, Android verifies and extracts only the first version. Therefore, the implications of this attack were huge—attackers could replace files within an application to launch arbitrary code without users detection.

Most recently, in January 2022, Moshe et al. [196] and Stenberg [263] found that there are two specifications commonly used to define URLs—RFC 3986 and the Web Hypertext Application Technology Working Group (WHATWG). The WHATWG URL specification was found to differ from RFC 3986 in some minor ways, such as handling both slashes the same. Additionally, Moshe et al. tested four URL-encoding confusions against 16 popular URL libraries across languages and browsers. They found that implementations sometimes used multiple URL parsing libraries in code that may be incompatible and led to denial-of-service vulnerabilities.

In summary, parser differentials and improper input validation are two broad categories of input-handling vulnerabilities that commonly appear in large systems. Using memory-safe languages mitigates many vulnerabilities commonly seen in C/C++ applications. However, parser differentials can occur in any programming language and data format. Therefore, we need better tools to aid developers in designing parsers that are less susceptible to these attacks. Similarly, we also need robust description languages to aid standardization bodies in developing machine-readable specifications.

Taxonomies of Input-Handling Vulnerabilities

In recent years, several popular vulnerabilities such as HeartBleed [82], Android Master Key [136], and Apple Mail's 0-click attack [300] have been parser errors of some

kind. Although all of these vulnerabilities are triggered via user input, we describe how they differ in the subsections that follow. Each of these vulnerabilities stems from an underlying anti-pattern followed by the developers.

In the rest of this section, we discuss various taxonomies describing LangSec anti-patterns and input-handling vulnerabilities. Such a discussion would provide the reader with an understanding of what kind of anti-patterns and vulnerabilities are common and how the LangSec methodology tackles them. First, we describe a taxonomy where we understand the root causes of vulnerabilities in code by capturing them as mismatches in interpretations or mismorphisms (Section 3.1). Then, we present a categorization of vulnerabilities based on the nature of the bugs as perceived by developers using software (Section 3.2).

Section 3.1

Mismorphism-based Approach[1]

Mismatches between the perceptions of the designer, the implementor, and the user often result in protocol vulnerabilities [14, 8]. The designer has a high-level vision for how they believe the protocol should function, and this vision guides the creation of the specification. In practice, the specification may diverge from the initial vision due to real-world constraints, e.g., hardware or real-time requirements. The implementor then produces code to meet the specification based on their perceptions of how the protocol should function and, in some cases, how the user will interact with it. However, incorrect assumptions may produce vulnerabilities in the form of bugs or unintended operation. A user—informed by their own assumptions and perceptions—may then interact with a system or service that relies upon the protocol. A misunderstanding of the protocol and its operation can drive the user toward a de-

[1]This section borrows from my previous work in [14, 8]. I primarily collaborated with Vijay Kothari and Sean Smith on this work.

cision that produces an unintended outcome. Ultimately, the security of the protocol rests on the consistency between the various actors' mental models of the protocol, the protocol specification, and the protocol implementation.

A *mismorphism* refers to a mapping between different representations of reality (e.g., the distinct mental model of the protocol designer, the protocol implementor, and the end user) for which properties that ought to be preserved are not. In the past, we have used this concept and an accompanying semiotic-triad-based model to succinctly express the root causes of usable security failures [253]. We now apply this model to protocol design, development, and use. As mentioned earlier, many vulnerabilities stem from a mismatch between different actors' representations of protocols and the protocol operation in practice, e.g., the HeartBleed [82] vulnerability embodies a mismatch between the protocol specification, which involved validating a length field, and the implementation, which failed to do so. Therefore, it is natural to adopt the mismorphism model to examine the root causes of protocol vulnerabilities. That is precisely what we do in this work.

We examined protocol vulnerabilities and the mismorphisms upon which they are rooted. We developed a logic to express these mismorphisms, which enables us to capture the human mismatches that produce in vulnerabilities in code. Finally, we used this logical formalism to catalog the underlying mismorphisms that produce real-world vulnerabilities.

To represent mismorphisms we need a way to express the relationship between interpretations of a predicate. Thus, we have the following:

$$[Predicate]\ [Interpretation\ Relation]\ [List\ of\ interpreters]$$

The interpretation relations over the set of all predicate-interpreter pairs are k-ary relations where $k >= 2$ is the number of interpreters there are in the interpretation

relation— and each k-ary relation is over the interpretations of the predicate by the k interpreters. The three interpretation relations we are concerned with in this work are: the interpretation-equivalence relation ($\underset{\text{interp.}}{=}$), the interpretation-uncertainty relation ($\underset{\text{interp.}}{\overset{?}{=}}$), and the interpretation-inequivalence relation ($\underset{\text{interp.}}{\neq}$).[2] These relations are defined as follows, where each P represents a predicate and each A_i represents an interpreter:

- $P \underset{\text{interp.}}{=} A_1, A_2, \ldots A_k$ if and only if P, as interpreted by each A_i, has a truth value that's either T or F (never U)— and all interpretations yield the same truth value.

- $P \underset{\text{interp.}}{\overset{?}{=}} A_1, A_2, \ldots A_k$ if and only if P takes on the value U when interpreted by at least one A_i.

- $P \underset{\text{interp.}}{\neq} A_1, A_2, \ldots A_k$ if and only if P interpreted by A_i is T and P interpreted by A_j is F for some $i \neq j$.

There are a few important observations to note here. One is that the oracle O always holds the correct truth value for the predicate by definition. Another is that if we only know the $\underset{\text{interp.}}{\overset{?}{=}}$ relation applies, we won't know which interpreter is uncertain about the predicate or even how many interpreters are uncertain unless $k = 2$ and one interpreter is the oracle. Similarly, if we only know that the $\underset{\text{interp.}}{\neq}$ relation applies, we do not know where the mismatch exists unless $k = 2$. That said, knowledge that the oracle O always holds the correct interpretation combined with other facts can help specify where the uncertainty or inequivalence stems from. Last, the $\underset{\text{interp.}}{=}$ relation will not be applicable if either the $\underset{\text{interp.}}{\overset{?}{=}}$ or the $\underset{\text{interp.}}{\neq}$ interpretations are applicable; however, $P \underset{\text{interp.}}{\overset{?}{=}} A_1, \ldots A_k$ and $P \underset{\text{interp.}}{\neq} A_1, \ldots A_k$ can both be applicable simultaneously.

[2]Note that for $k = 2$, if we confine ourselves to predicates that take on only T or F values, the relation $\underset{\text{interp.}}{=}$ is an equivalence relation in the mathematical sense, as one might expect, i.e., it obeys reflexivity, commutativity, and transitivity.

The purpose of creating this model was to allow us to capture mismorphisms. Mismorphisms correspond to instances where either the interpretation-uncertainty relation or interpretation-inequivalence relation apply.

Shotgun Parsers

Shotgun parsers perform input data checking and handling interspersed with processing logic.[3] Shotgun parsers do not perform full recognition before the data is processed. Hence, implementors may assume that a field x has the same value at time t and time $t + \delta$, but the processing logic may change the value of the field x in an input buffer B.

This mismorphism relation is seen below:

$$B(t) = B(t + \delta) \underset{\text{interp.}}{\neq} O, I \tag{3.1}$$

Implementors may expect the buffer to be intact across time, but that is not observed to be the case. Shotgun parsing can cause mismorphisms in two distinct ways. First, a partially validated input may be wrongly treated as though it is fully validated. Suppose Implementor 1 performs the shotgun parsing and know the input to be only partially validated. Then, Implementor 2 works on execution and assumes the input is fully validated by the time the code segment is executed. This type of a shotgun parser mismorphism can be represented as follows:

$$B \text{ is accepted} \underset{\text{interp.}}{\neq} I_1, I_2 \tag{3.2}$$

Second, the same implementor may perform shotgun parsing and be responsible for

[3]The term shotgun parser has to be rethought. Phrases associated with military actions or violence are not inclusive: `https://docs.microsoft.com/en-us/style-guide/bias-free-communication`. I propose using the term *sprinkled parsers* instead of shotgun parsers.

working on the execution code. But they may interpret the same protocol differently during those different times. This type of a mismorphism can be represented as follows:

$$B \text{ is accepted} \underset{\text{interp.}}{\neq} I^{t_1}, I^{t_2} \tag{3.3}$$

Parser Inequivalence for the Same Protocol

Designers of protocols intend for two endpoints to have the exact same functionality, and build identical parse trees. The Android Master Key bug [136] is an apt example for this type of a mismorphism. The parsers for the unzipping function in Java and C++ were not equivalent, leading to a parsing differential.

We describe this relation as:

$$\text{Parsers } P_1, P_2 \text{ are equivalent} \underset{\text{interp.}}{\neq} O, D, I \tag{3.4}$$

Implementor is Unaware that Some Fields Must Be Validated

Designers of protocols introduce new features in the specification of the protocol without describing them fully or accurately. The designer introduces a field x in the protocol, but the interpreter does not entirely understand how to interpret it. The HeartBleed vulnerability [82] was an example of this. The designers included the heartbeat message, but the implementors did not completely understand it and missed an additional check to make sure the length fields matched.

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I \tag{3.5}$$

Types of Fields in Buffer are Not Fixed During Buffer Life Cycle

The types of values that have already been parsed must remain constant. Sometimes, implementors assume field x is treated as type $t(x)$ throughout execution. In reality, the field may be treated as a different type at certain points during execution.

$$type(x) \text{ is fixed} \underset{\text{interp.}}{\neq} O, D, I \tag{3.6}$$

3.1.1. A Catalog of Vulnerabilities and Their Mismorphisms

Below, we provide a small catalog of some vulnerabilities and the mismorphisms that we believe produced them.

Shellshock: Bash unintentionally executes commands that are concatenated to function definitions that are inside environment variables [180].

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$$

The sanity check C here makes sure that once functions are terminated, the variable shouldn't be reading commands that follow it.

Rosetta Flash: SWF files that are requested using JSONP are incorrectly parsed once they are compressed using *zlib*. Compressed SWF files can contain only alphanumeric characters [257].

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$$

The specification of the SWF file format is not exhaustively validated using a grammar. The fix uses conditions such as checking for the first and last bytes for special, non-alphanumeric characters.

Heartbleed: The protocol involves two length fields, one that specifies the total length of the heartbeat message; the other specifies the size of the payload of the heartbeat message [82].

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$$

Sanity check C involves verifying the length fields l_1 and l_2 match.

Android Master Key: The Java and C++ implementations of the cryptographic library performing unzipping were not equivalent [99].

$$\text{Parsers } P_1, P_2 \text{ are equivalent} \underset{\text{interp.}}{\neq} O, D, I$$

Ruby on Rails - Omakase The Rails YAML loader doesn't validate the input string and check that it is valid JSON. And it doesn't load the entire JSON; instead, it just starts replacing characters to convert JSON to YAML [229].

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$$

Sanity check C should first recognize and make sure the JSON is well-formed, before replacing the characters in YAML.

Nginx HTTP Chunked Encoding: Large chunk size for the Transfer-Encoding chunk size trigger integer signedness error and a stack-based buffer overflow [175].

$$B \text{ is accepted} \underset{\text{interp.}}{\neq} I_1, I_2$$

The shotgun parser works on execution without validating the value of the length field, which could be much larger than allowed, thereby causing buffer overflows.

All implementors must work with the same knowledge, and the input must first be recognized fully.

Elasticsearch Crafted Script Bug: Elasticsearch runs Groovy scripts directly in a sandbox. Attackers were able to craft a script that would bypass the sandbox check and execute shell commands [195].

$$L \text{ is decidable } \underset{\text{interp.}}{\overset{?}{=}} O, D$$

Developers of Elasticsearch had to explore the option of abandoning Groovy in favor of a safe and less dynamic alternative.

Mozilla NSS Null Character Bug: When domain names included a null character, there was a discrepancy between the way certificate authorities issued certificates and the way SSL clients handled them. Certificate authorities issued certificates for the domain after the null character, whereas the SSL clients used the domain name ahead of the null character [141].

$$\text{Parsers } P_1, P_2 \text{ are equivalent } \underset{\text{interp.}}{\neq} O, D, I$$

Although having a null character in a certificate is not accepted behavior, certificate authorities and clients do not want to ignore requests that contain them. So they follow their own interpretations, resulting in a parser differential.

Adobe Reader CVE-2013-2729: In running length encoded bitmaps, Adobe Reader wrote pixel values to arbitrary memory locations since there was a bounds check that was skipped [93].

$$B \text{ is accepted } \underset{\text{interp.}}{\neq} I_1, I_2$$

The code used a shotgun parser where the implementor of the processing logic assumed all fields were validated. The bounds check was never performed.

OpenBSD - Fragmented ICMPv6 Packet Remote Execution: Fragmented ICMP6 packets cause an overflow in the mbuf data structure in the kernel may cause a kernel panic or remote code execution depending on packet contents [209].

$$\text{sanity check } C \text{ is performed} \underset{\text{interp.}}{\neq} O, D, I$$

Implementors of the ICMP6 packet structures in OpenBSD did not understand how to map it to the existing `mbuf` structure, and then validate it.

Conclusions

In this section, we proposed a novel approach to categorizing the root causes of protocol vulnerabilities. We created a new logical model to express mismorphisms, grounded in the semiotic-triad based representation of mismorphisms explored in our earlier work. We then used this logical model to develop a preliminary set of mismorphism classes for capturing LangSec vulnerabilities. Finally, we created a small catalog of vulnerabilities and demonstrated how our classification scheme could be used to classify the mismorphisms those vulnerabilities embody.

Section 3.2

Categorizing Vulnerabilities[4]

Mamot et al. [194] noted that several CWEs already define do cover a wide range of LangSec vulnerabilities. Such CWEs such as CWE-805 cover buffer overflows [191]. At the same time, there are CWEs to cover cross-site scripting attacks and SQL

[4]This section borrows from my previous work in [3]. I collaborated with Sameed Ali, Zephyr Lucas, and Sean Smith on this project.

injection attacks. Most LangSec bugs fall under CWE-20: Improper Input Validation [190].

However, this category of "Improper Input Validation" is a wide one. It combines many issues with input validation—first, situations when the validation is insufficient. The parser may have missed validating a few fields. Second, different parsers may interpret a specification differently. Such a situation may lead to a mismatch between the two parsers. Finally, the parser does not reject invalid input but accepts some versions of invalid input.

Although these are all different kinds of "improper input validation", CWEs tend to club them together with similar solutions. Each of these subcategories of poor input handling stems from different underlying causes.

We took a different approach from Mamot et al. to categorize vulnerabilities. We categorized vulnerabilities based on the LangSec anti-pattern that led to the vulnerability. The categories we came up with were:

- **Flawed and Insufficient Parsers: Basic LangSec Failures.** Often, flawed parser code implicitly assumes that the input is correctly formatted and acts on the input. The Ripple20 vulnerability was found in the Treck TCP/IP stack used in several embedded applications [60]. The DNS resolver in this Treck stack failed to validate data from external entities. Similarly, the Thales module used in billions of Internet of Things (IoT) devices was found to have a vulnerability of a similar nature [158]. The parser had a flaw in ignoring certain characters in the input, rendering the module vulnerable.

- **Official Grammar is Wrong: When specifications are incorrect.** Occasionally, the specifications can specify incorrect grammar concerning what is commonly used. For example, in the "Wallpaper of death" vulnerability, although the attacker uses a well-formatted image file since the Android op-

erating system converts the image from one file to another, the specification for this conversion was incorrect [35]. Similarly, ZecOps discovered zero-click attacks in default email applications in Apple devices [300]. Attackers craft valid but devious emails that can exploit various vulnerabilities in these email applications.

– **Parsing Differentials.** We already discussed parser differentials extensively in Section 3.

– **Semantic Differentials.** Although the underlying parsers may function correctly as per the specification, the results displayed to a user could be vastly different. For example, several attacks have been crafted against PDF viewers [177]. In addition, some PDF viewers may allow attackers to overlay text with their devious text in the PDF file. However, the PDF viewer uses the previous signature to tell a user that the file has been signed, so there is no reason for a user to worry. This semantic differential demonstrates a mismatch between what a user may think a PDF signature means and how PDF viewers interpret signatures—since signatures may not sign the entire PDF file.

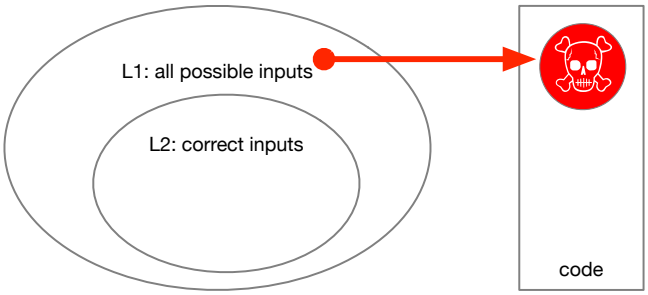


Figure 3.2: In the basic anti-pattern, the flawed code implicitly assumes the input is in L_2 but does not check; attacks are possible when the adversary crafts an input in $L_1 \setminus L_2$.

Figure 3.2 shows the basic LangSec anti-pattern, in which flawed code implicitly

assumes the input is correctly formatted and acts on invalid input as a result. The three vulnerabilities described below fit resulted from this anti-pattern.

Ripple20 June 2020 brought news of *Ripple20*, a set of vulnerabilities in the Treck TCP/IP stack. This TCP/IP stack is over 20 years old but used in "hundreds of millions" of embedded systems, including critical infrastructure such as power, aircraft, and healthcare [60]. Besides the direct threat to the large number of compromised systems themselves, Ripple20 has additional high impact downstream consequences because it is a networking stack, that, by definition, exposes an interface to the outside world. Consequently, Ripple20 vulnerabilities may enable adversaries to read and write data on the devices, as well as execute code, and thus compromise other systems. Remediation will be difficult as real-world embedded systems are notoriously hard to track down and patch.

Two of Ripple's vulnerabilities (i.e., "#1" and "#3" in [151]) are straightforward failures by the DNS resolver to validate external input.

- In the first vulnerability, flawed code constructs a requested hostname and copies it into a buffer. However, the adversary-supplied MX record provides both the `RDLENGTH` value used to calculate the size of the buffer, as well as the data from which the requested hostname is constructed. As a result, the adversary can supply a record that tricks the code into corrupting its heap by copying a large byte sequence into a too-small buffer. Correctly behaving code should have rejected this malformed record.

- In the second vulnerability, the flawed code does not correctly confine reads to data relevant to the adversary-supplied record—so a malformed record can trick the code into copying internal data to an output buffer.

Thales Module August brought news of a vulnerability in a Thales module potentially used in "billions of things" [158]. At its core, this vulnerability manifests as a variation on the basic LangSec anti-pattern. The variation is shown in Figure 3.3. In the case of the Thales module, flawed code determines access control by checking if an input parameter specifies something forbidden—but this test implicitly assumes the input parameter is correctly formatted (that is, a path with a single slash). If the adversary goes outside the safe input space by supplying what would be a forbidden parameter, except using two slashes instead of one, the flawed code accepts the parameter and then ignores the second slash.

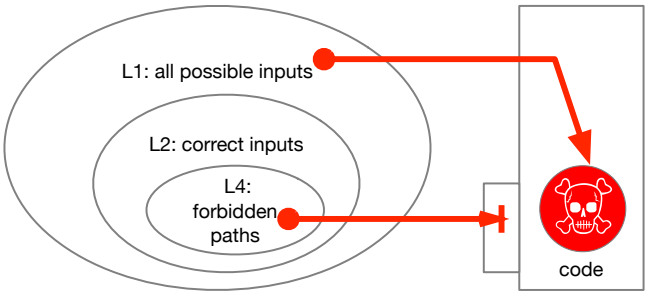


Figure 3.3: In a variation of the basic anti-pattern, the flawed code checks for disallowed requests (L_4) but implicitly assumes these requests are in L_2; the adversary can craft inputs in $L_1 \setminus L_2$ that bypass the checks but get interpreted as something in L_4

How LangSec can fix these problems In these examples based on exploiting the basic LangSec anti-pattern, the software programs expected their input to follow certain rules and the attacker capitalized on these expectations by providing data that was not structured according to these rules but processed nonetheless. This in turn caused unexpected and harmful computation to be performed. LangSec calls for these input rules to be formally described as a grammar so that a verifier could parse any potential input to see if it adheres to these rules before passing it on for processing. In the Ripple20 vulnerabilities, a parser would have recognized malformed records

and rejected them before any harmful computation would have been done. Similarly, in the Thales Module, an input parser would have rejected input parameters with excess slashes, therefore protecting the code from the forbidden paths.

3.2.1. The Official Grammar is Wrong: When Specifications are Incorrect

The standard LangSec defense pattern to the problems of Figures 3.2 and 3.3 is to formally specify an expected input language, L_2, to test whether an input is in L_2 before acting on it. However, this approach raises the question (depicted in Figure 3.4): just because an input is in the language the developer expects, does that it mean the code will handle it safely? This section considers some recent vulnerabilities for which the answer to the question (above) is "no."

Summer 2020 saw many flaws arising because the set of correctly handled inputs are only a proper subset of the inputs within the validation grammar. This can stem from either an error in formally describing the intended input, or from improperly handling all intended inputs.

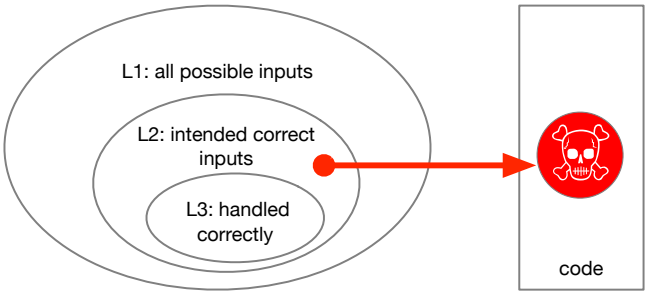


Figure 3.4: In many cases, the flawed code is intended to handle some target L_2 of valid inputs—but instead properly handles only a proper subset L_3. Attacks are possible when the adversary crafts an input in $L_2 \setminus L_3$. Simply validating against L_2 is not enough—we need tools to help ensure that $L_2 \setminus L_3$ is empty.

Wallpaper of death June brought news of the Android "Wallpaper of Death." [35] A particular picture, when set as the background image for a Samsung or Google Pixel (or other phones which use the default Android color engine), will softlock the phone by causing it to endlessly reboot. The picture contains a pixel whose color values cause an overflow error when the phone is calculating the luminescence value that the OS handles by rebooting the phone. During the reboot process, the background image is loaded, overflows and initiates another reboot. This renders the phone to be unusable until the user boots the phone in safe mode and deletes the image resetting the phone to the default background.

This raises some interesting LangSec questions as the image in question uses a properly formatted RGB color space—but the phone first transforms the RGB to sRGB, and the transform code can turn some valid RGB images (such as this one) into illegally formatted sRGB ones.

Non-Click iOS Email 0day In April this year, ZecOps reported that they found a zero-click attack on iPhone and iPad default email applications. [300] Users do not need to take any action; when the email shows up in their mailbox, it gives access to the attacker. The attacker can even delete the crafted email that gave them access. Such zero-click attacks are hence extremely difficult to detect. ZecOps reported that several top-level executives and journalists across the world might have been affected by this vulnerability.

ZecOps reported that correctly formatted (but deviously crafted) email could trigger vulnerabilities:

- Out of bounds write: the email application calls the `ftruncate` syscall to truncate or extend a file to a specific length, but does not check the error condition. With the right input, the application will write beyond the end of mapped space.

– Heap-based overflow: the email application tries to copy `0x200000` bytes into memory allocated using `mmap`. However, if this syscall fails, the application only allocates 8 bytes and copies over the much larger `0x200000` bytes, overflowing the heap.

Snapdragon August brought news of a large number of vulnerabilities within the Snapdragon Digital Signal Processor (DSP) used in many Android smartphones, including but not limited to Google, Samsung, OnePlus, Xiaomi, LG, Android and are accessible by third party applications. [222] Such devices have many separate processors; communication between them is handled by software generated by the Hexagon SDK. However, as Slava Makkaveev discussed at DefCon,[5] this code fails to handle certain kinds of correctly (but oddly) formatted input: data and buffer lengths are treated as unsigned integers, but tested as signed integers—so crafted input with a large enough length to be regarded as negative when interpreted as signed can result in heap overflow.

SigRed Bastille Day 2020 brought news of "SigRed," a 17-year vulnerability in the Windows DNS. The bug enables remote adversaries to execute code on the victim machine, on "the DNS servers of practically every small and medium-sized organization around the world." [107] Researchers at Check Point [272] found that when the code handles certain kinds of records sent in response to a forwarded DNS query, it calculates the length of the record using a 16-bit unsigned integer. However, the adversary can use compression tricks to construct a valid record whose length exceeds 2^{16}. Once more, the flawed code copies too many bytes into too small a buffer, resulting in heap overflow.

[5] `https://www.youtube.com/watch?v=CrLJ29quZY8&feature=youtu.be`

Ripple Again Another one of the Ripple vulnerabilities demonstrates this pattern ("#2" in [151]). The flawed code uses a 16-bit unsigned integer to track a "label length" value, but correct inputs can be constructed whose label length overflows that. As a result, the code will allocate a too-small buffer, and then copy a much larger byte sequence into it. This can also be viewed as a parsing problem: the flawed code uses two different ways to calculate a length, and there are correct inputs for which these two approaches do not agree.

How LangSec can fix these problems In these examples, the problem was not a failure to validate, but a failure to validate with the *correct* specifications. To fix problems like this, the grammar must be rewritten to match what is handled correctly, or the computation must be reworked to handle all of the expected input. Not only must we need to check if the input is within the specified grammar, but also we need to be careful the grammar we are checking against matches a program's "safe input." Therefore we need to develop tools that can discover grammars for the application, or at minimum be able to find potential errors with grammars that are considered "correct." Alternatively, if the legal inputs in $L_2 \setminus L_3$ have no legitimate use, the developers may wish to restrict the official input space and tighten the input filter accordingly.

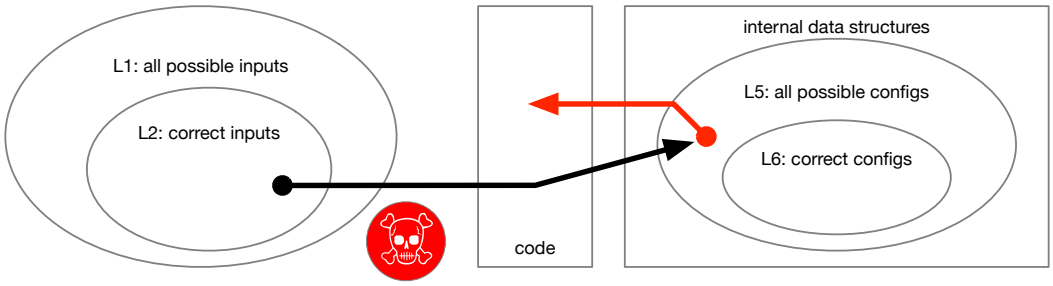


Figure 3.5: In some vulnerabilities, crafted but correct input can trick the flawed code into constructing incorrectly formatted internal state; when the code later acts on that state, it goes wrong.

3.2.2. LangSec on the Inside

Other Ripple vulnerabilities [150] pertain not to code parsing input, but rather to code parsing its own internal data structure (Figure 3.5).

When processing tunneled and fragmented IP packets, the code constructs an internal data structure with two different ways of representing the same length: an overall length in the initial part, and the sum of the lengths of each part. In "correct" configurations of this data structure, these match; however, correct inputs exist which can cause the flawed code to construct internal state in which these values don't match, leading to adversarial-directed memory corruption.

How LangSec can fix these problems Just as it's important for software to ensure that its external inputs are correctly formatted before acting on them, it's also important to check its internal inputs and internal outputs. Indeed, the basic programming tenet of defining and preserving the *invariant* for a data structure can be seen in a LangSec context: indicate precisely what one means as correct.

3.2.3. Differentials: Can Parsers Disagree?

Parsing Differential

In LangSec, the role of formal languages is not just confined to deciding whether an input is safe to act on; it's also key in *parsing*: how a program understands what an input means.

In the software world, the multiple programs often consume the same input—and there is an underlying assumption that different programmers read a specification and understood it the same way. Disagreements lead to *parser differentials* (Figure 3.6), perhaps documented initially in 2010 when Len Sassamen et al. [142] found that

Public Key Certificate Authorities could grant servers to domains that were invalid and some APIs accepted these invalid certificates.

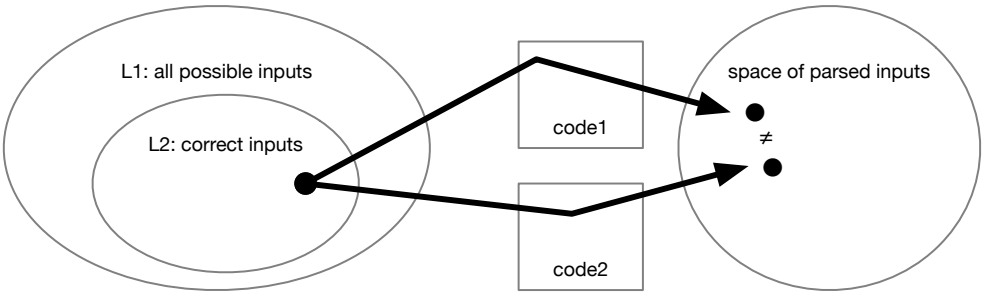


Figure 3.6: In *parser differentials*, two different programs can construct different parse trees—and take different actions—for the same input.

Psychic Paper May brought news of a parser differential vulnerability in iOS. [247]

In the Apple Ecosystem, *property lists* or *plists* are used to store serialized data in various contexts such as code signatures and configuration files. The list of properties the plist defines in code signatures are the *entitlements*. There are over 1800 entitlements available for the latest versions of the iOS (the mobile operating system), whereas the Mac OSX versions support just less than 1000 entitlements each. These entitlements also specify how the application interacts with the kernel. It can hold a list of drivers and system calls that the application can access.

The *Psychic Paper* vulnerability uses the fact that iOS uses "at least four" different XML parsers, and they have differentials. One example is how parsers handle comments in plists. Comments are usually enclosed in `<!--` and `-->` tags, and everything enclosed within these tags is considered to be a comment. However, three of the iOS XML parsers (IOKit, CF, and XPC) handle the not-quite-correct comment delimiters (a) `<!--->` and (b) `<!-->` differently

- IOKit sees the `<!--` in (a) and thinks it's the start of a comment. It ends the comment when it sees the end of (b) `<!-->`.

- The CF parser has a bug that only scans two places after seeing the `!` in (a). This means that a `-` is parsed twice and the first tag (a) opens a comment. The second tag (b) is treated as the opening of a new comment.

- XPC ignores both (a) and (b), but differs from IOKit and CF in other constructs such as the usage of double-quotes `''` and square brackets `[]`.

By being able to construct plists containing material that some parsers ignore as comments and others believe as real, the adversary can thus forge his or her own entitlements, and can get access to any memory location the user can access and execute system calls and drivers. The adversary can also alter thread register states and read and write process memory.

As mentioned earlier, the problem stems from the fact that the three parsers handle comments differently. One way to deal with this could be to specify the context-free grammar needed to recognize *plists*, and then build a parser to recognize this grammar. Instead of a specification, if developers start with the formal grammar needed to recognize *plists*, parser differentials can be minimized.

How LangSec can fix these problems With parser differentials, the *correct* specifications are not the same across various implementations. Using parser generators that use a data-description language or grammar could ensure that the parsers implement the same grammar across different implementations. Another approach would be to use parser combinators. They implement code that look like the grammar and can be easily verified. When different implementations use parser combinators, they can be visually verified to be equivalent, and equivalent to the grammar specification.

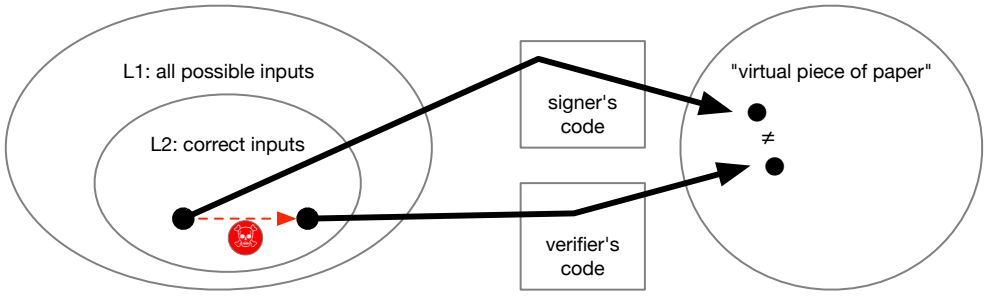


Figure 3.7: In the Shadow attack, the signer sees a "virtual piece of paper" and signs it; the verifier later sees a valid signature but a different virtual piece of paper. In between, the adversary modifies the PDF.

Semantic Differential

Shadow July also brought news of the *Shadow* attack [177], which might also be seen as a form of differential, albeit a semantic one.

A PDF viewer:

(a) can render a PDF file as a "virtual piece of paper" to the user;

(b) can allow a user to digitally sign the document they see.

(c) can allow a user to append incremental update to the document.

(d) can tell the user that a digital signature is valid.

The problem is that behavior 3 could permit certain harmful types of incrementally saved changes after behavior 2, and allow the attacker to configure the PDF so that the visible content of the document that the signer saw in behavior 2 differs from the content the verifier sees in behavior 4 (Figure 3.7). Attackers can construct a form, contract, bill, etc., and get it approved via a digital signature. However then it is possible for the attacker to modify the document to display different content from the one approved.

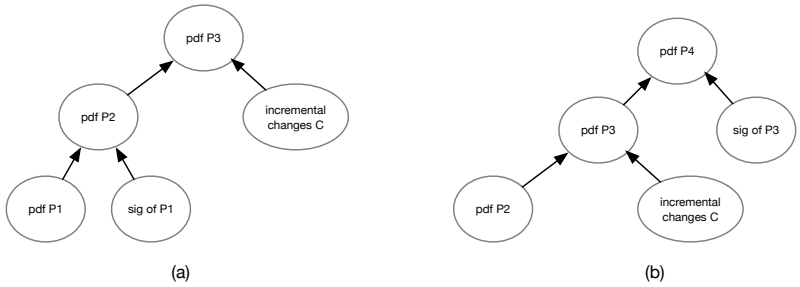


Figure 3.8: In this simplified sketch, the verifier may perceive that the signer signed P_3: the PDF P_1 modified by changes C. However, that only happened in (b); in (a), the signer only signed P_1.

The problem is that there's a gap between the core meaning of the PDF file—the virtual piece of paper the user sees—and the parsed PDF structure. As Figure 3.8 shows, the verifier cannot (easily) distinguish between two different ASTs with significantly different signature semantics the same way. (This was also demonstrated with earlier versions of PDF long ago [140]).

How LangSec can fix these problems A common solution that many PDF viewers currently use, is to remove any unused objects prior to signing, however, this only stops some flavors of the Shadow attack and not all of them. A stronger one (suggested in [177]), would be to reject any document which contains an non-signing incremental save after a signature. While this is effective, it is possibly too harsh, as there are use cases where changes made post signature are actually desired (for instance, filling out an approved form).

Work must be done to isolate exactly what is considered a "safe change"—and more deeply bind the action of producing/verifying a signature to the semantics of how the document is being presented to the user. Ideally, if two parse trees are semantically different, then the code consuming them should enable the user to distinguish the difference.

Section 3.3

Conclusions

This chapter detailed common input-handling vulnerability classes—namely, improper input validation and parser differentials. Then, I described related work on various LangSec fields such as parser combinators, data description languages, and parsing algorithms for data formats. Finally, I discussed two categorization approaches we organized to build taxonomies of various input-handling vulnerabilities seen in the wild. This chapter forms the basis for the rest of the thesis proposal—laying out all the fundamentals and the cutting-edge approaches in LangSec.

Chapter 4

Building Safer Parsers with Parser-Combinator Toolkits

Parser combinators are functions that allow developers to compose parsers using smaller building blocks. They were originally constructed for functional programmers to construct better and more composable recursive descent parsers. Subsequently, they have been adopted in other programming language communities, such as Scala, Rust, and C.

This chapter details our experiences building various tools using parser combinators. First, we built session-language-based parsers for Internet of Things (IoT) protocols. We define the state machines for various IoT protocols and define parsers specific to each state in the state machine (Section 4.1). We demonstrated that these parsers perform faster than the packet parsing in existing applications.

Second, we discuss CVD, a communication validity detector we built for various Power Grid or Supervisory Control and Data Acquisition (SCADA) protocols (Section 4.3). We deployed CVD as an intrusion detection tool and demonstrated how we could leverage parsers for various network protocols to detect intrusions. In addition to CVD, we also describe PhasorSec, where we built a parsing toolkit for the Phasor

Measurement Unit or PMU protocol and designed algorithms to place these parsers to get better performance from the network (Section 4.2).

Finally, we describe our experiences building a parser and a static analyzer for the iccMAX file format (Section 4.4). We built our parser using parser combinators and then analyzed the parser's syntax tree. The iccMAX file format supports a calculator element that implements various stack operations. Therefore, we check the stack for various underflows and overflows as a part of the static analyzer.

These early experiences of working with various parser combinator toolkits stressed the need for tools that perform better in several ways. We conclude this chapter by describing lessons learned and ideas we have subsequently incorporated in the later chapters of this thesis.

Section 4.1

Parsers for IoT protocols[1]

Current estimates show that the *Internet of Things (IoT)* will soon have billions of deployed devices. Approximately 50 million smart meters are in households in the USA at the moment. Several taxi companies, natural gas pipelines and industrial control systems make use of some of the popular IoT protocols. Some of these applications use personally identifiable information and any attack could lead to privacy concerns. Moreover, the widespread deployment of these protocols increases the probability of an attack as several recent attacks like the Mirai botnet [116] indicated. In the rush to deploy these IoT services, IoT vendors have limited or no focus on the required security mechanisms for their architectures.

Parser bugs have been haunting the Internet for the past several years, and the same trend is expected to continue in the IoT. Our current *Internet* works on a

[1]This section borrows from my previous work in [15].

"penetrate and patch" paradigm and, if the same paradigm is applied to the IoT, it would continue to lead to more parser-based vulnerabilities in the future [211]. Hence, we propose using *Language-theoretic security* to build hardened IoT end-devices.

Hammer [214] is a parser combinator toolkit that has been designed to aid developers in building code that can enforce input validation rules. This section discusses how *hammer* could be used along with protocol state machines to build hardened implementations of IoT clients.

Summary of Contributions:

- Our implementation demonstrates the efficacy of LangSec for Internet-of-Things protocols. Our technique could be deployed on IoT devices in the future.
- We show that the effort required to implement our technique is very reasonable given the security benefits offered to the IoT setting.

4.1.1. Background and Related work

The LangSec approach to security focuses on recognizing and handling all input safely. This functionality can be enforced by using a parser combinator toolkit like *Hammer*, which has bindings for several languages like C, C++, Python, Ruby and Java. To perform the task of building these hardened clients, we need to have a deep understanding of the target protocols.

Understanding application layer IoT protocols

XMPP. XMPP is a popular chat protocol that runs on the TCP/IP stack, and supports and enforces the use of TLS [235]. All clients connect to central servers.

XMPP messages are XML streams that need XML parsers. A stream is negotiated between a client and a server. The server supports some optional features, namely,

binding, TLS, and *SASL authentication*.

A trimmed version of XMPP could be a perfect fit for the IoT world [148]. Service infrastructures for the IoT based on XMPP have been proposed in the past [32]. XMPP maintainers have agreed to make use of TLS v2, disable the support for SSL v2, and make use of STARTTLS [236]. Despite the support for TLS, crafted messages could lead to exploits due to the lack of a clear parser and recognition boundary (shotgun parsers). Several secure architectures for XMPP have been proposed in the past [96, 64, 55], but none of them addresses the problem of shotgun parsers and untrusted input handling in XMPP implementations, which could have devastating repercussions in the IoT.

Some examples of XMPP messages are presented below.

```
<message type="chat" id="purpleb4dbd712"
to="arthur@sri.lit/host-134"
from="alice@sri.lit/XYZ567">
<active xmlns=
"http://jabber.org/protocol/chatstates"/>
<body>This is a sample XMPP message.
</body>
</message>

<success
xmlns="urn:ietf:params:xml:ns:xmpp-sasl"/>
```

The `<message>` is used to send a message from one client to another after the stream negotiation is complete. The `<success>` message is received by a client from a server if the SASL authentication was successful. The other relevant messages are `<stream>`, `<stream-features>`, `<proceed>`, `<auth>` and `<bind>`.

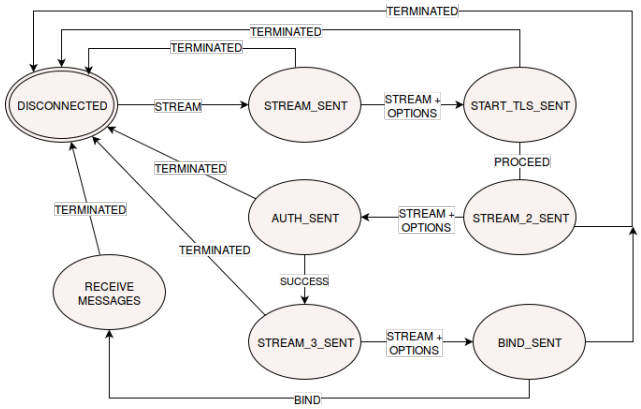


Figure 4.1: XMPP protocol state machine.

MQTT. MQTT is an asynchronous publish-subscribe architecture that works via a broker and was designed to consume less battery and bandwidth [124]. MQTT supports a limited number of messages - *Connect, Connect Ack, Subscribe Request, Subscribe Ack, Ping Request, Ping Response* and *Publish Message*. The message type is specified in the first 4 bits of the MQTT packet. The format of the MQTT packet is described in Figure 4.2. To initiate a connection with a broker, an MQTT client only needs to send one *Connect* message, in contrast to the large number of messages in XMPP. The client subscribes to channels using the *subscribe* message. The server sends messages *Connack, Suback, Pingresponse*. Figure 4.3 shows the state machine of the MQTT protocol.

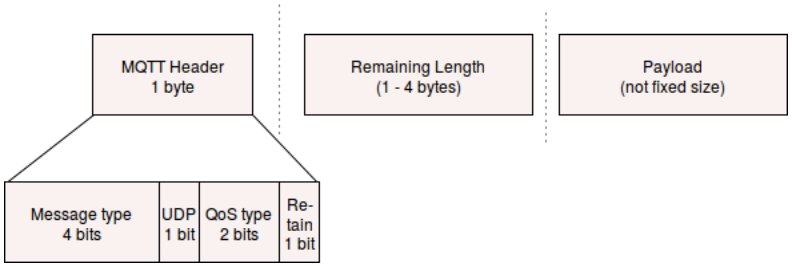


Figure 4.2: MQTT packet structure.

Hunkeler et al. proposed modifications to MQTT to be used in wireless sensor net-

works. MQTT has come under a lot of scrutiny for not being implemented correctly, with plenty of production systems not enforcing TLS or password-based authentication [174].

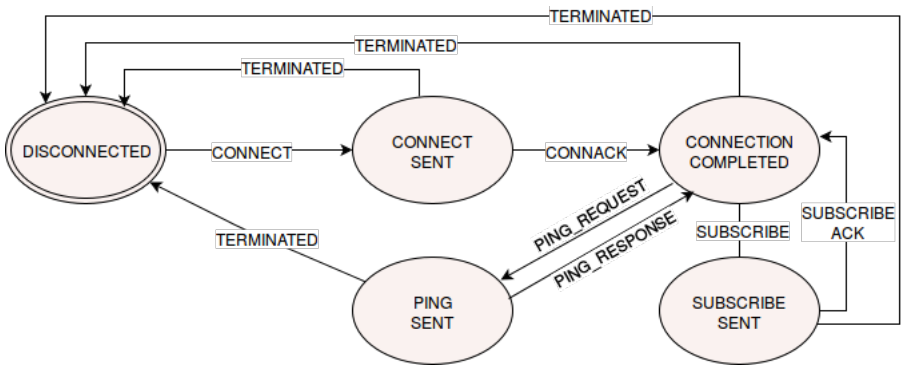


Figure 4.3: MQTT protocol state machine. The ovals represent states and the arrows represent MQTT messages.

We have seen past work on designing authorization mechanisms and making use of TLS to secure MQTT [203] [275]. Attribute-based encryption has been used in MQTT to encrypt MQTT packets for different channels [249]. Again, none of this work addresses the issue of handling untrusted input.

4.1.2. LangSec and Protocol State Machines

Shotgun parsers. Shotgun parsers perform data checking, handling, and processing, interspersed with each other. The shotgun parser pattern has led to several recent vulnerabilities as context-sensitive data formats can be dangerous and not easy to handle and recognize [45]. Instances of untrusted data propagation due to shotgun parsers were also found in Android applications in the past [273]. Shotgun parsers are only one of the possible *Langsec*-related weaknesses [194] [188].

Parsing errors encountered in XMPP and MQTT. Parsing errors could lead to memory corruption and logic errors, which could in turn lead to severe security

Client	Protocol	Vulnerabilities from Parsing Errors	Cryptographic Vulnerabilities
Prosody IM	XMPP	4	0
Cisco Jabber	XMPP	4	1
Pidgin	XMPP	3	0
Smack IM	XMPP	1	1
IBM Message Sight	MQTT	2	0
WebSphere MQ	MQTT	1	0

Table 4.1: Classification of vulnerabilities found in popular application-layer IoT protocols from 2013 – 2016 on Common Vulnerabilities and Exposures (CVE).

vulnerabilities. We studied the CVE database [188] for the "MQTT" and "XMPP" search strings and analyzed the type of vulnerabilities detected between 2013 – 2016. We found at least 14 vulnerabilities in XMPP implementations, out of which only two were cryptography-related (i.e., improper implementation of STARTTLS). Almost all other vulnerabilities included crafted XMPP input. The search string "MQTT" also yielded similar, but fewer results. MQTT had a higher percentage of crafted input vulnerabilities as well.

Table 4.1 summarizes a comparison of parser-related errors and cryptographic vulnerabilities in popular MQTT and XMPP implementations from the CVE database.

Protocol State Machines. In this work, we also make use of protocol state machines to define contextual parsers. In the past, there has been work in using protocol state machines for hardening protocols. Poll et al. [220] describe the difference in the state machines in various implementations of openssl. Graham et al. [106] state that finite state machines are sufficiently expressive for Internet protocols and are sufficiently performing for high-throughput applications.

Our work combines the concept of *protocol state machines* and the use of *parser combinators* to harden the implementations of IoT clients by making them less susceptible to input-processing vulnerabilities.

4.1.3. Methodology

In this section, we provide a comprehensive tutorial of how the state machine and the parsers hook together, and take a look at the limitations of this approach.

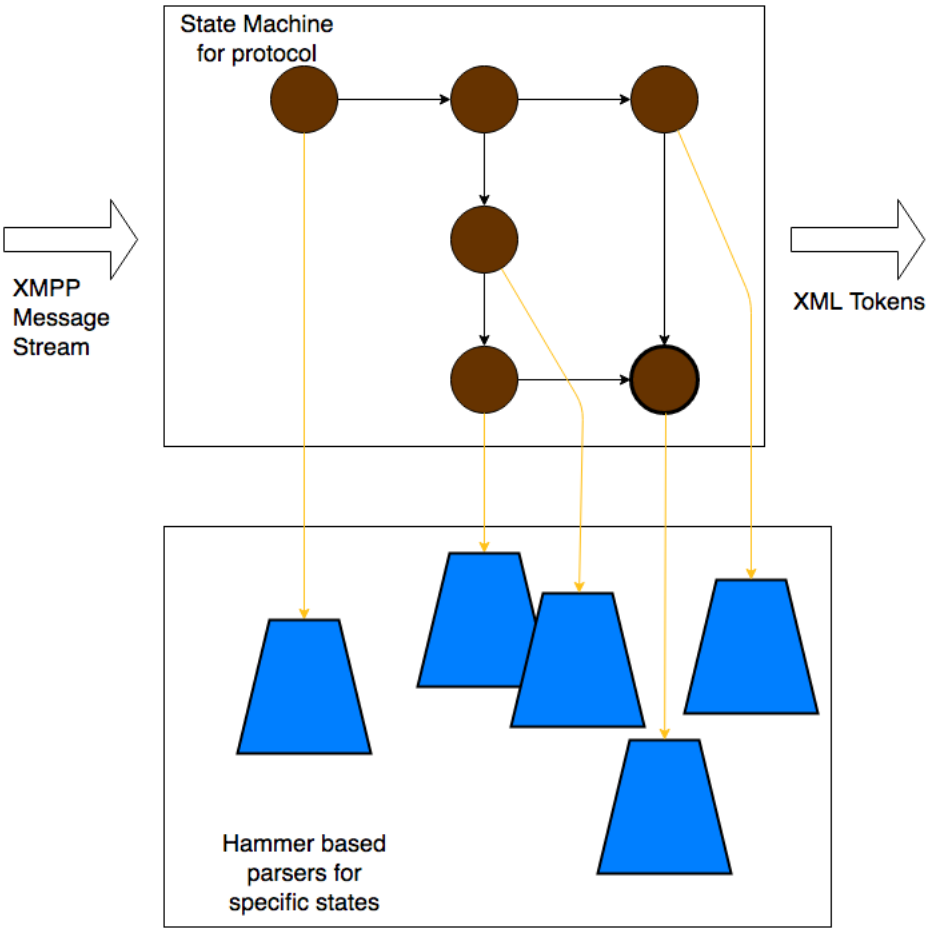


Figure 4.4: Overall architecture of a client: Depending on the current state, a different parser is called to recognize the message.

Our implementation is in **ruby**, and makes use of the *hammer ruby bindings* [214] and the *state_machines gem* [41], and hence assumes the correctness of both these implementations.[2]

[2]Gems are Ruby libraries that are packaged are shared using the RubyGems package manager: `https://rubygems.org/`.

Figure 4.4 gives an overview of the architecture of our IoT client. We implement the state machine defined in Figure 4.1. There are separate input recognizers for each of the states, which get called when the client receives a message at a particular state. Only once fully recognized, this input is further processed.

Our process of building a client involves four steps:

- Construct a simplified state machine from protocol specifications.
- Build the state machine using the *state_machine* gem.
- Identify the input language for each state.
- Define parsers for each of the receiving states of the state machine.

Constructing the protocol state machine. A protocol state machine is constructed from the specifications. To simplify the finite state machine, we need to enforce a few cases. For example, in our implementation of XMPP and MQTT, we enforce TLS and password based authentication because we think it is the right approach to follow as per the guidelines. The states and transitions that need to be handled in the case of not enforcing TLS and passwords can now be ignored. We make use of the *state-machine ruby gem* [41] to this effect. The gem includes a set of test helpers that could be used to aid in development as well. We implement the state machines in Figures 4.1 and 4.3.

The state machine gem could be used in this fashion with *before_transition* and *after_transition* methods used to trigger methods.

The above code snippet shows a part of the implementation of the state machine. The event methods show transitions between states. At each state that is receiving a message, we need to fully recognize the message received with the help of a parser.

Defining the receiving state parsers. We need to define a separate grammar for the messages we are expecting for each receiving state. For example, in the

```
state_machine :state, initial: :start do
  before_transition on: :stream_1_sending,
    do: :stream_1_sent_call
  before_transition on: :start_tls_sending,
    do: :start_tls_sent_call
  before_transition on: :ssl_negotiation,
    do: :ssl_negotiation_call
  before_transition on: :auth_sending,
    do: :auth_sent_call
  before_transition on: :bind_sending,
    do: :bind_sent_call
  after_transition on: :quitting,
    do: :quit_call
  event :stream_1_sending do
     transition [:start] =>
        :stream_1_sent
  end
  event :stream_1_received do
     transition [:stream_1_sent] =>
        :stream_1_received
  end
end
```

Code Snippet 4: State machine descriptions in Ruby

implementation of the XMPP protocol, if we sent the *stream* stanza to initiate an XMPP connection, we should be receiving a *stream* stanza back from the server along with some options. We need to use a parser to make sure this message is completely recognized. We need to define the language that is to be accepted at every state. Let us first look at the grammar for simple < *stream* > messages.

```
whitespace → \x20
doublequote → \x22
newline → \x5c \x6e
tab → \x5c \x74
gaps → whitespace | tab | newline
many_gaps → gaps | gaps gaps
zero_gaps → E | gaps | gaps gaps
starttag → zero_gaps \x3c zero_gaps
start_closetag → zero_gaps \x3c zero_gaps \x2f zero_gaps
endtag → zero_gaps \x3e zero_gaps
closetag → zero_gaps \x2f \x3e zero_gaps
stream_word → \x73 \x74 \x72 \x65 \x61 \x6d
stream_open_tag → starttag stream_word endtag
stream_close_tag → start_closetag stream_word endtag
```

Code Snippet 5: XMPP grammar in BNF syntax

The grammar defines whitespaces, double-quotes and all the other special characters as separate productions to help reusage of the combinators. The *stream_word* parser defines the word *stream* in the hexadecimal notation. The *starttag* production has the character < and the *endtag* has the character >. These productions can be reused while defining productions for the XML tags. All the tags allow zero or more spaces or special characters surrounding the tag symbols. This grammar was converted to hammer parser-combinator code in *ruby*, and is shown in the code above.

It is important to understand the combinators provided by hammer to represent grammar. The *many* combinator is used to apply the combinator given as an argument zero or more times. We use the *many1* combinator if it has to be used one or more times. The *sequence* combinator is used to group these combinators together in

```
@hammer = Hammer::Parser
whitespace = @hammer.ch('\x20')
doublequote = @hammer.ch('\x22')
newline = @hammer.token("\x5c\x6e")
tab = @hammer.token("\x5c\x74")
gaps = @hammer.choice(whitespace, tab, newline)
many_gaps = @hammer.many1(gaps)
zero_gaps = @hammer.many(gaps)
starttag = @hammer.sequence(zero_gaps,
        @hammer.ch('\x3c'), zero_gaps) # "<"
start_closetag = @hammer.sequence(zero_gaps,
    @hammer.ch('\x3c'), zero_gaps,
    @hammer.ch('\x3e'), zero_gaps # "</"
endtag = @hammer.sequence(zero_gaps,
    @hammer.ch('\x3e'),
    zero_gaps) # ">"
closetag = @hammer.sequence(zero_gaps,
    @hammer.ch('\x2f'), @hammer.ch('\x3e'),
    zero_gaps) # "/>"
stream_word = @hammer.token("\x73\x74\x72
        \x65\x61\x6d") # "stream"
stream_open_tag = @hammer.sequence(starttag,
                  stream_word,
                  endtag) # <stream>
stream_close_tag = @hammer.sequence(start_closetag,
                   stream_word,
                   endtag) # </stream>
```

Code Snippet 6: Hammer Ruby code matching stream objects in XMPP

a sequence. The *ch* combinator is used to represent a single character and the *token* combinator is used to represent a string. We make use of these basic building blocks to convert the grammar to the hammer parser combinator code.

Validating input. We will now look at how we use the *hammer parsers* we have already defined to recognize a message given as an argument. The hammer parser provides an object of class *HParseResult* if the parsing was successful, and a *nil* if the parsing was not successful. Upon successful recognition, we need to fire events to perform certain actions. In the example below, we are recognizing a message with the $< stream >$ message, and fire an event if the parsing was successful.

```
def validate_stream(args)
    if !stream_open_tag.parse(args).nil?
        true
    else
        false
    end
end

def stream_1_received(message)
    if validate_stream(message) and
          state == "stream_1_sent"
        fire_events(:starttls_sending)
    end
end
```

The above method would fire the *starttls_sending* transition, if the message was recognized fully and correctly.

File organization. As per the LangSec philosophy, all the input validators are placed in a separate class. These validators return a boolean if given a string to parse. These validators are called from all other files as and when input recognition is needed. Placing the validators in a separate class serves two purposes. Firstly, it

helps in re-usability of parser-combinator code. Secondly, it could aid developers and people performing code audits to easily identify mistakes in input recognition.

Limitations

- We can only define context-free grammars and regular expressions using parser-combinator toolkits. Hence, if the language to be recognized by our parser is either a context-sensitive grammar, or a Turing-complete language, we would have to simplify it down to a context-free grammar or a regular expression resulting in loss of functionality. However, we argue that this is essential since Turing-completeness or context-sensitivity of a language only results in more input processing errors.
- Since our implementations were lightweight, we see that our implementations were faster than a few other widely used open-source clients. In general, a layer to recognize input completely adds some overhead time. However, this cost is reasonable considering the benefits of this approach.

4.1.4. Evaluation and Discussion

To evaluate our implementation of the *MQTT* and *XMPP* protocols, we run timing experiments to determine how our implementation performed in comparison to other implementations. We also perform unit and preliminary fuzz testing, and describe our methodology.

Experiments

We compare the performance of our implementation and other open source implementations of XMPP and MQTT available. We analyze SleekXMPP [100] for python,

Smack [90] for Java, and Xrc [200] for ruby. We compare the total time to reach a connected state. These experiments were performed on a Raspberry Pi 2. For the server side of the XMPP protocol, we make use of *Vines - An XMPP chat server* [105]. In a similar setup, for the MQTT protocol, we use *ruby-mqtt*, *pyMQTT* and the *Mosquitto broker*.

Client	**CPU Time**
Our implementation	0.42 s
Xrc	0.59 s
Smack	0.30 s
QXMPP	0.41 s
SleekXMPP	0.90 s

Table 4.2: Comparison of average time to connect to the XMPP server.

Client	**CPU Time**
Our implementation	218 μs
ruby-mqtt	2.3 ms
pyMQTT	1.2 ms

Table 4.3: Comparison of average time to connect to the MQTT broker.

In Tables 4.2 and 4.3, we observe that our clients run in comparable time to most other XMPP and MQTT clients. Our clients are comparatively light weight and need fewer features than the traditional clients; hence, the similar CPU time values.

Message	CPU Time	Number of lines in hammer
Stream 1	17 ms	39
Proceed	18 ms	20
Stream 2	27 ms	39
Success	3 ms	20
Stream 3	24 ms	39
Bind	11 ms	25

Table 4.4: Comparison of average time to recognize various XMPP message inputs and the human effort in terms of lines of code.

Message	CPU Time	Number of lines in hammer
Connect Ack	39 μs	6
Subscribe Ack	126 μs	7
Ping received	97 μs	6

Table 4.5: Comparison of average time to recognize various MQTT message inputs and the human effort in terms of lines of code. MQTT messages for receiving acknowledgements from the server generally contain just 2 bytes and are easy to parse.

In Tables 4.4 and 4.5, we show that the amount of time used to recognize various inputs is minimal (i.e., in the order of tens of milliseconds).

We also observe that, with just under 250 lines of code, we can implement the input recognizers for a simplified version of XMPP, which indicates that the human effort required for such an implementation is not significant. Writing a separate layer of a parser and defining the protocol state machine explicitly are steps that are necessary to be taken and improve the process of code auditing.

4.1.5. Unit testing

We evaluate the correctness of the validation and state machine components using unit testing. We provide results of our unit testing in Table 4.6. We wrote unit tests for most of the parsers required for the two implementations, achieving a 100% code coverage for the state machine implementations and over 80% code coverage for the other parser implementations.

Protocol	Component	Coverage
XMPP	State machine	100%
XMPP	Parser validation	86.55%
MQTT	State machine	100%
MQTT	Parser validation	81%

Table 4.6: Unit test coverage of our XMPP and MQTT implementations via *coverage gem*.

4.1.6. Lessons Learned

We set out with the goal of demonstrating the efficacy of using LangSec in IoT protocols. We make use of the *hammer parser-combinator toolkit* to this end. It proved very easy for us to convert a context-free grammar to a parser-combinator implementation. The hammer library includes bindings for several programming languages and we made use of the ruby bindings for our implementations.

The specifications of protocols are in plain-text and are lengthy. Given the LangSec methodology, we have to read through these specifications and design a language compliant with the protocol specifications. Specifications could be more exact if they specified the protocol state machine and the input language for each receiving state. This would prevent parser differentials.

The hammer parser-combinator toolkit fared reasonably well in a constrained environment, showing that the overhead due to the recognizer was very negligible. The programmer does not have to write a lot of additional lines of code. The generation of the state machine could be automated with a tool to further reduce programmer effort. We simplified the language of the protocols to make it more strict, by enforcing rules that were otherwise optional in the protocol. This would reduce functionality of the protocol, but, at the same time, improve the security of the protocol.

4.1.7. Conclusion

In this work, we addressed the widespread problem of unprincipled input handling in IoT clients. We explore some of the popular application-layer IoT protocols and then discuss some of the recently known vulnerabilities in IoT protocols. We then describe the methodology used to build LangSec-compliant IoT clients. We analyze our implementation and show that the performance cost and the human effort in terms of lines of code added due to the hammer-based parser are a very reasonable price to pay for security. We generated large datasets of valid sequential XMPP and MQTT messages to see for what input our clients broke, and found no input that our implementation could not handle.

Section 4.2

PhasorSec: Protocol Security Filters for Wide Area Measurement Systems[3]

4.2.1. Introduction

Phasor measurement units (PMUs) are being widely deployed by power grid transmission owners and generator owners with the primary goal of improving situational awareness about the grid. These measurement devices are part of a larger critical infrastructure: wide-area measurement systems (WAMS). WAMS include PMUs, phasor data concentrators (PDCs), central data concentrators (CDCs), GPS receivers and communication and the networking infrastructure to facilitate better planning and operation of power systems. The measurements collected by the PMUs are sent upstream to PDCs over a wide area network where they are processed and used by applications. Every application has a desired timing bound on the freshness of data it requires. While some applications, like transient stability, require access to near real-time data, others like, postmortem analysis work with data that has been archived by the PDCs. Due to the real-time abilities of PMUs, there are plans to integrate the infrastructure with the current control system of the grid, thus making the security of the WAMS a high priority.

One of the primary tricks used by attackers to compromise devices is to find vulnerabilities in code that handles input. Regardless of how the code receives input, some processing on the input has to be done to make sure that the input is exactly as intended by the programmer. The lack of proper input recognition has commonly led to critical vulnerabilities like Heartbleed [82], where there was a particular length

[3]This section borrows from my previous work in [16]. I collaborated with Kartik Palani, Galen Brown, Rafael Brantley, Sergey Bratus, and Sean Smith on this work.

field that was not validated, enabling an attacker to read an arbitrary number of bytes from the buffer.

In the past, our investigation of current supervisory control and data acquisition (SCADA) protocols like DNP3 [42] revealed vulnerabilities in several implementations of the protocol by various vendors. Recently, vulnerabilities were found in implementations of the IEEE C37.118 [128], the most commonly used WAMS communication protocol. Operating on a malformed input can cause the device (PMU for example) to enter an unforeseen state which affects either the availability (device crash) or worse: increased privileges on the device for the attacker.

This section presents the initial design and implementation of PhasorSec, an input validation filter for WAMS. PhasorSec is an ingress-based network appliance that inspects packets in the WAMS network for potentially malformed inputs. PhasorSec is designed to filter out inputs that might compromise any WAMS devices by making use of Language-theoretic Security (LangSec) principles. Language-theoretic Security is a field of security that focuses on validating and handling input safely, using the principles of formal language theory [250]. Essentially, the set of acceptable inputs is treated as a language with a known grammar. Only inputs that conform to the grammar are operated upon, and everything else is either dropped or logged for analysis. Note that in this work the terms parser and filter are used interchangeably since the PhasorSec filter is essentially a hardened parser with added functionality to drop or log packets.

Several challenges exist in developing such a filter. We describe them and note the contributions of this work below.

- The IEEE specification of the C37.118 protocol [125] consists of verbose text, thus making it extremely hard and error-prone to implement a parser for the protocol. In this work, we provide an open source implementation of the parser

in C,[4] which is tested extensively against the state of the art AFL fuzzer.

- The protocol is also designed in a way that the PDC first receives a synchronization frame, and then receives a number of data frames that depend on the values in the synchronization frame. This dependency forces a validator to look at the state of the protocol stream, rather than look at the packets individually. Thus, such a filter would need to maintain state over multiple flows. There is a possibility of state space explosion and hence a concern over how much memory is available for the filtering process to maintain context. We mitigate this by building a finite state machine that only maintains state over what it expects to see given a certain input, thus conserving memory.

- Most ICS devices are comprised of proprietary software that cannot be modified without vendor involvement. This means long development cycles are needed to integrate hardened parsers into the device. PhasorSec is a bump in the wire solution that works independent of the vendor, thereby allowing flexible and timely deployment.

- Most power grid protocols prescribe a maximum permitted latency in the communication. Meeting these latency requirements and at the same time validating the input when the synchrophasor data that is collected at 60 to 240 measurements per second at the PDC becomes a challenging task. We build PhasorSec with stringent constraints and show that the overhead is a few microseconds. We also provide a mechanism to decide which network links are best suited to deploy PhasorSec such that the delay constraints are met.

It is important to note that PhasorSec filters don't match against an attack signature nor compare against a pattern of acceptable behavior, like in traditional intrusion

[4]The source code of our system is available at `https://github.com/Dartmouth-Trustlab/C37.118PMU`.

detection systems, but match each packet and its contents on a grammar, and on the context of the packet based on the protocol state. PhasorSec filters keep track of the actual values in the configuration frames, and use them to make sure that the subsequent values are well-formed as well. Also, PhasorSec does not prevent against false data injections in protocols by default. While it protects cyber components against compromise, it does not consider how attackers can affect the power system given that PMUs have been compromised. However, it is possible to extend PhasorSec functionality to do so by integrating it with [276] for example. This would come at a higher overhead, which we do not discuss in this work.

4.2.2. Background and Related Work

Language-Theoretic Security

LangSec posits that the design of any software must begin with gathering the protocol state machine and the grammar of the protocol. In the past, we have looked at a similar approach for protocols in the Internet of Things for the MQTT and XMPP [15] (Section 4.1) and the SCADA/ICS DNP3 protocol [42].

The DNP3 implementation was built as a proxy, that would consume DNP3 packets asynchronously, and would run the parser on it offline. This design decision was made since the DNP3 parser induced latency. To avail the security benefits of having a parser protect a system from input handling vulnerabilities, the parsing has to be performed in real-time. Also, the DNP3 implementation looked at parsing separate packets, without regard for the protocol state. SCADA protocols are traditionally designed such that they rely on a previous packet for context. The MQTT and XMPP implementations perform the parsing synchronously, but add significant overheads due to parsing.

PhasorSec overcomes these shortcomings in the DNP3 and IoT implementations

of LangSec. We build a placement tool and outsource the parsing to a separate device on the substation network to serve as a filter for the entire network, while at the same time maintaining the state for each of the connections on the network.

IEEE C37.118 protocol

The C37.118 protocol is used to gather synchronized phasor measurements to make better-informed decisions about the operation of the smart grid. Figure 4.5 shows the various types of C37.118 protocol messages. The PDC sends commands to the PMU, and the PMU in turn sends a configuration frame specifying the format of the data frames that would follow after that. Header frames can contain any sort of human-readable information (i.e., plain-text).

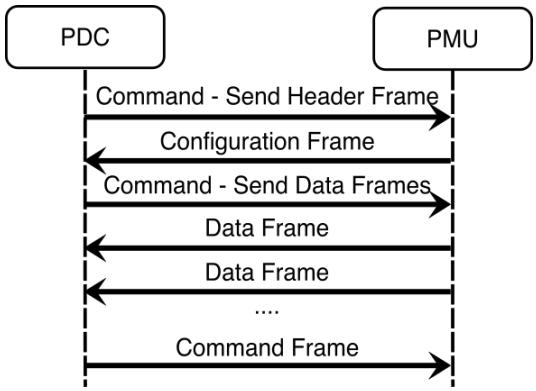


Figure 4.5: Flow of messages for the C37.118 protocol.

In the initial state of the PMU, only Command frames are valid. Receiving a Command frame requesting a Header or Configuration packet causes the machine to temporarily accept a single packet of the appropriate type. Due to the complexities of simultaneous communication, this packet need not be the next one received. Instead, the machine will continue accepting valid packets of any type until it receives the one requested, then stop accepting those packets until a second Command packet requests them.

At any point, receiving a Command frame requesting data causes Data frames to be accepted as valid, until a Command frame ending data is received, at which point all Data frames are rejected.

Security of Wide Area Measurement Systems

Intrusion detection in wide area measurement systems has been widely studied. Yang et al. propose an intrusion detection mechanism specific to the C37.118 protocol [293]. Their technique involved making use of a heterogeneous whitelist of known attacks, and a behavior-based approach to detect unknown ones. Stewart et al. provide a very comprehensive set of guidelines to describe what approach is to be taken by operators to ensure confidentiality, data integrity and availability of the grid [264]. Stewart et al. also note that the specifications of the C37.118 protocol do not include any security features, and that it is left to the network layers to enforce the security. The paper also provides several results demonstrating the effectiveness of the safeguards. Although the paper goes in depth about substation security and information security, Stewart et al. do not discuss the issue of input handling in the PMUs.

Coppolino et al. conducted a study of synchrophasor devices (PMU) and phasor data concentrators (PDC) [65]. Coppolino et al. note that telnet was being used to perform a lot of management tasks, and this is susceptible to man-in-the-middle attacks. Also, there was no input validation or sanitization in the PDC application that they examined, which was the open source OpenPDC application. The content of the messages were usually not verified, and the authors note that a host of input validation attacks and bugs are possible including SQL injection and buffer overflows. The authors recognize that the issue of input validation exists in the PDCs and PMUs, but do not talk about ways this could be addressed.

Another important source of motivation for our work is looking at past list of

Common Vulnerabilities and Exposures (CVE) and understanding why these errors occurred. In the database we find three CVEs that are specific to the C37.118-2 communication system and PMUs [192, 126, 127]. CVE-2013-2800 and CVE-2013-2801 signify exactly the same problem we are trying to address in this work. These bugs would expose the devices to a host of memory corruption and denial-of-service attacks. The most valid action would be to reject these messages since they are malformed, and don't satisfy the grammar of the protocol. Prior to the security investigation of DNP3 implementations by Chris Sistrunk and Adam Crain [42], the DNP3 CVE list had just one reported vulnerability. The researchers found over 30 vulnerabilities through their investigations in 2013-2014. We anticipate that a similar investigation of the C37.118 protocol would reveal several more of these vulnerabilities.

4.2.3. Approach

We are using a principled approach to build a high-assurance input validator, and provide it in a way that operators of constrained edge devices do not have to bother about the performance and CPU time. We want to do this by placing these parsers optimally on a level higher on the hierarchical structure followed by wide area measurement systems, and sometimes in the edge device itself when the PMU is known to be sending and receiving critical data.

Our development effort of PhasorSec was divided into three broad parts. First, we build the LangSec parsers for the IEEE C37.118 protocol for the revision of 2011. Second, we introduce PhasorSec as a bump-in-the-wire on a substation network. Finally, we find the most optimal location to place PhasorSec based on the importance of the PMUs and the network topology. Figure 4.6 demonstrates the purpose of PhasorSec in a substation network.

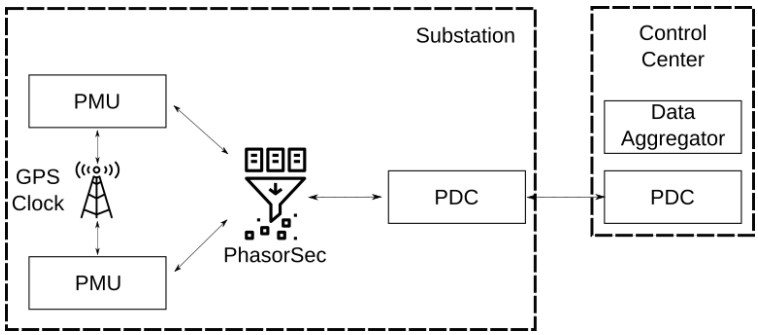


Figure 4.6: Overall architecture of PhasorSec. A single substation contains multiple PMUs, which are synchronized using a GPS clock. These PMUs communicate with a single PDC at the substation, which then aggregates data to a PDC in the control center. This diagram shows the placement of PhasorSec in a substation network in the control center.

Designing PhasorSec

The LangSec methodology involves a critical reading of the protocol specification to be able to extract information that is needed to build parsers. We start by understanding what states follow in the protocol, and the messages that are to be accepted by each of these states. The architecture of our parsing methodology can be seen in Figure 4.7. A clear and direct correlation between the packet format diagram, the grammar and the code validating this grammar is necessary, and is the goal of LangSec, since it helps programmers and auditors alike.

We make use of the Hammer parser combinator toolkit to both describe our extracted grammar for each state of the C37.118 protocol, and at the same time implement a parser for the protocol. Our coding style makes the C/C++ code more clean and concise, and more readable that the pointer arithmetic-based parsing code usually written.

Extracting the session language: To understand how the parser must be built, we need to understand the exact order in which the messages are sent and received. In some cases, messages that aren't supposed to appear one after the other or in a

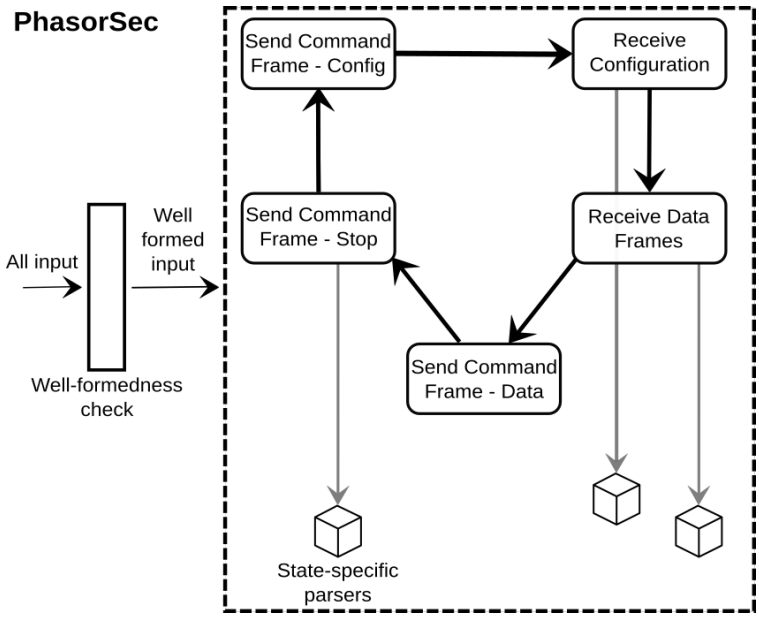


Figure 4.7: Parsing methodology in PhasorSec. For protocols involving complex session languages and states, we would first perform a well-formedness check to make sure the message overall conforms to the specification of the protocol, and only then parse the message with respect to the current state of the system.

certain order may appear so, and the session language must be able to reject such messages. Figure 4.5 shows the communication between the PMU and the PDC from which we can extract the individual state machines for both the PMU and the PDC. Figure 4.6 also shows the session language or the protocol state machine for an individual PDC.

Extracting grammar from specification: Usually all protocol specifications have clear descriptions of what are the various packet formats. The specifications also include some critical information such as what the boundaries of the various parts of the payload are, and which fields depend on other fields. Context-sensitivity must usually be avoided, and we advocate that we must stick to the easily recognizable regular and context-free languages. A LangSec analysis of the specification would reveal such issues as pitfalls in the specification of protocols [194]. Below is the grammar we were able to extract for the C37.118 protocol. A well-formed PMU frame can comprise any of the four frame types. One thing to note in the config_frame

description is that the Station_config field repeats NUM_PMU times. This leads to a complexity in parsing, since this value NUM_PMU has to first be extracted, following which the rest of the packet has to be extracted. The same occurs for the fields PHUNIT, ANUNIT and DGUNIT which all depend on the previous values PHNMR, ANNMR and DGNMR for their lengths.

$$
\begin{aligned}
\text{config\_frame} &\rightarrow \text{Header NUM\_PMU Station\_config}^{*} \\
\text{Header} &\rightarrow \text{sync framesize idcode soc} \\
&\quad\ \text{fracsec time\_base} \\
\text{Station\_config} &\rightarrow \text{name id\_code FORMAT} \\
&\quad\ \text{PHNMR ANNMR DGNMR} \\
&\quad\ \text{PHUNIT ANUNIT DGUNIT options} \\
\text{options} &\rightarrow \epsilon \mid \text{options DATA\_RATE} \mid \\
&\quad\ \text{options CHKSUM}
\end{aligned}
$$

Well-formedness check: We build recognizers for the chosen protocol, and recognize the overall syntax of the message, without actually taking into consideration the session state the receiver is in. This check is really important, since there are some semantic actions involved in checking whether a receiver is in a certain state, and making sure the correct parser is being run on it. When a device is on the receiving end of a stream of completely malformed and invalid messages, these messages get rejected directly at this step without being subject to any semantic actions hence saving CPU time and memory. The placement of the well-formedness check in the code can be seen in Figure 4.7.

Parsing based on the session state: Once the message is checked for overall

```
header = h_sequence(h_uint16(),
                    h_uint16(),
                    h_uint16(),
                    h_uint32(),
                    h_uint32(),
                    h_uint32(),
                    NULL);
station_config = h_sequence(name,
                            id,
                            format,
                            phnmr,
                            annmr,
                            dgnmr,
                            phunit,
                            anunit,
                            dgunit,
                            options,
                            NULL);
options = h_sequence(h_optional(h_uint16()),
                    h_optional(h_uint16()),
                    NULL);
initial_parse = h_sequence(header,
                        h_uint16(),
                        NULL);
next_parse = h_repeat_n(station_config,
                        num_pmu);
```

Code Snippet 7: The Hammer-based code for handling the C37.118 configuration frame as described in the grammar above.

syntactic validity in the well-formedness check, we check which state our system is in, and deploy that particular parser to be run on the given input to make sure that the message we received is not just structurally correct, but also valid with respect to the current state our system is in. In the C37.118 protocol, the data frame parsers need to be built based on the configuration frame that was received previously. Figure 4.7 describes how each state of the system has a parser of its own, that gets called if a message was received in that particular state. Code Snippet 7 shows a snippet of how the grammar in the previous section translates directly to a parser in our code snippet written in C.

PhasorSec as a bump-in-the-wire. One of the bigger goals of PhasorSec is to make sure that all the C37.118 packets in our scope goes through our parsers, and no device receives a packet that has not been parsed by one of our parsers. To introduce PhasorSec as a bump in the wire, we perform the following:

- The PhasorSec device performs an *arpspoof* on the PMUs telling them that it is the PDC, and the vice versa to the PDC.
- We use *scapy* to recover the packets the PDC is sending or receiving.[5]
- The state of each of the connection is maintained in the form of a finite state machine, and the correct parser is called.

In case of failure to parse a message, PhasorSec logs the message after dropping the packet. If the parsing is successful, PhasorSec forwards it on to the recipient.

4.2.4. Deployment

Applications that depend on synchrophasor data often have real time requirements in the order of seconds. Meeting these demands can be hard and hence security consid-

[5]Scapy helps us manipulate packets on the wire: `https://scapy.net/`

erations are often ignored in the pursuit of performance requirements. In deploying PhasorSec, we hope to guarantee end to end delay requirements while providing input validation. Another constraint is the strict security budget that utilities have which makes it expensive to deploy and manage a filter per network link. In Section 4.2.5 we show that PhasorSec adds a network overhead in the order of a few microseconds and hence even if we were to deploy a filter on every network link, the end to end delay requirements for even the most time critical of applications will not be exceeded. Hence, we define the problem of deploying PhasorSec as maximizing the number of links that are monitored by a filter while guaranteeing that the budget is met [143].

We represent the WAMS network as an undirected graph $G = (W \cup N, L)$, where W is the set of WAMS devices and N is the set of network infrastructure nodes. L is the set of links at which PhasorSec can be deployed.

Objective function: Not all network links are created equal i.e. some links carry data from PMUs that provide more valuable measurements than other PMUs. Thus, we can assign an importance to each network link. Let $w(l_i)$ be the importance of link i. The objective of PhasorSec deployment is then to maximize coverage of the most important links.

Deployment Cost: The cost of deploying the network appliance is affected not just by the cost of the appliance itself but also the cost of installation which can vary significantly due to geographical location or accessibility of the substation. Let $c(S_j)$ be the cost of placing a filter on a set of links $S_j \subseteq L$.

Hence, the formal definition of deployment is given by the following equation where $x_i = 1$ if link l_i is selected and $y_j = 1$ if a set S_j of links is chosen.

$$
\begin{aligned}
&\text{maximize} && \sum w(l_i)x_i \\
&\text{subject to} && \sum C(S_j)y_j \leq B; \\
&x_i \in \{0,1\}; \\
&y_j \in \{0,1\};
\end{aligned}
$$

Note that this is reformulation of the budgeted maximum coverage problem which is NP-hard [147]. There exists a $1 - \frac{1}{e}$ approximation algorithm to solve the problem [147]. It turns out that the greedy algorithm achieves the best approximation ratio.

4.2.5. Evaluation

We evaluate our parsers using three broad validation techniques. We performed a static analysis on our system, we fuzz-tested our system, wrote unit tests, and performed CPU-time analysis.

Unit Testing, Static Analysis and Coverage

We make use of the *Infer* tool to perform a static analysis of our system [52]. Our implementation was found to not have any of the categories of errors found by *infer*, namely, null de-references, memory leaks, premature nil termination arguments and resource leaks.

We used a set of unit tests to test our implementation of the C37.118 protocol. To validate our unit testing technique, we used *gcov* to assess the code coverage of our implementation. The results of our coverage are in Table 4.7.

Fuzzing

We make use of coverage-guided fuzz-testing using the *American Fuzzy Lop* fuzzer (AFL). Figure 4.8 shows that we ran AFL on the configuration frame parser.

Type	Lines	Percentage
Line Coverage	364/485	75.1%
Function Coverage	31/34	91.2%

Table 4.7: Code coverage of the C37.118 parser using *gcov* and *lcov*. This shows the number of lines and functions that could be reached by our unit testing suite.

Our results show that after 25 hours of fuzzing, there were no crashes. Although there were several hangs, these were mostly due to the fact that our parser was stateful.

Parser	**Total Run-time**	**Crashes**	**Hangs**	**Cycles**
Configuration Frame	25 hours	0	37	9452
Data Frame	25 hours	0	42	10300

Table 4.8: A summary of our AFL fuzz-testing results

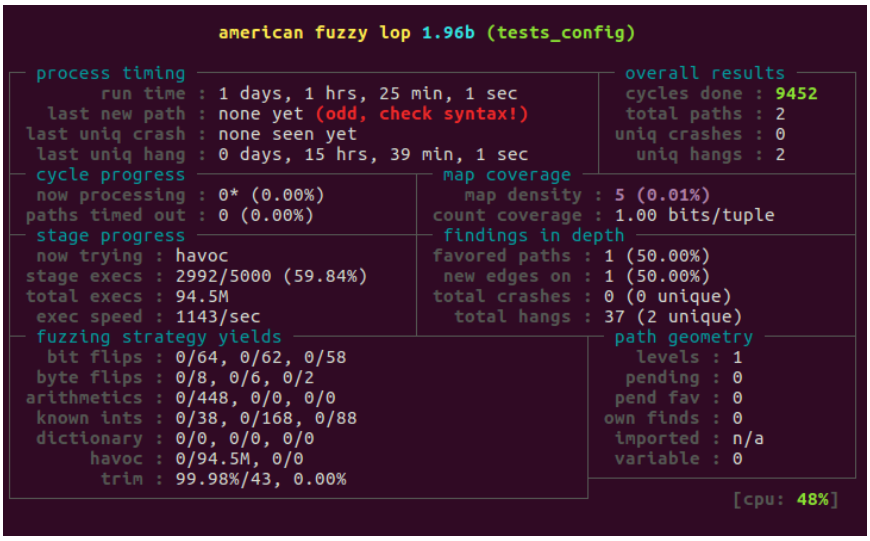


Figure 4.8: AFL Fuzzer screenshot showing results of fuzz-testing of our configuration parser written in hammer

Timing Analysis

The experiments were run on an Firefly Development Board with a Quad-core ARM Cortex-A17 processor and 2GB of RAM.

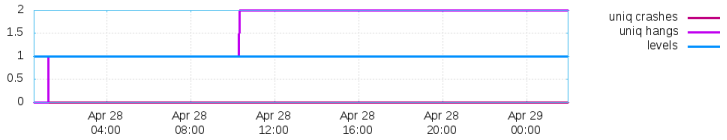


Figure 4.9: AFL did not detect any crashes and had only two unique hangs after over 24 hours of testing for both the configuration frame and command frame parsers.

Frame	**CPU Time**	**Lines of code**
Command Frame	20 μs	29
Configuration Frame	13 μs	71
Data Frame	27 μs	56
Header Frame	80 μs	30

Table 4.9: We compare the time taken to parse each of the frames and the number of lines of code in each of these parsers.

We perform CPU-time analysis on the individual parsers, to understand the overhead which would be introduced. The command frame and the header frame would have to be parsed at the PMU, and the configuration and data frames would be parsed at the PDC which aggregates the data from multiple PMUs. In Table 4.9, we see that the overhead due to the addition of these parsers is very minimal (in the order of micro-seconds) considering that it prevents input-handling vulnerabilities and bugs. We also note that the complex parser we have written was for the configuration frame, which contains the context needed for the data frames to parse the data correctly. Despite performing stateful parsing, we note that we can construct these parsers in under 75 lines of code each.

4.2.6. Conclusions

We showed that a context-aware parser can be built for the C37.118 parser, that is not only resilient to state-of-the-art fuzzing techniques such as AFL, but also does not add much overhead to the devices.

We started with a critical reading of the specification of the IEEE C37.118 proto-

col, understanding what devices are included in the protocol, and at the same time understanding the syntax and semantics of the messages. We implemented individual parsers for the different messages, that first look at the overall syntax of the message, and then look at the context of the device based on the previous messages.

Although our current design is inspired from our discussions with domain experts, we anticipate that the substation networks are moving slowly towards software-defined networking. In our future work, instead of using an ingress-based network appliance, we would instead like to move to software switches based on P4 [170]. To improve usability of *hammer*, we will also explore using domain specific languages to improve the usability of the parser building methodology.

Section 4.3

Communications Validity Detector for SCADA Systems[6]

Supervisory Control and Data Acquisition (SCADA) protocols are being used to make the Power Grid *smart* and *automated*. In such a modernized grid, substations are increasingly unstaffed and controlled from control centers via SCADA protocols such as DNP3 and IEC 61850 MMS.

Such critical-infrastructure systems usually boast of an IT-OT air-gap—physical separation of the critical infrastructure (OT) and IT systems using non-routable interfaces and legacy hardware. However, this air-gap is disappearing, given the need for remote access to control these devices [46].

In recent times, this air-gap has not helped either. Attackers were able to make centrifuges malfunction in the case of Stuxnet by taking control of PLCs [92]. These PLCs were attacked using multiple zero-day attacks using compromised USB sticks.

[6]This section borrows from my previous work in [186, 12].

We had no prior knowledge of these vulnerabilities and held no signatures of them. We also witnessed the use of such zero days in the Ukraine power grid attack [54]. However, these zero-day attacks have one thing in common: they are mostly input-handling vulnerabilities such as stack- or heap-based buffer-overflows [302].

Securing our SCADA Systems has been a challenge for various reasons. First, since any communication received by a SCADA device can have a physical effect, availability and timeliness are of paramount importance. The devices are usually too constrained, and security schemes often fail to meet some of the real-time guarantees required by Smart Grid networks [303].

Second, Anomaly and Intrusion detection schemes proposed for SCADA systems are either targeted at detecting anomalies using the physics of these systems or using the communication patterns. Such schemes cannot detect crafted-input attacks such as those used in the Ukraine attack in 2016 or the Stuxnet attack in 2010.

Hence, any forensic tool used to investigate cyberattacks must detect attacks that use invalid communications. In SCADA systems, invalid communications can stem from various causes. First, communications can exploit weaknesses in programs due to **insufficient syntax checking**. Programs often fail to implement communication protocols correctly, leading to vulnerabilities. An attacker can exploit this vulnerability to crash the program or gain complete access to the device running the program.

Second, forensic tools must detect syntactically valid but **semantically invalid communications**. For each device in a SCADA network, the SCADA operator holds a specification document listing all the IP addresses and the endpoints of that device. For protocols such as DNP3 and MMS, the device only supports a specific set of requests known as setpoints. A SCADA operator holds specification documents showing what setpoints each device supports. Communication violating any of these network or setpoint configurations is semantically invalid.

Finally, forensic tools must also help detect **communications triggered by a malicious program** or device, not a human. Differentiating between human actions and a malicious program is a difficult problem. SCADA forensic tools must provide visual feedback or confirmation on any human-triggered actions or evil actions.

To address these challenges, we present *CVD*, a communications validity detector. Our contributions are as follows:

- Our CVD implementation detects malformed packets in a wide range of protocols. CVD detects any packet that does not conform to the protocol specification, detecting potential zero-day attacks.
- CVD detects various Web-based, Telnet-based, and DNP3 actions that attackers take. Although these protocols are known to be insecure, they are used extensively in SCADA networks.
- CVD can detect various configuration and communication mismatches within a Smart Grid substation.
- With CVD, we propose a new forensics paradigm to permanently place our CVD devices in substations and control centers to monitor traffic and detect attacks early.

4.3.1. System Design

To build a useful forensic gathering tool for SCADA networks, we set the following technical goals for CVD:

- **Live Forensics**: CVD devices must be connected to SCADA traffic to continuously monitor and collect forensic information. In case there are any suspicious actions in a network, operators can view CVD's insights.

- **Detecting syntactically invalid messages**: CVD must detect messages that violate protocol specifications.

- **Detecting semantically incorrect packets**: Based on SCADA operator configuration files, CVD must detect communication flows violations.

- **Help in detecting non-human-triggered actions**: Most critical SCADA actions with physical effects such as opening or closing breakers and relays are human-triggered. If a compromised device communicates these SCADA commands over the network, CVD must detect and visualize all SCADA actions with physical effects.

To ensure that CVD is a tool that is extensible and usable, we set the following design goals for CVD:

- **Adaptive.** We must not depend on existing attack scenarios and heuristics, but be able to detect crafted packet attacks. Since zero-day attacks exploit patterns and vulnerabilities that we have never seen earlier, we do not want to rely on previously seen patterns but instead, use the LangSec paradigm to detect new attacks.

- **Scalable.** We need to support a wide range of Smart Grid protocols with an easy API to support future protocols. CVD detects syntactically malformed packets for protocols we implement. CVD must hold a flexible architecture so that in case a SCADA network supports protocols we do not support, developers can add parsers for this protocol with minimal effort.

- **Distributed.** Given that there is a large amount of data generated in every substation, we want to perform as much analysis as possible within the same substation. In the control center, we only want to perform certain aggregated

operations. This goal is vital to not add too much network overhead to the SCADA network due to our CVD devices.

– **Usable.** It must be able to provide alerts in a usable and visual way while not overwhelming users with information.

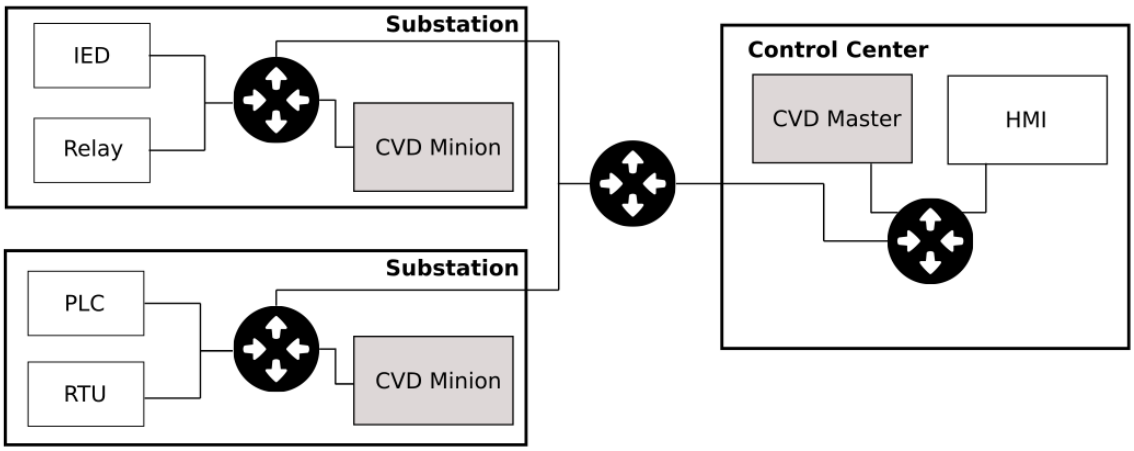


Figure 4.10: *CVD Minions* are connected to the routers within the substation, and collect traffic from two interfaces. Whereas, the *CVD Master* is present in the Control Center. The shaded boxes show our CVD devices.

Design

To realize our design goals, CVD uses various techniques. First, CVD uses a comprehensive set of Language-Theoretic Security-compliant parsers for different protocols commonly used in Smart Grid substations. Some of the protocols that CVD includes parsers for are DNP3, IEC 61850, and IEEE C37.118. Our parsers are **adaptive**, not relying on previous attack samples to detect crafted input attacks. We will discuss these parsers in more detail in Section 4.3.1.

Second, CVD uses a producer-consumer model within each implementation. This design using Apache Kafka makes CVD **scalable**. Any future protocol additions only

require adding consumers. The producers extract the payload portions of the packets and broadcast them to all the consumers simultaneously.

Third, CVD uses a **distributed** *master–minion* system, where the *CVD minions* are placed in every substation, whereas the *CVD master* is present in a control center. This design is shown in Figure 4.10. The CVD master performs only data aggregation and correlation, whereas the data collection and our parser checks happen in the minions.

Finally, CVD includes strong visual components in terms of various Web UIs and a command-line interface. The Web UIs present Smart Grid operators with multiple alerts and a visual representation of specific traffic, which gives operators an ability to monitor the network for traffic that may be well-formed but not sent by the operators. For example, attackers could then use a particular relay device compromised via a side-channel attack to send DNP3 commands to other breakers. Operators can easily detect such attacks via our visual component that displays various DNP3 protocol actions.

Continuous Data Collection and Monitoring Paradigm

CVD proposes a novel paradigm of continuous data collection and monitoring. Most forensic investigation tools are only deployed after a bulk of the attackers' actions are complete. We deploy CVD in SCADA networks to continuously monitor network traffic. In case of a suspicion of a cyberattack, operators can retrace the attackers' steps using the CVD database.

We require a live network tap interface that can be created by duplicating all the traffic going through a router. This duplication ensures that CVD does not add significant overheads to the network. CVD processes all the packets the router forwards asynchronously, generating alerts on all suspicious packets. We continuously push

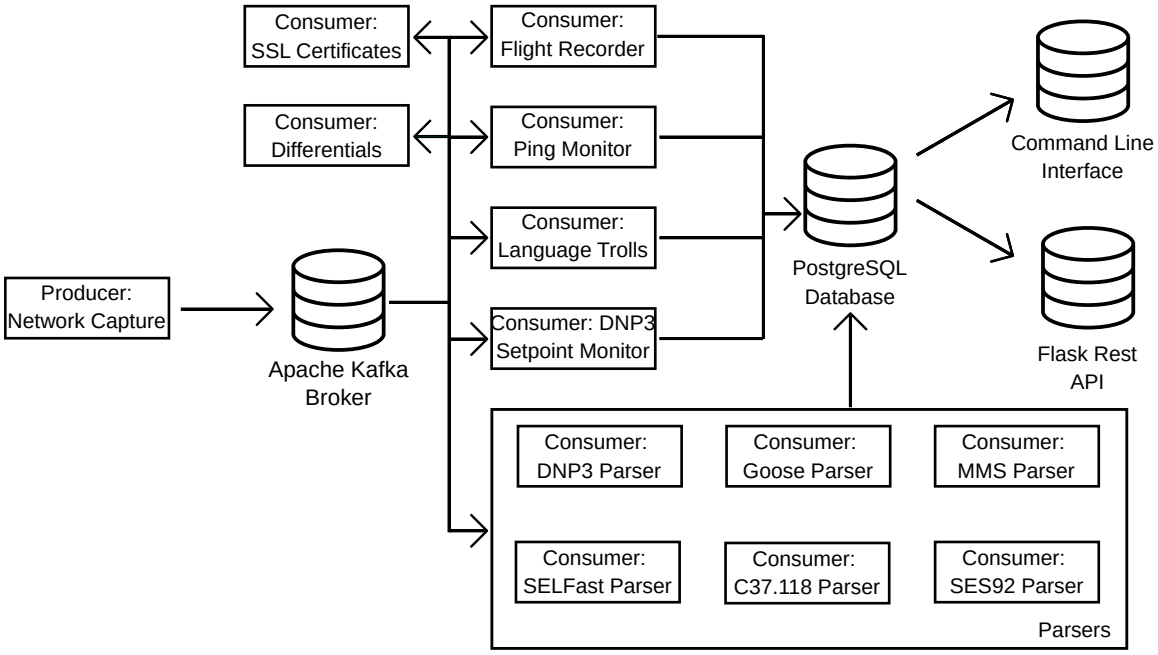


Figure 4.11: Overall Architecture of the CVD Minion. The producers process network packets and forward them to various consumers via the Apache Kafka Broker. The various consumers perform their analysis and store the results in a PostgreSQL database. Operators can consume information from the PostgreSQL database via interactive Web UIs or a command-line interface.

these alerts to our databases along with all network traffic metadata observed. This paradigm of leaving CVD connected to a SCADA network tap before any cyberattack to aid in forensics allows operators to reproduce the steps taken by attackers rapidly.

Distributed Data Collection

As seen in Figure 4.10, data is collected by CVD Minions present in every substation. These minions collect and store data within the substation, as well as across them. This data can include mostly various SCADA commands, but could also include other routing behavior such as ARP, NTP, and DNS.

Since the CVD Minion and Master are also a part of the substation networks, network traffic originates from these devices as well. However, this data does not contribute to any forensics gathered by our parsers since we whitelist all traffic orig-

inating from CVD.

CVD Minion Publish-Subscribe Model

One of the goals of CVD is to run the same packet received from the network through our various LangSec parsers in parallel to check if the packet conforms to any of those protocols. To do this, we adopt the publish-subscribe model.

We show this model in Figure 4.11. In this model, we have producers that capture network traffic on various interfaces and send them to the Apache Kafka Broker. The Kafka Broker then broadcasts it to all the consumers, that then decide what to do with these packets. We chose the publish-subscribe model since it gives us the *flexibility* to add future protocols and other analyzers easily without much effort.

Producers. We designed our Kafka producers to accept input in two formats. They can take packet capture files or attach to live network interfaces. In either case, our producers read each packet, and convert them to a `string` and then send them over to all the consumers. They do not do any pre-processing since each consumer analyzes the packet at a different layer of the protocol stack.

Consumers. Our Kafka consumers receive all the packets from the producers and process them in various ways. In this section, we describe the various consumers present in CVD. As seen in Figure 4.11, all our consumers store the results of their analysis in the PostgreSQL Database.

Detecting Syntactically Invalid Packets

We implemented LangSec-compliant parsers for the protocols shown in Table 4.10. To build these parsers, we purchased or procured the specifications of all SCADA protocols of interest. After carefully reading these specifications, we extracted the protocol state machine and the message formats in these protocols.

The protocol state machines specify what the correct sequences of packets must be and what sequences are prohibited. In contrast, message formats determine how a packet conforming to a specific protocol must look like. We converted these message formats to formal grammars.

```
IECGoosePdu = h.sequence(gocbRef,
        timeAllowedToLive,
        datSet,
        goID,
        T,
        stNum,
        sqNum,
        simulation,
        confRev,
        ndsCom,
        numDatSetEntries,
        allData)
```

Code Snippet 8: Code showing a portion of our IEC 61850 GOOSE parser. `h.sequence()` is a function provided by the Hammer parser-combinator toolkit.

We use parser-combinators to convert these formal grammars to code. Parser-combinators such as Hammer allow us to write parsers in code. As seen in the code snippet 8, the parser-combinators help us write parsers that visually resemble the formal grammar.

Our LangSec-compliant parsers detect packets that violate the formal specifications of a protocol. Although we cannot find or see specific semantic bugs, where well-formed packets crash the application, state-of-the-art fuzzers should find such semantic bugs. Our parsers or syntax validators cannot detect other types of attacks. However, based on our prior research, most new zero-days discovered are crafted input attacks such as buffer overflows that we prevent [14] (Section 3.1).

Our parsers are Kafka consumers that accept a raw byte string. We run our parsers on this byte string and decide whether the packet is safe or not. Often, packets may

conform to protocols we are yet to support. Based on the packet metadata (first two bytes of the payload and the Ethernet frame), we first shortlist packets to check if it may conform to a protocol we support.

These consumers run on Docker containers to ensure functional separation. Our parsers would return a parsed object if the parse were successful or `NULL` if the parse failed. The parsed object is essentially an abstract syntax tree (AST). Our parser interacts with the AST to store various information. Based on data extracted by our parsers, we keep all instances of failed parses due to malformed packets in our CVD database. After ascertaining if a particular packet is making any setpoint changes, we store these new setpoint values in our local CVD database.

Table 4.10: Various parsers included in CVD. The other LangSec parsers will also be made available soon.

Protocol	**Language**	**Code Availability**
DNP3	C/C++	Yes [42]
C37.118	C/C++	Yes [16] (Section 4.2)
SES92	C/C++	Not yet
GOOSE	Python	Not yet
MMS	Python	Not yet
SEL Fast Message	Python	Not yet

4.3.2. Evaluation

To evaluate CVD, we try to answer the following questions:

- Are the LangSec parsers CVD implements correct?
- Are the LangSec parsers in CVD resilient to fuzzers?
- Are CVD's network interfaces resilient to fuzzers?
- Can CVD detect crafted packet attacks for various protocols?
- Can CVD handle the high rate of traffic in SCADA networks?

– Can CVD visualize various human actions?

We ran our experiments on an Intel Xeon E31245 3.30 GHz processor with four cores and 16GB RAM. We used Apache Kafka version 0.10 and Hammer toolkit version 1.0-rc3.

Running our LangSec parsers through a large dataset

To ensure that our LangSec parsers cover a wide range of features in various SCADA protocols, we ran data collected from a SCADA tested through our parsers. Apart from running live network captures, CVD can also replay and process previously captured packet captures (PCAP files).

Dataset. We used a dataset provided by Yardley et al. [294]. They collected data from 20 substations and three control centers. These substations and control centers were not provided by an actual industry utility, but they were otherwise realistic, simplified physical substations with both SCADA communications and power equipment in a field-deployed testbed. These were experimental substations spread over two square miles representing three independent crank paths and fed by three generators connected by real overhead and underground wires. There were also high-voltage substations handling electricity at 13 kV.

Protocol	Number of Substations	Packets parsed Successfully	Total number of packets	Percentage of packets correctly parsed
DNP3	25	1888861	2007277	94.5%
MMS	1	35635	36262	98.2%
C37.118	2	1619479	1619582	99.9%
GOOSE	1	4501	4511	99.7%
SEL Fast Message	1	45802	46737	98.1%
SES-92	2	488147	503244	97%

Table 4.11: Our parser correctness experiments. We ran our parsers through a large dataset of SCADA traffic provided by Yardley et al. [294].

Each substation included at least four relays and an RTU. The three control centers had RTACs and HMIs. These devices were from at least four different manufac-

turers. Our dataset includes five hours of traffic. Although most of these substations and control centers ran DNP3, each of the following protocols—SES-92, SEL Fast Message, IEEE C37.118, IEC 61850 GOOSE, and IEC 61850 MMS—was running in one substation each.

The results of our experiment are in Table 4.11. Our parsers ran with a minimum accuracy of 94%, and most parsers had an accuracy of at least 98%. Our experiments demonstrate that our parsers cover an extensive feature set of these SCADA protocols successfully. Many practical DNP3 implementations support experimental and error-prone features that we are yet to support.

Fuzzing our parsers

Fuzzing SCADA devices can lead to crashes and denial-of-service attacks if the parsers are vulnerable [268]. As explained in Section III-D, CVD includes a set of LangSec parsers to ensure the syntactic validity of the SCADA network packets. Since CVD is designed to detect crafted-packet attacks for SCADA protocols, we want to ensure that fuzzing approaches do not crash CVD itself. Fuzzing CVD serves two primary purposes: (i) Parser resilience: ensuring that our parsers do not crash on any input—well-formed, malformed, or random, (ii) Network resilience: testing the network interfaces of our consumers to check if they can handle the high volume of traffic in SCADA networks.

Parser Resilience. We had to use separate fuzzers for our C/C++ parsers and our Python parsers. We fuzzed our C/C++ parsers using AFL++ [94] and our python parsers using Python Fuzz [213]; both coverage-guided fuzzers. To create a fuzzing target for AFL++, we created additional C files that called our parsers. AFL++ requires us to compile the program with instrumentation using an `afl-cc` compiler. We then run `afl-fuzz` on the binaries generated with a seed folder. We created a

	Parser Resilience			
Protocol	**Number of packets**	**Unique paths**	**Crashes**	**Hangs**
DNP3	3.62 Million	623	0	0
SES92	637 Million	6	0	0
C37.118	112 Million	5	0	0
GOOSE	1.2 Million	254	0	0
MMS	2.2 Million	13	0	0
SEL Fast	1 Million	6	0	0

Table 4.12: Results of our fuzzing experiment with our LangSec parsers.

corpus of valid packets for our seed.

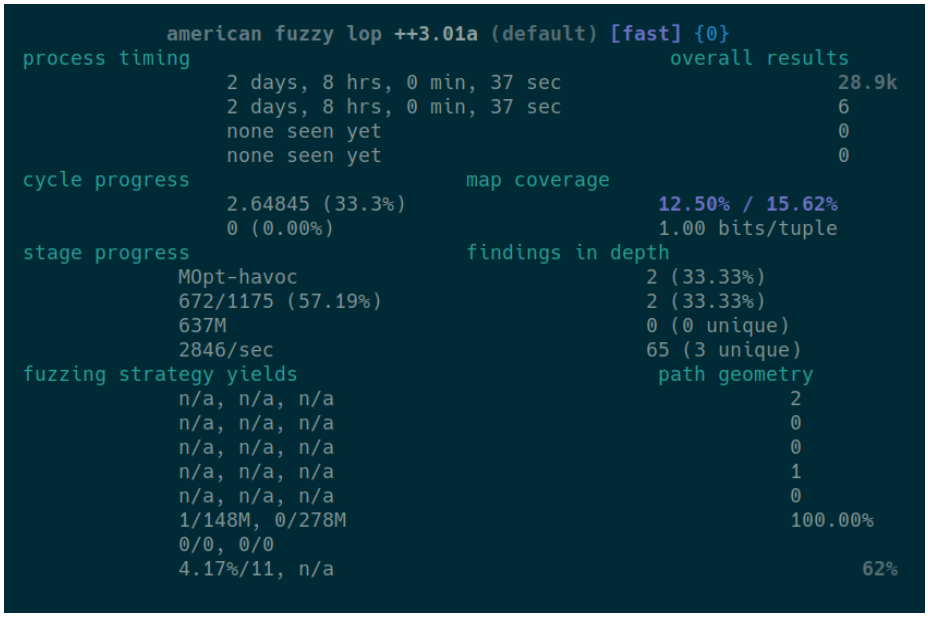


Figure 4.12: Fuzzing our SES-92 parser using AFL++.

Our fuzzing results are in Table 4.3.2 and Figure 4.12. We ran each of our fuzzers for 48 hours. None of our fuzzing executions led to any crashes or unresponsive parsers. Each of our python-fuzz targets ran at least one million permutations through our parser targets. In comparison, our AFL++ targets ran a minimum of three million executions through our parsers.

CVD versus Crafted Packets

To test CVD against crafted packets, we collected malformed DNP3 packets from the Aegis fuzzer [68]. Our sample consisted of 198 malformed DNP3 packets that were generated by mutating well-formed DNP3 packets. These malformed packets used some of the DNP3 vulnerabilities identified by Crain and Sistrunk [251]. Most of these vulnerabilities were structural or syntactic vulnerabilities.

Since Aegis only supports Modbus and DNP3, we mutated well-formed packets for SEL Fast Message, GOOSE, MMS, IEEE C37.118, and SES92 protocols. We generated 198 packets for each of these protocols. We mutated packets so that they mostly conform to the protocol but violate the specification in specific locations.

We fed these generated packets to our CVD producers, which passed on these packets to the respective CVD consumers. We found that CVD was able to detect all the mutated packets as malformed. Also, none of these malformed packets led to any crashes on any of CVD's parsers.

CVD's Visualization Capabilities

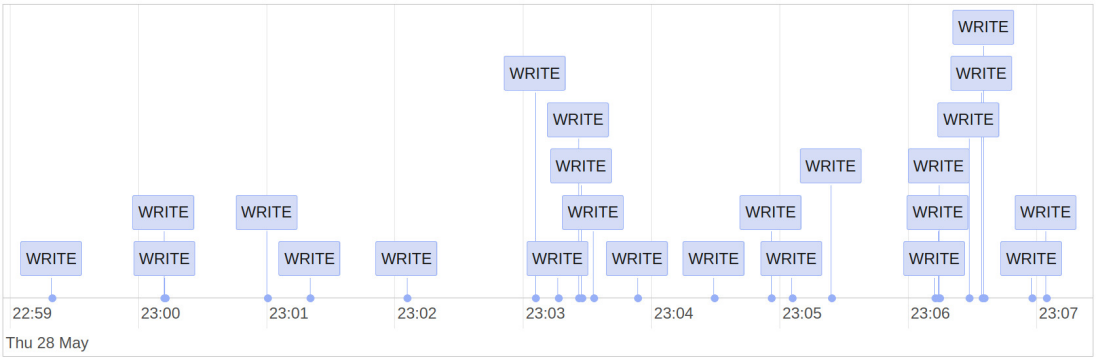


Figure 4.13: CVD's Web UI displaying the DNP3 Write operations captured within a ten minute window.

One of CVD's core features is an added visual component so operators can confirm

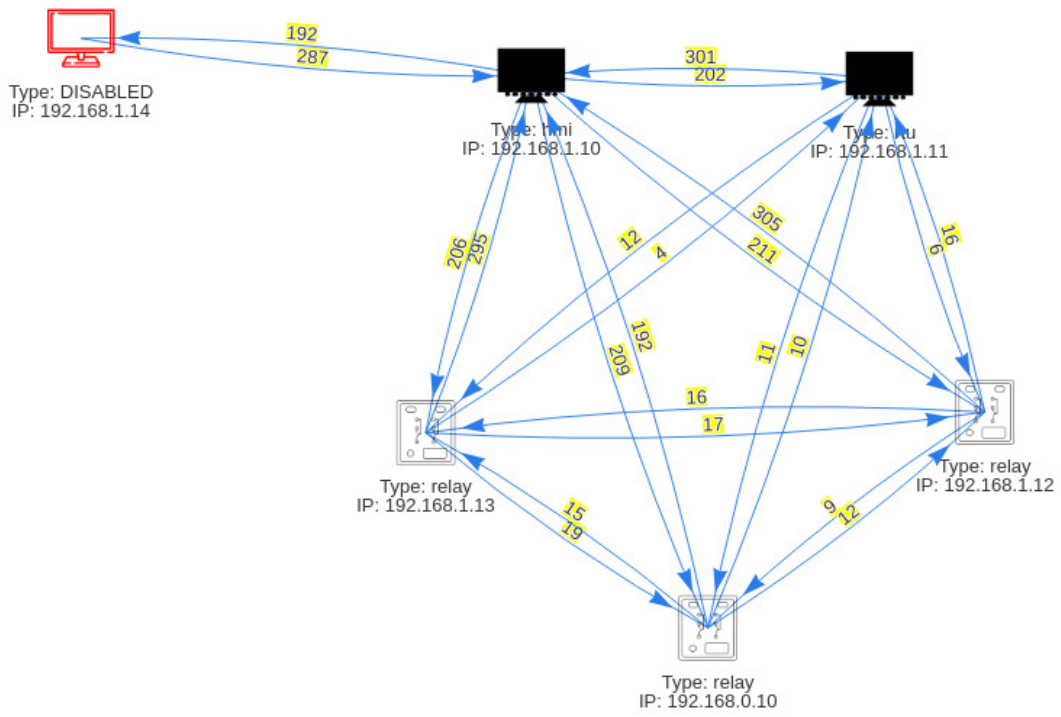


Figure 4.14: CVD's network UI. Devices not identified are shown in red. The highlighted numbers on edges shows the number of packets observed.

user actions. To validate our various UIs, we constructed several scenarios. We created network configurations for our test network with three relays, two RTUs, and one RTAC from three different manufacturers. This network was a SCADA-only network. Our relays were not controlling any live power settings.

We removed various devices from our network configuration and ran CVD on the network. We found that CVD was successfully able to detect network devices not present in the configuration. Our network configuration mismatch analyzer triggered alerts to alert the user with the network diagram in Figure 4.14.

Next, to evaluate our DNP3 timeline UI, we crafted a scenario where a malicious program injected various DNP3 WRITE commands in the network. Operators must monitor our DNP3 timeline to ensure no malicious WRITE or OPERATE commands enter the network. Only users can trigger these commands.

When we injected DNP3 WRITE commands and ran CVD, we found that CVD generated various alerts and generated the timeline UI in Figure timeline. If an operator notices the timeline UI in such a state, it could imply that one of the network devices could be sending malicious commands.

4.3.3. Conclusions

This section presented CVD, a novel, distributed tool to monitor SCADA communications networks for crafted input attacks and malicious operations. We presented our novel paradigm of continuous monitoring and data collection to gather forensics from the SCADA network. We showed how we use LangSec-compliant parsers to detect crafted input attacks and malformed packets.

We demonstrated CVD Minion's parsers on a wide range of SCADA protocols and a large dataset of valid SCADA traffic. We also built a GUI that would enable SCADA operators to make security decisions. We fuzzed CVD to show that it is resilient and can detect various SCADA misconfigurations.

Section 4.4

Parsing ICC Max

The International Color Consortium (ICC) provides a file format to create and interpret color data [202, 265]. The consortium was established in 1993 with nine founding industry members to standardize how different vendors handle color profiles and rendering. In 2019, ICC approved the iccMAX specification, which is also ICC.2:2019. ICC also provided a reference implementation to go along with the iccMAX specification. The reference implementation and specification are publicly available [76, 130].

Implementing a parser for the iccMAX file format presented several challenges. First, since the specification is recent (2019), we could not find many ICC profiles

to test our parser. This is especially a problem because several vendors have yet to adopt iccMAX and provide support for these profiles.

Second, the iccMAX specification defines a calculator element that uses a stack and defines several operations on the stack not entirely different from assembly code. Therefore, to correctly parse a calculator element and ascertain that the entire iccMAX file is safe, we need to perform two checks: (1) syntactically validate the iccMAX file using parsers, and (2) statically analyze the calculator element to ensure safe stack operations.

Finally, the iccMAX specification and the reference implementation contain numerous errors that researchers find and fix every day [131, 291]. The iccMAX specification is a moving target. Other than implementing the specification, we also had to be on the lookout for inconsistencies and bugs in the specification itself.

In this section, we demonstrate how you can parse an iccMAX file using parser combinators, and build a static analyzer to model the stack effects of various operations inside the iccMAX calculator element.

4.4.1. iccMAX parser

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

Cursor control. Complex file formats often require users to search for a magic character to begin parsing. For example, the actual file contents start only after the magic string `%PDF` in the PDF format. Similarly, the contents of a PDF file following

the `%EOF` tag are ignored.

Hence, control over the cursor's location is crucial in a parsing library. For example, we may need to set the cursor's location to a specific location in some formats. Similarly, we may have to skip over a particular set of bytes until we find a magic string, and depending on the offsets we find, we may need to parse the bytes we have already skipped.

Creating views. Like the PDF format, the iccMAX format contains a tag table with a list of tag entries. Each entry comprises a tag, the offset, and the size. The iccMAX format uses this pattern often. Each calculator element also contains a list of offsets and sizes relative to the calculator element. These offsets and sizes show where the main function and all the subelements of a calculator element are located.

The main function and the subelements need not be contiguous with the calculator element header—there could be arbitrary data between these blocks of an iccMAX element. These offset-size structures allow us to define a view—a fragment of the complete parsing buffer, such that the cursor can only move within this fragment.

Creating a view using the Parsley combinators is a two-step process. First, we define the transformation using the TransformView structure. It follows the syntax `TransformView::new(position, size)`. We then call the transform function with an existing view as an argument. Views are recursive structures; we can create additional views from an existing view.

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. At every parsing operation, these checks are enforced on the parsing buffer and the view.

4.4.2. iccMAX Static Analyzer

In addition to a parser for the iccMAX format using our Parsley Rust combinators, we added a static analyzer to further analyze these iccMAX profiles, and in particular, the calculator elements. The static analyzer uses the syntax tree produced by the parser to analyze the calculator element. The static analyzer checks the syntax of various operations and the stack effects of each operation.

The iccMAX specification defines a minimum stack size of 65 535. To ensure interoperability, every iccMAX implementation must adhere to this minimum stack size requirement. Therefore, given an iccMAX profile, we must detect stack overflows and underflows along multiple paths of calculator functions.

Each calculator element contains the main function with a set of operations and subelements. 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.

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

The calculator element and each subelement define how many input and output channels they use. When a subelement is invoked from the main function, we check the number of input channels used by the subelement and pop those many elements from the stack. Similarly, we add the output channels back to the stack when the subelement 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 `switch` 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 `switch` 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 `switch` statement does not hold a default condition. In that case, we compare the stack size before the `switch` 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 with these malformations to help future iccMAX implementors test their parsers with these malformed files. We were able to detect these malformations in the given files successfully.

4.4.3. Showcase

We uncovered several issues with the iccMAX reference implementation and the specification while implementing our parser and static analyzer. We detail the issues we

found in this section. In addition, we reported all these issues to the ICC and PDF Association.

In implementing our iccMAX parser and static analyzer, we have uncovered several errors in the specification and the demo implementation. In addition, we have already 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 demo implementation.

Conditional operations are insufficiently defined

Conditional operations such as if and switch conditions were not sufficiently described in the specification. These operations could have nested structures, where a switch operation could have an if condition or a switch 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 in-

formed 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.

Subelement type mappings missing

The iccMAX specification uses the following language: "The curv, mtx, and clut operators require that the indexed subelement has the appropriate type." The subelement types for each of these operators are not specified in this text. We proposed adding a mapping for these operators to subelement types. The specification must contain the following mappings: mtx/matf, curv/cvst, clut/clut, and calc/calc. The calc operation and subelement type were missing from the original list. The ICC working group has acknowledged these changes.

Operators missing in specification

Since the iccMAX specification and demo implementations are relatively new (since 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 demo 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 about missing specifications.

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

Incorrect subelement index implementation

In the previous section, we described that subelement indices are a four-byte value. We can invoke any subelement by referencing the index number and the correct operator type.

When we inspected various iccMAX profiles as part of the demo iccMAX implementation, we found that several of these files were malformed. Instead of using a four-byte index as an argument to the subelement operations, the demo iccMAX implementation converted this argument to two two-byte arguments where the second argument is always zero.

Any reasonably crafted calculator profile must not need more than 65536 (2^{16}) subelements. However, this implementation of iccMAX clearly violates the specification. Given that extant data has already been produced and several vendors are building on top of the demo iccMAX implementation, the ICC working group has two options.

First, alter the specification in multiple places. For example, make sure that the subelement 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 subelement 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.

Section 4.5

Armor Within: Defending against Vulnerabilities in Third-Party Libraries[7]

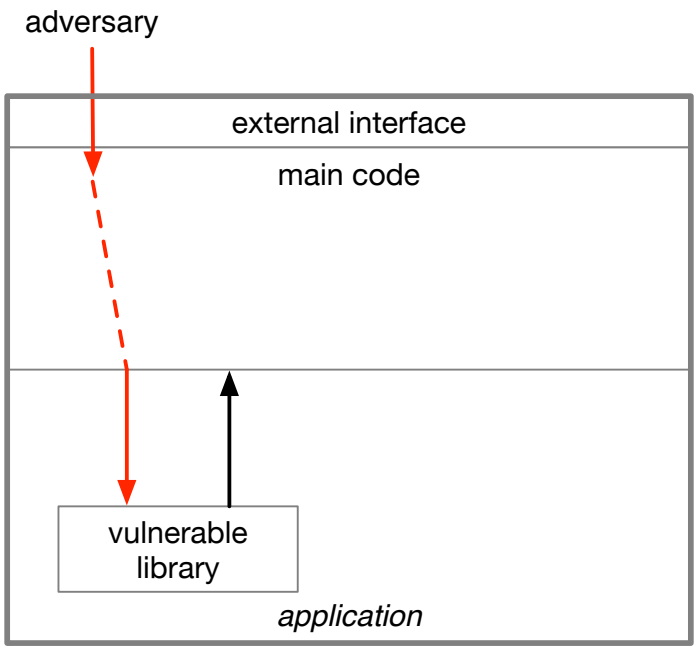


Figure 4.15: In the general model we consider, an internal library has a crafted input vulnerability; we must defend against the adversary tricking the main program into calling that library with suitably crafted input.

The standard way to defend against crafted input attacks is validate the input that comes in at the adversary's attack surface. However, when an internal library is vulnerable (Figure 4.15), it's not clear what to filter for at the adversary's attack surface. We therefore propose enforcing strong protections at the software module boundaries—e.g., between the main program and each library it uses. These protections will provide a strong guarantee that even if a third-party library is vulnerable to a crafted input attack, the overall application will not be exploited; the adversary

[7] This section borrows text and images from [4]. Also, Sameed Ali led this work, and I collaborated with Sameed and Sean.

may try to trick the main program into passing the right crafted input to the vulnerable library, but our defense will ensure that no unvalidated input crosses that channel. This allows us to reason strongly about software security. We can be sure that any crafted input attack which requires the input to be syntactically invalid will not be able to exploit the software module—thus reducing the severity of another libpng-like 0-day [189], if it were to happen again. Furthermore, these drop-in filters could easily be added to applications after a vulnerability is discovered to protect them from being exploited by a malicious attacker.

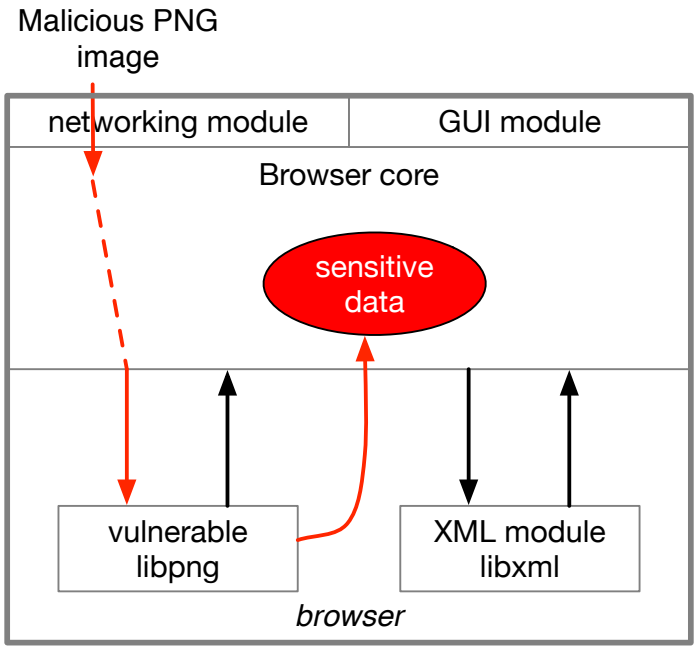


Figure 4.16: In CVE-2004-0597, the adversary tricks the browser into sending a malicious PNG file into the libpng library. The exploited software module can then access sensitive information in other parts of the address space.

Attacks Third-party libraries are used by multiple widely used software programs and are often targeted by malicious actors who try to find exploits in these widely used modules so they can reach a larger target audience for their malware.

Programs accept arbitrary input from users, sockets or files. These programs are

expected to fail safely and reject malicious inputs. However, in the case of the remote code execution bug found in libpng (CVE-2004-0597), if the adversary can trick the main program into passing a specially crafted PNG to the library, the library overwrites the stack of the main program. Figure 4.16 visualizes the data-flow of this attack. Traditional security considers hardening the outer boundary of the main program, and treats libraries as black boxes; preventing these attacks requires hardening the boundaries between the main program and the libraries.

Existing Defenses Most solutions that have been proposed—such as Write-XOR-Execute [193], stack cookies [89], DEP [240], SafeSEH [183], CFI [1], and ASLR [215]—do not prevent malformed input from being consumed. Indeed, they attempt to prevent exploits *after* the exploit starts execution by detecting a memory corruption or a control flow integrity violation.

Although these techniques do manage to stop some exploits midway in execution, they offer no protections that crafted or malformed input will not be consumed by the application in the first place. In contrast, our drop-in LangSec filters will detect and prevent a crafted input attack before the exploit executes itself. Moreover, sophisticated exploits are often able to bypass these defense techniques, so there is a need for a more effective exploit mitigation mechanism.

The existing software hardening approaches also do not take advantage of software modularity. A defensive mechanism which attempts to use this aspect of software architecture to its advantage will prove highly useful as we shall demonstrate with our approach.

We tackle these issues using a suite of techniques to provide *Armor Within* an application. First, we propose enforcing strong protections at software module boundaries. We do this by injecting LangSec filter before data enters the untrusted modules. A LangSec filter is the software component which validates the input by parsing it as

a formal language.

Second, we use ELFbac to compartmentalize software and enforce fine-grained access control on code and data sections and across software module boundaries. By ensuring that programs do not jump to code sections that are not allowed via policy, ELFbac enforces that entries to a vulnerable library must first go through the filter.

Finally, we convert legacy binaries to include ELFbac metadata and LangSec filters to reduce the human effort required to deploy our tools. Our LangSec filters consume CPU time in the order of micro-seconds.

I refer the reader to the paper [4] for a full treatment of this work.

Section 4.6

Conclusions

This chapter described some of my earlier work where I used parser combinators to build recognizers for various network and file formats. We evaluated our parsers against corpora of data to ensure that the implementations were correct. However, our approaches did have some pitfalls.

The need for verified tools. We found that the Hammer parser combinator, just like any other C library, is not immune to various memory safety bugs. Although efforts are underway to fuzz the various Hammer backends to build assurance, the lack of termination proof and the huge variation in functionality across different parsing backends confuse developers. A verified parser combinator toolkit would guarantee various correctness properties and termination properties.

Hammer does not implement garbage collection. As we were fuzzing the Hammer-based parsers we wrote using libfuzzer, we found that Hammer does not provide any functionality to free memory after it has been allocated to a parser

object. Although we can avoid this from being a major issue in software with clever engineering, it presents a challenge to support fuzzing efforts efficiently.

Chapter 5

SPARTA: A Strict PDF Type Checker

Section 5.1

Introduction

In Chapter 4, we studied how we can use parser combinators to build parsers to ensure that we fully recognize input before operating on them. In this chapter, we switch gears to study parser differentials and present an approach to finding sources of parser differentials.

Parser differentials occur when two parsers implement a specification in slightly different ways that may be interoperable. In addition, the tools creating input may also contain these differentials. For example, two producers of JPEG files may both misinterpret portions of the specification—producing slightly malformed files. This chapter studies Portable Data Format (PDF) creation tools, how they may produce malformed PDFs, and how we find these malformations.

PDF is one of the most commonly used data formats in today's age. We use this format to share documents such as this thesis proposal, presentations, and home

leases. The format has iterated and evolved over the years since Adobe introduced it in June 1993. Adobe maintained the specification until 2008, when it released its patent to the International Organization of Standardization (ISO). There were six versions of the PDF standard until 2008. As part of ISO, a committee of volunteer industry experts votes on changes to the PDF specification.

Since 2008, ISO has released two versions of the specification: ISO 32000-2:2017 and ISO 32000-2:2020—both of which are versions of the PDF 2.0 specification. Both of these versions of the specification hold no normative references to any proprietary files or formats.

The PDF format is extremely complex—it contains various objects spread out across the file with a cross-reference table (`xref` table) at the bottom of the file containing pointers to each of these objects. A parser first looks at the cross-reference table and jumps to each location in the table to parse an object. Hence, we need a mechanism to read a value and jump to a location based on that value to parse PDF files. This construct can lead to cyclic references—which in turn makes it difficult to build a verified parser for the format [38].

These objects in a PDF file can be of several types: (1) Text stored in content streams, (2) Font stored as Embedded Font objects, (3) Vector graphics, (4) Embedded images, and (5) other special objects such as forms and encrypted objects. The format uses several base types that are extensively used.

Specifically, the dictionary and array objects in PDF require a specific syntactic structure. For example, the Catalog dictionary is the root of all the objects connected in the form of a tree. In addition, the Catalog dictionary holds other information such as the version, language, and page layout.

Keys in a dictionary are optional, mandatory, or forbidden. The PDF specification includes tables specifying the keys and their corresponding types for each dictionary

and array type in the PDF specification. Each key in a dictionary or location in an array can correspond to a fundamental type (number, integer, string, etc.) or a complex type (dictionary, array, stream, number tree, name tree).

Different PDF readers implement various features differently. For example, Rohlmann et al. [233] found that 15 of the 26 PDF viewers they tested, would display visible content over previously certified content. They also found that different implementations test permissions in PDF structures very differently.

Similarly, Muller et al. [197] define various attacks against PDF readers by crafting attack paths. They demonstrate that different PDF viewers implement various PDF constructs differently—leading to infinite loops in some tools. Similarly, they also found that some PDF creation tools may reveal information about users via metadata. Different PDF creation tools and viewers implement features of the PDF specification differently. Due to these inherent differences, viewers often display different outputs for the same PDF files.

Hence, detecting parser differentials between different PDF creation tools is essential. Muller et al. [197] and Rohmann et al. [233] use pixel comparison and Optical Character Recognition (OCR)-based text comparison to check if PDFs are rendered differently. Even when implementations deviate from specifications in subtle ways, the PDFs display and render differently.

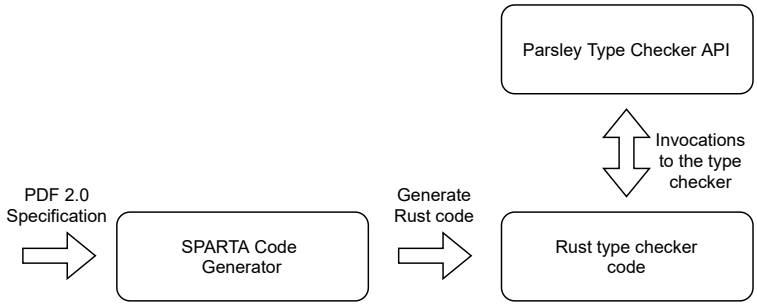


Figure 5.1: Overall overview of SPARTA and the Parsley PDF Type Checker API.

I present two tools: (1) A PDF type checker API in Rust, and (2) SPARTA—a tool

to generate Rust PDF type checker code from the Arlington PDF Document Object Model (DOM). This DOM provides the tables in the PDF specification in a machine-readable (XML and TSV) form. Figure 5.1 provides an overview of our project. SPARTA generates Rust code that includes invocations to the Parsley type checker API. Our Rust type checker API supports most annotations required in the PDF specification, such as indirect references, required fields, and heterogeneous arrays.

My contributions in this chapter are as follows:

- I built SPARTA—a first-of-its-kind tool that generates a type checker from the Arlington PDF DOM: a machine-readable PDF specification (Section 5.3).
- SPARTA has already contributed to several corrections to the Arlington PDF DOM: some logical errors and other typos (Section 5.5). In addition, SPARTA has also led to corrections to other commercial PDF creation tools and the PDF 2.0 ISO specification.
- I also contributed to the Parsley type checker and built the transformer, and serializer. Our Parsley Rust type checker provides an API that closely aligns with the fields in the Arlington PDF DOM (Section 5.4).
- I present the PDFFixer, a tool to dynamically modify PDFs to fix commonly seen type errors. The PDFFixer produces PDF files that produce fewer errors and more text when run through text extraction tools (Section 5.7).

Section 5.2

PDF Format

PDF files, in their simplified form, comprise five basic structures [216].

- **Header**: this specifies the PDF version this file follows.

- **Objects**: a list of objects each conforming to a predefined PDF type.
- **Cross-reference table**: also known as *xref* table, this table contains a list of absolute locations of the objects in the PDF file.
- **Trailer**: this dictionary stores references to the *Root* object and *info* dictionary. The trailer also contains other metadata fields such as the size of the *xref* table.
- **End of file marker**: this marker shows where the *xref* table is stored.

Each object in a PDF file are identified by an *object* number and a *generation* number. For example, let us consider the following object.

```
13 0 obj
Object contents
..
endobj
```

In the above example, the object takes the object number 13 and generation number 0. This object can be referenced from other objects or the trailer via the reference `13 0 R`.

Dictionaries PDF dictionaries are surrounded by `<<` and `>>` characters. The keys in these dictionaries must be of the *name* type, whereas the value portion can hold any predefined PDF type. For example, `<<</Foo1 (bar) /Foo2 13 0 R>>` is a dictionary with two keys, `Foo1` and `Foo2`, where the corresponding values are of types string and reference respectively. The ISO 32000-2:2020 PDF specification specifies several rules on various dictionaries. The Arlington DOM captures these rules in machine-readable specifications.

Arrays These are predefined objects that are surrounded by brackets (`[` and `]`) and contain multiple elements separated by spaces. Arrays can be homogeneous or heterogeneous and can be fixed or variable length. For example, `[(foo) /bar 13 0 R]` is a heterogeneous array of size three where the first object is a string, the second object is a name object, and the third object is a reference.

Streams These predefined objects contain a dictionary and a compressed stream. The adjoined dictionary holds keys that specify the compression algorithm used and the length of the stream. In most cases, stream objects are *indirectly* referenced, i.e., they are entirely stored in a different object and not embedded in another object. The code snippet below shows an example of a stream object.

```
13 0 obj
<<
stream dictionary objects
>>
stream


endstream
endobbj
```

The Parsley PDF parser follows several steps to validate a PDF file thoroughly. Figure 5.2 shows these steps. First, we syntactically validate all objects in a PDF file by following the objects from the xref table. This step ensures that the dictionaries, streams, arrays, and other predefined-typed objects are well-formed.

In the second step, we type check the various objects starting with the catalog dictionary and info dictionary. Then, iteratively, we type-check all the objects referenced in these dictionaries based on type specifications.

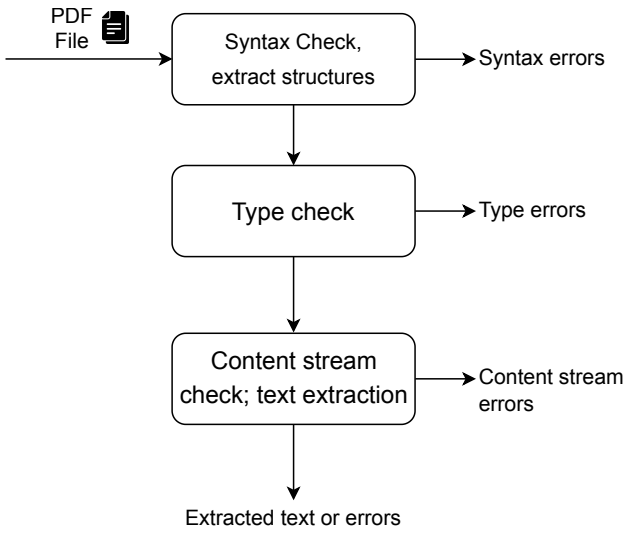


Figure 5.2: We show the steps followed by the Parsley PDF parser to validate PDF files.

5.2.1. Related Work

PDF readers are also susceptible to crafted packet attacks like any other system. In 2021, several CVEs were issued since PDF readers were vulnerable to various crafted packet attacks. For example, Polaris Office had a vulnerability where an attacker could cause remote code execution by making the victim open a crafted PDF file [161]. Similarly, in Corel PDF Fusion, an attacker could use a heap corruption vulnerability to execute arbitrary code if the victim opens a crafted PDF file [245]. Finally, various Adobe Acrobat versions also had a stack overflow vulnerability that could be exploited by opening a crafted file [157].

Validating PDF files is an ongoing problem. Given how complex the PDF format is, it must not be a surprise that we see so many vulnerabilities in various PDF readers from time to time. In addition, different PDF readers implement different validation and rendering algorithms.

These subtle differences lead to differentials—the same file produces different results on different readers. These differentials could be in terms of validation, render-

ing, or security.

Caradoc Caradoc [87] is a PDF parser freely available on GitHub. The Parsley project started with the goal of making parsing file formats easier. Caradoc, on the other hand, targeted the PDF format. Caradoc implements three passes to check PDF files. For lexical analysis and parser generation, Caradoc uses MENHIR and OCAMLLEX. They implement a strict type checker and a content stream checker for the other two passes.

Although the Parsley PDF parser does the same three passes, it uses a parser combinator library in Rust to check the syntax. In this chapter, we focus on the Parsley type checker. In addition, we provide an extensible interface to support code generation. Any changes to the PDF specification will need significant changes to the Caradoc code base. The Parsley type checker uses SPARTA-generated code to alleviate this issue of code changes.

Table 5.1 compares various features in Caradoc and Parsley. The Parsley type checker is only one component of the Parsley PDF parser. For the sake of this table, we decouple the type checker from the rest of the parser. The Parsley PDF parser goes above and beyond many of Caradoc's capabilities.

Caradoc includes a *normalizer* to patch broken PDF files. In Section 5.7, we test the efficiency of this normalizer by running files through the normalizer in an additional pass before running the PDF file through the Parsley parser. Caradoc also checks a PDF file for cycles. They implement indirect references in files as a graph and look for cycles in these structures. A PDF file with cycles can lead to infinite recursion or nontermination if not handled carefully.

Property	Caradoc	Parsley Type Checker	Parsley PDF Syntax Checker
Overall PDF Syntax	Yes	No	Yes
Cycles in PDF objects	Yes	No	No
Duplicate keys	Yes	No	Yes
Normalizer	Yes	No	No
Type checker API	No	Yes	No
Dictionary Specifications	Partial	Yes	No
Array Specifications	Partial	Yes	No
Stream Specifications	Partial	Partial	No
Number Tree Specifications	Partial	Partial	No
Name Tree Specifications	Partial	Partial	No

Table 5.1: A comparison of Caradoc features and Parsley features. We extracted the list of features from [87].

5.2.2. Extant Data and the de facto standard

The SAFEDOCS program is a DARPA program designed to build better parsing and parser monitoring tools. The SAFEDOCS program broad agency announcement states the following "dominant implementations of these formats extend the [published] standards by deliberately accepting non-compliant inputs without any indication to the users that the document contains malformations silently presumed benign" [217].

A de facto standard is the standard implemented by dominant implementations that vary from the official standard in tiny, non-malicious, and subtle ways [67].

Similarly, files that implement these de facto standards are called extant data. These files mostly conform to the specification, and most commercial PDF readers cannot detect these malformations. However, some PDF readers print warnings when encountering some of these constructs.

In Section 5.5, we outline various extant PDF files and their constructs. We reported these issues to the PDF association. These issues led to the official specification and the de facto standard converging, leading to multiple proposed edits to the ISO 32000-2:2020 PDF specification.

5.2.3. Arlington PDF Model

The PDF 2.0 ISO 32000-2 specification spans over 1000 pages with several normative references. PDF files have been in use for over 25 years, and via a differing understanding of the specifications, contradictions in the specification, and errors of omission, various PDF implementations have deviated from the specification in various ways.

Wyatt et al. at the PDF Association proposed creating a machine-readable specification, the *Arlington PDF Model* (also known as the Arlington DOM or Document Object Model),[1] for various objects commonly used in PDFs. This specification can help reduce deviations and inconsistencies in implementations. The Arlington PDF Model captures the keys present in Arrays (indices), Dictionaries, and Stream objects.

The DOM defines 515 TSV files with a total of 3544 rows of data. Each row includes 12 columns. We describe the columns that are vital to SPARTA and supported by the Parsley type checker.

- Key: For each row in the DOM, this column defines the key portion of the dictionary.
- Type: This column defines the type of the value portion of this dictionary field. There are 18 pre-defined types. This field cannot contain any other type.
- PossibleValues: For basic predefined types such as integers and names, this column lists all the possible values the field can take.
- Required: Takes a true or false value to denote whether the field is required.
- Indirect: Specifies whether a field must be indirect. There could be cases when the field must be indirect for only specific choices (such as streams) and be

[1] Peter Wyatt, one of the lead creators of the DOM said, "it is named after the DARPA HQ location as a nod to their funding support."

direct for other choices.

– Link: Links help define a type further into a a TSV class. For example, the "Type" column may define a particular field to be a dictionary. What fields this dictionary must contain and their properties are defined by files listed in the "Link" column. The types arrays, dictionaries, name trees, number trees, and streams always require links.

In the rest of this chapter, we differentiate between three types in the Arlington model.

(a) Basic predefined Arlington types: These are types that do not need additional descriptions. These types are bitmask, boolean, date, integer, matrix, name, null, number, rectangle, string, string-ascii, string-byte, and string-text. Our type checker implements these types out of the box.

(b) Complex predefined Arlington types: These are types that hold links in the DOM. Although they hold an overall structure, additional descriptions are needed to completely capture these definitions. Occasionally, these types may hold links to a `_UniversalDictionary` or a `_UniversalArray` type. These types describe generic dictionaries and arrays with no constraints on the keys or values present.

(c) TSV class: Links in the TSV files correspond to other TSV files. The type derived from the entire TSV file corresponds to the link in the original TSV file. For example, the Catalog TSV file holds a link to a PageObject TSV type which corresponds to the PageObject TSV file. The row in the Catalog TSV file specifies it is a dictionary, and hence the PageObject TSV type must be interpreted as a dictionary. In summary, TSV classes specify additional constraints on the complex predefined Arlington types.

Section 5.3

SPARTA Code Generator

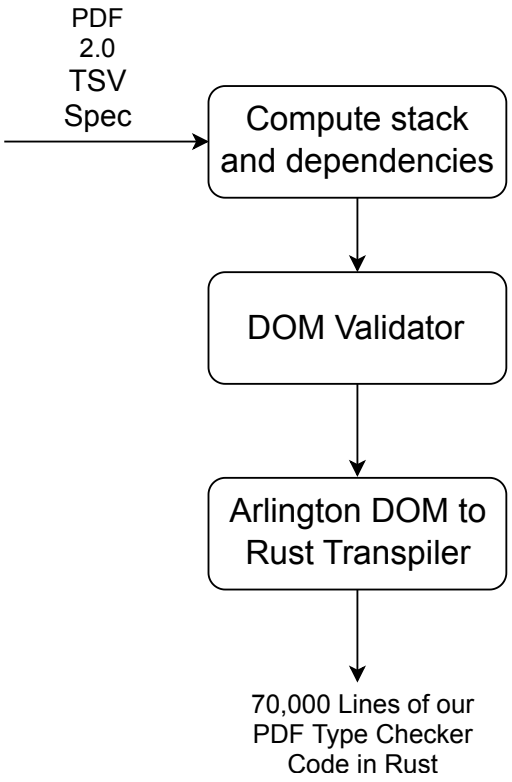


Figure 5.3: Overall architecture of SPARTA. We generate Rust code using the PDF 2.0 machine-readable specification in TSV format.

Figure 5.3 shows that SPARTA follows three steps to generate Rust type checker code. We use the passes to compute metadata and stack configurations needed to generate code that terminates. We implemented SPARTA in Python 3, and the source code uses about 1000 lines of code. As we will discuss later, we found several specification errors using the DOM validator pass.

5.3.1. Computing a list of classes

We start with a stack comprising one element—the catalog. Then, we process each key in the catalog dictionary and add the classes a key references to the stack. For example, the catalog contains an entry to an "Extensions" field and a corresponding class called "Extensions." Once we see a reference to this class in the "Link" column, we add it to our stack and explore this new class.

Once we add it to our stack, we also take a snapshot of the current stack to keep track of all the various classes higher up the tree. For example, if the extensions dictionary holds a reference to a catalog dictionary, we can create an infinite loop of class definitions. We use a naming convention for class definitions to avoid such cyclic class definitions. Similarly, we also keep track of all the sibling classes. Again, the extensions and pages fields in the catalog dictionary could hold cyclic references to each other, leading to an infinite loop.

In summary, we use a full-stack to keep track of all the class definitions (tsv files) we must explore. Simultaneously, we use a live stack to keep track of all the classes higher up the tree. Finally, we also keep track of siblings of a particular class to ensure there are no cyclic definitions.

5.3.2. DOM Validator

Once we extract type definitions from the Arlington PDF model, we perform a static check on the model to ensure certain integrity properties.

(a) The same type must not be defined as an array and a dictionary.

(b) Each predefined complex type must hold a corresponding tsv file.

(c) Dictionaries must not contain duplicate keys.

(d) Arrays must only consist of numbers from 0 to n-1 or a * entry.

(e) "*" must always be the last entry in an array or dictionary definition.

These validation steps are covered in the Arlington PDF model's internal specification and language document. Since we treat these properties as invariants in the Arlington specifications, we must make sure that these properties hold before generating code from this data.

Since the code generation process varies, if an Arlington class is a stream, dictionary, or array, we must handle these cases differently and run the above integrity checks to ensure that the same type is not used as multiple of the above-predefined types. Likewise, interpreting them becomes complicated if arrays contain non-numeric and non-star values.

The DOM validation step is vital in generating Rust code from the Arlington model. Unfortunately, we found three significant errors in the Arlington model—typos and contradictions—using the DOM validator. These issues are reported in more detail in Section 5.5.2.

5.3.3. SPARTA Transpiler

Since the Parsley type checker provides an API that can model heterogeneous and homogeneous arrays, dictionaries, and all predefined PDF types, we generate Rust code using templates. These templates store the overall syntax of these types and can be called when we encounter a particular type.

First, we use information gathered in the first pass. We ascertained what predefined types are called with each Arlington class (tsv file). The same class cannot be defined as an array and a dictionary. We check if the array is homogeneous or heterogeneous for arrays to call the correct code generator.

Homogeneous Arrays The Parsley API provides support for an `Array` structure that accepts an optional size parameter and a type parameter to specify what checks

to apply on each array element.

Heterogeneous Arrays These arrays use the `HetArray` structure in the API. It takes a list of types as an argument. The type checker goes through this list and applies each check in this array sequentially, ensuring the array's length and the type at each location are accurate.

Hybrid Dictionaries Dictionaries by nature in Parsley implement both the star and the individual keys. We use the `DictEntry` and `DictStarEntry` structures to specify the individual keys that go in a dictionary. As shown in the code snippet below, we specify the type for each key in a dictionary and whether it is required, optional, or forbidden.

Indirect References Each class in the Arlington DOM is implemented as a separate function. These functions follow a similar pattern—taking two arguments, a context variable, and a boolean to denote if a class needs to be indirect. We needed to use an additional argument since several classes get called indirectly and directly depending on the callee.

The snippet below shows a portion of the Arlington specification for the catalog dictionary (Figure 5.4) and the code generated using SPARTA (Code Snippet 5.3.3). For classes such as `extensions_type` and `pagetreenoderoot_type`, the functions use the two arguments we described earlier. For keys that define an allowlist of possible values, the function signatures use the `_possible_` keyword. We also use the `opt` parameter in these structures to denote if the field is required, optional or forbidden.

1	Key	Type	SinceVersion	DeprecatedIn	Required	IndirectReference	Inheritable
2	Type	name	1.0		TRUE	FALSE	FALSE
3	Version	name	1.4		FALSE	FALSE	FALSE
4	Extensions	dictionary	1.7		FALSE	FALSE	FALSE
5	Pages	dictionary	1.0		TRUE	TRUE	FALSE
6	PageLabels	number-tree	1.3		FALSE	FALSE	FALSE
7	Names	dictionary	1.2		FALSE	FALSE	FALSE
8	Dests	dictionary	1.1		FALSE	TRUE	FALSE
9	ViewerPreferences	dictionary	1.2		FALSE	FALSE	FALSE
10	PageLayout	name	1.0		FALSE	FALSE	FALSE
11	PageMode	name	1.0		FALSE	FALSE	FALSE
12	Outlines	dictionary	1.0		FALSE	TRUE	FALSE
13	Threads	array	1.1		FALSE	TRUE	FALSE
14	OpenAction	array;dictionary	1.1		FALSE	FALSE	FALSE

Figure 5.4: A snippet of the Arlington PDF model

5.3.4. Version-specific type checkers

The Arlington model currently provides separate specifications for each version of the PDF standard. However, since the goal of the PDF standard is to make sure that the specifications are backward compatible and implementations are forwards compatible, we used the PDF 2.0 standard to generate code using SPARTA.

We generate a separate type checker for each version in the PDF standard and use the version number in the catalog or the header to choose the correct version of the type checker to use. In our experiments so far, only the pdfcpu [120] tool implements version checks for keys. However, pdfcpu only supports PDF versions up to 1.7. The current active version of the PDF specification is 2.0. Table 5.2 shows the number of files for each PDF version in the Arlington DOM and the lines of code generated by SPARTA for each version.

```
    let entry_type = DictEntry {
        key: Vec::from("Type"),
        chk: mk_type_possible_typchk(
                            tctx),
        opt: DictKeySpec::Required,
    };
    let entry_version = DictEntry {
        key: Vec::from("Version"),
        chk: mk_version_possible_typchk(tctx),
        opt: DictKeySpec::Optional,
    };
    let entry_extensions = DictEntry {
        key: Vec::from("Extensions"),
        chk: extensions_type(tctx, false),
        opt: DictKeySpec::Optional,
    };
    let entry_pages = DictEntry {
        key: Vec::from("Pages"),
        chk: pagetreenoderoot_type(tctx, true),
        opt: DictKeySpec::Required,
    };
    let entry_pagelabels = DictEntry {
        key: Vec::from("PageLabels"),
        chk: number_tree(tctx),
        opt: DictKeySpec::Optional,
    };
```

Code Snippet 9: SPARTA-generated Rust code

Section 5.4

Parsley Type Checker

Our Parsley type checker closely follows the PDF specification, defining the fundamental and complex structures. In addition, we provide an API to define homogeneous and heterogeneous arrays. Finally, we support dictionaries with a fixed set of keys and values or dictionaries where the types of values are defined, but keys can be any name.

Version	Number of type files	Lines of code in Rust
1.0	46	6389
1.1	78	10617
1.2	147	23591
1.3	237	41415
1.4	264	47677
1.5	349	64626
1.6	385	71370
1.7	414	76114
2.0	511	96263
Total	2431	438062

Table 5.2: Number of type files and Rust code generated for each PDF version in the Arlington PDF Model

Since some fields in the specification hold a set of possible values, we support choice predicates where the list of names is provided as an array. The type checker checks each of these possibilities sequentially till a match is found. Similarly, we support disjunctions for more complex types. Fields in arrays or dictionaries can often take multiple types. Just like an ordered choice in PEGs, we iteratively go through these disjunctions till we find a match.

The PDF 2.0 specification defines fields in arrays with annotations such as "Required" or "Optional" and "must be indirect" or "indirect preferred." These annotations are vital in defining what fields are necessary for PDF dictionaries for syntactically valid PDF files. We support annotating each dictionary key with Required, Optional, or Forbidden keys. Additionally, we support an annotation to define that an indirect specification is required for a particular field in each type definition.

Our type checker runs a breadth-first search to match types. We start with the Catalog dictionary and Info dictionary in a PDF file. Then, since the catalog holds indirect references, we add these type definitions and the indirect object to our queue. Finally, we keep removing these items from the queue to check types.

Section 5.5

Showcase: Specification Violations

When we ran SPARTA on the Arlington PDF model and SPARTA-generated Rust code on a corpus of PDF files, we found several errors in the Arlington PDF model and errors in PDF files. We reported the errors in the Arlington PDF model to the PDF Association. Subsequently, they acknowledged these errors and fixed them. Since creating the machine-readable PDF 2.0 specification was a manual process, most of these errors were a result of data-entry errors. In the rest of this section, I detail the PDF model errors, and the errors in the PDF files. Table 5.3 summarizes the various errors and their outcomes.

Error	**Date Reported**	**Outcome**
Name-/Number-tree can hold links	9/20/21	Specification fixed
Incorrect types in DOM	12/05/21	Specification fixed
Array/Dictionary mismatch in DOM	12/05/21	Specification fixed
Name/String mismatch in Lang	7/20/21	Specification fixed
Indirect specifications not followed	7/21/21	Changes proposed
Type key missing in Fonts	12/15/21	More investigation needed
Page Tree Node keys missing	7/20/21	More investigation needed
Incorrect Date Format in `htmltopdf`	11/15/21	Fixed in immediate release

Table 5.3: A summary of errors uncovered using SPARTA and Parsley and their current status with PDF association (as of May 3rd, 2022).

5.5.1. Test Dataset

To test our type checker, we use a corpus of PDF files provided by the SAFEDOCS program [6, 217]. This corpus includes files from Common Crawl [48], GovDocs [101], and files artificially generated by the SAFEDOCS Kudu team. We used 10,000 files from this dataset to evaluate our type checker. These files vary in length, language, and, most importantly, the PDF creation software.

5.5.2. Errors in the PDF model

The DOM sanitizer in SPARTA (Figure 5.3) performs several wellformedness checks that we have detailed in Section 5.3. These checks allow us to be certain that the model is correct and does not contain contradictory information. The machine-readable specification must also conform to the text-based specification of ISO 32000. Discrepancies between the machine-readable specification and the ISO document could further lead to extant data.

Number trees and Name trees can hold links The Arlington PDF model holds a documentation file detailing all the fields in the model and what values each field must hold.[2] The documentation specified the following: "The following predefined Arlington types ALWAYS REQUIRE a link: array, dictionary, stream." and "The following predefined Arlington types NEVER have a link (they are the basic Arlington types): bitmask, boolean, date, integer, matrix, name, null, number, rectangle, string, string-ascii, string-byte, string-text, name-tree, number-tree." However, when we implemented this validation check, we found that name-trees and number-trees always contained links. We provided a pull request to the Arlington DOM Github repository to fix this issue.[3]

Same type defined as an array and a dictionary When we checked the Arlington model for cases where the same type is defined as multiple predefined types, we found that there were 25 instances where a type specification (tsv file) corresponded to multiple predefined types.

[2]Arlington PDF Model Grammar Validation Rules: `https://github.com/pdf-association/arlington-pdf-model/blob/master/INTERNAL_GRAMMAR.md`

[3]Pull request: `https://github.com/pdf-association/arlington-pdf-model/pull/4`

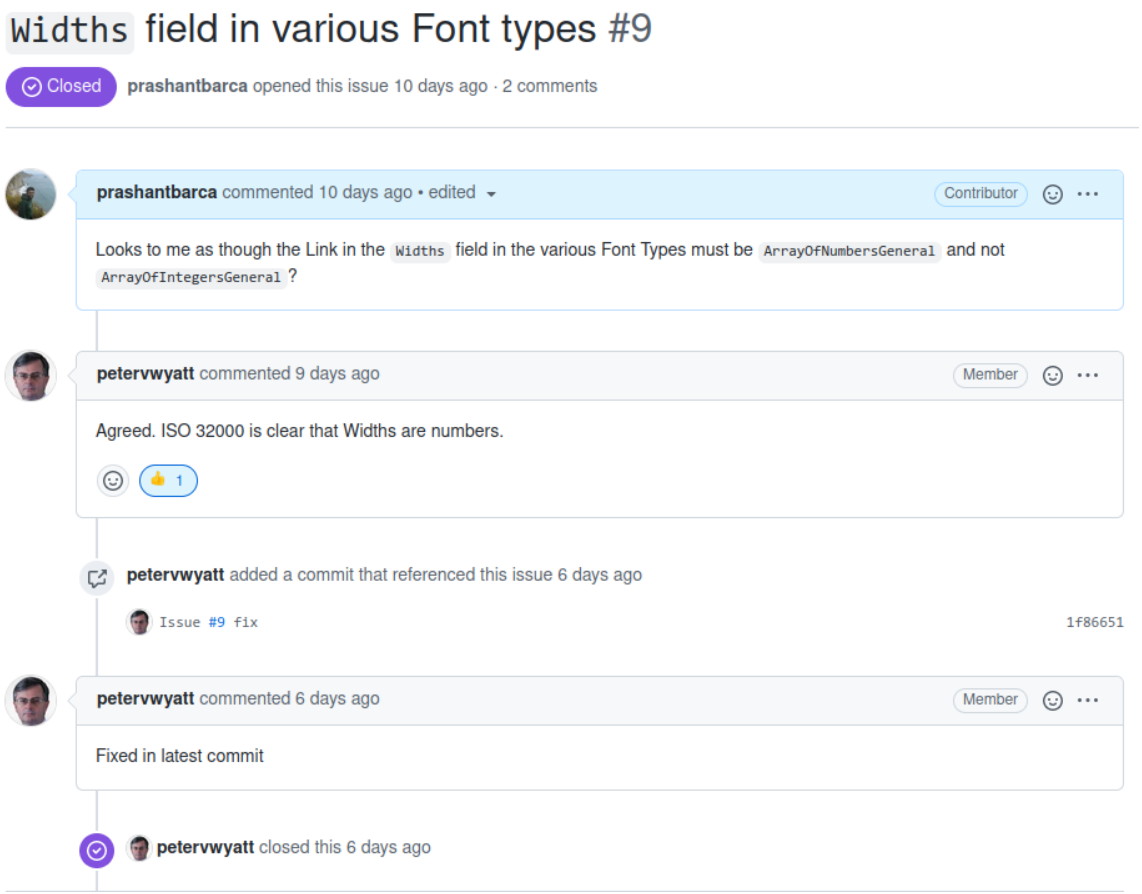


Figure 5.5: The Widths key was incorrectly described in multiple places in the Arlington PDF DOM.

Incorrect "Widths" key Various font dictionaries in the PDF format include a `Widths` key, which is an array of numbers. The number type signifies that these numbers can be a real number (decimal) or integer. An earlier version of the Arlington PDF DOM specified the `Widths` field in various dictionaries to be an array of integers.

We ran SPARTA and ran the generated Rust code on our 10,000 file dataset. When we investigated the various specification violations, we found that the key was incorrectly specified in the DOM and subsequently in our generated code. We raised an issue, which has been fixed in a later version of the Arlington DOM. Figure 5.5 shows our interaction with the PDF association.

5.5.3. Common Specification Violations in Extant Data

We found these errors by running our Parsley type checker on the dataset of 10,000 PDF files. Most of these errors demonstrate a poor understanding of the PDF standard on the part of the developers of PDF creation tools.

Type Errors This category of errors does not seem extremely common. However, occasionally, we find that specific keys in dictionaries do not contain a value of the correct type. For example, we found that several PDF files contained a catalog dictionary with the `Lang` containing a corresponding value of the name type instead of the string type [290]. An incorrect `Lang` key can confuse a text extraction tool and could lead to poor text extraction results. Figure 5.6 shows the issue created to illustrate this issue and how an incorrect value for this key can lead to confusion.

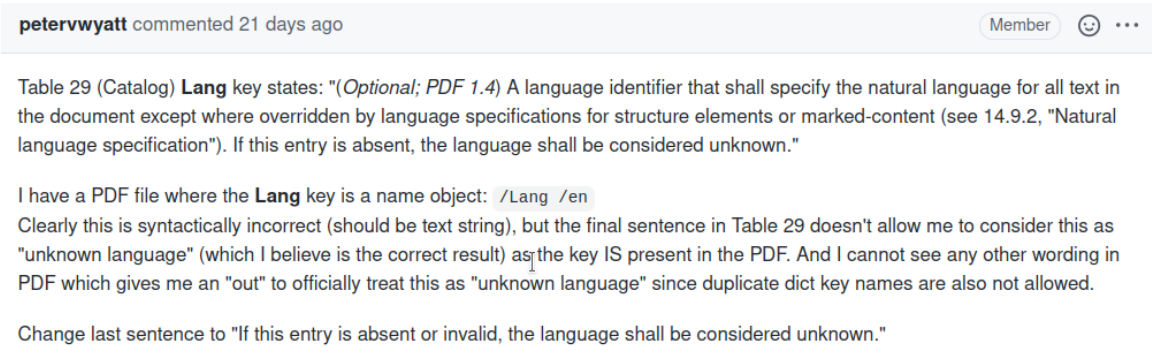


Figure 5.6: We found several files with invalid languages. The Lang key needs a value of type string. However, we found several files using the Name type in PDF.

Missing Keys PDF dictionaries contain several mandatory keys to ensure that a PDF file created on one tool can still be read and rendered on another tool. Unfortunately, missing required keys are the most common error format in our experiments. Two examples of this error occur frequently. First, out of the 10,000 files, we found more than 200 failing due to missing the count or kids keys in page tree nodes. Both

of these keys are mandatory as per the standard.

Similarly, we find that often font descriptor dictionaries are missing the `Type` key. Each PDF dictionary includes a type key to differentiate between the various classes. However, around 40 files missed this key. Most PDF readers do not provide errors or warnings when such mandatory keys are missing. Since error handling is not covered in detail in the specification, different implementations differently handle the lack of required fields.

Indirect References Our type checker enforces an indirect reference check if the specification requires an object to be indirect. We do this by ensuring a reference for that key, not a dictionary, array, or stream, in place. We found that the `Outlines` and `Dests` keys in the catalog dictionary appear to be in place quite commonly [289]. Figure 5.7 shows the GitHub issue we created for this issue. 96 files out of the 10,000 files failed our type checker with this error.

This issue on GitHub is still currently open. The PDF community is still discussing how this needs to be addressed in the next iteration of the PDF specification. Currently, the proposal is to weaken the specification to allow both direct and indirect references for these keys.

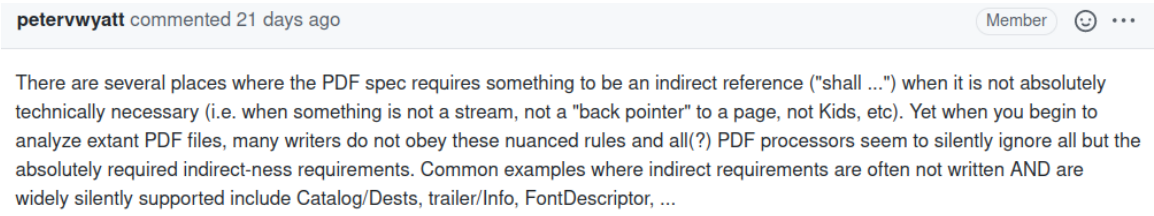


Figure 5.7: Issue opened by PDF Association to address the problem of indirect references raised.

Proprietary page layout and modes The page layout and page mode fields are a part of the catalog dictionary. They are both of the name predefined type, and the

specification provides a list of options for these values. We enforce a strict check to ensure that this allowlist is followed. However, we found several instances where the dictionary used keys that violated these lists. Some of these are proprietary values that are used in some PDF readers, whereas others were simple typos such as using `none` instead of `UseNone`. In addition, other keys with allowlists such as the `TR` key in procedure sets also include an allowlist that we observed was violated time and again.

Incorrect Date Syntax In the PDF syntax, date strings must start with a `D:` at the beginning, followed by the rest of the date, ending with the timezone information. We found that several files violated this date specification. Upon further investigation, we found that the `htmldoc` tool still implemented an incorrect date format. Therefore, we created an issue on their GitHub page to ensure this issue is addressed [9]. Subsequently, the maintainers have promptly fixed this issue to ensure future versions do not produce extant data.

Errors in Font definitions We found that this error was prevalent mainly in files generated via latex tools. These files sometimes did not contain Type, BaseFont, or SubType tags in the font definitions—these tags are mandatory in font dictionaries.

Missing Keys in Data Dictionary The Data dictionary is referenced from the PieceInfo key in the Catalog dictionary. This dictionary stores two keys: LastModified and Private. The LastModified key stores the timestamp of when the file was last edited. Whereas, the private key stores other private data in a dictionary. The LastModified key is required in this dictionary type. However, we found instances of files where this key was absent, leading to differentials.

5.5.4. Skia Errors

The Skia open-source graphics library serves as a graphics engine for multiple browsers and operating systems. In addition, it provides ways to describe graphics in C++ and generate PDFs and SKP files. In our dataset, we found a cluster of 53 files that were produced using the Skia graphics engine milestone 90, which Google released in August 2021. In this section, we examine these errors.

Error	**Frequency**
Type spelled as Typ	16
Content Stream reference absent	12
Parent key missing from Page	10
Page Tree Node has key Page (should be Pages)	1
Kids and Type key missing from Page Tree Node	1
Typo in the keyword Page	4
Typo in the keyword Catalog	3
Root points at Info instead of Catalog	1
Root points at XObject instead of Catalog	1
Page Tree Root contains Parent key	4
Typo in string "Normal"	1

Table 5.4: Skia/PDF errors summarized

Type misspelled We found that 16 files contained a typo in the word "Type." This key in the dictionary is mandatory to understand the type of the dictionary. For example, a catalog dictionary has the string Catalog corresponding to the key Type. A typo in this key makes parsing a PDF unambiguously a colossal challenge.

Typos We found several keywords such as the string "Normal" or "Type" were misspelled or were spelt with incorrect cases. For example, we found instances of the typo "Typ" instead of "Type" and "norMaL" instead of "Normal."

Content stream objects missing Several files contain content stream references that we were not able to resolve to an actual stream object.

Root does not point to Catalog The trailer in a PDF file specifies the Root object ID, which must reference a Catalog dictionary. However, files created by Skia/PDF contained references to the Info dictionary instead of the Catalog. Similarly, we also found references to XObject types.

Malformed Page tree nodes A page tree node must contain three keys: the Type, Kids, and Count keys. An additional Parent key must be present in page tree nodes that are not the root. In addition, the Type key must correspond to the value Pages. This key occasionally contained the value Page instead of Pages. Also, there were instances where the required Parent key was absent in non-root nodes, and the Kids and Type key were missing.

In summary, we found several files that contained the producer Skia/PDF. These files contained numerous minor defects and typos, as we outlined in Table 5.4. However, not all of these errors are easy to fix dynamically. For example, errors in the Type field or the absence of the Type field make it difficult to identify the object type.

Section 5.6

Evaluation on arXiv dataset

The Kaggle team has made a dataset of arXiv documents available [270]. This dataset is composed of over 1.7 million PDF files available on arXiv. Although this data was made available so researchers can perform machine learning research on such a vast corpus, we leverage this dataset to understand differentials in PDF creation tools. We used all papers uploaded to arXiv from 2022 January to March, which amounted to over 40000 unaltered files.

When we analyzed the type errors we found in these datasets, we found some

Dataset	Number of files	Type Errors
January 2022	4379	11
February 2022	16336	35
March 2022	19105	46

Table 5.5: Summary of our evaluation on the arXiv dataset

trends. The arXiv dataset predominantly consisted of files generated using various latex tools. We found these errors in files submitted to arXiv in 2022—making it highly likely that such error-introducing behavior is present in the latest releases of some of the latex to PDF conversion tools. Table 5.5 summarizes our datasets and the number of type errors we found in each of our dataset when we applied our version-specific type checker.

In the rest of this section, we study the type errors we found in detail.

- **Incorrect PageMode:** We found several instances of this issue in our analysis of other datasets as well. The PageMode key includes an allowlist and values corresponding to this key must follow this rule. In our arXiv evaluation, we found that there are files containing incorrect values, such as `None` instead of `UseNone`. This issue was found in files generated by TeX Live 2020 and Dvips used with GPL Ghostscript 9.22 (2017). Following is a list of all the incorrect values we found corresponding to the PageMode key in these datasets: `None`, `noneh`, `Fit`, `Fullscreen`, `none`, `UserNone`, `empty`, and `TwoColumnsRight`.

- **Incorrect Data Syntax:** PDF files store dates using a specific syntax. This syntax handles the data, time, and timezone. We found that PDFs generated by the WPS office, the word processing software, had incorrect syntax for date strings that store the time the file was last modified.

- **Missing objects and streams:** The PDF specification uses several indirect references to improve readability and spread objects across a PDF file. Using a

lazy reading technique, we only read objects currently in use instead of loading the entire PDF file in memory. However, in our evaluation of the arXiv dataset, we found several instances where there were references to missing objects.

– **Type key missing in FontDescriptor dictionary** Like most dictionary types in the PDF specification, the Font Descriptors must include a Type key. Most font types require font descriptors to describe the font in further detail, such as the font weights, family, and the name if it is a base font (The PDF specification defined 14 standard base fonts). We found that many files generated by PDFium did not include the Type key for Font Descriptor dictionaries. This makes identifying the dictionary type difficult since most other keys in the dictionary are not mandatory.

Section 5.7

PDFFixer: Transforming PDF files using a Format-Aware Reducer

In previous sections, we understood how the PDF syntax is defined (Section 5.2), how we can generate code from the machine-readable specification (Section 5.4), and how PDFs today contain numerous errors (Section 5.5). This section explores ways to fix common type errors in PDF files.

There is a need to have an organized way of applying fine-grained rewriting rules to *parsed* input, particularly for data formats. For example, software engineers that deal with a complex file format like PDF need a toolchain to enable them to rewrite an arbitrary, complex PDF file to a notional "PDF/Safe" dialect by removing or rewriting slight malformations or by substituting risky instances of particular format features with an equivalent representation that carries less risk. Engineers who deal

in data languages often lack a principled framework for applying transformations to instances of that data in a safe and systematic way [115, 43].

Current data rewriting systems in file and network security are usually based on regular expression search-and-replace, such as URL rewriting schemes when they occur in email. This technique "works" because typical URLs do not have deep, nested structure.[4] This grep-like approach will not work for complex structured input. Moreover, the post-parsed input is likely contained in a tree-like or similar data structure, thereby losing the ease of applying file and line-oriented Unix-style string search-and-replace tools. Such tools and tracers lack format awareness: they do not "see" the meta-language structure on top of the symbols built to scan.

One way of visualizing this problem is to apply a regular expression-aware trigger-action programming pattern. Xpath [62], M4 [146], and Gremlin [232] are standard tools used to achieve this task. However, we would need to build tools to operate on complex graph or tree structures instead of text or XML data in the more generic case.

This section argues for the need for format-aware tools to transform, sanitize, and serialize data. There are several reasons to design such tools. First, not all data formats are similar. For example, the transformations we must apply to a PDF file differ significantly from DNS network protocol message transformations. File formats often contain offsets—data spread throughout the file, and the locations and size of this data are maintained in a table. Designing an intermediate representation common across various data formats is also challenging.

Second, we need tools that can generate files from an intermediate representation. This operation is called unparsing or serializing [231]. However, with the complex

[4] Although they can be surprisingly complex, as evidenced in M. Zawelski's Browser Security Handbook [299], due to the various legal representations of IPv4 and IPv6 addresses along with the permitted encoding schemes for GET and POST parameters.

structures in data formats, programmers need to design their serialization tools for a particular format to convert the internal representation to bytes.

Finally, the PDF specification has evolved a lot over the last three decades, resulting in several ad hoc implementations that create PDF files that violate the specifications in subtle ways. We investigated commonly occurring errors in PDF files and found that a substantial number of errors require sophisticated methods to fix these errors. For example, in a dataset of 1 million PDF files, we found that over 3000 of them contained page tree node malformations of some sort—meaning keys mandatory in the page tree nodes were missing. As a result, these otherwise benign files do not open on several PDF readers.

Additionally, Muller et al. [197] demonstrated that PDF files contain metadata, personally identifiable information, and redacted data. For example, they found that over 58% of PDF files in their sample included author information. Furthermore, PDF tools also include the creation time and the name and version of the tool used to create the file. An adversary can use this information about the victim's PDF tools to craft exploits.

Although the most common application for data transformation tools is fixing minor malformations in data, there could be other use cases. For example, we can transform valid files to violate format specifications. We can then use these files to test parsers for these formats. We can also use a parser and serializer to redact data transiting network filters. We can achieve this by applying transformations to the parsed data structures and then serializing the parsed structure back to a network packet.

To this end, we present the Parsley normalization tools that address the above challenges using several approaches. First, we introduce the Parsley Intermediate Representation (Parsley IR), a representation to capture the syntax of PDF objects in

use. Developers can use this IR to debug PDF files or use it with other normalization tools that are a part of the Parsley toolchain.

Second, we develop the PDFFixer, which comprises the Parsley Reducer API to allow developers to specify various transforms on the PDF structure. A *reducer* is a tool that can dynamically execute transformation functions over data. These transformation functions are defined as a set of reducer rules. Then, a reducer applies these rules to transform data. This section proposes a reducer API targeted at the PDF format. The API allows a developer to specify selection, filtering, and replacement rules.

Our Reducer API provides a customizable, scriptable rewriting system for the PDF format. This design pattern is similar to the pattern used by Pin [173] and Valgrind [201] and dyninst and DynamoRio [34] for x86 code. These tools use this pattern for other purposes, such as performance profiling, static and dynamic memory and cache analysis, and bug finding. They dynamically inject x86 code to monitor the program's performance. In contrast, we invoke our Reducer API to apply transformations to the PDF structure.

Third, we build a serializer to convert Parsley IR to a transformed PDF file. *Serialization* is the process where an internal representation of data is converted to a set of bytes that can be stored on disk or sent over the network [231, 285]. In the case of a file format, it converts an internal representation to a file format. For example, we can convert a JSON representation of a PDF file to a PDF file. Similarly, in the case of network protocols, a machine beginning to transmit a message will create a structure with all the fields locally. Subsequently, these fields are converted to bytes in a sequence predefined by the serializer. Our serializer for the PDF format does not preserve the object order but retains all the in-use objects while computing the correct offsets and generating syntactically valid PDFs. We automatically remove

objects not used.

This section makes the following contributions:

- We created a radically new capability we call the Parsley Reducer API, a novel tool to allow developers to programmatically redact or modify portions of PDF files. This concept uses a design pattern similar to Pin and DynamoRio, allowing scriptable dynamic data operations on PDF files. For example, this new capability provides something akin to a graceful dynamic exception handling mechanism that is external to the codebase it is protecting, similar to seminal work in matching exploit signatures in network stacks [280].

- Our research over a large corpus (1 million files) of real-world documents, led to the discovery and characterization of a number of malforms present in PDF files that are amenable to safe rewriting. We use these files to create a set of case studies that successfully demonstrates that this tooling repairs malforms and other conditions of concern in real PDF documents.

- We conducted experiments that explore several dimensions of comparison between existing PDF transformation tools and our Reducer. We discover that these existing PDF (normalization) transformation tools lack any ability to dynamically script or react to malforms present in their input and are unsound (i.e., they introduce errors during their rewriting).

- We provide conceptual contribution: there are many divergent unlabeled dialects of PDF. Our work shows and provides tooling so that one can begin to merge unlabeled dialects into labeled, normalized representations, where the overall organization is provided by the set of transformations defined and executed by the Reducer. In other words, we can begin to formulate dialects that are labeled by the set of transformations applied and/or malforms rectified.

The benefit of this is to reduce poor outcomes like parser differentials and other format confusion.

Definitions

The Parsley normalization tools are a part of the larger Parsley system. The Parsley system first validates PDF files to ensure they are syntactically valid. As a result, we can successfully serialize well-formed files or those that we can transform into well-formed files using our reducer.

Definition 5.1 (Well-formed PDF file)**.** A PDF file is well-formed if the Parsley syntax checker decides that the overall syntax of the file is valid as per the PDF 2.0 specification.

Since we transform a PDF file, we want to ensure that the text in the original PDF is fully preserved. In addition, since we have fixed particular objects by applying transformations, pages the text extraction tools could not previously access will now be accessible. Hence, we hypothesize that a safe normalization tool must produce the same amount of text or more text than the original file. Additionally, the transformed file must also be syntactically valid.

Definition 5.2 (Safe Normalization)**.** The original PDF file $P1$ is transformed to produce a new PDF file $P2$. This transformation is safe if:

- $P1$ and $P2$ are well-formed PDF files, and
- The text extraction output of $P2$ is identical or a superset of the text extraction output of $P1$ across multiple tools.

We describe six case studies we constructed to demonstrate the utility of a format-aware normalization tool for the PDF format. The normalization tool uses a reducer and serializer in conjunction to produce transformed PDF files.

Claim. The Parsley reducer and serializer produce safe normalizations.

We back this claim up by evaluating our normalization approach against two datasets, each over 6000 PDF files. Furthermore, the reducer rules we evaluated were designed to fix specific malformations commonly seen in PDF files but not commonly fixed by PDF rewriting tools.

5.7.1. Background

Running Example

In the rest of the section, especially Section 5.7.3, we will use the following running example to discuss transformations to PDF files. This example includes two PDF objects that are surrounded by the `obj` and `endobj` tags. The first object in the snippet takes ID 1, whereas the second one takes ID 0. Both the objects have the generation number 0.

```
1 0 obj
<</Names <</Dests 4 0 R>>
/Outlines 5 0 R /Pages 2 0 R
/Lang (en-US)
/Type /Catalog
/PageLayout /OneColumn
/PageMode /UseOutlines>>
endobj

2 0 obj
<</Count 13
/Kids [6 0 R 25 0 R 33 0 R
35 0 R 38 0 R 45 0 R 49 0 R
57 0 R 60 0 R 63 0 R 66 0 R
68 0 R 70 0 R]
/Type /Pages>>
endobj
```

Catalog In the above example, object 1 is a Catalog dictionary. Dictionaries, in PDFs, are surrounded by << and >> symbols. They hold key-value pairs that are separated by whitespaces. In this example, the keys are Names, Outlines, Lang, Type, PageLayout, and PageMode. There are several constraints on a Catalog dictionary to ensure cross-compatibility: a file rendered on various PDF readers or text extraction tools must produce nearly identical results.

The Type and Pages keys are the only mandatory keys in a Catalog. Additionally, each key has a type associated with it. For example, the Lang key must have a corresponding string, and the PageLayout and PageMode keys must have a corresponding Name object chosen from a list of options. The Names key has a corresponding dictionary object in this example. The Outlines and Pages keys must always include indirect references to a dictionary that is stored in another object. The Outlines key refers to object number 5, while the Pages key refers to object number 2.

Page Tree Node Similarly, Page Tree Nodes also include required and optional keys. The tree's root node must not include a Parent key but include Kids, Type, and Count keys. If any of these keys are absent, the Page Tree Node is invalid, and it can get challenging to traverse the tree. Other non-root nodes in the tree must include a Parent key.

Like in the case of the Catalog dictionary, the specification assigns types for these keys. For example, the Count key must have an associated positive integer, and the Kids key has an array of references to other Page Tree nodes or pages.

Meriadoc Recognizer

In this work, we rely on the Meriadoc Recognizer to test a PDF file against a set of PDF parsers [259]. The recognizer instruments PDF parsers such as Mupdf, Mu-

tool [23], Qpdf [33], PDFMiner [246], and Poppler [205], to generate meaningful errors that help debug malformations.

It applies techniques such as parser watchpoints—monitoring parsers in runtime to find out which piece of code leads to errors. These parser watchpoints are format-aware tracers—explicitly designed to handle PDF files. These format-aware tools use a higher-level representation of the parsing events. With this representation, we can provide meaningful, explainable errors as output.

There is a strong coupling between the Meriadoc Recognizer and the Parsley Reducer, as we discuss in Section 5.7.2. The Recognizer provides feedback to pinpoint where other PDF parsers reject a file Parsley generates. With this insight on the error types, we revise future versions of our serializer to avoid buggy files. We also used this Recognizer to compare a file transformed by other PDF repairing or cleaning tools (such as Caradoc and Mutool clean) in Section 5.7.5.

5.7.2. Parsley Normalization Tools

There are two possible architectures we envision for the normalization tool. First, we can use the reducer as an additional pass to the PDF checker. This way, before files are type-checked, we fix the objects based on the set of reducer rules. Second, the reducer can be used in conjunction with the serializer. In this approach, we focus on making fixes in the intermediate representation and provide a transformed file.

Background

The Parsley PDF parser applies several checks to ensure that a PDF file is well-formed. We built the individual components of the parser to follow the PDF 2.0 specification. Figure 5.2 shows the steps followed by our PDF parser.

First, we syntactically validate all objects in a PDF file by following the objects

from the xref table. This step ensures that the dictionaries, streams, arrays, and other predefined-typed objects are well-formed. Next, the syntax checker extracts an intermediate representation that provides the following structures.

- The object numbers and the corresponding objects in use
- The root (Catalog) and info dictionaries
- PDF file version (extracted from the PDF header and the Catalog dictionary)

In the second step, we type-check the various objects starting with the Catalog dictionary and Info dictionary. We define specifications for each dictionary, array, and stream object to apply the correct types. For example, in arrays, we define the array size and the object type at each location in the array. Similarly, for dictionaries and stream objects, we define the keys that must be present and their corresponding types. Then, iteratively, we type-check all the objects referenced in these dictionaries based on type specifications. Finally, we discuss several reducer rules in the case study based on the errors we identified using our type checker.

In the third step in the Parsley PDF checker, explore content streams—descriptions of the page's contents. These content streams include text, shapes and drawings, and images. We validate content streams to ensure that the text and the metadata are well-formed. We also provide the functionality to extract this text.

In summary, the Parsley PDF checker performs three levels of validation to ensure that files are well-formed. Our second and third passes generate a binary output by checking the internal representation. Therefore, our normalization tools operate on the internal representation generated after the first pass. In definition 5.1, we consider a file well-formed if it passes the first step of validation in Figure 5.2.

This work focuses primarily on the Parsley normalization tools that we use in conjunction with the rest of the Parsley PDF checker. Figure 5.8 demonstrates the

overall architecture of the Parsley normalization tools. We designed the Parsley normalization tools (the reducer and serializer used in conjunction) with several goals in mind:

- **Usability: Provide an API to support scriptable editing**

  There are several reasons to edit PDF files. We want to design an API that provides flexibility to identify objects of interest and edit them.

- **Composable Design: No shotgun serializers**

  We must not place the burden of serializing the internal representation on the developer. We envision developers would use our serializer as a black box. Additionally, we also want to avoid the anti-pattern of a shotgun serializer—when a code performs validation or transformations while also serializing portions of the internal representation.

- **Soundness: Produce syntactically valid PDF files**

  Our serializer must produce syntactically valid PDFs that pass the first syntax check the Parsley system makes. However, these files may not pass the other two validation checks we place since PDF files may already contain type or content stream malformations in them. Additionally, developers can also introduce malformations using the reducer API—such as removing mandatory keys from dictionaries or deleting objects.

Parsley Reducer

The Parsley PDF parser and our normalization tools are implemented in Rust and require the compiler version 1.55.0 or higher.

We require that the developer specify a selector hash map, use one of the filtering functions we have built-in as a part of the reducer API, and then describe the type

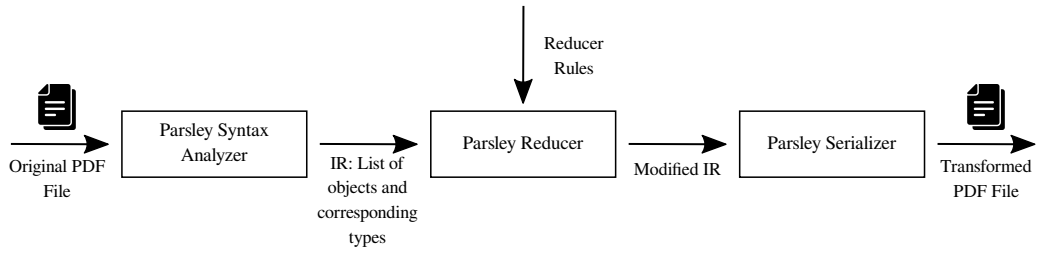


Figure 5.8: Changing the architecture of Parsley to support PDF rule-based transformations.

we must replace the filtered keys with.

Parsley IR

The Parsley Intermediate Representation (IR) is made available as an output of the Parsley syntax checker. This IR comprises of four items: (1) a hash map containing a mapping of PDF object IDs and their corresponding type-annotated objects, (2) object ID of the root object, (3) object ID of the Info dictionary, and (4) version number of the PDF file extracted from the header and the version string in the Catalog dictionary.

This IR does not store information about the exact locations of the various objects in the original file. Since PDF objects can be updated, traditionally these older, unused objects are still stored in a PDF file. However, our IR does not represent these unused objects—only storing the used objects.

Code snippet 10 demonstrates the Parsley IR syntax for our running example. Essentially, it is a hash map of PDF object IDs and generation numbers and the object contents.

```
(1,0): {
 Names: PDFType:Dict(map: {
        Dests: PDFType:Ref((4,0))
    }),
 Outlines: PDFType:Ref((5,0)),
 Pages: PDFType:Ref((2,0)),
 Lang: PDFType:String(val: "en-US"),
 Type: PDFType:Name(Catalog),
 PageLayout: PDFType:Name(OneColumn),
 PageMode: PDFType:Name(UseOutlines),
}

(2,0): {
 Count: PDFType:Integer(13),
 Kids: [PDFType:Ref((6,0)),
    PDFType:Ref((25,0)),...],
 Type: PDFType:Name(Pages)
}
```

Code Snippet 10: Parsley IR example of our running example

Selector

The selectors we use as input to the reducer take in a hash map with the list of keys and the corresponding values for selecting an object. For example, if we need to apply transformations to the Page Tree Node objects where the objects have the type key "Pages," we create a hash map with the key "Type" and value "Pages."

The Selector traverses the Parsley IR hash map to find the appropriate objects. Occasionally, this could also result in traversing the dictionary structure of a certain object. For example, a selector rule to find all the Page Tree Nodes would need to find the Type key of every object to find if it is a Page Tree Node. The selected objects are then passed on to the filtering rules.

Filtering rules

Once the intermediate representations of selected objects are available, we can find specific structures of interest. Our filtering mechanism can extract specific objects based on dictionary keys or array indices. For example, we may want to filter out objects missing a particular key or inspect the type of the value corresponding to a key. Our reducer API supports several built-in functions to find objects within an outer object.

We list a few of the filtering rules we provide as a part of the reducer API.

- **Delete a key from a dictionary:** This rule is useful when some data needs to be redacted: such as personally identifiable data in the file metadata.
- **Select particular keys from dictionary or array**: This rule is useful when we need to replace specific keys or array locations. The transformed type is defined separately in the transformer.
- **Select version string**: The version string is specified in the file header and the Catalog dictionary. Hence, a modification to the version string needs to reflect in both locations. Hence, we provide a separate filtering rule and a corresponding transformer function to aid in this transformation.
- **Select keys in the Catalog/Info dictionary**: These dictionaries are special and are the root nodes. Hence, we provide a separate way to get access to keys in these nodes instead of traversing through the intermediate representation.

Transformer

We provide new values to the filtered keys or array locations using the transformer functions. The function follows the syntax where we provide a filtered object O and

the object to replace O, O'. Then, the transformer applies these rules to each filtered object individually. Finally, we return the modified IR with the correct transformations applied.

Finding well-formed PDFs

Parsley IR is produced by the first step of the Parsley PDF parser—the syntax checker. Hence, to ensure that our Reducer rules can be applied, we must ensure that the file is syntactically well-formed. To find well-formed PDF files in our dataset, we separated the first pass of the PDF parser and ran it over all files in the dataset. Therefore, we only operate on this subset of well-formed files from our datasets in the rest of this section.

Parsley Serializer

We built a serializer for the PDF format, starting with the Parsley IR. The files we generate must be syntactically valid PDFs to comply with the soundness property we set for our normalization tools. [5] We implemented our serializer in less than 200 lines of code in Rust.

We follow a recursive structure to serialize each PDF object. First, we traverse the Parsley IR, which contains a hash map of all object IDs to their corresponding objects. Then, we identify the type of each of these objects and then call a recursive function to serialize the internal objects within the object. For example, dictionaries can be nested. Dictionaries may also contain other arrays and stream objects.

Name objects in PDF can contain special characters. To handle these special characters, we included an additional check to ensure that we escaped these special characters. Similarly, String objects may also contain special characters that are

[5] PDF parsers may still reject our modified PDF files for type malformations originally present or introduced as a part of our reducer rules. We only ensure that the syntax is correct.

escaped using a certain syntax. We escaped all characters in PDF strings to avoid further confusion. However, this approach makes debugging hard, and the modified files significantly diverge from the original files.

For Stream objects, we kept the dictionary and the compressed portions intact—not redacting or decompressing any objects to continue to preserve space. Finally, we remove all comments from the file by not serializing any comment objects. We do this because comments, metadata in the info dictionary, and unused objects collectively contain much unredacted information.

In summary, our serializer produces files compliant with the PDF 2.0 syntax. Furthermore, we recursively implemented the Parsley serializer since PDF objects contain complex types such as streams, dictionaries, and arrays containing other complex or basic types such as boolean, strings, numbers, and name types. We evaluate our serializer in Section 5.7.5 to ensure that the files we generate are syntactically valid by running our generated files against a host of PDF parsers.

Using Meriadoc Recognizer as a feedback loop

We found several bugs in our serialization approach after running our generated files against the set of parsers in the Meriadoc recognizer toolset. We list some of the findings and then discuss how we fixed these issues.

Incorrect understanding of the Size field in the cross-reference table The specification says this when describing the Size field in the PDF file trailer dictionary, "this value shall be 1 greater than the highest object number defined in the file." Unfortunately, in the first version of our serializer, we wrongly computed these values as the total number of entries in the Xref table.

Qpdf and Mutool parsers that are a part of the Meriadoc recognizer spotted this

malformation and provided warnings specifying that the file may be malformed. We subsequently fixed our serializer to generate these values correctly.

Special characters in the Name and String objects need to be escaped The Name type in PDF contains strings that start with a slash (`/`). These name types are extensively used in dictionary keys. To use characters that are not alphanumeric, we can represent these values using hexadecimal. For example, the ASCII "`+`" character would be denoted as the hexadecimal string "`#2B`." Similarly, names can also include whitespace characters escaped using this syntax. The specification uses the following example: `Lime#20Green` must be interpreted as `Lime Green`. Our serializer did not implement this hexadecimal encoding for these special characters producing the string `/Lime Green` in PDF files. This led to several parser errors since the string `Green` here does not conform to any PDF type.

Likewise, PDF string objects use octal or hexadecimal encoding to store special characters. Any parenthesis used in a string must be balanced and require no special treatment. However, any special character must be escaped using an octal syntax `\ddd`—where `ddd` is the octal character code.

In an earlier implementation of the serializer, we wrote the internal string representation to bytes in the PDF objects. Unfortunately, similar to the case with name objects, special characters in strings were serialized as is. This led to several errors due to the incorrect handling of numerous special characters in these PDF objects. We have subsequently patched this issue by representing all string characters in octal encoding.

5.7.3. Case Study

Given the age of the PDF formats, several PDF creation tools have deviated from the specification in various ways. These deviations expose parser differentials in

PDF readers. In this section, we study several PDF specification violations and data redaction cases and how we added reducer rules to handle these patterns and anti-patterns.

Finding type check errors in PDF datasets

Table 5.6: PDF Type check errors in PDFs

PDF Errors	**Occurrences in 1 million files**
Incorrect Lang key	1k
Page Tree Node error	20k
Incorrect PageMode keys	500
Null references	5k

We ran the Parsley PDF type checker—closely designed for the PDF version 2.0 on 1 million PDF files collected via CommonCrawl and GovDocs digital corpora. We logged errors we encountered on files in this dataset and clustered them based on the type of error. Table 5.6 provides a summary of all the errors we found and their frequency in our dataset.

As a result of our study, we've designed case studies C1-4 to serve as a way to correct these commonly seen errors in PDFs. Subsequently, C5, C6, and A1 are other use cases such as redacting data or generating files that contain errors for better testing.

C1: Incorrect `Lang` key syntax

The language key in the Catalog dictionary aids text extraction. Some PDF creation tools use incorrect syntax to describe the language. For example, several files use the following Catalog dictionary syntax.

This dictionary differs from the running example in only the `/Lang` key. The value `/en` uses the name type in PDF. However, the specification notes that this field must

```
<</Names <</Dests 4 0 R>>
/Outlines 5 0 R /Pages 2 0 R
/Lang /en
/Type /Catalog
/PageLayout /OneColumn
/PageMode /UseOutlines>>
```

be a string—not specifying what readers must do if this field is incorrect.

The specification does not offer a default value for this key. Instead, it specifies that if this key is absent, the language is considered to be unknown. Therefore, we remove this key from the Catalog dictionary in an attempt to remove this common anti-pattern from PDF files.

Selector	**Filtering rule**	**Transformer**
Catalog	Delete: `Lang`	-

The above table shows our reducer rules to remove the `Lang` key since the specification does not provide any default values for this key. Therefore, we specify that the filtering rule we apply is a delete operation and specify the key that must be deleted.

C2: Missing keys in Page tree nodes

Page tree nodes in PDF files must contain the `Kids`, `Type`, and `Count` keys. However, we found several PDF files in the wild with some of these keys missing. When PDF viewers attempt to walk the page trees in documents with these malformations, they encounter the page tree nodes with either the `Count` or `Kids` key missing—and often fail to render the PDF file. The count key holds an integer value. The kids key stores an array of references to other page tree nodes or pages. For example, following is an example of a malformed Page Tree node—differing in some ways from our running example.

In the above example, the Count key is missing. Other variations of this error

```
<<
/Kids [6 0 R 25 0 R 33 0 R
35 0 R 38 0 R 45 0 R 49 0 R
57 0 R 60 0 R 63 0 R 66 0 R
68 0 R 70 0 R]
/Type /Pages>>
>>
```

could be a missing Kids key or both Kids and Count keys missing.

We transform these malformed page tree node dictionaries using our reducer syntax. We select dictionary objects where the `Type` key is set to `Pages`. Then, we check if the mandatory keys are present. We consider three cases:

- Both the keys are missing: we create an empty array for the kids key and set the count key to 0.
- Only the count key is missing: we count the number of pages in the kids array. We then set the correct count value. The count value may also be incorrect in the previous version of the file.
- Only the kids key is missing: we create an entry for the kids key in the dictionary. We then reset the count key to 0.

Selector	**Filtering rule**	**Transformer**
`Type= Pages`	Select if `Kids` missing	Set `Kids` to `[]` and `Count` to 0
`Type= Pages`	Select if `Count` missing	Set `Count` to `Kids.length()`

The above reducer rules account for all three cases we listed. The first rule we list accounts for both cases—irrespective of whether the Count value is present or not.

C3: Incorrect `PageMode` keys

The Catalog dictionary also contains the `PageMode` key. The PDF 2.0 specification provides a list of allowed values for this key. This field can only take the following values: UseNone, UseOutlines, UseThumbs, FullScreen, UseOC, and UseAttachments. Additionally, the specification also specifies the default value as UseNone.

```
<<</Names <<</Dests 4 0 R>>
/Outlines 5 0 R /Pages 2 0 R
/Lang (en-US)
/Type /Catalog
/PageLayout /OneColumn
/PageMode /None>>
```

We found hundreds of PDF files with incorrect page mode values not in this list. The code snippet above demonstrates such a value. So we wrote a reducer rule to rewrite the Catalog dictionary with the page mode value reverted to the default value. The serialized PDF does not contain this malformation and can produce reliable viewer output—since different PDF readers may handle invalid values differently.

Selector	**Filtering rule**	**Transformer**
Catalog	Select: `PageMode` if not in list	Replace with: `UseNone`

In our rule, we select the Catalog object and select the `PageMode` key in the Catalog. Since this key is optional, it may not be present—in which case this rule is not applied. We use an allowlist of the values this key in the Catalog can take. If the value for this key is not in the allowlist, we replace it with the default value.

C4: Removing Null References

Let us consider the second object in the running example in Section 5.7.1. In this example, The Dests, Outlines, and Pages key all have an indirect reference to another object stored in the same PDF file. However, as we demonstrated in Table 5.6, there were several cases where these indirectly-referenced objects were missing from the file.

Most PDF readers ignore these keys if they are not mandatory keys. They arrive at this interpretation following clause 7.3.10 in the PDF 2.0 specification. First, any undefined object is considered to be a Null object. Furthermore, in a dictionary, if a key is associated with the Null type, we must treat the dictionary as if this key was absent.

However, this is not easy to implement in a type checker. Although in the running example, three keys are indirect references, any key can hold an indirect reference unless otherwise specified in the specification. For example, the Lang key could be in its object with the following syntax.

```
25 0 obj
(en-US)
endobj
```

To capture clause 7.3.10, we must treat every key as a disjunction between a Null object and the correct type definition. We must also include a flag to ensure that the keys that do not need such a disjunction are skipped.

Instead, we rely on creating reducer rules to remove any Null references in a PDF dictionary. We iterate over every object in the Parsley IR and over every key in the dictionary to find Null references. These objects are then modified to remove the key with the Null reference. A drawback of this reducer rule is that we may inadvertently remove a mandatory key. For example, suppose the Pages key in the running example holds a reference that does not exist. In that case, our reducer rule may remove this

key from the dictionary—creating a type check violation since the key is mandatory.

C5: Change PDF version

PDF files contain a version string in the header and a field in the Catalog dictionary. However, if both of these version strings are not identical, the higher of the two values takes precedence. If a file's version needs to be updated, either of these two values can be updated to a higher version.

We consider the cases of downgrading and upgrading the version number of a PDF file. For example, we found that the Caradoc PDF parser [87] and PDFCPU [120] do not accept files that are higher than version 1.7. Therefore, to run a PDF file through both these parsers, we must downgrade to version 1.7 from 2.0.

We provide two reducer functions—both take a list of old version numbers and a new version number. We rewrite both the version string in the Catalog and the string in the header in both cases. We did this to ensure there are no parser differentials if specific parsers only read one or the other. We demonstrate our reducer rules below.

Selector	**Filtering rule**	**Transformer**
Version	Select if not `2.0`	Replace with: `2.0`
Catalog	Select `Version` if not `2.0`	Replace with: `2.0`

C6: Change values in the Info Dictionary

The Info dictionary in a PDF file contains nine optional fields. It contains metadata such as the title, author, creation tool, producer, and creation and modification date. Some PDF creators and conversion tools may add metadata about a user, such as their name and username, and email ID [197, 25].

```
<< /CreationDate (D:20200407135742Z)
/Creator (John Doe; TeX)
/ModDate (D:20200407135742Z)
/PTEX.Fullbanner (This is pdfTeX,
Version 3.14159265-2.6-1.40.20
(TeX Live 2019) kpathsea
version 6.3.1)
/Producer (pdfTeX-1.40.20)
/Trapped /False >>
```

The above code snippet demonstrates the contents of the Info dictionary for a file generated from Latex. We see that it can disclose software versions and creator information in this dictionary.

Hence, we need mechanisms to redact these keys from the Info dictionary to ensure that such information is not exposed when publishing PDF files. Removing these keys from the dictionary does not render a file invalid since all of these keys are invalid.

We identify that the author, creator, producer, and creation date keys can be particularly problematic in exposing details about the origin of a file. Hence, we design rules to redact these crucial keys to ensure better privacy. Unfortunately, since the Info dictionary does not have any mandatory keys, we could not use a selector rule to find it in the IR. Instead, we rely on the trailer of a PDF file to tell us which PDF object contains the Info dictionary. Below, we demonstrate the reducer rules to change the Author and Creator keys in the Info dictionary.

Selector	**Filtering rule**	**Transformer**
Info	Select `Author` and `Creator`	Replace with: `Manul` and `Bobcat`

A1: Adversarial file generation

We propose another application of the Parsley Reducer API—generating files with deliberate errors to understand if parsers can detect anti-patterns and clear specification violations. For example, we generated a test case to inject an incorrect object ID in the Kids array of every Page Tree Node dictionary in a PDF file.

Since PDF parsers often ignore missing objects, we did not randomly inject a nonexistent object ID. Instead, we append the Catalog or Root dictionary ID to the Kids array. Unfortunately, this action leads to malformed PDFs in three ways.

First, the Kids field in a Page Tree Node must point at other Page Tree Nodes or a Page object—not a Catalog dictionary. Any parser that checks the well-formedness of this tree structure must find this error in the syntax of the tree.

Second, a naive PDF parser that recursively walks the Page Tree structure can enter an infinite loop. As a result, we could trigger a denial of service attack against PDF parsers that do not keep a list of objects already checked.

Finally, a naive text extraction tool can run into an infinite loop in such an infinite tree structure and generate the same text repeatedly. This way, the extracted text would differ from the text extracted from other tools—leading to text-extraction differentials.

Next, we randomly selected 4200 PDF files from our dataset to evaluate our adversarial file generation use case. Finally, we ran each generated file through Qpdf and Mutool clean to see if these tools found our injected malformation.

Table 5.7: Testing files we generated with adversarial rules against PDF parsers. Dataset included 4200 PDF files randomly selected.

Cases	**Mutool**	**Qpdf**
Files Rejected previous accepted	3177	221
Files Timed out, previously ran correctly	18	1

As shown in Table 5.7, we were able to force qpdf to run over the timeout value (one minute) for one file when the original (unmodified) file runs efficiently within the timeout. The original file ran in 50 seconds, while the modified file did not terminate after running over three minutes in multiple tests. Similarly, we found 18 modified files that caused an infinite loop in the Mutool clean command. Each of these files ran in less than 10 seconds in the unmodified case.

Upon more investigation, we found that this issue was fixed in the latest Mutool release 1.19, following a report in 2020 [206]. However, the Mutool shipping with the package managers on most operating systems is version 1.16. Therefore, all versions of Mutool before 1.19 are vulnerable to a denial of service attack, where a PDF file with a wellformed content stream can cause an infinite loop.

However, we also found that Qpdf does not flag most of the adversarial files we generated as malformed. In contrast, Mutool overwhelmingly finds them malformed with the error "non-page object in page tree (Catalog)." Qpdf provides a warning with the message "expected /Type /Pages, found something else" for the files it rejects.

Developer effort

Table 5.8 demonstrates the lines of code needed to use the Parsley normalization tools to apply transformations of varying degrees of difficulty. Using the Reducer API requires some degree of understanding of the PDF specification since we require the developers to specify the transformed type in cases where data is replaced.

We observe that the harder cases of modifying the Page Tree Nodes using cases C1 and A1 require over 100 lines of code to be added. This is because we must account for many combinations of missing keys in Page Tree Nodes, causing an increase in the lines of code. However, the cases of redacting data in objects or replacing them

require far fewer lines of code.

Table 5.8: Developer effort needed to write reducer rules

Case	Lines of code added	Complexity
C1	15	Easy
C2	150	Hard
C3	32	Easy
C4	65	Hard
C5	15	Easy
C6	55	Medium
A1	120	Hard

5.7.4. The need for a reducer

Some of the examples we discussed in our case studies were simple—especially the version editing and fixing page modes. However, C2, C4, and C5 are complex rewriting rules where simple grep- or m4-like expressions would not suffice. For example, to redact keys from PDF dictionary objects, we need to parse the dictionary as a whole and then select the keys for modifications. These dictionaries can also be recursive: dictionaries may contain other dictionaries, streams, or arrays either inline or as a reference.

5.7.5. Evaluation

We apply the reducer rules for C1, C2, and C3 as a part of the Parsley reducer since these rules made significant modifications and correct malformed constructs in PDFs. We evaluate the Parsley reducer and serializer to answer the following questions.

- Does the reducer fix bugs and malformations in PDF files?
- How does the Parsley reducer compare with other PDF cleanup tools?
- Do text extraction tools produce the *similar* output for the original and transformed PDF files?

Dataset

We used two datasets of 10,000 files each. Both these datasets contain files randomly selected from 1 million files collected from GovDocs [78] and CommonCrawl [48].[6] These 20,000 PDF files are representative of a real-world sampling—randomly selected from 1 million files.

We then filtered out the well-formed PDFs from these datasets using the methodology described in Section 5.7.2. As a result, we found that Dataset 1 contained 6753 well-formed files, whereas Dataset 2 comprised 7171 well-formed files. We used the files from both datasets to evaluate our Parsley reducer and serializer methodology.

Parsley Reducer fixups

This section examines the fixes and changes made by our Parsley reducer. We closely observe files in each of the categories specified in Section 5.7.3. Unfortunately, most popular PDF readers cannot render PDF files suffering from C2.

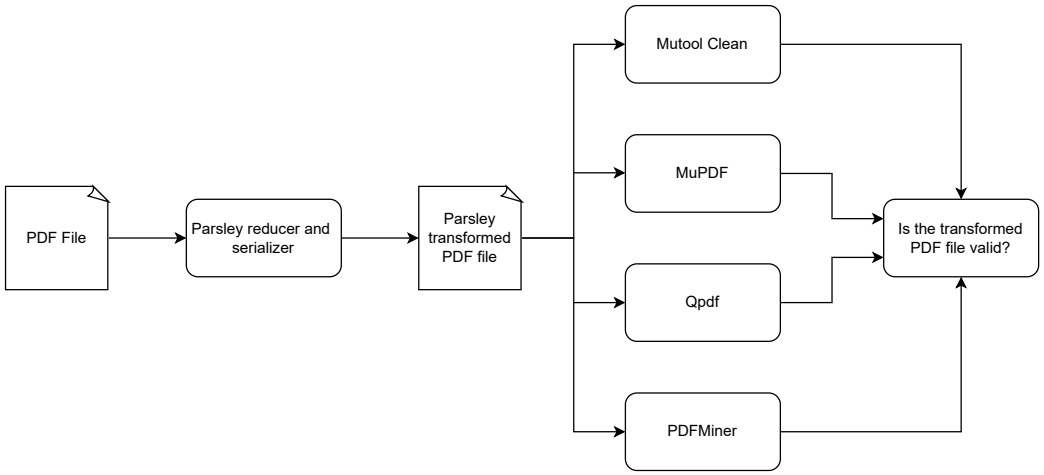


Figure 5.9: We attempt opening an original PDF file and the transformed file using three PDF tools. We log cases where previously malformed files now open using the various readers.

[6]These datasets were collated by Dan Becker of Kudu Dynamics for the DARPA SAFEDOCS project.

Figure 5.9 outlines our approach to examining the fixes made by the Parsley normalization tools. We rely on the Meriadoc recognizer to compare the parser output of original PDF files and Parsley modified PDF files. After running these PDF files through the Parsley normalization tools, we tried to open the modified files using Qpdf [33], PDFMiner [246], and Mutool [23].

Table 5.9: Comparing the output of various parsers on original PDF files and the Parsley generated files.

Evaluation tool	**Fixups after transformation**		**Errors after transformation**	
	Dataset 1	**Dataset 2**	**Dataset 1**	**Dataset 2**
MuPDF	48	31	0	2
Mutool clean	726	96	0	0
Qpdf	5	16	1	2
PDFMiner	217	401	11	25

Table 5.9 shows the results of our evaluation. We found that when we ran Mutool clean on these files, many files in both datasets were fixed. Similarly, PDFMiner also fixes a number of files in our datasets with malformations—especially ones following the pattern we described in Case Study C2.

However, we also observe that PDFMiner throws errors on some files that were earlier valid. PDFMiner errors were ASCII decoding or parser errors exposed in the parser. None of the other parsers threw errors on the files PDFMiner failed to parse. We will explore these parser errors in PDFMiner in more detail and suggest fixes in future work.

Does the serializer work correctly?

In this work, we set out with the goal of achieving *safe normalizations*. To test our claim, we compare the text extraction output for the original PDF file and the transformed PDF file to ensure that our Parsley serializer works correctly. We use

three PDF to text tools: Parsley's text extractor, PDFMiner's pdf2txt [246], and Ghostscript [22]. Figure 5.10 demonstrates our overall approach of comparing text output.

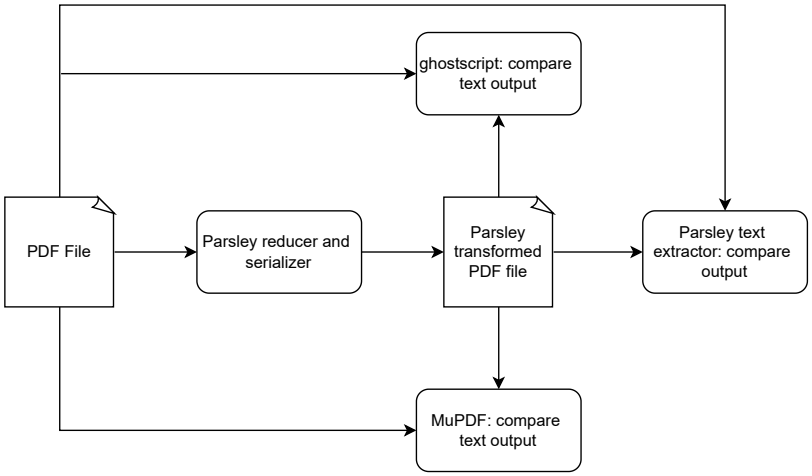


Figure 5.10: Comparing the text extraction output from the original PDF file and the transformed PDF file to ensure we did not damage the PDF files during the transformation.

Mismatches in extracted text There can be two reasons the extracted text did not match the original text. First, our fixes can provide access to new objects in the PDF file. The text extraction tool may have encountered errors, skipping these objects earlier. Second, the modified file could now contain errors we introduced. Our Parsley serializer *could* introduce certain bugs in the PDF objects.

Hence, we differentiate between these two mismatches by first comparing the extracted text from the original and modified PDF files. If there was a mismatch, we see if we extracted more text or lesser text than the original file. If we extracted more, we would flag this as a fixup. Similarly, we flag cases with lesser text extracted as errors we introduced.

Figure 5.11 and Figure 5.12 showcase the results of our text extraction comparison experiments. Since these datasets contain very dissimilar PDFs, we see a significant

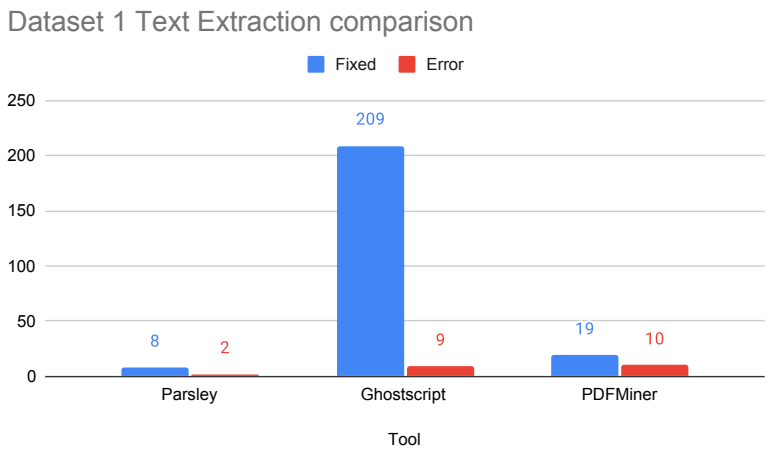


Figure 5.11: Text Extraction results from Dataset 1 (6753 files). This chart only displays mismatches between running a text extraction tool on the original file and running the same tool on the Parsley-modified file.

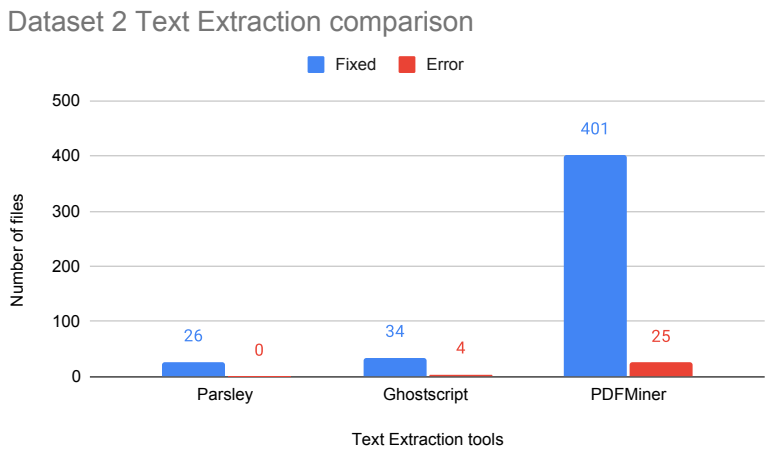


Figure 5.12: Text Extraction results from Dataset 2 (7171 files). This chart only displays mismatches between running a text extraction tool on the original file and running the same tool on the Parsley-modified file.

difference in the results. In Dataset 1, we observe that Ghostscript produces more text in over 200 files while throwing errors in 9 files. In contrast, in Dataset 2, we see that PDFMiner generates more text in over 400 modified files than the original files.

These text extraction tools use different algorithms to traverse the page tree nodes and extract text. Therefore, we employed three separate tools to observe different fixes and malformations. We demonstrate that files we've modified generate either the same amount of text or more text in almost all cases—producing safe transformations.

Why do text extraction tools generate more text. We applied case study reducer rules C1 to C4 to evaluate our normalization approach. As we discussed earlier, these rules fix malformations we found in PDF files. For example, C2 finds page tree nodes that are malformed and missing mandatory keys. However, the behavior of text extraction tools on these malformed page tree nodes is not defined. Fixing these malformed page tree nodes allows the text extraction tools to proceed further down a page tree and render text from more pages in the tree.

Comparing Parsley Reducer with `caradoc clean` and `mutool clean`

Other than using the Parsley reducer, we also ran each file in our Dataset 2 through `caradoc clean` and `mutool clean`. We then validated the transformed PDFs generated by all the three transformation tools through the Meriadoc recognizer. We designed this experiment to check whether Caradoc and Mutool can fix our identified issues.

Figure 5.13 shows our experimental approach. After transforming the PDF files using three different tools—one of which we built—we run the transformed files through the Meriadoc recognizer with the Qpdf and Mutool parsers.

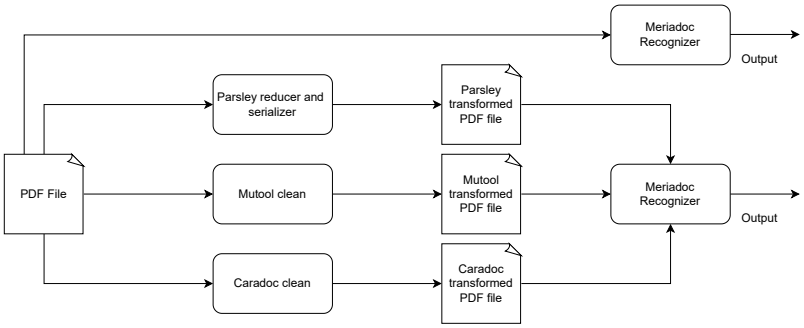


Figure 5.13: Comparing the transformed files of the Parsley Reducer to files generated by Caradoc clean and Mutool clean.

Table 5.10: Comparison of cleanup transformations applied by Parsley, Caradoc, and Mutool against Qpdf and Mutool on Dataset 2.

Serializer	**Fixups after transformation**	**Errors after transformation**
Caradoc	24	0
Mutool	24	32
Parsley	47	4

Table 5.10 presents the results of our comparison. We find that Caradoc and Mutool both fix files in our dataset. We also find that our Parsley reducer fixes more files that were previously rejected by Qpdf or Mutool than Mutool clean and Caradoc. We investigated the bugs introduced by Mutool clean (32 across Qpdf and Mutool). We found that several of these files generated by Mutool clean were empty, containing no data.

Additionally, we found that a particular type of file X_1 was initially rejected by Mutool but accepted by Qpdf. Mutool complains about several issues, ranging from a broken Xref table, content stream syntax errors, and a corrupt JPEG data segment. Here is a snippet of the error log from running Mutool on X_1.

```
$ mutool clean -s -d x_1
error: cannot recognize xref format
warning: trying to repair broken xref
```

```
warning: repairing PDF document
warning: PDF stream Length incorrect
error: syntax error in content stream
error: unknown keyword: 'e'
error: syntax error in content stream
error: syntax error in content stream
error: syntax error in content stream
error: syntax error in content stream
error: syntax error in content stream
error: syntax error in content stream
error: unknown keyword: 'r7529.751'
error: syntax error in content stream
error: syntax error in content stream
error: zlib error: invalid block type
warning: read error; treating as end of
file
error: syntax error in content stream
Corrupt JPEG data: premature end
of data segment
```

Mutool does, however, produce a modified PDF file X_1'. Mutool and Qpdf are both also unable to open the file, saying it requires a password—when Qpdf was able to process the original file—and several PDF readers could display the original file.

```
$ mutool clean -s -d x_1_prime
error: cannot authenticate password:
x_1_prime
```

Similarly, we also investigated the four files that Parsley produced that were malformed. We found that these files contained referenced objects but were not present in the file. Mutool rejected these files on these grounds.

Meriadoc recognizer feedback loop

As we discuss in Section 5.7.2, we used the Meriadoc Recognizer to find errors in our serializer and then fixed them. Table 5.11 shows that the number of errors introduced by the Parsley serializer according to both Qpdf and Mutool reduced in the second run, while the number of fixed files marginally increased. This table considers a file to be fixed if the original file was rejected and the modified version was considered valid.

Table 5.11: Demonstrating the feedback loop from Meriadoc to Parsley on Dataset 2

Run	**Qpdf**		**MuPDF**	
	Fixed	**Errors**	**Fixed**	**Errors**
1	24	15	33	101
2	26	1	35	2

Summary

In summary, we evaluated our Parsley reducer approach to ensure the following.

- First, we evaluated files generated by Parsley to ensure we do not introduce many malformations in PDF files as judged by a selection of PDF parsers.
- Next, we ran multiple text-extraction tools on our generated PDF files. We found that in a vast majority of cases, we generated more text or the same amount of text as the original file. We have an implementation that is capable of performing *safe normalization*.

– Finally, we compared the fixing capabilities of the Parsley reducer approach to popular tools such as Caradoc and Mutool clean.

5.7.6. Discussion

Reducing Parser Differentials

Since more PDFs fail to include the "Creator" and "Producer" tags in the Info dictionary, it is not easy to pinpoint the software used to create a particular file. Therefore, tools such as the PDF observatory have included a classifier to find the software used to create a given PDF based on specific structural properties of PDF files [67]. The Parsley PDF checker is strict and can pinpoint fine-grained malformations such as missing keys in dictionaries, wrong type implementations, and missing indirect references. Hence, we believe we can use the PDF observatory in conjunction with the Parsley PDF checker to design Reducer rules to remove entire classes of malformations.

In other words, different PDF tools produce different types of malformations. Therefore, we can transform the unlabeled dialects of the PDF specifications that these PDF tools implement and normalize them into a compliant dialect of the PDF format. By transforming several different malformations introduced by a variety of tools into a single, normalized form that is compliant with the PDF specification, we are ensuring that we encounter fewer parser differentials. However, Reducer rules can be used to formalize the de facto specification implemented by a PDF tool and to formalize *grammar drifts*.

Reducing Developer Effort

Our Parsley Reducer API follows a mechanism similar to a graceful "exception handling" mechanism where we transform data using customized handlers. Large classes of document malforms can be represented using these reducer rules and would not require updates to large codebases. Our methodology standardizes the methods of transforming patterns in documents.

In the future, we wish to explore generating these reducer rules dynamically. We need to use a PDF type checker to find out where specification deviations occur. Such a type checker must not stop the type checking process once it encounters an error—instead continuing to process other objects in the file. Such a process would allow the ability to find all deviations in a file rather than just the first.

We would then need to understand how to fix a particular error. For example, if an object is missing a Type key, but holds every other key, essentially allowing us to predict what the dictionary type is, then we must be able to patch the dictionary with the Type key. However, if a dictionary holds a key that must hold a value from a set of values, such as PDF versions, reverting to a default value would be a direct fix.

Errors that Caradoc and Mutool fix

We closely inspect the transformations Mutool clean and Caradoc apply in their tools. We find that the fixes by Mutool clean in our dataset fit broadly into three categories. First, we find that Mutool clean fixes truncated files without a cross-reference table or trailer. It calculates the correct offsets and inserts a valid cross-reference table and a trailer.

Second, Mutool fixes files with incorrect string encodings, causing errors. This

malformation is similar to the ones we introduced in an earlier version of the Parsley serializer that we subsequently fixed (Section 5.7.2). The Mutool clean command adds the correct octal encoding to special characters that were previously unescaped.

Finally, we observe that Mutool fixes several off-by-one errors in stream encoding and compression. Given the number of compression, image, and font libraries, such off-by-one errors causing parser differentials are fairly common in PDF files.

We found that Caradoc fixed files in which Mutool could not find objects. Caradoc expands compressed object streams to place all the objects from these compressed streams in the PDF file. We find that several errors caused by compression errors are fixed by decompressing object streams.

Comparing visual output

In our evaluation, we compared the text output of the original file and the modified files to ensure that we did not break the file in a way text extraction tools can no longer extract any text. Another possible evaluation approach is to compare the visual output of the PDF files [149, 102, 282]. There have been several attempts to compare the visual rendering of PDF files using machine learning or a pixel-by-pixel comparison. In future work, we will explore such a comparison as an additional soundness measure.

5.7.7. Related Work

This section relates our Parsley normalization approach and Meriadoc with other approaches. Caradoc [87] and ICARUS [67] are the closest tools to our approach, and they set out with similar goals. We ensure that we can support an API to allow developers to script the rewriting process.

Caradoc

Caradoc [87] is a PDF parser freely available on GitHub. Caradoc implements three passes to check PDF files. For lexical analysis and parser generation, Caradoc uses MENHIR and OCAMLLEX. They implement a strict type checker and a content stream checker for the other two passes. This way, the Parsley PDF parser follows a similar overall architecture to that of Caradoc.

Caradoc includes a *normalizer* to patch broken PDF files. In Section 5.4, we tested the efficiency of this normalizer by running files through the normalizer in an additional pass before running the PDF file through various parsers. Additionally, we also discuss the errors that the Caradoc tools fix in the generated files in Section 6.3.

Caradoc also checks a PDF file for cycles. They implement indirect references in files as a graph and look for cycles in these structures. A PDF file with cycles can lead to infinite recursion or nontermination if not handled carefully.

This work goes beyond the normalization work Caradoc does by reducing developer effort by providing a structured format with an API to apply transformations. We also demonstrate several use cases such as adversarial file creation that Caradoc does not support.

Other PDF fixing tools

Most of the prior work is focused on detecting malicious PDF files using various techniques such as machine learning [69, 204] and structural features [254, 260]. Cowger et al. presented ICARUS, a tool to extract the de-facto specification of the PDF format from a corpus [67]. They also propose converting a PDF file to a safe subset. They require users to define bi-directional lenses to convert a file from one version of the specification to another. This ICARUS approach to using lenses is the closest related

work to our Reducer API. Instead, we take the approach of allowing developers to transform objects in a file with more flexibility. We also created a new serializer from the Parsley IR to a PDF file to only serialize objects in use—and to avoid polyglot files [28, 176].

PDF comparison tools

There are multiple tools available to compare the output of various PDF parsers on one PDF file. These tools are not much different from VirusTotal—a website that runs a host of antivirus scanning engines to make a decision [117]. Allison et al. built a file observatory to analyze the properties and syntax of various PDF files [6, 5]. For example, they find misspellings of various fields in PDF files by using the Levenshtein edit distance. They were also able to identify the tools used to create a PDF based on the syntax it follows.

Cowger et al. presented the PDF observatory—a tool to run a set of parsers against each file in a corpus [67]. For each file, they then iterate over `stdout` and `stderr` to decide whether the parser rejected a file or accepted it. The Meriadoc recognizer follows a similar approach to comparing parser output.

Ambrose et al. introduce the idea of topological differential testing, a mechanism to decipher the behavior of a set of programs on a corpus of inputs [7]. They use this mechanism to learn the de-facto specification of the input format implemented by these programs. Meriadoc uses a similar consensus approach to finding patterns but does not extract a de-facto specification. The PDF observatory and topological differential testing could be used in place of Meriadoc since all three projects share similar goals and approaches.

5.7.8. Conclusions

This section presents a novel approach to a principled, scriptable rewriting of PDF objects. First, we demonstrated our normalization tool on a set of case studies we designed—informed by our research on a large corpus of PDF files. Then, we evaluated the normalization tools against two datasets of PDF files and demonstrated that text extraction tools generate more text from the modified files than the original files.

Much of the future work remains. As we discussed in Section 5.7.2, our serializer generates a normalized PDF file that contains all the in-use objects in the original file. However, we remove linearization and incremental updates, forcing the modified PDF to diverge significantly from the original file. We will explore how to provide users with flags to support these features in future work.

We will explore another direction of creating a tight-nit feedback loop between the format-aware tracing tools of Meriadoc and our Reducer API. With such a tool, we can dynamically generate these reducer rules with little developer input to fix commonly seen malformations in PDFs.

Section 5.8

Comparison with Caradoc

Error Type	Caradoc	Parsley
Name and String mismatch	No	Yes
Missing PageTreeNode keys	Yes	Yes
Invalid DateString	No	Yes
Invalid PageMode	Yes	Yes
Indirect Reference Missing	No	Yes
Malformed Fonts	No	Yes

Table 5.12: Comparing Caradoc and Parsley

We ran the Parsley type checker and Caradoc on the same set of files (as outlined

in Section 5.5.1) to analyze the differences in results. We found that Caradoc does not catch several errors that the Parsley type checker catches. We detail these errors in Table 5.12.

We found that both Parsley and Caradoc check the Page Tree nodes carefully to ensure all the required fields are present. Whereas only the Parsley type checker carefully enforces mandatory indirect references and date string syntax. Caradoc also follows a safelist of options for the PageMode and PageModes fields. The specification clearly says the allowed values for this field, and both Caradoc and Parsley find violations to this field.

	MuPDF	**Caradoc Strict**	**Caradoc**	**Parsley**	**Caradoc Cleanup & Parsley**	**Parsley with Transformer**
Files Accepted	9801	3442	6483	7010	7189	7436
Files Rejected	199	6558	3517	2990	2811	2564

Table 5.13: Results of running 10,000 files through these PDF parsers.

We ran every PDF in our 10,000 dataset through five PDF parser configurations. Next, we used MuPDF to compare the results with the other parsers. Caradoc provides three modes of operation: strict, relaxed, and cleanup. The cleanup mode takes a broken PDF file as input and normalizes it to a fixed PDF output. We used this cleanup mode in conjunction with the Parsley parser.

Table 5.13 shows the results of our experiments. We see that the MuPDF parser only rejects around 2% of the files. In contrast, the stricter Caradoc parser rejects around 66% of the files. We also see that the Caradoc cleanup step improves the number of files Parsley rejects. The fixups by Caradoc alleviate some of the issues earlier seen by Parsley.

The Caradoc parser results also check for content stream errors other than syntax and type check errors. However, since it is not in the scope of this chapter, we suppressed content stream errors in Parsley. Most of the files rejected by MuPDF were rejected by Parsley and Caradoc, barring around thirty files for each Parsley

and Caradoc.

Section 5.9

Conclusions

In this chapter, I presented our tool SPARTA, which generates PDF version-specific Rust code from the Arlington PDF Model. This Rust code invokes the Parsley type checker that applies types to each object in a PDF file. This process uncovered how several popular implementations violate the PDF specification and how we can find such violations by applying our strict parser. Our findings have led to changes and modifications to the specifications to handle such antipatterns, and modifications to implementations to conform to the specification.

In the process, we also found that PDFs often contain minor malformations that can be corrected by implementing rules to fix these PDFs. Subsequently, we build the PDFFixer, comprising a reducer and serializer, to transform the intermediate representation and produce serialized PDFs. When we run the PDFs generated by PDFFixer, we find that we produce more text than the original PDFs.

5.9.1. Future work and open problems

Although SPARTA and PDFFixer have come a long way in improving the accuracy of the specifications implemented by PDF tools and bringing the specification closer to how real-world PDF implementations handle the specification, much of the work remains. The PDF specification has been around for decades without such strict type checkers to ensure PDF files generated from these tools conform to the specification. That has led to several different clusters of deviations from the specification.

Documenting these grammar drifts As we first documented in Section 5.7.6, it is important that we create an accurate and easily interpreted taxonomy of how different implementations handle the PDF specifications. The PDFFixer and the reducer rules provide ways to reverse the errors introduced by a PDF tool. However, these rules are not comprehensive. One option is to document the differences in the form of changes to the Arlington DOM. For example, suppose an object type is a Name object instead of a string in some implementation. In that case, the transformations are then applied to the Arlington DOM, and we maintain a taxonomy of the deviations from the specification. I believe this is an achievable follow-on work to this thesis.

Machine-readable syntax definitions This thesis primarily focuses on applying types to various objects in a PDF file. However, before applying these types, we must extract the overall syntax and understand the basic types present in the PDF object. Therefore, we used the Parsley tools to syntactically analyze the file and extract the Parsley IR. However, the syntax checker was written by hand and can be prone to errors in the interpretation of a specification. Hence, one direction the LangSec community must focus its efforts on is building rigorously-tested and accurate machine-readable specifications.

Detecting cycles in PDF files. PDF files could contain cycles. This is because several data structures are designed to be like a linked list—with pointers to the next object. However, after a critical reading of the specification, we observed that the specification does not explicitly prevent cycles [288]. Subsequently, Muller et al. [197] also identified that several PDF viewers are susceptible to these infinite loops. Therefore, we believe that a strong PDF validator must detect such loops. In the future, such a tool must also check for such loops and cyclic references in PDF files.

Chapter 6

ParseSmith: Parsing Real-World Data Formats

Previously, Chapter 4 discussed the parser-combinator approach to building parsers. As we outlined in Chapter 1 and Chapter 2, the data description language approach is different from the parser-combinator approach in that we generate code from a grammar description rather than implementing the grammar in code. This chapter describes a parsing toolkit that can generate code in various target languages. Developers can then import the generated parsers into their applications to ensure that they fully recognize input before using them in their applications.

Real-world data formats—network as well as file formats—use constructs not commonly found in formal grammars. For example, let us consider the Domain Name Service (DNS) protocol. In this protocol, a client requests IP addresses for a fixed set of domain names. The client, however, needs to specify in the same packet how many queries are in the packet before including the queries. With this arrangement, the server knows how many queries are present and can parse the packet.

The parser reads an unsigned integer n and then expects n domains. Such a field—commonly called a *repeat* field—cannot be parsed using the common formal

language constructs such as regular grammars, context-free grammars, and parsing expression grammars without a state-space explosion.

Parsing Expression Grammars (PEGs). In 2004, Ford proposed the idea of Parsing Expression Grammars (PEGs) to avoid the ambiguities in parsing Context-Free Grammars (CFGs) [97, 98, 37]. While parsing CFGs, we often run into certain ambiguities, i.e., we can construct multiple parse trees from the same string. PEGs resolve ambiguities by using a prioritized choice. PEGs also implement the *and* and *not* operations that provide lookahead capabilities to PEGs.

The deterministic execution of PEGs and the lookahead capabilities make PEGs an excellent candidate to model data formats. However, off-the-shelf PEGs do not make an appropriate candidate for data formats. They do not support context-sensitive constructs often used in data formats such as repeat fields, dependent fields, and constraints.

Attribute PEGs. Both Mercer et al. [182] and Dos Santos Reis et al. [79] propose the idea of Attribute PEGs. Dos Santos Reis et al. use properties of YAKKER [138], such as using attributes as variables in constraint expressions. They also define the concept of *attribute expressions* to compute attributes for non-terminals.

This chapter builds on the above fundamentals to design Parsley grammars—a variation of Attribute PEGs more suited to describing data formats. We also introduce two algorithms that build on existing algorithms for PEGs.

Parsing PEGs faster. Several researchers have explored improving the speed and performance of PEG parsing algorithms. Henglein et al. [113] presented progressive tabular parsing (PTP), an execution model for PEGs that resolves the leftmost expansions of the parse tree dynamically, enabling us to discard the table columns

corresponding to already resolved portions of the string. In the future, ParseSmith can leverage a variation of this algorithm to get superior memory performance.

Other researchers focused on how we can leverage the previous memoization table while parsing PEGs if the input changes only incrementally. Dubroy et al. [80] implemented an incremental Packrat parser for JavaScript that outperformed non-incremental parsers. Guillermo et al. [110] further improve these results by reducing copy operations and using additional data structures. Yedidia et al. [297] also improve Dubroy et al.'s results by changing the memoization structures.

Dubroy et al., Guillermo et al., and Yedidia et al. focus on the same application of incremental parsing. They are interested in a situation where a user makes minor edits to a program, and the parser must reuse previously memoized data so that the entire program does not need to be parsed again. Our application differs from this since each network packet or file data is independent and may not even be similar in size or structure. Hence, we cannot leverage incremental parsing in ParseSmith to gain any performance upgrades.

Verified Parser Combinators. The Narcissus parser combinator framework allows developers to specify encoders and decoders in Coq [75]. This framework supports a wide range of network formats and proves that the encoder and decoder are inverses of each other. In ParseSmith, we focus on using a more usable interface to the parsing library and supporting software in multiple programming languages to provide verified libraries in their language.

EverParse also implements verified parser combinators using a combination of F-star and low-star [226]. They use similar encoder-decoder proofs to ensure that the parser and serializer are inverses of each other. Additionally, they also prove that their implementation is memory-safe. Finally, they implemented a zero-copy parser for better security and performance. We take a broader approach than EverParse,

implement a more extensive set of combinators, and use a strongly-typed attribute and constraint system to capture context-sensitive languages.

Data Description Languages. Several types of data-description languages (DDLs) are available that can capture various properties of languages. For example, Nail supports dependent grammars and dependent fields [27]. In addition, they use streams and transformations to implement format nesting and demonstrate their system on the DNS protocol and ZIP file format. Finally, they rely on the Hammer parser combinator library under the hood to capture the syntax of these formats.

The Data Format Description Language (DFDL) relies on an XML syntax to capture data format syntax [181]. DFDL supports both text and binary formats out of the box—and provides the parsed structure in multiple formats to the application. DFDL also supports serializing the AST—a feature none of the other popular tools support.

Kaitai Struct relies on YAML syntax to describe formats [295]. In addition, Kaitai supports user-defined constraints and creates a class with all the corresponding fields correctly named. However, due to branching, formal grammar schemas in BNF syntax do not translate well to the Kaitai Struct syntax. Moreover, since they provide a class instance and not an AST as the parsing output, we cannot easily serialize the Kaitai Struct output.

PADS is a strongly-typed data description language supporting data-dependent parsing and user-defined constraints [178]. Parsley grammars use inherited attributes instead of the let-binding constructs used in PADS. The YAKKER system also supports data-dependent parsing and user-defined constraints and again does not support attribute definitions the same way as Parsley [138].

ParseSmith. In this chapter, we design and implement a high-level parsing toolkit to describe and parse data formats. Developers interact with ParseSmith by defining Parsley grammars and providing input to parse. The ParseSmith toolkit contains two parsing algorithms to parse these inputs as per the grammar definitions.

ParseSmith has four high-level goals.

- **Security**: We provide strong security guarantees using correctness and termination proofs.
- **Usability**: We help developers by providing better error messages and a debugger.
- **Parsed Data Access**: Developers can access parsed data via synthesized attributes that are made available. We also provide methods for developers to read, transform and serialize an abstract syntax tree.
- **Code Generation**: Finally, relying on Dafny's code generators, we generate parsers in various target languages such as C++, Go, and JavaScript.

My contributions. I present the following contributions in the rest of this chapter:

- I formalize Parsley grammars: Attribute PEGs that support arbitrary constraints and additional properties that we describe in Section 6.1.
- I present two parsing algorithms: a bottom-up algorithm based on the Packrat algorithm and a top-down recursive descent algorithm to parse *Parsley* grammars in Section 6.2.2.
- I implement ParseSmith: a toolkit to parse these real-world network formats. We produced verified implementations of both the parsing algorithms in Dafny

proving that the algorithms terminate for well-formed Parsley grammars (Section 6.1.1).

– I demonstrate ParseSmith on the Real-Time Publish-Subscribe wire protocol and the PFCP protocol and generate Go, C#, and JavaScript parsers for these protocols using ParseSmith (Section 6.4).

– Finally, I present a first-of-its-kind parser debugger to pause and inspect parser data structures in runtime (Section 6.3). This debugger supports several operations in runtime that resemble state of the art binary debugging tools, such as GDB.

Section 6.1

Parsley Grammars

Mundkur et al. [198] proposed Parsley grammars in 2020 to capture the syntax and semantics of various data formats. They use an attribute and constraint system to capture context sensitivity. The design of Parsley grammars relies on several prior works.

Parsley grammars make use of the core PEG and Attribute PEG ideas. In addition, we use L-Attributed grammars to use attributes in constraints and inherited attributes later in the production. We also build on the YAKKER system to use user-defined attributes as variables in the parsing context [137]. Additionally, Parsley grammars define several fundamental data types such as integers of a varying bit length, characters, character strings, and byte strings.

The Parsley language relies on the above fundamentals to describe various data formats [198]. Specifically, it holds the following properties:

– We use a EBNF notation to describe grammars in the PEG format. As described

earlier, using the PEG constructs instead of CFGs provides determinism and results in better formalism.

- The Parsley language uses a standard attribute system and assigns types to various fundamental non-terminals commonly used in data formats. For more complex non-terminals, the user must specify how the attributes are assigned.

- Finally, we use arbitrary constraints on attributes to specify various data dependencies. Using such constraints, we can implement various cyclic dependencies and integrity checks (such as length fields and checksums) commonly found in data formats.

Definition 6.1. A Parsley grammar is a 6-tuple $G = (V_N, V_T, R, e_S, \mathcal{A}, \mathcal{P})$, where:

- V_N is a finite set of non-terminals,

- V_T is a finite set of terminal symbols,

- R is a finite set of rules, and each rule $r \in R$ takes the form $A \leftarrow e$,

- e_S is the starting expression, and

- $\mathcal{A}$ holds the set of attribute types.

- $\mathcal{P}$ holds the set of binding variables.

We inductively define expressions e in Parsley grammars as follows. $e \in V_N$

(a) ϵ, empty strings

(b) $(a, assign)$, where $a \in V_T$ and $assign(a) \in \mathcal{A}$.

(c) $(List(sequence), assign)$, where *sequence* is in turn defined as:

  (i) $p_i = e_i$, where $e_i \in V_N$, and $p_i \in \mathcal{P}$.

(ii) $constraints(constraint\_name)$, where constraints operate on all p_i defined so far.

(iii) $p_i = (p_j \wedge e_i)$ is the repeat operator, where $p_j \in \mathcal{A}$ and $e_i \in V_N$. If p_j takes the type $\mathcal{A}_j$, then p_i takes the type $List(\mathcal{A}_j)$.

(iv) END is a special operator we introduce to denote the end of the input string.

(d) $e_1/e_2/e_3...e_n$ where $\forall i \in 1\ to\ n,\ e_i \in V_N$, prioritized or ordered choice.

(e) e^*, and the type the attribute takes is $List(\mathcal{A}_i)$ where $\mathcal{A}_i$ is the attribute type of e. This expression matches zero or more repetitions of e.

(f) e^+, and the type the attribute takes is $List(\mathcal{A}_i)$ where $\mathcal{A}_i$ is the attribute type of e. This expression matches one or more repetitions of e.

(g) $\&(e)$, the *and* predicate does not consume input but attempts to match a string. This expression can be used as a lookahead operator. This expression returns an attribute with the same type as e.

(h) $!(e)$, the not operator also does not consume any input and can be used as a lookahead operator. If e succeeds at the current location, then $!(e)$ fails, and vice-versa. This expression returns an attribute with the same type as e.

(i) $e?$, the optional operator. This expression returns an attribute with the same type as e if successfully matched, else returns $Null$.

6.1.1. Well-Formed Parsley Grammars

Although we rely on the Dafny compiler to find syntax errors in ParseSmith grammar input, grammars can contain other malformations that we must detect to prove properties about the parsing algorithms. Therefore, we implement preconditions to our

parsing functions. We check these preconditions prior to calling these functions—if these preconditions are violated, the program must exit early and not call the parsing functions.

Definition 6.2. A Parsley grammar is well-formed if it satisfies the following conditions:

- Every dependent variable must be defined earlier in the execution model.
- The Repeat construct must include an attribute of an integer type.
- No left recursions must be present. It is impossible to have complete proof of lack of left recursions. We propose using a check that is sound but not complete [98].
- Check for cyclic references: two nonterminals may be mutually recursive, making it hard to prove that they will terminate.
- `End` type must always occur last in a sequence.

Section 6.2

Implementation

We implemented various components of ParseSmith in the Dafny programming language. The Dafny programming language allows developers to annotate functions and methods with preconditions and postconditions. These annotations enable users to prove various correctness properties on their code.

The Dafny programming language also supports loop invariants and annotations for recursive functions. These constructs are vital in proving that a program terminates for all possible input. Most importantly, the Dafny verifier does not proceed

with code generation if any of the above properties—loop invariants, recursion termination, preconditions, and postconditions—cannot be proven.

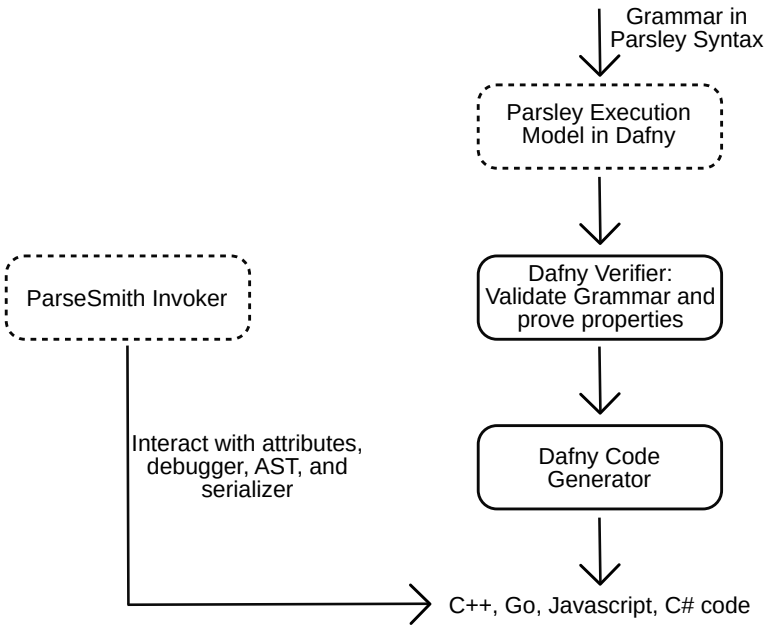


Figure 6.1: Workflow of our Dafny ParseSmith implementation. The dotted components show our contributions. Other components use Dafny-provided tools.

Figure 6.1 shows the overall architecture of ParseSmith. The dotted lines show components we contribute in ParseSmith—the other components are provided by the Dafny verifier and compiler. The execution model accepts input via Dafny syntax that closely resembles Parsley grammars.

The execution model implements two parsing algorithms—first, a bottom-up algorithm based on the packrat algorithm. Next, a top-down algorithm that starts with nonterminals while recursively reaching the terminals needed to parse the input. The second algorithm consumes lesser memory while also supporting inherited attributes. However, the first algorithm only supports synthesized attributes with Parsley grammars. We describe these algorithms in detail in Section 6.2.2.

The ParseSmith invoker combines several utility tools we built around the execu-

tion model to improve debugging and usability of ParseSmith. First, we describe a serializer. The serializer allows us to serialize an AST back to the input bytes. The developer can define transformation rules on the AST output. Following this, they may need to serialize the set of bytes. For example, an everyday use case is that network packets occasionally contain confidential data that may need to be redacted before packets or files leave the network.

Next, the invoker includes a parser debugger. This debugger provides functionalities for users to pause and examine the parsing table at any given step and then step through iterations of the various algorithms. In addition, the user can place breakpoints on nonterminals and input locations. This tool is hugely vital to aid in debugging grammars on large inputs.

Finally, the invoker provides access to the synthesized attributes to the start nonterminal. Using the API, users define this start nonterminal. Subsequently, the invoker provides access to the synthesized attributes and the AST for this nonterminal. The serializer and the transformers previously described can be called from this AST. The synthesized attributes provide a developer-defined attribute structure. Developers can invoke this structure to gather file metadata and contents from their target programming language.

Figure 11 shows the datatypes we use to implement Parsley grammars in ParseSmith. We used inductive datatypes in Dafny to closely model our Parsley grammar inductive definition in Definition 6.1. `F_terminal`, `star_plus_f`, and `update_attr` are all function types used to assign attributes to non-terminals. A comprehensive list of these function types are in Table 6.1.

Dafny does not contain integer types for integers of varying lengths. It also does not contain types for unsigned and signed variants of integers—the `int` type is signed. We created the type `int8` to add an additional constraint on the `int` type to create

```
datatype grammar = epsilon
    |  any(update_terminal: F_terminal)
    |  terminal(a : int8, update_terminal: F_terminal)
    |  range(a : int8, b : int8, update_terminal: F_terminal)
    |  sequ(assignments: seq<sequence>, update_attr: F)
    |  star(f1 : string, update_attr_ter: star_plus_f)
    |  plus(f1 : string, update_attr_ter: star_plus_f)
    |  opt(f1 : string)
    |  notP(f1 : string)
    |  andP(f1 : string)
    |  prior(f: seq<string>)

datatype sequence = Sequence (d1: string, f1: string)
    | Constraint (con: constraint)
    | Repeat(d1: string, f1: string, f2: string)
    | SequenceWithError(d1: string, f1: string, error: string)
    | RepeatWithError(d1: string, f1: string, f2:string, error: string)
    | ConstraintWithError(con: constraint, error: string)
    | End(error: string)
```

Code Snippet 11: ParseSmith implementation of Parsley grammars

unsigned 8-bit integers (i.e., 0 to 255). The `int8` type can be used to denote a byte of input.

The `sequence` datatype defines how we can concatenate productions. The grammar format covers all the types present in parsing expression grammars, including *and*, *not*, *star*, *plus*, and *prioritized choice*. Additionally, we also support additional annotations in the sequence type.

`End` type. PEGs can consume input while not matching with the entire input. As a result, the parsing may not fully fail but partially fail. A common way is to check if the entire input was consumed by comparing the number of bytes consumed to the input size. The execution model supports such execution; however, we also support a different construct.

The End construct takes a string as an input. When the parser encounters this

construct as part of a sequence, it checks if we have reached the end of the input. If we still have input when encountering this operation, we stop parsing and print out the string given as an argument. Therefore, this operation must appear only at the end of a sequence.

`WithError` versions. We add support for these operations extending the simple Constraint, Sequence, and Repeat operations. The WithError format of these operations provides additional help for developers to debug their parsers with meaningful errors. For example, instead of the parser stating that a constraint at a certain byte and nonterminal failed, we can print out a meaningful error message with a byte number and nonterminal. The developer can use this information to debug the grammar using the ParseSmith debugger.

Attribute system. Users define the attribute types needed for each non-terminal in ParseSmith using an inductive data type in Dafny. For example, the following code sample shows the attribute definition we used to capture the language $\mathcal{L} = a^n b^n c^n, \forall n \geq 0$.

```
datatype Attributes = AStar(size: int) |
                      BStar(size: int) |
                      CStar(size: int) |
                      Nil |
                      Byte(val: int8) |
                      Int(intval: int)
```

Users must also define a function to define these attributes for each non-terminal. The function type definitions for each of the different non-terminal types are defined in Table 6.1. The other grammar types— optional fields, and, not, and prioritized

choices—do not take these assignments. Instead, they assign the same attribute as their argument to the defining non-terminal.

Non-terminal type	Function type	Dafny Name
Terminals	`(int8) -> Attributes`	`F_terminal`
Any	`(int8) -> Attributes`	`F_terminal`
Sequence	`(map<string, Attributes>) -> Attributes`	`update_attr`
Constraints	`(map<string, Attributes>) -> Attributes`	`update_attr`
Plus	`(Attributes, Attributes) -> Attributes`	`star_plus_f`
Star	`(Attributes, Attributes) -> Attributes`	`star_plus_f`

Table 6.1: Type definitions for attribute creating functions.

6.2.1. Parsley ASTs

Parsley ASTs form the backbone of our Completeness proof. For each expression type in Parsley grammars, we define a rule to create the AST and a subsequent rule to serialize the AST to bytes. Figure 12 demonstrates the inductive datatype we use to define Parsley ASTs.

This AST follows closely from the ASTs in Bleadeau et al. [37]. We store the successful and the unsuccessful paths in the prioritized choice or sequences. Each node in the AST stores the number of bytes consumed and whether it corresponds to a successful or failing expression. A sequence parser stops when it encounters a failing non-terminal. The AST corresponding to the failing non-terminal would be the last AST in the list corresponding to all the non-terminals in a sequence.

Similarly, the prioritized choice parser tries the choices in order—following the prescribed sequence. We store the AST at each failure case. The prioritized choice parser halts once a successful case is found. Hence, the successful AST is always the last node in the list of ASTs in a prioritized choice AST type.

The terminal types—`ast_terminals`, `ast_any`, and `ast_range`—include an additional parameter value that holds a byte. These terminal types form the leaves of the

AST. We do a depth-first search (DFS) on the AST to find the bytes corresponding to the string. This DFS ignores all the nodes where the success flag is set to false.

```
datatype ast = ast_nil(consumed: int, success: bool)
    |  ast_epsilon(consumed: int, success: bool)
    |  ast_any(consumed: int, success: bool, value: int8)
    |  ast_terminal(consumed: int, success: bool, value : int8)
    |  ast_range(consumed: int, success: bool, value : int8)
    |  ast_sequ(consumed: int, success: bool, T: seq<ast>)
    |  ast_repeat(consumed: int, success: bool, T: seq<ast>)
    |  ast_prior(consumed: int, success: bool, T: seq<ast>,
                 choicename: string)
    |  ast_star(consumed: int, success: bool, T1 : ast, T2: ast)
    |  ast_plus(consumed: int, success: bool, T1 : ast, T2: ast)
    |  ast_opt(consumed: int, success: bool, T : ast)
    |  ast_notP(consumed: int, success: bool, T : ast)
    |  ast_andP(consumed: int, success: bool, T : ast)
```

Code Snippet 12: Dafny inductive datatype showing our Parsley ASTs

6.2.2. Parsing Algorithms

To interpret Parsley grammars, we create an interpreter model with a relation $\Rightarrow_G$ that maps an expression (e, x), where $e \in G$ and $x \in V_T^*$. e is one of the inductively defined expressions in G, whereas x is a subset of the input string. The relation $\Rightarrow_G$ maps each (e, x) to a tuple $(n, \mathcal{A}_e, \mathcal{T})$, where $n \geq -1$, $\mathcal{A}_e \in \mathcal{A}$, and $\mathcal{T}$ is a well-defined abstract syntax tree (AST). We inductively define $\Rightarrow_G$ following the syntax in Definition 6.1 as follows.

To evaluate each expression, we use an input string $x \in V_T^*$. We use the Nil attribute type to capture failure cases in the AST.

(a) **Empty strings**: $(\epsilon, x) \Rightarrow (0, Nil, ast\_epsilon(consumed : 0, success : true))$, where $Nil \in \mathcal{A}$.

(b) **Terminals**:

- **Success**: $(a, ax) \Rightarrow (1, T(a), ast\_terminal(consumed : 1, success : true, value : a))$, where $T(a) \in \mathcal{A}$.
- **Failure**: $(b, ax) \Rightarrow (-1, Nil, ast\_terminal(consumed : 0, success : false, value : a))$, where $Nil \in \mathcal{A}$.

(c) **Sequences**: If $e = e_1 e_2 .... e_n$ and $x = x_1 x_2 .... x_n$. And the substrings $x_1 ... x_n$ hold the length s_1, s_2... s_n. Then,

- **Success case**: If $(e, xy) \Rightarrow (s_1 + s_2 + ... + s_n, Seq(dependent\_variables), ast\_sequ($
  $s_1 + s_2 + ... + s_n, success : true, T : seq < ast >))$
- **Failure case**: If $(e, xy) \Rightarrow (-1, Nil, ast\_sequ(s_1 + s_2 + ... + s_i,$ $success : false, T : seq < ast >))$, where the last value in the sequence T is the failing AST. We can reach a failure if $\exists e_i \in e$, s.t. $(e_i, x_i y)$ did not lead to a successful parse.

(d) **Prioritized choice**: If $e = e_1/e_2/..../e_n$,

- **Success case**: If $(e_i, xy) \Rightarrow (s_i, typeof(e), ast\_prior(s_i, success : true, T : seq < ast >))$, where e_i is one of the choices in e. e_i is tested if $\forall j in 1\ to\ i-1, (e_j, xy)$ led to a failure.
- **Failure case**: If $(e, xy) \Rightarrow (-1, Nil, ast\_prior(-1, success : false, T : seq < ast >))$, where the every value in the sequence T is a failing AST.

(e) **Star**: This operation can consume ϵ, and hence cannot fail. If $e = e_1^*$ and $(e_1, x_1 x_2 y) \Rightarrow (s_1, typeof(e_1), T : ast_1)$ and $(e_1*, x_2 y) \Rightarrow (s_2,$
$seq < typeof(e_1) >, T : ast_2)$.

(f) **Plus**: If $e = e_1^*$ and $(e_1, x_1x_2y) \Rightarrow (s_1, typeof(e_1), T : ast_1)$ and $(e_1*, x_2y) \Rightarrow (s_2, seq < typeof(e_1) >, T : ast_2)$.

- **Success**: If both the above tests succeed, we will return the following $(s_1 + s_2, Plus(dependent\_variables), ast\_plus(consumed : s_1 + s_2, success : true, T1 : ast_1, T2 : ast_2))$
- **Failure**: If the first test does not succeed, then the operation fails. We will return the following $(-1, Nil, ast\_plus(consumed : -1, success : true, T1 : ast_1, T2 : ast_2))$. ast_1 here must be a failure case.

(g) **And**: If $e = \&(e_1)$ and $(e_1, xy) \Rightarrow (s_1, typeof(e_1), T : ast)$ then,

- **Success**: $(e, xy) \Rightarrow (0,\ typeof(e_1), ast\_andP(consumed : 0, success : true, T : ast))$, where ast is the same as the ast matched for e_1.
- **Failure**: $(e, xy) \Rightarrow (-1, Nil, ast\_andP(consumed : -1, success : false, T : ast))$.

(h) **Not**: If $e = (e_1)$ and $(e_1, xy) \Rightarrow (-1, Nil, T : ast)$ then,

- **Success**: $(e, xy) \Rightarrow (0, Nil, ast\_notP(consumed : 0, success : true, T : ast))$.
- **Failure**: $(e, xy) \Rightarrow (-1, typeof(e_1), ast\_notP(consumed : -1, success : false, T : ast))$, where ast is the same as the ast matched for e_1.

(i) **Optional**: If $e = e_1?$ and $(e_1, xy) \Rightarrow (s_1, typeof(e_1), T : ast)$ then,

- **Success**: $(e, xy) \Rightarrow (s_1, typeof(e_1), ast\_opt(consumed : s_1, success : true, T : ast))$, where ast is the same as the ast matched for e_1.
- **Failure**: $(e, xy) \Rightarrow (0, Nil, ast\_opt(consumed : 0, success : false, T : ast\_nil(0, true)))$.

We use three scaffolds to store the bytes consumed (similar to a standard Packrat algorithm), attribute generated from the expression, and AST corresponding to the expression. We keep three large scaffolds of size $(n + 1) \times k$, where n is the input size, and k is the number of productions. We use the above inductive definition to compute the entries for our scaffold tables.

Bottom-up algorithm

Our bottom-up algorithm implements an algorithm similar to Packrat [97]. The algorithm needs the order in which the non-terminals must be evaluated. We rely on the user to specify this order in the current implementation. If the input size is n, we use columns from 0 to n. Additionally, we use 0 to $k - 1$ rows for the non-terminals. We fill the scaffold from right to left, the rightmost entry (column n) signifying ϵ—and bottom to top.

For both the loops, we implement loop invariants to cover all the paths our parsing function can take based on the inductive definitions described earlier. In addition, we use invariants to ensure our completeness property—by ensuring that if the AST value in the scaffold holds a success value of true, then the serialized output corresponds to the substring matched.

Top-down algorithm

An evident problem of using our bottom-up algorithm is that we fill every entry in the scaffold despite using only a tiny portion of them to compute the final scaffold cells. Hence, we implement a top-down algorithm that uses a stack to implement the parsing algorithm. Using this version of the algorithm means we do not need to specify the order of the non-terminals anymore.

This approach leverages a stack. Our stack only contains the starting non-terminal

and the input location we attempt to match. For example, if the non-terminal is S and the location is 0 (this would be the starting state for any parser), then we place $(S, 0)$ in the stack before calling the parser.

The parser pops the stack and checks for all the dependencies needed to fill this scaffold entry. Next, we place the popped entry back in the stack and push each dependent entry onto the stack. As it may be evident, left recursions and cyclic references would mess up the scaffold computations.

The parsing function stops when the stack is eventually empty. Most importantly, the stack may only contain one copy of an entry. As the parser completes, we check the $(S, 0)$ entry in the scaffolds to extract the parsed attributes or AST. The top-down algorithm drastically reduces memory coverage and CPU time by not filling every unnecessary cell in the scaffolds. We use a targeted approach by only filling the cells necessary to get to the $(S, 0)$th location.

Complexity analysis. We compute three scaffold tables during our parsing functions: the usual packrat scaffold, an attributes scaffold, and an AST scaffold. Our bottom-up algorithm computes every location in each scaffold and hence consumes much space and time. If we have k productions and the input size is n, both the algorithms follow a space complexity of $O(n \times k)$. However, the algorithms differ in the time complexity due to the *Repeat* operator. The bottom-up algorithm requires $O(n^2 \times k)$ time. Whereas the top-down algorithm uses the same $O(n \times k)$ time. However, in practice, the top-down algorithms save far more space and time by only computing needed entries.

6.2.3. Properties

We prove the following properties for our implementation:

– *Termination*: For a well-formed grammar, our parser terminates on all in-

puts [152, 37]. We rely on the Dafny verifier to prove termination. In ParseSmith, we use user-defined functions to define constraints and attribute declarations. Therefore, the Dafny verifier forces the user to prove termination for each function—user-defined or a part of the ParseSmith library.

The Dafny verifier requires that we provide "decreases" clauses for recursive functions. Although we do not implement many recursive functions in ParseSmith, we use a recursive implementation to serialize ASTs. Therefore, we had to annotate the serializing functions to ensure that the AST supplied continuously decreases in size. We also annotate loops using invariants to prove termination.

– *Completeness*: A traversal of the AST produces the same input string [226]. Each cell in the AST scaffold contains a tree of the AST type. Furthermore, each tree datatype contains a field to denote the number of bytes that were consumed to form that subtree. For example, let us consider that at cell $(S, 5)$, the "consumed" field in the AST shows 6. It implies that the non-terminal S consumed 6 bytes to form that AST. Hence, the serialized form of the AST must contain the substring starting with location 5 and ending with location 10.

  To verify *completeness*, we use loop invariants to ensure that the serialized version corresponds to the correct substring for every cell in the AST.

– *Correctness*: We prove that we implement all the parsing rules for Parsley grammars. We do this by specifying loop invariants for all expressions in Parsley grammars. We use a *correctness* clause for each expression type defined in the earlier inductive definitions. The program only invokes the parsing function and the loops if the pre-conditions for the grammar well-formedness are met. Hence,

the correctness clause only holds if the parser's pre-conditions are satisfied.

Section 6.3

Parser Debugger[1]

Developing grammars for data formats is an iterative process that may require trial and error. Since specification documents are long and verbose, developers may misinterpret portions of the specification and not implement them correctly. Therefore, to ensure that most cases in a specification are implemented correctly, we must construct a large corpus and test our grammar on it.

As we noted earlier in Chapter 5, two outcomes may occur from running our parsers on data. First, data constructed by tools in the wild may be producing malformed data. Second, the developer could have introduced errors in the grammar. To help developers debug parsing errors that may arise from malformed input or grammars, we provide debugging options with ParseSmith.

Print columns after update We support this operation in ParseSmith. Our algorithms match every nonterminal to an input substring starting from an input location. Then, we print the values computed corresponding to a nonterminal after matching an input location. This approach does not pause the execution but instead prints these values for debugging logs.

Supporting breakpoints We also support a separate debugging mode, where users can place a breakpoint on a nonterminal. Each time the parsing function is computing this nonterminal, our parsing halts for user input. At this point, a user can choose to inspect specific memory locations in the scaffold or continue the execution. The

[1]Patrick Norton helped build the ParseSmith Debugger over the summer of 2021, when he interned with our group before joining college that fall.

execution will pause again when it reaches the same nonterminal for another input location.

Command	Arguments	Function
b	Nonterminal name	Add a breakpoint at a nonterminal
c	None	Continue execution
p	Input location and nonterminal	Prints the values in the table
s	Number of steps	Take steps and pause again for input

Table 6.2: Commands supported in the ParseSmith debugger

Figure 6.2 provides a complete list of commands we support in debug mode. In addition to adding breakpoints and continuing execution, we also support printing and stepping through the parsing function sequentially.

Our debugger helped us debug several errors in our ParseSmith specifications and helped us identify the root causes of why we rejected some RTPS packets in our case study. Furthermore, by allowing users to step through the nonterminals and their assignments, we provide the first-of-its-kind parser debugger that exposes the internals of the parsing data structures.

Section 6.4

Case Studies

To evaluate ParseSmith, we conducted two case studies. We implemented a parser for the Real-Time Publish-Subscribe and the Packet Forwarding Control protocols. We evaluated our parsers on an Intel i7 4th Generation Processor with 16 GiB RAM.

6.4.1. The Real-Time Publish Subscribe Protocol (RTPS)

The Object Management Group (OMG) released the first version of the Real-Time Publish-Subscribe (RTPS) protocol in 2006 [262]. The current active version of the RTPS specification is version 2.3, and it was released in 2018. Publish-subscribe

protocols such as RTPS are used to transmit data unilaterally. Several users subscribe to a server (subscribers). Whenever users (publishers) publish data, the server broadcasts this data to all subscribers.

Each RTPS packet includes a 160-byte header followed by multiple sub-messages. Additionally, each sub-message ends on a 32-byte boundary. There are thirteen sub-message types of which only the Data and the DataFrag sub-message types vary in length—all other types are fixed-length messages. RTPS messages and sub-messages do not contain length fields—hence, Data and DataFrag sub-messages must always occur last in the sub-message list.

People have explored building parsers for the RTPS protocol in the past. For example, ElShankiry et al. [86] use the Structure and Context-Sensitive Language (SCL)—an extension to the ASN.1 description format—to describe various binary protocols. SCL adds additional types and markups to support more constructs found in network protocols. They demonstrate their approach to generate LL(k) parsers using SCL on the RTPS protocol.

Lavorato et al. [160, 159] built a technique to optimize LL(k) grammars that are used to describe network protocol formats. They produce parsers that perform better than ElShakankiry et al. [86] by optimizing backtracking using lookaheads. Their focus was to improve the performance of LL(k) parsers, and they used RTPS to demonstrate their techniques.

We implement the RTPS protocol syntax for all submessage types in ParseSmith. We describe these packets entirely in Dafny. This format does not require any inherited attributes but very strongly relies on synthesized attributes to pass values to the end-user.

Figure 13 shows the overall syntax of RTPS packets in ParseSmith. Packets here is the starting nonterminal. The equivalent Parsley grammar is shown in Figure 14. As

we have previously covered, ParseSmith provides additional syntax to describe better error strings and using the End construct to denote reaching the end of a packet.

The sub-messages nonterminal is defined using the plus construct. Let us consider the following packet structure.

```
Header
GAP submessage
DataFrag submessage
```

A DataFrag sub-message follows a GAP sub-message in the above structure. If the DataFrag sub-message is malformed, the sub-messages nonterminal still succeeds since it parsed with one sub-message. Hence, we include an additional End construct to ensure that when we encounter this construct in the sequence, we have reached the end of the packet—implying that we have parsed the DataFrag sub-message.

```
peg := peg["Packets" := sequ([
        SequenceWithError("r", "R", "Did not match magic character R"),
        Sequence("t", "T"),
        Sequence("p", "P"),
        Sequence("s", "S"),
        Sequence("version", "Int16"),
        Sequence("vendor", "Int16"),
        Sequence("guid", "GuidPrefix"),
        Sequence("submessages", "SubMessages"),
        End("Expected more characters reached end")],
        update_packet)];
```

Code Snippet 13: Code sample showing the RTPS packet structure in ParseSmith.

We will evaluate the performance of our RTPS parser using a dataset of RTPS packets we generated using open-source tools in Section 6.6.

```
Packets p {s: Submessage, v: int} := r=!"R"!
                t=!"T"!
                p=!"P"!
                s=!"S"!
                version=Int16
                vendor=Int16
                guid=GuidPrefix
                submessages=(Submessage)+
                {
                    p.v := version ;;
                    p.s := submessages
                } ;;
```

Code Snippet 14: Code sample showing the RTPS packet structure in the Parsley language syntax. This code represents the same structure shown in Code Snippet 13.

6.4.2. Packet Forwarding Control Protocol (PFCP)

The Packet Forwarding Control Protocol (PFCP) plays a crucial role in the 5G Next Generation (5GC) suite of protocols [155]. Earlier, it found a prominent role in worldwide implementations of the 4G/LTE protocol to implement Control and User Plane Separation. Essentially, packets between control and user planes are processed and forwarded or discarded depending on PFCP connections. The PFCP protocol is UDP based and uses the reserved port 8805 for its communications [266].

We chose this protocol for our case study given its wide-ranging application and increased adoption. The use of CFCP in cellular networks increases throughput in several ways:

- By separating the control plane and user plane, we reduce latency.
- We can use software-defined networking to deliver user plane data more efficiently.
- The evolution of control plane protocols is independent of the user protocols—not all control plane nodes need to be updated time and again.

Additionally, attacks against CUPS systems can lead to significant network degradation [187]. Therefore, not only will users be directly affected by such an attack, but such an attack can lead to violations of quality of service (QoS) requirements and service level agreements (SLAs), potentially costing a service provider millions of dollars. Ensuring the syntactic validity of packets in this protocol would go a long way to eliminating certain classes of vulnerabilities from code.

Syntax Code Snippet 15 demonstrates a portion of the syntax to describe the PFCP protocol in ParseSmith. PFCP packets start with a 1 byte header. The header specifies the version and holds flags. We check the header separately using the constraint `check_first_byte`. A PFCP packet comprises several information segments. Each of these information segments follow a specific type and structure. Since we use the star operator to match information segments, we may end up with bytes remaining at the end of the buffer that have not been matched. To ensure that the length field is set correctly, we use a `validate_length` constraint and a `End` type at the end to ensure we have to trailing bytes. The information segments contain more length fields that need to be thoroughly validated.

```
peg := peg["PFCP" := sequ([
                    Sequence("b", "Byte"),
                    Constraint(check_first_byte),
                    Sequence("length", "Int16"),
                    Sequence("s", "Int24"),
                    Sequence("z", "Zero"),
                    Sequence("infopackets", "PFCPInformationMultiple"),
                    Constraint(validate_length),
                    End("Unmatched bytes at the end present")
                ], update_seq)];
peg := peg["PFCPInformationMultiple" :=
                star("PFCPInformation", info_packets)];
```

Code Snippet 15: Code sample showing the PFCP packet structure in ParseSmith

Section 6.5

Testing the generated parsers

In building ParseSmith, we relied on the Dafny verification toolkit and compiler to produce code in Go, C#, and JavaScript. However, the generated code may, in turn, contain bugs that were introduced by the code generator. We conducted two sets of experiments to test the generated code to look for errors, bugs, and security issues.

6.5.1. Static Analysis

Static analysis is the process of evaluating code and checking for common anti-patterns and bugs without actually running the code. Static analysis engines are usually rule-based and parse the code and check for patterns. Static analysis tools construct code graphs to performs various analyses. We use these techniques on our generated code to ensure that the code is bug-free and provides the required guarantees for the rest of the application.

Go Static Check The Go-tools Static Checker [119] uses the single static assignment (SSA) syntax produced by Go binaries. This tools contains more than 150 patterns designed to find common bugs, performance issues, and vulnerabilities. This tool can find misuses of standard library functions, errors in regular expressions, improper variable use, and several other common errors observed in Go production code.

Insider Insider [129] focuses on covering the vulnerabilities in the OWASP Top 10 list.[2] These vulnerabilities are commonly occurring and, can quite often, be exploited with not much difficulty. Although we used the command-line tools available, devel-

[2] `https://owasp.org/www-project-top-ten/`

opers can also set up a continuous integration pipeline on GitHub to run the Insider tests every time someone commits to a repository. Insider supports JavaScript, C#, and Java source files for analysis. Hence, we use Insider to analyze our generated C# and JavaScript code.

Findings Go Static Check found that six variable were initialized in the generated code for the RTPS and PFCP files each that were never used in the code. In addition to a Go file for the general repository, Dafny also generates internal files containing helper functions. These files do not change between projects, and also contained six variables that were initialized but never used.

Insider contains rules for each programming language in its set.[3] We tasked Insider with scanning the folders containing generated JavaScript and C# code for our RTPS and PFCP case studies. Insider checked for 76 known anti-patterns and vulnerabilities in C# code and 56 rules in the JavaScript code and found no violations in the code generated by Dafny.

Based on the static analysis tools that we used, the generated Go code contained some minor bugs—some variables that were initialized were never used in the code. As a result, memory allocated for these variables could be misused by an adversary modifying the program's address space. The fix for this bug is straightforward. Our tools did not find issues in the C# and JavaScript code, indicating that the code produced by Dafny is generally of high quality.

6.5.2. Fuzzing

Fuzzing or fuzz testing is a testing paradigm where a program is supplied with random input to try to reach not-so-well-tested code paths that may contain crashes or bugs.

[3]We also found that Insider requires a `tech` flag to choose a set of rules to apply. However, a folder containing code from multiple programming languages, for example, C#, Java, and Go, Insider still applies the correct rule set based on the file extension.

Fuzzing tools are extremely effective in finding memory corruption, integer overflows, and other common bugs seen often in C and C++ applications. However, the use of fuzzers in other languages requires additional effort. For example, in Python and Rust, fuzzers are used to find paths to exceptions.

Go Fuzz The Go programming language happens to be one of the few in the world that supports fuzzing as a part of the language [118]. Go Fuzz requires a corpus that is then mutated to find interesting new paths. Like Python fuzzers, Go Fuzz is designed to find panics, stack overflows, paths to exit functions, and timeouts.

Go Fuzz requires that we declare a fuzzing function using the following syntax.

```
f.Fuzz(func(t *testing.T, orig string) {
...
}
```

We then invoke the testing function using the Go module system with the following command and the expected output is shown below.

```
go test -fuzz=Fuzz
fuzz: elapsed: 0s, gathering baseline coverage: 0/3 completed
fuzz: elapsed: 0s, gathering baseline coverage: 3/3 completed,
    now fuzzing with 8 workers
fuzz: elapsed: 3s, execs: 418168 (139372/sec), new interesting: 0
    (total: 3)
fuzz: elapsed: 6s, execs: 844639 (142172/sec), new interesting: 0
    (total: 3)
```

Approach We invoked Go fuzz on the ParseSmith engine by defining the input to be Go arrays that we can then transform into a Dafny sequence. In the Dafny-

generated Go code, we need to use the following syntax to define sequences to seed our fuzzer.

```
_dafny.SeqOf(uint8(82), uint8(84), uint8(80), uint8(83), uint8(2),
uint8(4), uint8(1), uint8(3))
```

A Go array containing `[82, 84, 80 ... 1, 3]` needed to be converted to the above Go-Dafny syntax in a function. In our test harness, we tested this function and the core parsing functions called with the transformed arrays. We fuzzed the generated Go code for 24 hours and found no crashes or exceptions.

6.5.3. Discussion

Verifying the correctness of the generated code from the Dafny compiler is difficult. The discussion of whether the generated code and the original Dafny code are equivalent leads to another discussion to understand the trusted computing base [234] of the ParseSmith project. We inherently assume that the operating system and the compiler we use on the generated code are trusted and secure.

Verifying the verifier The Dafny compiler and verifier are written in C# and comprises thousands of lines of code. Therefore, verifying this would be a massive undertaking. For example, the CompCert compiler, a C compiler verified in Coq, contains over 200,000 lines of code [165]. C compilers have not, however, been without errors. In 2008, several C compilers were found to misinterpret the volatile keyword and erroneously compiled it [85]. Hence, verifying compilers, verifiers, and code generators is a worthwhile pursuit.

A possible approach One approach that would also require a significant engineering effort is to guarantee the same invariants in the generated code as in the original

Dafny code. For example, the Viper project includes verification tools for the Go language [284], Rust, Python, and several other commonly used languages.[4] The Viper project uses a similar set of invariants, pre-conditions, and post-conditions in our Dafny implementation. Support for a mapping from the invariants and pre-and post-conditions would greatly increase the trust in the generated code since it would then guarantee the same properties in the generated code and the original Dafny source.

Section 6.6

Performance Evaluation

The Dafny command-line tool provides numerous options to customize the output. For example, we can opt to generate code in a target language or directly execute the code. In addition, we can also verify the code to check if all the proofs are valid. As we discussed earlier, Dafny also provides several options for target languages. We chose C#, JavaScript, and the Go runtimes for our evaluation. The C++ runtime for Dafny has been abandoned and is no longer usable.

Input selection We computed the median size of packets for PFCP and RTPS using datasets containing over one thousand packets each of both these protocols. We found that the median packet size for RTPS was 120 bytes and for PFCP it was 35 bytes. We set these packet sizes for our performance evaluation.

Lines of code After running the Dafny code generator for three programming languages, we computed the lines of code generated for our ParseSmith input. Table 6.3 showcases the results of our study. We found that Dafny generates the fewest code for JavaScript and the most lines of code for Golang.

[4] `https://github.com/viperproject`

In addition, we also found that the Dafny compiler does not generate code from separate Dafny source files into separate target code files. Instead, it places the entire codebase in one file in the target language. This pattern significantly complicates debugging.

Format	**Dafny**	**C#**	**JavaScript**	**Golang**
RTPS	230	5350	2286	10486
PFCP	107	4378	2224	4760
ParseSmith Library	727	-	-	-

Table 6.3: Lines of code for different target languages in ParseSmith

Task	**Go version**		**JavaScript version**		**C# version**	
	Memory (KiB)	**CPU time (s)**	**Memory (KiB)**	**CPU time (s)**	**Memory (KiB)**	**CPU time (s)**
RTPS						
Verification	177201.16	14.76	176768.36	14.52	219812.52	18.67
Verification + CodeGen	176826.60	14.81	176681.04	14.54	219899.36	18.66
Verification + Exec	177305.20	15.00	176899.64	14.63	222192.32	18.80
Verification + CodeGen + Exec	177480.68	15.03	177051.32	14.59	221912.56	18.80
Exec	42970.50	0.14	41180.56	0.14	46072.71	0.15
PFCP						
Verification	144815.52	10.90	144038.20	10.38	181891.40	14.58
Verification + CodeGen	144786.72	10.91	144116.08	10.46	181123.08	14.64
Verification + Exec	144869.16	11.04	144072.32	10.42	177710.36	14.68
Verification + CodeGen + Exec	144703.64	11.06	143973.88	10.50	181033.92	14.72
Exec	41764.12	0.14	41936.36	0.14	46156.72	0.15

Table 6.4: Comparing the performance of the Dafny-generated code for the RTPS and PFCP parser across target languages.

To make our performance measurements, we ran the Dafny executable with the configurations in Table 6.4 1000 times each, with different data packets of the same fixed size. These performance numbers reflect successful parses—our top-down algorithm can terminate earlier if it encounters errors. The default compiler option included in Dafny verifies, generates code, and then executes the generated code.

Memory Usage In Table 6.4, we see that the memory usage of our verification process is significant, but it is one time. Once we have run our verifier and generated code, we do not need to verify the code again. However, the memory usage of the

verification and code generation operations is over 170 megabytes for RTPS and over 140 megabytes for the PFCP packets. On the other hand, we find that the memory consumption during execution is only around 42 megabytes for Golang and JavaScript and around 46 megabytes for the C# code. We believe the memory consumption is reasonable for a scaffold-based algorithm where we use three scaffolds to prove strong guarantees.

CPU Time We note that the JavaScript and Golang version of code consume a lot lesser memory and take slightly lesser time than the C# version of the code. We believe this results from the Mono runtime used in Linux, which may be slower than the native code generated by Golang or the Nodejs runtime.

Section 6.7

Conclusions

This chapter presented ParseSmith, a parsing toolkit that can handle real-world data formats. To build ParseSmith, we first designed Parsley grammars and described two network protocol formats in the Parsley grammar syntax. We then designed two algorithms to parse data in the Parsley grammar syntax.

We implemented ParseSmith in Dafny to verify the correctness and termination properties we were interested in. For all well-formed Parsley grammars, we can prove the traversal of the AST produces the original input, the algorithm is implemented correctly for all the conditions in the grammar, and the algorithm terminates for any input.

We demonstrated and evaluated ParseSmith using two case studies: the RTPS and the PFCP protocol. We found that the verifier and the construction of three scaffold tables are very cumbersome and memory intensive. The issues we encountered in our

experience with ParseSmith leads us in several future directions.

Runtime interpreter for Parsley DDL Currently, we modeled our input grammar language to resemble Parsley grammars closely. Creating a compiler and an interpreter for the entire Parsley language still requires some effort. The Parsley language supports extensive cursor and view operations that we do not support yet. The PEG-based algorithms we adapt in this chapter are not amenable to view and cursor modifications.

Backend for Parser Combinators Another potential future direction is to use the ParseSmith interpreter as a backend for the Hammer or Nom parser combinator toolkit [66, 214]. Unfortunately, neither of these parser-combinator toolkits are verified for correctness or termination. Hence, a way to invoke ParseSmith via parser combinators would be a good addition to Hammer and Nom.

Designing more efficient algorithms As we noted in the evaluation, one of the significant issues with our two algorithms is that they have significant memory requirements [199]. For most network protocol formats, simple recursive descent parsing algorithms with backtracking would suffice. Hence, we believe that supporting a simpler algorithm as default would significantly improve performance and lower memory requirements.

Chapter 7

A Systematic Comparison of Data Description Languages

Earlier, in Chapter 4 we studied the parser-combinator approach and built several parsers for network protocols and file formats. Later, in Chapter 6 we build a parsing toolkit that can generate code in multiple languages. This chapter studies Data Description Languages (DDLs) in detail. DDLs present one of the most comprehensive methods to enforce various LangSec principles. More importantly, by generating parsers in multiple languages from the same source, we minimize parser differentials. This chapter aims to systematize knowledge about state-of-the-art DDLs.

DDLs are a means to describe data formats such as network protocols and file formats. DDLs often involve three components: the description format, the parser generator, and generated code. Developers describe grammars in the description format and use the parser generator to generate executable code. Figure 7.1 shows how developers interact with these DDL and the parser generation tools. Typically, DDL users are developers.

Although some DDLs can also be used to describe programming language structures, they are relatively domain-specific and designed to capture properties in net-

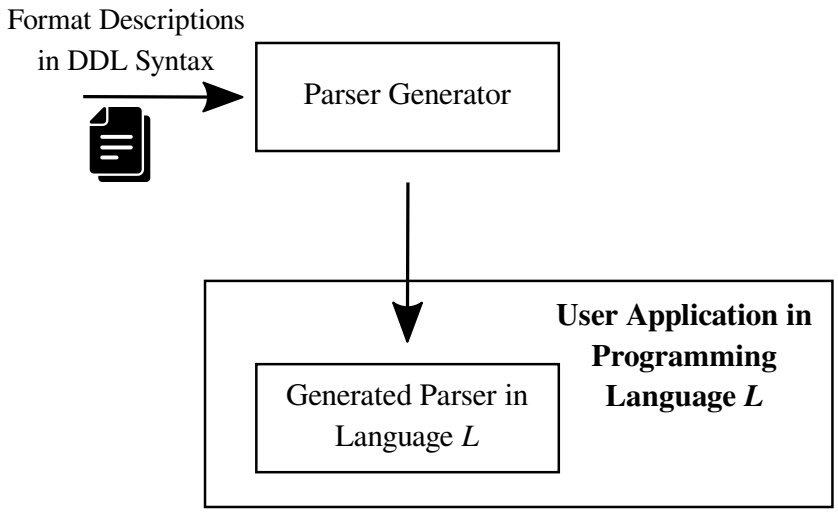


Figure 7.1: An overview of how developers interact with DDLs and their corresponding parser generation tools.

work protocols and data formats. Network protocols are often driven using an overall protocol state machine to ensure that the correct parser is used for the expected packet.

Similarly, network protocols also include repeat constructs or length fields. For example, in the DNS protocol, the header specifies a question count (QDCOUNT), a 16-bit value [271]. The header is followed by QDCOUNT number of questions in the DNS packet. Such constructs are not easily represented in formal grammars.

The tag-length-value (TLV) construct provides another similar example. In these constructs, a tag is followed by a length l, and l number of bytes follow. Unfortunately, such TLV formats are also not representable in formal grammars (Chapter 6). Therefore, any dedicated format used to describe network protocols must support at least these two formats to capture binary network protocols successfully.

On the other hand, file formats use different constructs to describe data. For example, let us consider the Portable Document Format (PDF) [216]. To parse a PDF file, we scan the file backward from the end of the file till we encounter a `%EOF`

tag. We then locate a cross-reference table (xref table). This xref table holds the locations of the objects present in the PDF file. The parser must then jump to each of these offsets and parse the PDF objects present there.

Using offsets and seeking to various locations is extensively used in other file formats such as TIFF, ZIP, and ICC (Chapter 4). Hence, the parsing library must control the *cursor* and provide support for creating smaller parsing buffers. The parser cursor needs to be moved to parse additional constructs at another offset in the file. To this end, several DDLs were created to tackle various constructs in data formats. Most DDLs can support the TLV format and repeat constructs. Support for controlling cursors and seeking to offsets in the files are much rarer features in comparison.

The primary object of using DDLs and parser generators is to reduce the parser code developers write by hand. Using a code generator that has been extensively tested, we ensure that common human errors can be minimized. There are two relatively popular DDLs—Kaitai Struct [295] and DFDL [181]—and several other relatively lesser-known ones, such as PADS/ML [178], SCL [225], and RecordFlux [228]. DFDL has several applications in military domains where serializing bytes from the parsed syntax is critical. Unfortunately, the other DDLs do not offer serialization functionality. However, developers often design their own domain-specific language to parse a format of their interest rather than rely on general-purpose DDLs.

This chapter describes an expert elicitation study we conducted to understand various features that must be present in usable DDLs. First, we transcribed interviews with 15 experts who answered questions about DDLs they built or DDLs they used. They also spoke about what features they expect to see in any usable DDL. Finally, we propose using the list of features provided by various experts to categorize these DDLs.

Next, we compare various DDLs based on several factors. First, we compare them based on the data format features they support. For example, we describe whether a DDL can support network protocols or file formats. This comparison provides insight into which DDLs can be used to describe complex formats.

Last, we also evaluate DDLs and their parser generator implementations using provided image format descriptions. As shown in Figure 7.1, the generated code parses the input based on the format descriptions provided in DDL syntax. Kaitai Struct and DFDL provide descriptions of various image formats such as JPEG, GIF, BMP, and PNG as a part of their format galleries. We generate code using these descriptions as input and perform two sets of evaluations on the parser code generated.

(a) We measure the *accuracy* and *correctness* of the descriptions provided for various image formats. Then, we use a large corpus of image files to compare the results of the DFDL- and Kaitai-generated parsers with open-source image processing libraries available in Python.

(b) We fuzz the DFDL- and Kaitai-generated parsers for all the image formats using AFL++ [94] and apply various static analysis tools to point out weaknesses in code.

My contributions in this chapter are as follows:

– First, my collaborators and I conducted the first expert elicitation study to understand the design of data description languages and parser generators. We believe that this study can aid future data description language developers to understand vital features in DDLs and design decisions made by previous developers (Section 7.1).

– Next, we produce the first taxonomy and categorization of multiple DDLs based on common DDL properties and supported features (Section 7.2).

– Finally, we compare implementations of various DDLs and their corresponding parser generators by conducting a case study of several image formats (Section 7.4).

Related Work. Levillain et al. [166] present a preliminary design for a platform to compare binary parser generators. This work proposes comparing these parser generators' efficacy, robustness, and expressiveness. However, they do not evaluate their technique against one of the most popular data description languages available—DFDL. Also, this paper only focuses on the DNS protocol and does not tackle any file formats. They measure the time taken by the parser to evaluate performance.

This chapter goes beyond the work of Levillain et al. by conducting a user study and picking file format descriptions and network format packets to evaluate and fuzz the DDL implementations. They use Nom and Hammer that are parser-combinator libraries and not DDLs. We do not include these libraries in our evaluation because of two reasons. (1) We are primarily focused on DDLs. (2) These parser-combinator libraries do not provide sufficient documentation or examples to describe file formats—they focus primarily on network protocols.

Section 7.1

Expert Elicitation Study to Understand Important Features[1]

Data formats, such as PDF, have complex structures that cannot be described easily using formal language theory. Section 2.6 and Section 5.2 provided additional information on such complexities in data formats. Nevertheless, there are at least two popular DDLs that programmers extensively use to describe these data formats. Therefore, we wish to understand which DDL features affect usability.

[1]For this work, I collaborated with Vijay Kothari, Ross Koppel, and Sean Smith.

7.1.1. Objectives

This section presents a novel and first-of-its-kind expert elicitation study solely focused on getting the perspectives and thoughts of experts who create specifications, develop new features for DDLs, or implement parsers for various protocols and file formats. We started this study with various goals in mind.

First, we would like to gather a consensus on what features must be present in a DDL. Creators of DDLs have made different design decisions—included some features, excluded a few others—we gather reasons for some crucial design choices made in these DDLs.

Second, we investigate whether there are certain features or patterns in DDLs that make them easier or harder to use. For example, DDLs often use a general-purpose data-encoding format such as XML or YAML to encode the format. However, developers may not be very familiar with tools that make it easier to describe and use XML schemas.

Finally, we want to understand what resources developers look for when they want to start using a DDL. Participants provided insight into what resources they provide as part of their DDLs or what resources they find helpful when trying to pick up something new.

In the rest of this section, we describe our methodology and summarize the results from our study.

7.1.2. Recruitment

We use the Snowball sampling technique [104, 56]. In this approach, we begin with a small pool of participants, and each participant is asked to suggest additional participants. Typically, this technique is used to make statistical inferences about relationships in the population.

However, we use this approach differently. We first approached five data format and parser experts and sought their answers to our questions. At the end of our interview, we asked them to suggest more people to interview. Each interviewee suggested 1-3 participants who we then contacted with information about the study. We totally contacted 35 people and 15 of them participated in our study.

Minimizing community bias

The first five participants we chose were from different industries and fields. We made sure that they did not know each other, and if they were academically inclined, we made sure they published in different subfields. These five participants were experts concerning different parts of the specification writing to parser development life cycle of data formats. They, in turn, provided contacts of participants who mainly worked in their closed subfield. This approach ensured that we got participants for a wide range of subfields.

Participant Demographics

Table 7.1 provides participant demographic details. Most of our participants were software developers or researchers in the industry who worked on parsers and data format specifications. Based on their answers to our question asking them to describe their role relating to DDLs, we see that most of our participants have implemented parsers for file formats. In contrast, a few have worked on standardization committees to develop specifications and create new parsing algorithms better suited to data formats.

Our participants also have varying education levels. For example, two-thirds of our participants did not have a Ph.D. Our participants also demonstrated varying levels of experiences with parsers. For example, we had users building a DDL for

15 experts		
Professional Role	Government:	1
	Industry:	12
	Academia:	2
Roles pertaining to DDLs	Writing Specifications:	2
	Writing parsers for a specific protocol/file format:	13
	Developing better parsing methodologies:	4
	Other:	2
Education	Completed high school:	1
	Completed College:	5
	Completed Masters:	4
	Completed PhD	5
Experience pertaining to parsers	30+ years:	5
	25-30 years:	2
	20-25 years:	2
	10-15 years:	3
	Less than 5 years:	3

Table 7.1: Summary of participant demographic characteristics. The rightmost column shows the number of participants who fall under that category.

their use case last year, and they had not encountered DDLs and parser combinators prior to that. On the other end of the spectrum, we had participants who have focused on defining specifications for file formats and network protocols and parsing data formats for over 30 years.

7.1.3. Ethics and Anonymity

We are committed to ensuring that the benefits of this expert elicitation study outweigh the privacy risks posed to the participants. We have taken steps to ensure the anonymity of our participants by removing any references to their employers or projects directly. All our quotes are accounts of participants documenting their experiences interacting with or building DDLs. We altered some word choices to ensure that unique word choices would not reveal the participant's identity.

We believe that these experts building DDLs could use findings from our study to improve upon their tool to provide more tools. Also, improve the usability and accessibility of their tools by using insights from our study.

Our meetings were recorded locally using Zoom. We transcribed our meetings and anonymized any references they made to their employer or affiliation. We are storing our transcripts and audio recordings to comply with GDPR. Dartmouth's Committee for Protection of Human Subjects (CPHS) approved our study on April 10th, 2021. As part of CPHS's requirements, we needed to provide participants with an information sheet with information about the study and our practices. Additionally, to comply with GDPR, we needed to provide European participants with an additional information sheet with compliance information. Each user was compensated for their time using a $30 Amazon gift card.

7.1.4. Approach

Before starting our semi-structured interviews, we asked several research questions that we wanted to explore. Parser experts have made several unique contributions to parsing algorithms, methodologies, and data description formats. Therefore, we designed questions to help us document their insights, design choices, and knowledge in general.

Research Questions

RQ1: What are aspects in DDLs experts like? We asked participants to list DDLs they have used and then talk about the aspects of these DDLs that they liked. Most participants went beyond the list of features we had accumulated. Instead, they described features they desired in DDLs but have not found any that matched their requirements.

RQ2: What are aspects in DDLs experts dislike? Similarly, we asked participants to describe any DDL features that they disliked. We prompted the users to discuss difficulties with the language and the tooling built for parser generation.

RQ3: What learning resources do experts think are necessary? We provided participants with a list of learning resources such as tutorials, IDE integration, descriptive errors, and Web IDE support. Then, we asked them to specify which of these resources they find valuable and necessary. Some of these learning resources, such as Docker containers and Web IDE support, aim to provide a simple way for someone new to try the DDL in an accessible setup environment. Other features such as IDE integration, error reporting, and tutorials may help users dive into specifying formats using the DDL by easing the learning curve.

Hypotheses formed

We also created several hypotheses informed by our research questions and our experiences with DDLs and parsers. We list our hypotheses and describe each of them.

H1: Experts would like to have control over underlying parsing algorithms. Parsing algorithms differ in the class of grammars they accept and the space and time complexity. For example, the Packrat algorithm is designed to accept parsing expression grammars (PEGs) and runs in $O(n)$ time and consumes $O(n)$ space. On the other hand, the CYK algorithm can recognize arbitrary context-free grammars and runs in $O(n^3)$ time while consuming $O(n^2)$ space.

Some tools, such as Hammer, allow developers to switch between these parsing algorithms. Given that different algorithms can recognize different classes of grammars, we set the hypothesis that developers want to describe a data format and prescribe a particular parsing algorithm to be used with it.

H2: Experts prefer dedicated data description languages over general-purpose data encoding languages such as JSON/YAML/XML. The two popular DDLs—Kaitai Struct and DFDL—both rely on general-purpose data encoding

standards (YAML and XML, respectively). However, other languages such as Nail, RecordFlux, and PADS/ML use dedicated syntax that a developer must learn from scratch.

Given that describing a data format in a general-purpose data encoding standard can make the syntax verbose, we hypothesized that developers would not mind learning new syntax to achieve their task of parsing a data format.

H3: Experts expect a "proof of termination" in DDLs. DDLs are used in conjunction with code generation tools. Developers can then use the generated code to parse untrusted input. Most such programs do not provide any guarantees that a given function will terminate. However, using various formal methods tools (such as the one we used in Chapter 6), we can argue that provided a set of preconditions are met, we can prove that a function will terminate.

Such guarantees are pivotal to a program's correctness. For example, any parsing function can take several branches, and each branch can lead to various paths. A formal "proof of termination" ensures that the function always terminates irrespective of the path taken. We hypothesize that developers require such guarantees in generated parser code.

H4: DDLs must support module systems. Module systems allow users to build data-format specifications compositionally. There are two major applications to a module system. First, a specification in a DDL could get extraordinarily large if the format is complex. Second, some complex formats, such as PDF, contain several other embedded formats. A PDF file relies on correct implementations of image format parsers, font parsers, Javascript parser, string encoders, and compression algorithms.

A module system would let developers import other previously defined specifications to simplify an overarching specification, such as PDF. Hence, we hypothesize

that developers would think that a module system is critical to any DDL.

H5: Parsers must implement zero-copy parsing. Zero-copy operations help improve the performance of any software. A zero-copy parser would not copy any parse buffer content from its original location. Instead, it would directly operate on the original parse buffer to provide the security guarantee—the untrusted input is not being copied multiple times in the program's address space. Also, it can improve performance since copying memory is a time-consuming process [59]. We hypothesize that zero-copy parsing is a crucial DDL feature given these two properties.

7.1.5. Qualitative Codebook

Our interviews resulted in 13 hours of audio recordings that were transcribed to 40 pages of transcripts. These transcriptions were done manually. Three of our participants said they preferred a chat-based or email-based interview. We accommodated the requests of these participants and had them participate in our study. For the email participants, we sent another follow-up email to clarify any confusing answers they gave.

We used an inductive approach to analyze our data. Since our interview was semi-structured, we extracted specific codes for each question in our interview (for example, "XML is verbose," "roundtripping from grammars to parsers," and "serialization."). As a result, our analysis of the transcripts resulted in over 50 codes across all questions.

7.1.6. Results of our expert elicitation study

Hypotheses

After transcribing and coding our interviews, we analyzed our data to test our hypotheses. Table 7.2 summarizes the results of our study. Of our five hypotheses, only three were valid based on our interviews. However, the experts also provided more profound insight into why they thought two of our hypotheses were incorrect.

Hypothesis	*Result*
H1: Experts would like to have control over different underlying parsing algorithms	False
H2: Experts prefer dedicated DDLs over general-purpose JSON/YAML/XML	True
H3: Experts expect a "proof of termination" in DDLs	True
H4: DDLs must support module systems	True
H5: Parsers must implement zero copy parsing	False

Table 7.2: Results of our hypotheses. This table shows the hypotheses that evidence we collected support.

H1: Experts would like control over parsing algorithms *(False)*. We found that experts did not find this factor very important in DDLs. For example, Participant $P1$ said,

> *I wouldn't necessarily think about it like that. PDF, SVG, we are switching between different parsing algorithms. But we are switching between data streams. Our code might call out to another library and calling out things. I don't think worrying about which type of algorithm is that important. We do it sort of accidentally. ($P1$)*

As we noted earlier, different parsing algorithms are used to recognize different classes of grammars. By supporting different parsing algorithms, we inherently support different languages classes and assume that the syntax is at the intersection of

these language classes. For example, as $P2$ notes, a programmer may not notice how the behavior of the underlying parsing algorithm has changed unless they run into corner cases.

> *Other than to compare formats, why should you? It sounds like a nice thing to have, but usually you just settle on one and set it forever. As long as you can do it transparently, it is great. Packrat and CYK parse different language classes, if you switch out then you need to make sure that you are operating on the intersection of the two. The programmer may not realize that they switched from ordered choice to unordered choice. As a testbed for underlying algorithms, it is great. ($P2$)*

Since most developers do not have a deep understanding of the language classes and their corresponding algorithms, it might be wise for a DDL to stick to one as default (Hammer uses the Packrat backend as default). However, only provide other algorithms as options for more advanced users who know what they are doing. $P4$ said,

> *Leave other options apart and stick to one. Most developers don't understand the meaning of these algorithms. ($P4$)*

Most data formats use sequential structures that can be recognized using straightforward parsing algorithms. Moreover, for most grammars, these parsers can run in linear time. Some experts, such as $P5$, noted that they have never had to use anything more complicated than a recursive descent backtracking parser.

> *For most binary formats, you can just use a recursive descent backtracking format. You shouldn't need anything more complicated. So I wouldn't say this is super important. ($P5$)*

H2: Experts prefer dedicated DDLs over general purpose syntax *(True)*. We found that this hypothesis was valid. Even though DFDL currently uses concrete

XML syntax to describe formats, developers expressed interest in taking it towards a concrete syntax based on a domain-specific language. In addition, the verbosity of XML, JSON, and YAML was a common complaint among our participants. Participants $P6$ and $P2$ said,

> *I find XML schema, as annoying and verbose as it is, I am used to it now. The future of DFDL is that it is going to be a domain-specific language. It is going to be its own thing with its own syntax. It can really be specific towards what it needs to be and nothing else. (P6)*

> *I prefer that the specifications should be in some easy to understand or slurp in format. But what I've noticed in my work is that people want a concrete syntax and an interface where people can type in a data structure which is indeed a data description. For users you want a concrete syntax, but for underlying representations we need some abstract syntax. (P2)*

Using general-purpose serialization formats to describe data formats give programmers two opportunities to make errors. First, errors can occur in the data-encoding formats, such as YAML/JSON/XML. Second, errors may occur within the correctly encoded general-purpose serialized data. For example, they may miss mandatory keys or incorrectly nest structures. Using a dedicated language with its syntax to describe the data format means that we must build a compiler to describe this format sufficiently. Participant $P15$ explained this by saying,

> *I personally prefer dedicated languages. Composition is really important in defining parsers. To me, XML and JSON aren't really all that well situated. Everything kind of blurs together. XML schemas have their advantages. But most of the bugs that arose out of that were basically just because an element that was supposed to be nested in one place got accidentally nested in another place. That ended up producing errors later on in the pipeline. The verbosity is not only bad with XML, but really anything in Java. (P15)*

On the other hand, a minority of experts insisted on using general serialization

formats since they can use parsers for that format in any language to extract the syntax. Participant $P14$ said,

> *I prefer DDLs based on a general serialization formats, because it greatly simplifies interfacing with custom tools. I think of such grammars not as of "grammars based on YAML", but as of "grammars based on other grammars". (P14)*

H3: Experts expect a "proof of termination" in DDLs *(True)*. Experts believe this is a crucial feature irrespective of the language class and the corresponding parsing algorithm. If a parsing toolkit advertised this feature, developers could assume that this property holds across multiple algorithms they can invoke using the toolkit—and not be bothered by the fine print. As participant $P4$ notes, it is important not to have a switch to disable the proof of termination, as developers may create code without this guarantee and not realize it.

> *If I assume that the very secure parser always terminates, I will assume it as a developer. You have to ensure this. Don't give them the option to switch termination on or off. They will create insecure parsers and forget about it. (P4)*

Similarly, participant $P2$ notes that proving termination is primarily a formal and academic experience. However, defining termination for streaming formats—where data is not available all at once but in chunks—is a research problem.

> *Yes. I can see that when it is possible it is important. It is a formal experience. When you do stream parsing, you do not have proof of termination. But if it is possible, it should be there. (P2)*

It is critical to have deterministic algorithms that terminate for the set of expected input and reject the unexpected input. Participant $P5$ said that their government clients do not allow them to add any new features to their parsing toolkit that would remove the proof of termination guarantee.

> *It is certainly something important to our government customers that we can ensure that things always terminate. Customers have also said that we cannot make any changes that will remove the proof of termination. (P5)*

H4: DDLs must support module systems *(True)*. Importing different DDL specifications to construct a larger specification is an essential factor. Our hypothesis was valid—almost all the experts agreed that module systems are vital in any DDL. Some experts who primarily work on network formats and not file formats noted that they have not have run into this issue.

Application-layer network protocols do not commonly embed other formats in them. However, in some rare cases, such as using the real-time publish-subscribe protocol (RTPS, Chapter 6), you can embed other image and video formats. Embedding other formats within a format is a much more common occurrence in file formats. Participant $P2$ said,

> *We haven't needed it. But it is an issue in computer science when you scale. You want to reuse code and not have multiple different copies of the same thing. (P2)*

Programmers often apply the paradigm Do not Repeat Yourself (DRY) [26]. Hence, reuse and composition are important to developers. Participant $P4$ who has interacted with many developers notes,

> *Reuse is always important, but for DDLs it is even more important. For example, reusing UTF8 can be nice. (P4)*

H5: Parsers must implement zero-copy parsing *(False)*. We found that this hypothesis was not valid. Experts do not believe that zero-copy parsing is an important factor in the generated parser code. Although when we asked this question, we believed that the security and performance benefits of using zero-copy parsers

would outweigh not having it. We found that most experts only associated zero-copy parsing with its performance benefits. For example, participant $P2$ said,

> *We should strive for zero-copy recognition. Minimal copy is desirable. Parsing of any language class requires a stack. Is a stack copy? Can you run it in constant space? That is a criteria that is useful I guess. ($P2$)*

Participant $P2$ draws an interesting difference between input recognition and input parsing. They claim that checking if the algorithm runs in constant space is a more useful metric. Similarly, participant $P5$ said,

> *I don't think that is actually a requirement. The requirement is speed. This is one way to try to achieve that speed. Some people will say this is the only way that I can get the speed, nothing else can be fast enough. ($P5$)*

> *DFDL does not do it at all. There is a lot of copying going on. That is one of the biggest performance issues. It is not necessarily critical. ($P6$)*

Parsers for context-free grammars (CYK, Earley, etc.) and parsing expression grammars (Packrat) often rely on memoization tables to recognize input. These memoization tables help reuse previously computed values. However, it is unclear if zero-copy implementations of these algorithms are feasible. Participant $P15$ added,

> *Packrat requires so much memoization that the only way to do zero-copy packrat would be to throw pointers to the input stream everywhere. Which I think is kind of dangerous. We found that hashing functions are far more expensive that doing copies. ($P15$)*

7.1.7. Detailed results

Data Description Languages participants used. Although we were familiar with the most commonly used DDLs and some lesser-known academic work, it was enlightening to ask experts to list all DDLs they have used. In addition, we learned about several other DDLs that we were not familiar with. For example, although we

were familiar with the Flex/Bison tools used in parsing programming languages, we were unaware that developers still used these tools to parse data formats.

Another toolkit we were unaware of as being used to recognize data formats was Antlr. Antlr provides robust tools to generate parsers in various target languages. Synalysis provides an interface to visualize data formats. For example, developers specify a grammar in XML syntax and provide data files as input. The Synalysis tool visualizes the data and dissects it into its correct fields.

> *I am also somehow familiar to Synalysis grammars (I have implemented a converter from them into Kaitai Struct specs). (P14)*

> *I'll just add that my best/most productive experience was with ANTLR up until using Daffodil. I've used DFDL. And I've used IDL (interface description language) a long time ago, for wire protocols. (P3)*

> *I would count Hammer a data description language. The BNF format of bison and antlr are data description languages. I've also used PADS. (P15)*

Aspects of DDLs they liked. Experts provided accounts of their experiences interacting with various DDLs. Participant $P1$ said,

> *I like ones that are very concise. You don't have to spell everything out. I guess I come with more baggage of existing knowledge and understanding. I also like DSLs that don't build in platform or language specifics..... That is where XML and DFDL fail me. Lots of tags and attributes and it is not succinct. Not really tied to how a human would read XML. (P1)*

Another participant ($P4$) states that they trust parsers that implement complex constructs like lookaheads. Lookaheads and backtracking are needed to parse certain classes of languages, and can be particularly hard to implement from scratch.

> *I always trust those that take care of special cases. I don't want to do those lookaheads and stuff by myself. (P4)*

Finally, DFDL users describe that they have used DFDL to describe both text and binary protocols and found the same ease of use while using it in either context. For example, text-based network protocols primarily use ASCII strings to describe protocol messages. In contrast, binary protocols may or may not be human-readable since they focus on using binary representations to encode data.

> *The aspect I like most about DFDL is that it's nearly an universal data format description language - it's powerful enough for both text and binary data formats. (P3)*

Aspects of DDLs they disliked. In contrast to the features experts liked, most experts focused on three categories of DDL features they disliked. First, experts spoke about the verbosity of some DDLs since they use XML syntax. Synalysis and DFDL both use XML syntax. For example participant $P13$ said,

> *I had known about DFDL a while before I discovered Kaitai Struct, but it completely overwhelmed me with its long-winded syntax with lots of keywords, attributes and namespaces, so I rejected it as unusable, compared to the brevity of the Kaitai Struct language. I tried to read the BMP specification in the DFDL language, but I can't clearly see the structure like I can in a .ksy file — I'm constantly distracted by the unnecessary ballast around. (P13)*

Second, code generation is a hard problem. DDLs often include code generation to generate usable parsers. Since such code generators are not as thoroughly tested as compilers, their errors may be harder to surface. Using a code generation step means that the generated code and the original specification essentially implement the same thing, and the code generation process is correct. To this effect, participant $P4$ said,

> *You have two places that is the truth, not just one place. The problem is that the generator has bugs. No one says that a compiler has bugs. You don't really care about compiler bugs, because they get fixed all the time. (P4)*

Finally, some formats may not be very appropriate to be parsed using DDLs. One common format many experts discussed was PDF. Several experts believe that it is very difficult to capture the syntax and semantics of PDFs using DDLs, and hence, we must not set the bar so high.

> *Some formats cannot really be parsed with DDLs. ($P14$)*

Bidirectional change propagation from DDL to programming language and reverse. In our interview with participant $P1$, they mentioned that they would like to see a DDL supporting bidirectional change propagation. Often, the generated code contains bugs. Fixing these bugs directly in the generated code must also lead to the bug fixes in the DDL specification. Bidirectional programming presents a mammoth research and engineering challenge. However, we asked subsequent participants what they thought of this feature.

We found that DDL creators do not envision that users would be making changes to the target code. Instead, if they spot a bug, the expectation is that the users would fix the bug in the DDL specification and regenerate the code. Participant $P6$ described this saying,

> *The output of our specification compilation is not human readable or human understandable in any way. Generated code is usually pretty complicated, I would not want someone modifying that. I would be concerned that the modifications are wrong and have security implications. Every time someone needs to make a change, they would change the specification, and regenerate code. ($P6$)*

As the participant notes, the generated code is often not human readable, and hence not an appropriate candidate to edit. Therefore, experts believe that a developer must resort to changing the target code only as a last resort. Participants $P13$ and $P14$ say,

> *I don't think that Kaitai users make changes to the target code too often, they usually try to avoid it and take advantage of the automatic compiler generation. Editing the generated code is of course a possibility when the automatic parsing code would be limited by some missing Kaitai Struct features, but I believe it should be the last resort. (P13)*

> *I guess instead the language should be designed the way eliminating the need in such changes completely. (P14)*

As more developers interact with a DDL and use additional DDL features, we will uncover more errors and corner cases. Although thorough testing and formal methods tools can help uncover some of these issues, they will not eliminate them.

Serialization. Several participants discussed that serializing or unparsing data is critical for them in DDLs. As a participant noted, although modifying the intermediate AST to redact certain information has its uses (also seen in Chapter 5.7), there may be certain cases when serializing intermediate parsed data to raw bytes may still be necessary.

> *Another thing interesting in DFDL is the unparse option. We can modify an infoset and unparse to bytes. The inability to round-trip that data is a non-starter for us. We get data that is untrusted coming from a foreign country. We parse that data to an infoset. We remove the data we consider a problem or dangerous. We then write it back to bytes. We have data generated by the American military. We want to share this data with some allies. We want them to know within a single latitude-longitude precision, but not exactly where the plane is. We send the redacted data to our allies. They don't know it has been converted. We've fuzzed the data a little bit in a way that they don't know. Redaction is the another common use case. Another case is when we take data from an unclassified domain to a classified domain—the only way to take the data across the domain is through XML. We send a parsed XML across the secure channel. On the other end we unparse it back to the original byte format. XML is an intermediary language here. (P6)*

> *The ability to reuse the same spec for generating libraries for multiple programming languages, and the ability to map AST nodes to raw bytes, allowing creation of rapid prototyping and analysis tools. (P14)*

On the other hand, a network filter that parses and serializes data can be susceptible to covert side-channel attacks where people can observe the packets before and after the network filter to infer security rules. Additionally, attackers can observe how long a network filter takes to change the data and serialize it to find the network's bandwidth.

> *Data can be used to exploit all kinds of security holes. The data can crash a system, its not malware... Just causes a crash. It is still bad. A lot of people don't want the network security to change the data. A secure system will canonicalize the data and remove leading zeros. Suddenly, you could have a covert channel where people can figure out the bandwidth. (P5)*

Although the above two participants ($P5$ and $P6$) mention that a DDL without a serializer is a nonstarter for their use-cases, they are also cognizant of the security risks posed by an AST modifier tool.

Key properties. We asked participants to talk about each of the factors in Table 7.3. We covered the proof of termination, module systems, control over parsing algorithms, and zero-copy parsing results in the hypotheses. As shown in the table, we found that experts support code generation in many languages. Programmers look for code generation support in the language of their choice. All experts mentioned that it would be good to support many languages.

Although most experts mentioned that they expect the code generator to produce memory-safe code, a minority of experts mentioned that they do not expect a code generator to do this. The developers need to test their code to find such issues thoroughly.

Finally, experts do not believe DDLs need to support arbitrary regular expressions. DDLs usually support syntax that goes beyond regular expressions. Developers need to transform their regular expressions into something that the DDL can accept. Four of our study participants mentioned that a subset of regular expressions could be good to include, but the performance costs can be pretty massive.

Feature	**Vital**	**Good to have**	**Not important**
Proof of termination	10	0	5
Module system	12	3	0
Code generation in any language	10	5	0
Memory safety	12	0	3
Arbitrary regular expressions	0	4	11
Switch between parsing algorithms	1	0	14
Zero-copy parsing	1	3	11

Table 7.3: Summary of technical features from our study. The columns show number of participants who said a particular feature was vital, good to have, or not important. We prompted each participant to comment on every feature.

Asynchronous operations such as jumps and offsets. Asynchronous operations such as jumps and offsets pose a tough problem. We asked experts if they prefer specifying these operations within their DDL or would they rather call these operations from within a programming language. The second approach would involve specifying only portions of the data format using the DDL and relying on the programming language to glue the offsets together. We found that of the 15 participants, two did not have an opinion on this. Additionally, two participants said they would keep these operations in the programming language. Finally, the remaining 11 mentioned that they expect the DDL to support these definitions.

Participant $P13$ mentions that the stateless principle in Kaitai Struct has been sufficient to describe data formats thus far.

> *It would be nice to have, but the simple stateless concept in Kaitai Struct is usually enough for the vast majority of purposes. There are only very specific*

> *cases where you would truly need synchronous operations. I also haven't heard of any language supporting this. (P13)*

Participant $P14$ added that they do not have a strong opinion on this but mentioned that you could achieve this by modifying the Kaitai Struct runtime. The Kaitai Struct runtimes are in programming languages of a developers' choice. So, essentially, this proposal is the same as relying on a programming language for these operations.

> *I have no opinion on that since I don't write apps based on event loops much. In the case of KS I guess that this can probably be achieved by customizing the runtime. (P14)*

Participant $P15$ defined jump and offset operations as a semantic operation and not a syntax check. They mention that they prefer keeping their semantic operations in the programming language.

> *This is a tough question because it is right on the boundary between syntax and semantics. It is much better to keep it in the programming language. I would like to say that I want to keep my semantics in the programming language. Jumps and offsets are semantic operations. (P15)*

Feature	**Vital**	**Good to have**	**Not important**
IDE support	10	3	2
Docker containers	0	2	13
Tutorials to build real-world formats	8	2	5
Web IDE/compiler	0	5	10
Descriptive syntax and type errors	15	0	0

Table 7.4: Summary of our learning curve and usability factors. The columns show number of participants who said a particular feature was vital, good to have, or not important. We prompted each participant to comment on every feature.

Learning curve and usability. We formed a list of usability features informed by existing DDLs we came across. Then, we asked participants to speak about what

they think of each of these features. Table 7.4 summarizes the participant responses for various usability factors we put forth.

Most participants rated IDE support as either "good to have" or "vital." The two participants who said this was not important mentioned that there are commercial tools and IDE tools available to write better XML and YAML. For example, participant $P3$ said,

> *You can buy commercial off the shelf tools to make writing XML Schemas easier for DFDL. ($P3$)*

Docker containers. Docker containers provide an easy interface for new users to try a DDL and explore its features. Most of our participants agree that Docker containers are not very important in DDLs. It is more important to have an easy build system and ensuring that the software can compile easily on most systems without much difficulty.

Tutorials. Our participants differentiate strongly between tutorials, reference manuals, and examples of real-world formats. We found that although most participants mention that these tutorials are important, they also believe that the other resources are vital.

Most participants believe that all three of the above resources are important from an onboarding point of view. In contrast, some participants believed that tutorials are not very important if you had a comprehensive reference manual and real-world examples. For example, participant $P4$ said,

> *Just show developers some example code, and include a proper documentation for all the standard libraries. You don't need to write tutorials. ($P4$)*

Web IDE/compiler. We found that most participants did not think a Web IDE was useful. However, some users said that it could be handy from an onboarding perspective when some people want to try the DDL out without installing it.

Descriptive errors. All our participants overwhelmingly supported this category. For example, participant $P13$ said,

> *It is inevitable that users make errors in the language, and it's great when the compiler can be a bit smart and assist at least in the most common cases of mistakes - try to guess how the error happened, provide a clear human-readable error message and even suggest a fix if it's possible. ($P13$)*

Descriptive errors are the backbone of any compiler. The user would have no idea how to fix an error if the error is not descriptive and does not accurately pinpoint where the parser syntax error occurred. However, the compiler only catches errors in the DDL syntax themselves. Therefore, another category of errors to tackle is the syntax and semantic errors in the data input. Parser libraries must support runtime debugging tools that can help identify root causes of errors in data concerning the DDL specifications.

Interesting errors introduced in parsers. We asked users to talk about some parser bugs they or their colleagues introduced while interacting with a DDL. The most exciting bugs fall into two categories. First, how thorough should a parser for data formats be?

> *The very first DFDL schema someone wrote for the JPEG format, they didn't really look inside the bytes of the segments. That is when we learned our lesson that not only do you have to accept what the specification accepts, but more importantly you have to reject what the specification rejects too. ($P5$)*

These DDL-generated parsers need to implement a format specification as closely as possible. In addition, a parser must accept well-formed input and, more importantly, reject malformed input that can cause various security issues.

Although various DDLs rely on generic data-encoding standards and use their (of the data-encoding standard) parsing libraries to parse specifications, these standards often follow rules that may not be very obvious to a new user.

> *Usual problems on Kaitai Struct are with not understanding YAML - people do not often realize that YAML doesn't allow duplicate map keys, or that they have to wrap a string value containing a colon ":" into the quotation marks. (P13)*

Using generic standards saves development time for the creators of the DDL. Likewise, it also makes things more accessible for people already well-versed with standards. However, for users unfamiliar with the rules of the data-encoding standard and the DDL, the learning curve is much harder.

Other interesting features. Finally, we asked participants if they could think of any other DDL feature that would make them want to try something new. Multiple participants mentioned that a comprehensive format library with many already-defined formats would be a feature. If developers are already familiar with a particular data format and its specification is already available as a part of a DDL, they would be able to incorporate the generated parser into their codebase a lot more easily. Also, developers can familiarize themselves with the syntax more quickly if they see formats they recognize.

> *If they could find schemas already written for their data formats, they would want to try a new DDL. (P3)*

> *A large standard library with a bunch of format implementations and protocol message implementations would be really good for someone to get started. (P15)*

Moreover, DDLs solve a very niche problem. Most programmers attempt parsing data formats without using any parsing tools—either relying on regular expression libraries or by parsing the format manually. Both these approaches have their drawbacks. Regular expression libraries are often overused and misused. It can be hard to implement all the constraint checks correctly and recognize the full syntax.

> *I was unaware of DDLs until a year ago. You need to have access and motive. When the problem comes along you look for DDLs. (P2)*

Parsing algorithms that memoize previous results consume space that is in the order of the input size. For example, the Packrat algorithm consumes $O(n)$ space. Let us consider a situation where this input of size n is in the order of gigabytes or terabytes. The parsing algorithm, in turn, would consume as much memory to memoize prior results. This nature of parsing algorithms presents a research challenge. We must support streaming formats, where the entire data is not available when the parser is called, but the parsing must be completed chunk by chunk.

> *I was playing with the idea of adding support for buffered/asynchronous reads. For example, to allow reading large files (like 4 GiB) over the HTTP protocol with range requests, transferring only the data that need to be....This feature alone would probably make me to want to try a DDL. (P13)*

> *A killer feature would be that a DDL is integrated into a major programming language. For example, I use Java a lot. Whatever DDL is integrated in Java, is the best for me. When developers choose a framework, they see what the programming language can give them. Then they scan the first page of the Google search. (P4)*

7.1.8. Discussion

Since we interviewed a diverse set of experts with varying background and expertise, we received results from our study that was also rich and diverse. We started with five hypotheses and three research questions for this expert study. Unfortunately, only three of our hypotheses were verifiable from input from our experts.

Emerging Trends. Although there were some outliers in our study results, we see that several exciting trends emerge that open up avenues for compelling research. First, switching between parsing algorithms is not considered an essential feature by our experts. We found two lines of reasoning for this when we probed our experts further. (1) Experts would like to consider parsers the same way cryptographic al-

gorithms are considered. *"Just the way we do not roll out our own crypto today, we must not roll our own parsers."*

Hence, we must not place the burden of understanding the algorithms on the developer and instead set good, fast, and reliable defaults. (2) Some experts do not see why this is useful. Differentiating parsing algorithms are more valuable to parse programming languages and natural languages. Our experts hypothesize that simple recursive descent algorithms should suffice for data formats.

Although memory safety stands out as an essential property on its face, this concern can be alleviated easily using memory-safe programming languages. Code generation is another property that is worth discussing. Most experts believe a DDL must support code generation in as many programming languages as possible. However, this presents a challenge. Developers usually only write code generators for the languages they would use. Supporting a large set of popular programming languages presents a mammoth engineering challenge.

Section 7.2

Categorization of DDLs based on Features

Before starting our study, we considered the DDLs of interest to be DFDL, RecordFlux [228], and Kaitai. To the best of our knowledge, these are the DDLs actively used in the industry. In our study, participants pointed us to Parsley, DaeDaLus, EverParse, ANTLR, PADS/ML, and Synalysis and told us they use these tools to parse data formats. We subsequently added these also to our list of DDLs.

We will now compare these 9 DDLs (DFDL, DaeDaLus [111], EverParse [226], Parsley [198], RecordFlux [228], Kaitai Struct, ANTLR, PADS/ML [178], and Synalysis) across all of the features extracted from our study. The features are separated broadly across advanced user features and learning resources.

Table 7.5 summarizes the status of these 9 DDLs we study. Only EverParse and RecordFlux include a verifier and produce code with property guarantees. Synalysis and PADS/ML are open-source projects that are not maintained as of May 3rd, 2022.

Languages	**Verifier**	**Active**
DFDL	✗	✓
RecordFlux	✓	✓
Kaitai Struct	✗	✓
Synalysis	✗	✗
EverParse	✓	✓
ANTLR	✗	✓
Parsley	✗	✓
PADS/ML	✗	✗
DaeDaLus	✗	✓

Table 7.5: Data Description Languages: which ones include verifiers of some sort, and which ones are actively maintained.

Languages	**Formats Supported**
DFDL	File Formats, Network protocols
RecordFlux	Network protocols
Kaitai Struct	File Formats, Network protocols
Synalysis	File Formats
EverParse	Network protocols
ANTLR	Programming languages
Parsley	File Formats, Network protocols
PADS/ML	Serialized Data (such as CSV)
DaeDaLus	File Formats, Network protocols

Table 7.6: Data Description Languages and the types of formats they support

7.2.1. Code Generators

DDLs traditionally include code generators (as shown in Figure 7.1 earlier). The generated parsers can be incorporated into an existing application in the target programming language. Hence, we studied the code generators and the languages supported by each parser generator of interest. Not every DDL had a corresponding code generator, but several DDLs supported code generators in multiple languages. Table 7.7

Languages	Languages Supported
DFDL	C, Java
RecordFlux	SPARK
Kaitai Struct	C++, Python, JavaScript, Java, Go, C#, Ruby
Synalysis	- (only visualization)
EverParse	C, F*
ANTLR	C++, Java
Parsley	- (only interpreter)
PADS/ML	OCaml, C
DaeDaLus	C++, Haskell

Table 7.7: Code generators that ship with DDLs and the languages they support

summarizes our findings.

Although DFDL supports C and Java, the runtime2 implementation of DFDL only implements a small subset of the entire language. In addition, we found that none of the image format descriptions available for DFDL complied with the runtime2 syntax for DFDL. Hence, although a C++ code generator is available, it is not fully functional. Kaitai supports the broadest range of code generators and format descriptions. The code generated by the Kaitai compiler is easy to use and can be incorporated into existing applications without much difficulty.

Parsley supports an interpreter model—where you can provide data and an input format description. The interpreter then parses the data based on the format description. The parsed data is available for an application to use in the form of a data structure.

Unlike Parsley and other DDLs, Synalysis uses only a graphic interface to provide the output of parsed data on a particular grammar. This GUI is helpful to visualize the fields present in data. It can also point out data malformations using error messages showing data failed to render. However, without any code generated and command-line output, the Synalysis tools are harder to incorporate into existing applications.

The PADS/ML project has been inactive for the last 11 years—the compiler and

language have not been altered. The PADS/ML project includes code generators for two languages.

7.2.2. Properties Proven

As we saw in Table 7.5, only two of the DDLs we studied included a verification component. In this section, we study the properties these verification tools prove.

RecordFlux RecordFlux provides an Ada-like syntax to describe message format specifications. The RecordFlux syntax primarily focuses on network formats. They use the `rflx` tool to generate SPARK [2] code with invariants that ensure strong properties such as lack of runtime errors [227]. They verify the following invariants using the `rflx` tool.

- Field conditions must be mutually exclusive and not contradict each other
- No field condition must be statically false, and every field must be reachable on some path
- The size of each field must be non-negative
- All bits in the message must be accounted for in the message description on all paths.

EverParse EverParse is a toolkit comprising a set of parser combinators written in Low* and a formalization of these parser combinators in F*. The Low* compiler, Kremlin, generates C code that is verified not to contain common bugs such as use-after-free, buffer overruns, and integer overflows. Additionally, the formalization in F* proves properties such as the parser and the serializer are inverse functions. Low-level protocol developers can implement parsers and serializers using EverParse by interfacing with the verified C code generated.

Most recently, in EverParse3D, the authors of EverParse have presented an extension of the original work while adding more combinators and applying it to the Hyper V virtualization framework in the Windows environment [267].

7.2.3. Grammar Syntax

We found a diverse set of grammar syntax as seen in Table 7.8. Most DDLs have designed their own syntax, not relying on traditional data storage formats such as JSON, XML, or YAML. However, the more popular DDLs, DFDL and Kaitai, use XML and YAML to store their grammars. EverParse uses grammars in BNF form that are defined in RFC documents. Synalysis uses XML as well but allows users to define functions in Python inside the definition files. Parsley and ANTLR adopt formal grammar-like notations to describe formats. Whereas DaeDaLus, PADS/ML, and RecordFlux have adopted an approach to specifying the fields in the format sequentially, rather than going for a syntax that looks like formal grammars.

Languages	**Syntax Format**
DFDL	XML/XSD
RecordFlux	Dedicated
Kaitai Struct	YAML
Synalysis	XML
EverParse	RFC Packet Descriptions
ANTLR	Dedicated
Parsley	Dedicated
PADS/ML	Dedicated
DaeDaLus	Dedicated

Table 7.8: Syntax used by DDLs

Section 7.3

Categorization based on Expert Study properties

In our expert elicitation study, we discussed several features of DDLs and the tools associated with them. Experts generally had an overwhelming opinion about specific properties, such as proof of termination and module systems. Similarly, experts also thought that good error reporting and IDE support are crucial for DDL usability.

This section categorizes DDLs and their associated tools based on whether they offer the features we discussed with experts. Unlike the previous section, where we discussed what languages and formats these DDLs support, we emphasize the expert features in this section. We discuss whether a DDL provides a particular capability or not. We separated our features into two categories: DDL features and usability features.

7.3.1. DDL Features

Languages	Proof of Termination	Module System	Code Generation	Memory Safety	Zero-Copy Parsing	Regular Expressions	Serialization
DFDL	✗	✓	✓	✗	✗	✓	✓
RecordFlux	✗	✓	✓	?	✗	✗	✗
Kaitai Struct	✗	✓	✓	✗	✗	✗	✗
Synalysis	✗	✗	✗	?	✗	✗	✗
EverParse	✓	✗	✓	✓	✓	✗	✓
ANTLR	✗	✓	✓	✗	✗	✓[1]	✗
Parsley	✗	✓	✗	?	✗	✓	✗
PADS/ML	✗	✓	✓	✗	✗	✗	✗
DaeDaLus	✗	✓	✓	✗	✗	✗	✗

[1] Some regular expressions are supported in the Lexer.

Table 7.9: Comparison of DDLs based on Expert Study Features

Table 7.9 presents the results of our categorization. We discuss each of these features in detail.

Proof of termination Although most developers want to empirically ensure that their program terminates for all input, only one implementation—EverParse—was verified to guarantee this property. RecordFlux is another implementation that was verified to ensure other properties, but termination is not guaranteed.

Module System Any DDL that needs to be able to capture complex formats such as PDFs must be able to support a module system. The PDF format uses numerous embedded formats that can be imported using the module system syntax of the DDL. DaeDaLus uses the `import` keyword to include other descriptions in the current format. In Kaitai, the `imports` tag under the `meta` field specifies a list of subformats to

import. Parsley uses the keyword `use` to include other specifications. RecordFlux uses the keyword `with` to include other specifications. DFDL also supports importing modules using an XML syntax: `<xs:import schemaLocation="imported_type.dfdl.xsd"/>`. Most interestingly, the PADS/ML language follows a "types as modules" paradigm. They rely on the OCaml module system by generating an OCaml file for each data description file. ANTLR also uses the `import` keyword to include nonterminals from a file.

Code generation DDLs often have adjacent tools that can be used to generate code in various target languages. Earlier, Table 7.7 discussed the languages supported by each code generation tool. Of the DDLs under consideration, DFDL includes a C code generator for a subset of the language and an interpreter written in Scala for the entire language. Parsley, in its current stage, only supports an interpreter. Similarly, DaeDaLus includes a Haskell-based interpreter and a C++ code generator. The other DDLs, RecordFlux, Kaitai Struct, EverParse, ANTLR, and PADS/ML focus on generating code in some target language. A complete list of languages supported by each DDL is in Table 7.7.

Memory Safety Memory unsafe languages such as C, C++, and assembly allow out-of-bounds reads and writes. The Low* compiler guarantees that the C code it generates is memory safe. EverParse exploits this feature in Low* to ensure that the parser code generated is memory safe. Unfortunately, other applications using C or C++ do not provide this guarantee. We use the "**?**" symbol to denote that either no code is generated or the generated code is in a memory safe language.

Zero Copy Parsing Zero-copy parsing is recognizing an input without making any copies of it. However, most parsing algorithms that are commonly used keep track of

a scaffold table and make copies of either input or metadata. Therefore, a zero-copy parser needs to be efficient in keeping track of the data locations and would need to use pointers extensively. Only one DDL implementation claims to use zero-copy parsing to achieve better security and superior performance. Previously, in Table 7.3, we had identified that experts did not think that the zero-copy parsing property was vital to a DDL.

Regular Expressions We explored the reference manual for each of the DDLs under consideration. We studied whether there is direct support to describe regular expressions using the particular DDL. The reference manual will usually contain an entry or at least references to regular expressions if these are supported. DFDL allows the use of regular expressions using a special XML tag. ANTLR supports the use of some regular expressions in the Lexer. Parsley supports some restricted regular expressions to aid developers. Supporting arbitrary regular expressions presents the risk of a developer trying to do "`.*`" leading to the entire remaining input buffer to be matched to the regular expression.

Serialization Serialization is the process of generating binary output from the parsed data structure. Traversing the parsed structure should produce the original input format again. We may need to modify the parsed structure to redact data or change the syntax in some cases. For example, if this redaction happens in a network filter, we may need to generate binary input from the parsed data. EverParse includes such functionality. In fact, in EverParse, the authors include proof that the parsers and serializers are inverses of each other with no transformations. DFDL also includes an "unparsing" functionality, where we convert the parsed infoset back to data.

7.3.2. Usability Properties

People learning new DDLs may require plenty of resources to get started and familiarize themselves with describing formats in these DDLs. In our expert study, we had asked experts to talk about which resources helped them and why. Furthermore, what resources do they make sure to include in DDLs where they have contributed. Table 7.10 summarizes our survey of DDLs and the resources currently available with each of them.

Languages	IDE support	Docker containers	Tutorials	Web IDE	Descriptive Errors
DFDL	✓[1]	✗	✓	✗	✓
RecordFlux	✓	✗	✓[3]	✗	✓
Kaitai Struct	✗[2]	✓	✓	✓	✓
Synalysis	✗	✗	✗	✓	✗
EverParse	✗	✗	✗	✗	✓
ANTLR	✓	✗	✓	✗	✓
Parsley	✓	✗	✓	✗	✓
PADS/ML	✗	✗	✗	✗	✓
DaeDaLus	✓	✓	✓[3]	✗	✓

[1] IDEs are easily available for XML syntax.

[2] IDEs are easily available for YAML syntax.

[3] DaeDaLus and RecordFlux include comprehensive reference manuals and usage guides, but not tutorials.

Table 7.10: Comparison of DDLs based on Usability Features

IDE support We surveyed the website of each of these parsing tools and searched for a language server implementation, an IDE plugin, or an extension to an existing IDE. Parsley, DaeDaLus, and RecordFlux include a Language Server implementation that makes it easy to build plugins for various IDEs. ANTLR provides an IntelliJ

plugin to aid with developing grammars in the Java API. DFDL provides a VSCode IDE plugin. In addition, since DFDL relies on core XML/XSD syntax, we could use IDEs and IDE plugins designed for XML development.

Docker containers Docker containers make it easy for people starting work on a new DDL to try things out without bothering with the installation. Often, developers try things out on the Docker container available to familiarize themselves with the environment and ensure the tool is appropriate for their requirements. We reviewed the source code repositories and websites of DDLs to check if they provide Docker containers to help developers try the DDL. Upon review, we found that only Kaitai Struct and DaeDaLus included Docker containers. In addition, the Kaitai Struct Docker image has not been updated in over three years, whereas DaeDaLus provides simple scripts for us to build the Docker images.

Tutorials and Reference Manuals Tutorials, reference manuals, and examples are great stepping stones for any new developer. A good manual for the standard libraries allows any developer to understand the true extent of the DDL and all the features it supports. Tutorials, where you can build gradually more difficult formats, are another useful way to allow people to explore the features of a DDL. Finally, an extensive gallery of example formats where many language features are exercised also allows developers to find use cases for various features.

Different DDLs support a different set of these features. Most implementations include a reference manual for the standard library and language features. In particular, DFDL, Kaitai, Parsley, and ANTLR include comprehensive, progressive tutorials for real-world examples. In addition, DFDL and Kaitai contain the most extensive format galleries supporting a wide range of file formats and network protocols.

Web IDE Web IDEs, just like Docker containers, provide an easy way to try out features of a language. These IDEs also provide syntax highlighting and error detection. Web IDEs also help visualize data files based on a grammar file, which can help debug malformed input and inadequate grammars. Synalysis and Kaitai Struct provide Web IDEs and a visualization component. When given a grammar and a data file, these tools can visualize the data file and separate it into the correct fields.

Descriptive Errors Since error descriptions are subjective, we marked this field as valid if the compiler or code generation toolkit provided syntax errors or type errors that would aid in debugging the error. A comprehensive user study would be needed to understand genuinely how usable these parser generation tools are. Since Synalysis does not have an offline compiler, only a visualization tool, it does not provide any information on syntax errors in the grammar descriptions. We found that all other tools provide some error description.

7.3.3. Discussion

This section categorized DDLs based on properties we considered in the expert elicitation study.

DDL Features

In our study, experts concurred that proof of termination, module system, and code generation are essential features that a DDL must include. Additionally, regular expression and serialization support are features that would be good to have in a DDL. However, when we studied the popular DDLs, we found that proof of termination is not provided in most popular DDLs today. Most DDLs support module systems and code generation, but not all. Support for regular expression and serialization is still very limited.

Our study highlights that to increase adoption in the parsers and file formats communities, researchers must focus their efforts on providing these functionalities in DDLs. In addition, serialization is a critical feature for those filtering and redacting packets in network middleboxes.

Usability Features

In addition to the core features of DDLs, usability features determine what resources are available to make it easier to learn a new DDL or try its features. We found that the results of our categorization lined up very well with the results of the expert study in Table 7.4. Most popular DDLs included some IDE support, tutorials, and descriptive errors.

Some of the IDEs also included web compilers and support for Docker containers. However, more DDLS must provide tutorials, comprehensive reference manuals, and a library of data formats. We believe that IDE support and a collection of formats would propel the adoption of these DDLs.

Section 7.4

Case Studies and Evaluation on Image Formats[2]

Of the DDLs we've considered so far, we looked through the format galleries of these DDLs to find formats they all support. Unfortunately, we did not see much of an overlap. Kaitai, DFDL, and Synalysis provide descriptions for several image formats commonly used.

First, we analyzed these specifications to compare the lines of code. Table 7.11 shows the lines of code for each image format in the three DDLs we consider. Next, we built a corpus of images across these formats from CommonCrawl and evaluated

[2] I collaborated with Kris Udomwongsa for this section of the thesis.

these DDL implementations.

Parser	JPEG	PNG	GIF	BMP
DFDL	836	730	398	601
Kaitai	221	428	203	590
Synalysis	3153	386	1086	-

Table 7.11: Comparing image format descriptions in various DDLs lines of code

7.4.1. Constructing the dataset

To evaluate the image specifications in various DDLs, we first set out to extract sufficient image files. The GovDocs1 corpus released by Digital Corpora [78] contains 109,282 JPEG files. Unfortunately, this dataset does not contain other image formats at all. Hence, we had to seek other sources for image files.

We built scripts to extract images from the Common Crawl dataset [48]. This dataset is organized in the form of web crawling results based on individual months. To gather images for our study, we must extract all the data (stored as WET files) for each month and then look through the HTTP MIME headers to find the images. Additionally, the dataset also contains numerous duplicates. Therefore, we build an SQLite database with SHA256 hashes to ensure that we can find unique pictures. Our scripts are available with instructions at `https://github.com/Dartmouth-Trustlab/commoncrawl-images`.

These WET files are databases containing URLs. After analyzing all the WET files from one month's captures, we gathered unique, reachable URLs corresponding to each file of our file types of interest. Table 7.12 shows the number of images we were able to gather from one month of common crawl data.

Image Formats	Unique Images
JPEG	78591
PNG	50260
GIF	974
BMP	211

Table 7.12: Number of files captured from one month of Common Crawl data and GovDocs. We used the GovDocs corpora for JPEG images, and the CommonCrawl dataset for other formats.

DDL	Source
Kaitai Struct	`https://formats.kaitai.io/`
DFDL	`https://github.com/DFDLSchemas/`

Table 7.13: DDLs and the source of the image format descriptions for JPEG, PNG, GIF, and BMP.

7.4.2. Evaluation

To look for parser differentials between the DFDL and Kaitai implementations of various formats, we first identified descriptions of image formats in these languages that are available as a part of the official projects. The sources of these descriptions are listed in Table 7.13.

Next, we created a testbench in Python to run each file through Pillow, Apache Daffodil, and Kaitai with the corresponding descriptions. Pillow is an image library in Python that can detect the image format using an internal parser. We use Pillow as ground truth to detect differentials.

7.4.3. Case Study Results

We investigated the mismatches we found between Pillow (served as our ground truth), DFDL, and Kaitai. These mismatches were an indication that the specification did not implement all the cases required to comply with the format rules. After documenting these errors, we investigated the source of these mismatches and proposed fixes to format descriptions to improve compliance. Table 7.14 displays the

File Format	$\neg\mathcal{V}_{kaitai} \wedge \neg\ \mathcal{V}_{DFDL}$	$\mathcal{V}_{kaitai} \wedge \neg\ \mathcal{V}_{DFDL}$	$\neg\mathcal{V}_{kaitai} \wedge\ \mathcal{V}_{DFDL}$	Total Files
JPEG	100	3212	66	78591
PNG	1336	1844	8	50260
GIF	87	10	0	974
BMP	19	0	0	211

Table 7.14: Examination of differences in output from DFDL and Kaitai image format specifications

results of our case study evaluation.

Definition 7.1. We define $\mathcal{V}_{DFDL}(x, format)$ to be $\top$, if according to the DFDL specification for the $format$, file x is valid. For simplicity, we will shorten this notation to say $\mathcal{V}_{DFDL}$, without the arguments to the predicate functions explicitly stated.

Similarly, we define extensions for $\mathcal{V}_{pillow}$ and $\mathcal{V}_{kaitai}$.

Table 7.14 demonstrates the results of our case study. Our JPEG evaluation was the most comprehensive one given the abundance of data available. The Kaitai implementation of JPEG failed in several instances where the Pillow implementation did not throw errors. A deeper examination of these errors led to a major fix to the Kaitai JPEG specification. Similarly, we also examined the errors exposed by DFDL and the issues in the specification.

7.4.4. Fixes to descriptions in Kaitai

JPEG JFIF/JFXX not implemented in Kaitai Upon closely examining the JPEG specification evaluations, we found that the Kaitai Struct implementation does not handle the JFIF/JFXX fragments correctly. First, the Kaitai specification does not check any of the magic strings. These fragments start with the magic string `JFIF\x00` to denote a JFIF segment.

JPEG APP0 segments can also contain an optional JFXX fragment.[3] Making these changes to the Kaitai Struct specifications to make it more accurate presented

[3] `https://www.w3.org/Graphics/JPEG/jfif3.pdf`

```
segment_app0:
  seq:
    - id: magic
      type: str
      encoding: ASCII
      size: 5
```

(a)

```
segment_app0:
  seq:
    - id: magic
      type: str
      encoding: ASCII
      size: 4
    - id: extra_null
      contents: [00]
    - id: body
      type:
       switch-on: magic
       cases:
        '"JFXX"': segment_app0_ext
        '"JFIF"': app0_jpeg_format
```

(b)

Code Snippet 16: (a) The original Kaitai Struct JPEG code, (b) Modified Kaitai Struct JPEG code

challenges. First, the Kaitai struct syntax does not allow for a mix in the cases of hex and ASCII text. Therefore, we had to create a workaround to handle the ASCII and the hex characters separately.

Next, Kaitai Struct syntax does not allow for a single optional nonterminal. So, for example, in the BNF notation, it would follow the following syntax.

```
APP0 → JFIF (JFXX)?
```

The above syntax implies that the JFXX nonterminal is optional, and may not need to be matched. However, the Kaitai Struct syntax does not allow for such an option. We created workarounds to handle the above two issues by treating the construct as the following:

```
APP0 → JFIF | JFXX
```

Code Snippet 7.4.4 demonstrates our additions to the Kaitai JFIF additions. First, we add a switch case to check for the magic string. Next, we add a test for the null character. And finally, unlike the original Kaitai specification, we handle the JFXX

extension and the JFIF APP0 objects separately. Our parser correctly parsed the JPEG files that followed this syntax. We will submit our Kaitai Struct descriptions to the Kaitai community for review soon.

Shortcomings of the DFDL JPEG schema The authors of the JPEG schema for DFDL acknowledge that their parser is over permissive and accepts files that may not be JPEG.[4] They explain that they do not validate the data inside the JPEG tags, instead only focusing on the different types of packets available. We found the following error frequently occurred while invoking the JPEG parser on valid JPEG files (marked valid by Pillow and Kaitai Struct).

```
[error] Parse Error: Assertion failed: Does not end with an EOI segment.
Schema context: sequence[2] Location line 63 column 18 in file jpeg.dfdl.xsd
Data location was preceding byte 57278
```

Upon further investigation, we identified that the length field in the JPEG segments is not sufficiently implemented. Other than the start and end of image segments, each JPEG segment includes a length field specifying where the packet ends. In addition, the scanning function incorrectly identifies this location as the end of the buffer. With the features currently available in DFDL, it would be a monumental task to implement the correct specification in DFDL. Hence, we leave this exercise to future work when hopefully, the DFDL creators adopt a more expressive language or provide additional functionalities.

The need for parser debuggers

Debugging the JPEG errors and finding the causes these tools reject a particular JPEG file presented significant challenges. Unfortunately, these tools do not include

[4] https://github.com/DFDLSchemas/JPEG/issues/2

good debugging tools. Using traditional debugging tools such as GDB does not provide an easy interface to debug parser errors and the values in parsing tables.

Our experience of using these tools motivated us to build the ParseSmith debugger in Chapter 6.3. It provides a more straightforward interface to interact with a parser and provides ways to inspect the parser's internal data structures to understand errors faster. We believe that these commercial tools would benefit from providing such a debugger. In fact, it would be good to have a single set of debugging operations and options that all parser tools follow to provide a unified interface across tools.

Discussion

Specifications written in various DDLs must be evaluated thoroughly using an extensive dataset of valid data to ensure that all the features commonly used in a message or network format are handled correctly. Our experience in evaluating these specifications on image formats has revealed that these specifications are made available but not thoroughly evaluated. Several constraints were left unevaluated, and several features were not handled correctly—leading to several parser differentials.

We believe our evaluation framework provides an extensible framework to evaluate future specifications to find differentials. We hope the community of DDL users use our framework to evaluate their tools and compare their specifications against other implementations of the same protocol or data format.

7.4.5. Fuzzing

Fuzzing or fuzz testing is an approach to ensure software resiliency to crashes and semantic issues [168]. The method uses randomly generated input that may be known to be invalid to stress different paths in an application. Fuzzing has been extremely effective in finding bugs in various software because it exercises boundary and corner

cases in applications to find crashes, memory leaks, and infinite loops.

The term fuzzing was coined in 1988 by Barton Miller in a class project where they exercised many Unix utilities with random character streams in the command-line arguments [185]. Subsequently, AFL and AFL++ have become the go-to fuzzing tools for C and C++ applications and have found numerous popular vulnerabilities, such as Shellshock and in Firefox.

Most recently, Google and Microsoft provided fuzzing-as-a-service to fuzz open-source software using libfuzzer and AFL++ continuously [244]. AFL++ operates in a black-box or grey-box mode by either randomly mutating input or tracking the program's control flow based on instrumentation [94]. Libfuzzer also instruments code to mutate input to gain more code coverage [243].

Symbolic execution approaches, in contrast, analyze the program to understand what input would execute branches in the program [219, 167]. Instead of real input, the program is supplied symbolic input, and a program tracks the changes to this symbolic input. Using these symbolic expressions, we can define what inputs need to be used to arrive at a particular branch in the code. Symbolic execution engines require access to source code and code changes. Additionally, they require enormous memory since it leads to state-space explosions.

Hence, in our evaluation, we used fuzzing approaches to ensure the resilience of the generated code. We used the AFL++ fuzzer against the C code generated from DFDL (Apache Daffodil), DaeDaLus, and Kaitai Struct. In addition, we ran Python-AFL against the Python code of RecordFlux. Unfortunately, the fuzzers for other programming language targets are not well maintained, and hence, we could not fuzz them. Table 7.15 discusses the fuzzing tools and static analysis tools we deployed against generated code for various DDLs.

Languages	Fuzzing tools	Static analysis
DFDL	AFLPlusPlus	Infer, CppCheck, FlawFinder
RecordFlux	Python-AFL	Bandit
Kaitai Struct	AFLPlusPlus, Python-AFL	Infer, CppCheck, FlawFinder, Bandit
DaeDaLus	AFLPlusPlus	Infer, CppCheck, FlawFinder

Table 7.15: Fuzzing tools used against code generated from these DDLs

AFLPlusPlus We ran AFLPlusPlus against the C source code generated from DFDL and Kaitai's various image formats. After running for 24 hours AFLPlusPlus did not find any vulnerabilities in either implementation.

Python-AFL Python-AFL uses AFL++ under the hood to fuzz Python applications. We must import the AFL library and call an init function. Subsequently, we invoke the `py-afl-fuzz` binary on the Python code. Unlike AFL++, the Python code does not have to be explicitly instrumented to run with Python-AFL. Although we were able to run Python-AFL against RecordFlux source code, the generated code uses the SPARK programming language, that we were not able to fuzz.

Python-AFL is designed to tweak input to find paths to exceptions within the source code—since, in Python, we cannot reach memory safety violations. However, when we ran Python-AFL against Kaitai-generated code and RecordFlux, we found numerous crashes since every path to an exception is marked as a crash. To ensure that we find true crashes, we need to define a list of paths to exceptions that are valid and the exceptions that we must not reach because they expose issues in the core parser libraries.

7.4.6. Static Analysis

Programs may contain bugs. A compiler may uncover syntactic and semantic bugs, but more complex bugs that are syntactically valid but can produce runtime errors in some cases are harder to identify. Static analysis tools can automatically reason

about the runtime properties of code without actually executing it. Instead, they traditionally build a call graph or an AST and reason about conditions the programmer may not have envisioned. Of course, no static analysis tool is perfect, and hence we deploy a host of these static analysis tools against code generation tools to identify errors. Table 7.15 summarizes the static analysis tools we use to evaluate code generated using DDLs. We will first discuss these tools and then look at the results of running these tools on generated code.

Infer Infer is a static analyzer for Java, C, and C++ applications written in OCaml [51]. Infer effectively identifies commonly occurring problems such as null-pointer dereferences, memory leaks, and sources of potential buffer overflows. Hence, Infer provides a way to statically analyze programs to find sources of input-handling vulnerabilities.

We statically analyzed code produced using DDLs where C, C++, and Java were involved. Hence, we were not able to target all the languages proposed. DFDL's runtime1 uses `sbl` to build the JVM based Scala code—and this command is not supported by the infer toolkit. Instead, we analyze the C code generated using Apache Daffodil's runtime2.

```
infer -- make
```

We need to make sure that all the library dependencies needed by the make command are satisfied. We used the Infer docker container to run our experiments.

FlawFinder FlawFinder is a simple static analysis tool that checks for library functions that can lead to input-handling vulnerabilities such as buffer overflows and string formatting attacks [283]. FlawFinder recursively checks all the C and C++ files in a folder to find these security issues. We invoke FlawFinder using the following command:

```
flawfinder /path/to/code
```

CppCheck This is a static analysis tool that is focused on finding undefined behavior caused by various pointer operations [179]. The premium (paid) version can detect infinite loops as well. Cppcheck internally implements unsound flow-sensitive analysis, where the data flow is bidirectional. Like other static analysis tools, it can find null pointer dereferences, out-of-bounds checks, integer overflows, and division-by-zero errors. We invoke CppCheck using the following command:

```
cppcheck --force /path/to/code
```

Bandit The Bandit tool is designed to analyze Python code for security issues [223]. Bandit builds ASTs for each Python file in the directory and scans them for common security issues. These security issues include weak cryptographic algorithms, antipatterns, dangerous functions such as exec and eval, and allowing new processes to start with a shell. We ran Bandit on RecordFlux and Kaitai Struct's Python implementation. Bandit is invoked with the following command:

```
bandit -r /path/to/python/code
```

Results

In this section we summarize the results of running various static analysis tools on code generated for DDLs under consideration.

DFDL We relied on the Apache Daffodil runtime2 implementation, which uses C. Since this format only supports a subset of the entire DFDL standard, we could not incorporate the image formats to comply with runtime2. So instead, we relied on the corpus of examples in the runtime2 library, which generated between 400 and 1300 lines of C code.

FlawFinder reported 19 weaknesses in C code generated by Apache Daffodil. Most of it were warnings on the use of functions that can be dangerous. Attackers would use functions such as `strlen` and the statically-sized arrays to trigger buffer overflows.

Weaknesses	**Reported by**	**Count**
Format Strings can be exploited	FlawFinder	4
Buffer overflows possible in `getopt`	FlawFinder	1
Use of `fopen`	FlawFinder	1
Statically sized arrays can lead to overflows	FlawFinder	10
Use of `strlen`	FlawFinder	3
Dead code: variable assigned never used	Infer	1

Table 7.16: Weaknesses reported by various tools in Apache Daffodil generated C code

Infer produced only one alert, and CppCheck failed to find any issues with the runtime2 C code generated. Upon further examination, we found that the alert produced by Infer was incorrect. The C code used the following syntax:

```
char* nibble = text;


*(nibble++) = '\0';
return text;
```

The infer engine pointed out that the variable `nibble` was modified, but never read. For those familiar with C pointers, this error is not valid—the pointer `nibble` modifies the input in `text`.

DaeDaLus To evaluate DaeDaLus, we used a description of the Musical Instrument Digital Interface (MIDI) format. This description was provided as a part of the DaeDaLus code distribution. We generated code using the `daedalus` executable using the following command.

```
daedalus tests/midi/midi.ddl --compile-c++ --out-dir=generated-c/
```

Weaknesses	Reported by	Count
`open` can be exploited with a symlink	FlawFinder	1

Table 7.17: Weaknesses reported by various tools in the DaeDaLus generated C++ code

Next, we ran the static analysis tools on the `generated-c` folder. The tools analyzed 14545 lines of code each. Table 7.17 demonstrates the results of static analysis.

CppCheck and Infer did not find any issues with the generated C++ code. FlawFinder found one minor weakness: an alert to ensure that developers check the file being opened using the `open` system call.

Kaitai Struct The Kaitai image format descriptions and the C++ runtime parsing library accounted for 3559 lines of C++ code. Table 7.18 demonstrates the results of our evaluation of Kaitai generated code and the Kaitai C++ standard library.

CppCheck and Infer did not find any weaknesses in the Kaitai-generated code or the C++ library. The weaknesses identified using FlawFinder were in the Kaitai Struct C++ library that contains the core parsing functions. FlawFinder did not find bugs in the C++ code generated from image format Kaitai specifications. Similarly, Bandit analyzed 322 lines of Python code and did not find any issues.

Weaknesses	Reported by	Count
Should check buffer boundaries in `read`	FlawFinder	20
Statically-sized arrays need bounds checking	FlawFinder	2

Table 7.18: Weaknesses reported by various tools in the Kaitai Struct generated C++ code

RecordFlux We studied the RecordFlux Python source code and code generated for a TLV format provided with the distribution. Invoking Bandit on these Python sources produced 1427 warnings: of which 1425 were low severity issues involving

uses of assert in code. One medium severity issue was on using the MD5 hashing algorithm—which is known to be vulnerable [255]. Finally, a severe issue was concerning using "shell=True" when creating subprocesses in Python. If attackers can control the input sent to this function, they can gain shell access.

Most of the issues found in the generated code are warnings or alerts asking a developer to be careful when using certain functions or patterns. There were no serious security or LangSec anti-patterns uncovered that need to be patched immediately.

7.4.7. Summary

In this section, we evaluated the runtime implementations of various DDLs for the reliability of the code generated and the accuracy of the descriptions of various image formats. To do this, we first built a corpus of image files and evaluated the image format descriptions in DFDL and Kaitai against our corpus. Next, we statically analyzed and fuzzed the code generated from DDLs to find security issues in the implementations. Finally, we cataloged the issues we found, but none of these issues were major and needed to be fixed urgently.

Section 7.5

Conclusions

In this chapter, we conducted an expert elicitation study where we interviewed parser experts with varying backgrounds to understand what features must be present in DDLs to support a wide range of data formats and for adoption. We then studied the popular DDLs through the lens of these features to deduce if these DDLs satisfy the requirements of the experts.

We then conducted case studies to generate code using these DDLs where various image formats were described. First, we evaluated the *correctness* of these specifi-

cations using a large-scale study from data collected from CommonCrawl. Next, we evaluate the *reliability* of these tools by running fuzz testing and static analysis tools on the generated code.

As we observe in this Chapter, no DDL available today supports all the features and usability tools that experts wish to see. A DDL must support several of these features and support several data formats commonly used. DaeDaLus and Parsley are tools that have come a long way: designed to support network protocols and complex file formats such as PDFs. However, based on our expert study and the categorization using the features from the expert study, a few trends emerge, paving the way for future work.

Bidirectional Programming and Parsing Data Formats Bidirectional programming has been studied quite extensively in programming languages, where transformations made to one version of the code must automatically translate to another version [95]. UML tools such as Rational Rose provide another example of such bidirectional programming where an addition to the diagram in the form of new functions must translate to the Java class definition [224].

Given that code generation tools are often buggy—producing incorrect code—developers would benefit from having bidirectional tools that can modify the source data description when the generated code is modified. Most of the experts in our study mentioned that the generated code is so complex that they did not envision that any developer would want to modify it directly—instead, changing the description and regenerating code from it.

One specification → Multiple versions in DDLs → Multiple parsers At a glance, this seems like a good paradigm. Once a specification is released, different developers read it and describe it in the DDL of their choice. We then run code

generations on the data description to generate parser code. However, our study shows that the same format is described differently by different parsers, leading to numerous parser differentials.

One proposed solution followed by the PDF Association is to use machine-readable specifications. For example, the PDF association used a TSV format to describe constraints on the complex data types in PDFs using the Arlington PDF Model. This solution, however, does not extend well to other formats. For example, the image formats we studied in this thesis do not follow a syntax that can be easily captured using only TSV files.

DDLs can be used to serve as the tool to describe these complex data formats in a machine-readable format. Additionally, we would benefit from tools to translate specifications in these global DDLs to the other commonly used DDLs.

Chapter 8

Conclusions

Given the Coronavirus pandemic for the last 2-3 years, our society is increasingly embracing remote work and automation, which means industries and devices previously working offline must support Internet-connected technologies. These computers increasingly provide convenience and efficiency through the Internet of Things (IoT) technologies.

Over time, Internet access has gradually increased in the developing world. As a result, more and more people in the developing world now own multiple computers and smartphones. Additionally, the cost of these devices has also been lowered—allowing lower-income societies to embrace these new technologies. However, as IoT's scale reaches the order of billions of devices, the threat to human life and digital privacy also increases as nation-states devote more and more resources to adversarial research.

As I discussed in this thesis, parsers are the first line of defense for any software application. Since we now use Internet-connected devices in critical situations such as delivering life-saving drugs like insulin or controlling dangerous remote machinery, we must ensure that these devices have well-written parsers that can protect the devices from malicious input.

This thesis considered the various threats posed to software and discussed how parsers are the most targeted portion of any software system—a vulnerable or insufficient parser exposes the entire system to attacks. Studies examining how vulnerabilities are traditionally patch bugs demonstrate how we still follow a reactionary, band-aid approach to fixing bugs. Such approaches are undesirable since they do not address root causes and parsers remain challenging to audit.

Contributions This thesis aimed to understand what tools are needed to protect systems from input-handling vulnerabilities and how we can build robust, performant, and verified tools that developers can use. This thesis covered four major chapters, each of which takes a step towards demonstrating how we can achieve our grand objective.

Chapter 4 I discussed parser-combinator tools in detail and how they can be used to describe various data formats. This chapter presented a first-of-its-kind parser and protocol state machine for the Industrial IoT protocols XMPP and MQTT. We also built CVD, a communications validity detector for SCADA networks. CVD was deployed to real power grid substations and was able to detect attacks and events in real-time. Finally, we demonstrated how we could use parser combinators effectively to describe file formats such as iccMAX. I believe our experiences using parser combinators to describe various network protocols and file formats can motivate more work in this direction.

Chapter 5 We built the Parsley PDF checker, which included a comprehensive type checker. We generated type checker code directly from the machine-readable PDF specification—the Arlington model. As a result, our type checker was able to find several errors in popular PDF implementations—leading to numerous bug fixes and major specification changes. This work demonstrates

that using machine-readable specifications accelerates the process of getting parser and validator implementations right. Additionally, it also reduces the possibility of parser differentials if each parser is generated from the exact machine-readable specification. We hope that future specification creators design machine-readable specifications, given their benefits.

Chapter 6 I built ParseSmith, a parser interpreter for formats described in the Parsley Data Description Language. Our parser interpreter builds on traditional Parsing Expression Grammar (PEG) parsing algorithms to capture complex constructs commonly used in data formats. I demonstrate how to argue about various correctness properties in these parsers and how to build an effective debugger for parsers. I believe that developers can use ParseSmith to generate efficient, verified parsers for various data formats. I also hope that the ParseSmith debugger will serve as a case study on building practical parser debuggers that can inspect the internal data structures of parsing machines.

Chapter 7 We provided the first comprehensive categorization and evaluation of data description languages. The metrics for this categorization were inferred based on an expert elicitation study we conducted. We interviewed 15 experts who are well-versed with data formats and parsers. Our evaluation explored implementations of various data description languages and the descriptions of image formats in these languages. We found that different languages include varying versions of image formats, covering different portions of specifications. Our experiments have led to corrections to format descriptions in Kaitai Struct and DFDL.

Section 8.1

Future Work

Moving forward, I envision research examining parsers and better parsing methodologies to continue in full swing. Each of my chapters outlined some future work (Sections 4.6, 5.9, 6.7, and 7.5). In the rest of this section, I summarize broad future directions and open problems for the field to explore.

Parsing Abstract Machines

Parser implementations follow different algorithms with differing guarantees. Parsing functions can be standardized using a dedicating parsing machine that can be isolated using sandboxing techniques to ensure that exploitation of this machine does not affect the rest of the program's address space.

We must create an instruction set that can implement the core parsing functions, such as matching a nonterminal to a location in the input. Data formats also require operations on the buffer—where we create a window or view into a buffer to perform parsing operations. A parsing machine would need to keep track of a stack of views and instructions and write output to attributes and abstract syntax trees.

Such an abstract machine could serve as a backend for multiple parsing combinator toolkits and data description languages. Additionally, by verifying components of this abstract machine, we can ensure correctness, interoperability, and isolation.

More user studies

In this thesis, we conducted a user study to understand experts' experiences working with Data Description Languages (DDL). Experts, however, discuss their goals and ideas concerning what they want to see in DDLs and what they think is useful.

Their view of DDLs and combinators may not translate directly to the developers' experiences.

Additionally, DDLs and their tutorials may set expectations on what a developer must know to be able to use it. Deep knowledge and familiarity with formal language theory and compiler design concepts are not common among computer science graduates.

Hence, there is a need to conduct a large-scale study on programmers to understand how they implement parsers for data formats. Furthermore, a user study would need to compare the usability parser combinators and data description languages and the current common methodology of building parsers from scratch. Such a study could set a clear standard on what users find useful and how DDL toolkits must structure their tutorials to ensure developers adopt them and do not commit many errors while developing parsers.

Hardware parsers

I also envision research implementing hardware parsers to continue. Pegmatite places a foot in this direction, implementing parsers for Calc-Parsing Expression Grammars [171]. Parsers implemented in FPGAs can be more performant and efficient than those implemented in software. Hence, such FPGA-based parsers can be a good candidate for resource-constrained IoT devices. Such FPGA-based parsers have been used in printers for data formats such as PDF and Postscript for specific purposes and can be used in updateable IoT devices.

FPGA-based parsers also serve as an avenue for intrusion detection and packet filtering on the router level. Wang et al. [279] described an approach to generate FPGA code using P4 descriptions—where network headers are described in P4, and the packets are parsed at line speed. Programmable network forwarding planes also

serve as an avenue for future research where line-speed parsers could be deployed to filter packets in a network [221, 123].

Additionally, most prior work focused on hardware parsing focus on network protocols. The following future directions could be useful avenues to explore for the community.

- **Building a framework to parse file formats in FPGAs**: Firewalls processing email attachments for malware are often pattern- or signature-based. As a result, they may miss zero-day exploits since they will not meet any known patterns. To catch exploits in file formats such as image and document formats, we need to parse them fully in the network endpoint firewalls. Generating FPGA parsers from syntax, such as Parsley and Kaitai, that already support file format descriptions, can be a significant first step in achieving this goal.

- **Formalizing how streaming formats should be parsed**: Streaming formats often use delimiters to denote where new packets begin, and the previous ones end. A full data frame, such as a video file, could be spread out across numerous network packets. Most parsing tools available today cannot handle such chunked data that needs to be reassembled. This problem is exacerbated in FPGAs, forcing a need to understand how we can formalize principled streaming format parsing and how the buffer must be controlled in hardware.

- **How would a parsing abstract machine map to FPGAs**: In a previous section, we discussed how parsing abstract machines provide an avenue to formalize parsing algorithms and isolate parsers. However, such machines do not translate well to FPGAs and would be incredibly inefficient. Compiler tools, such as Xilinx Vivado, perform several rounds of optimization to ensure that the circuits generated are efficient and match timing requirements. Understanding

the circuits for various standard parsing operations in abstract machines is a natural step in that direction.

Parser Differentials and Symbolic Execution

In Section 7.4, we found that image format specifications in DFDL and Kaitai differed for several constructs in various formats. We compared these interpretations to a ground truth implementation to determine which of these two implementations misinterpreted the specification. However, pointing out accurately why a particular implementation is incorrect involves a lot of manual debugging and finding the right inputs.

I believe we can automatically determine differences between implementations.

- The first approach to tackle this problem is to a symbolic execution engine. Linda Xiao explored this approach in [292]. They used the KLEE engine [50] to construct grammar inputs that conform to a grammar specification. KLEE could be a useful symbolic execution engine to generate input that satisfies one parser but not another.

  Unfortunately, most constructs in the DFDL format are not supported by the C-code generator. To define a constraint such that a parser $P1$ accepts an input while parser $P2$ rejects it in KLEE, both the parsers need to be in C or C++. Hence, we need to create a client-sever protocol to pass input to parsers of any programming language. We need a grammar-based engine that can pass input to these implementations exploring different paths in the grammars.

- The second possible approach is to define a grammar format as an intermediate representation. For example, the Parsley language [198] encompasses the features offered by other grammar formats. We need to build compilers that

can convert DFDL, Kaitai, and other specifications to a Parsley language. A normalized or uniform representation of grammars allows us to formalize and point out where grammars differ by generating input.

Formalizing evolution of formats as grammar drifts

In chapter 5, we studied how different tools implement major portions of the PDF specification incorrectly. Also, in chapter 7.4.3, we explored how different parsing tools contained incorrect representations of various image format syntax. Both these chapters motivate us to understand how grammars and data formats evolve.

When we parse an input on a grammar, we build an abstract syntax tree describing how the input was parsed. We can formalize how one interpretation differs from another by defining transformations on the AST. For example, suppose an implementation P' incorrectly implements some fields and another implementation P serves as the ground truth. We can create transformations from the AST produced by P' to a normalized AST produced by P.

By creating a tree of transformations on where these implementations differ from one another and a ground truth implementation, we can create a genealogy chart of the evolution of a data format since its inception. Creating such a chart would serve several purposes.

First, it allows us to understand how humans make mistakes and introduce errors and typos in data format parsers and creation tools. Understanding this would help us build tools to detect human errors and produce patterns to detect them. Second, it can allow us to fingerprint software. By traversing the tree and applying transformations, we can guess which software was used to produce a certain incorrect AST. Argyros et al. [21] used a similar approach to fingerprint web application firewalls based on their regular expressions rules and the errors in them.

Ensuring generated parsers are readable

Currently, Kaitai Struct, DFDL, and Dafny do not generate human-readable code (Chapters 6.5 and 7.4.3). This paradigm forces the user to generate code again whenever a specification is incorrect as a part of the development process. In most cases, a significant portion of the generated code is the same, and a small portion of code is rewritten.

This process of debugging and re-generating code is slow and inefficient. Additionally, developers need to trust code they did not directly write but generated from a grammar specification. The generated code might also contain implementation errors that the developer does not foresee. Ensuring the human readability of the generated code should hence, be of paramount importance. A developer must be able to debug the generated code and have the changes propagated to the original specification (bidirectional programming).

Learning File Formats from Input Samples or Code

Prior work in grammar learning has focused on learning network protocols and formal languages—and not on learning data formats such as PDF or image formats. More recent work on learning data formats focuses on building a format recognizer using machine-learning techniques [67]. However, much of the work on learning file formats remain. This is because file formats often include complex constructs such as offsets that are not traditionally found in network protocols or formal grammar.

Section 8.2

Final Remarks

In closing, I believe the contributions of this thesis are vital and will aid developers in designing better parsers, which would help reduce bugs and vulnerabilities in code. Furthermore, since Smart Buildings, Smart Cities, Driverless cars, and Internet-of-Battlefield-Things are all up and coming concepts that people seem to be very invested in, the need to secure communication endpoints with parsers is more serious than ever.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information Systems Security*, 13(1) article Article 4 (40 pages), November 2009. DOI 10.1145/1609956.1609960.

[2] AdaCore and Capgemini Engineering. SPARK: A Software Development Technology Specifically Designed for Engineering High-Reliability Applications. https://github.com/AdaCore/spark2014, 2014.

[3] Sameed Ali, Prashant Anantharaman, Zephyr Lucas, and Sean W Smith. What We Have Here Is Failure to Validate: Summer of LangSec. *IEEE Security & Privacy*, 19(3) pages 17–23, 2021. DOI 10.1109/MSEC.2021.3059167.

[4] Sameed Ali, Prashant Anantharaman, and Sean W Smith. Armor Within: Defending against Vulnerabilities in Third-Party Libraries. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 291–299. IEEE, 2020. DOI 10.1109/SPW50608.2020.00063.

[5] Tim Allison, Wayne Burke, Valentino Constantinou, Edwin Goh, Chris Mattmann, Anastasija Mensikova, Philip Southam, Ryan Stonebraker, and Virisha Timmaraju. Research Report: Building a Wide Reach Corpus for Se-

cure Parser Development. In *IEEE Security and Privacy Workshops (SPW)*, pages 318–326, 2020. DOI 10.1109/SPW50608.2020.00066.

[6] Tim Allison, Wayne Burke, Chris Mattmann, Anastasija Mensikova, Philip Southam, and Ryan Stonebraker. Research Report: Building a File Observatory for Secure Parser Development. In *IEEE Security and Privacy Workshops (SPW)*, pages 121–127, 2021. DOI 10.1109/SPW53761.2021.00025.

[7] Kristopher Ambrose, Steve Huntsman, Michael Robinson, and Matvey Yutin. Topological Differential Testing. https://arxiv.org/pdf/2003.00976.pdf, 2020.

[8] Prashant Anantharaman. Mismorphism: The Heart of the Weird Machine (Transcript of Discussion). In *Cambridge International Workshop on Security Protocols*, pages 125–131. Springer, 2019. DOI 10.1007/978-3-030-57043-9_12.

[9] Prashant Anantharaman. Missing single quote character in date encoding. https://github.com/michaelrsweet/htmldoc/issues/455, November 2021.

[10] Prashant Anantharaman, J Peter Brady, Patrick Flathers, Vijay H Kothari, Michael C Millian, Jason Reeves, Nathan Reitinger, William G Nisen, and Sean W Smith. Going Dark: A Retrospective on the North American Blackout of 2038. In *Proceedings of the New Security Paradigms Workshop*, pages 52–63, 2018. DOI 10.1145/3285002.3285011.

[11] Prashant Anantharaman, J Peter Brady, Ira Ray Jenkins, Vijay H Kothari, Michael C Millian, Kartik Palani, Kirti V Rathore, Jason Reeves, Rebecca Shapiro, Syed H Tanveer, et al. Intent as a Secure Design Primitive. *Modeling and Design of Secure Internet of Things*, pages 529–562, 2020. DOI 10.1002/9781119593386.ch23.

[12] Prashant Anantharaman, Anmol Chachra, Shikhar Sinha, Michael Millian, Bogdan Copos, Sean Smith, and Michael Locasto. A Communications Validity Detector for SCADA Networks. In *Proceedings of the International Conference on Critical Infrastructure Protection*. Springer, 2021. DOI 10.1007/978-3-030-93511-5_8.

[13] Prashant Anantharaman, Steven Cheung, Nicholas Boorman, and Michael Locasto. A Format-Aware Reducer for Scriptable Rewriting of PDF Files. In *Proceedings of the IEEE Security and Privacy Workshops (SPW)*. IEEE, 2022.

[14] Prashant Anantharaman, Vijay Kothari, J Peter Brady, Ira Ray Jenkins, Sameed Ali, Michael C Millian, Ross Koppel, Jim Blythe, Sergey Bratus, and Sean W Smith. Mismorphism: The Heart of the Weird Machine. In *Cambridge International Workshop on Security Protocols*, pages 113–124. Springer, 2019. DOI 10.1007/978-3-030-57043-9_11.

[15] Prashant Anantharaman, Michael Locasto, Gabriela F Ciocarlie, and Ulf Lindqvist. Building hardened Internet-of-Things clients with language-theoretic security. In *Proceedings of the IEEE Security and Privacy Workshops (SPW)*, pages 120–126. IEEE, 2017. DOI 10.1109/SPW.2017.36.

[16] Prashant Anantharaman, Kartik Palani, Rafael Brantley, Galen Brown, Sergey Bratus, and Sean W Smith. PhasorSec: Protocol Security Filters for Wide Area Measurement Systems. In *Proceedings of the IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, pages 1–6. IEEE, 2018. DOI 10.1109/SmartGridComm.2018.8587501.

[17] James P. Anderson. Computer Security Technology Planning Study. http://csrc.nist.gov/publications/history/ande72.pdf, October 1972.

[18] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2020.

[19] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2) page 87–106, nov 1987. DOI 10.1016/0890-5401(87)90052-6.

[20] Dana Angluin. Queries and Concept Learning. *Machine learning*, 2(4) pages 319–342, 1988. DOI 10.1023/A:1022821128753.

[21] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-Box Differential Automata Learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1690–1701, New York, NY, USA, 2016. Association for Computing Machinery. DOI 10.1145/2976749.2978383.

[22] Artifex Software, Inc. Ghostscript: An interpreter for the PostScript Language and PDF Files. https://www.ghostscript.com/doc/current/Use.htm, 2016.

[23] Artifex Software, Inc. MuPDF: A lightweight PDF, XPS, and E-book Viewer. https://mupdf.com/docs/manual-mutool-clean.html, 2016.

[24] Hala Assal and Sonia Chiasson. "Think Secure from the Beginning": A Survey with Software Developers. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–13, New York, NY, USA, 2019. Association for Computing Machinery. DOI 10.1145/3290605.3300519.

[25] Tuomas Aura, Thomas A Kuhn, and Michael Roe. Scanning Electronic Documents for Personally Identifiable Information. In *Proceedings of the 5th*

ACM Workshop on Privacy in Electronic Society, pages 41–50, 2006. DOI 10.1145/1179601.1179608.

[26] Michael Bächle and Paul Kirchberg. Ruby on Rails. *IEEE Software*, 24(6) pages 105–108, 2007. DOI 10.1109/MS.2007.176.

[27] Julian Bangert and Nickolai Zeldovich. Nail: A Practical Tool for Parsing and Generating Data Formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 615–628, Broomfield, CO, October 2014. USENIX Association. Online at https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert.

[28] 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.

[29] Aditi Barthwal and Michael Norrish. Verified, Executable Parsing. In *European Symposium on Programming*, pages 160–174. Springer, 2009.

[30] Marshall A. Beddoe. Network Protocol Analysis using Bioinformatics Algorithms. http://phreakocious.net/PI/PI.pdf, 2004.

[31] Steven M Bellovin. Security Problems in the TCP/IP Protocol Suite. *ACM SIGCOMM Computer Communication Review*, 19(2) pages 32–48, 1989. DOI 10.1145/378444.378449.

[32] Sven Bendel, Thomas Springer, Daniel Schuster, Alexander Schill, Ralf Ackermann, and Michael Ameling. A service infrastructure for the Internet of Things based on XMPP. In *IEEE International Conference on Pervasive Computing*

and Communications Workshops (PerCom Workshops), pages 385–388. IEEE, 2013. DOI 10.1109/PerComW.2013.6529522.

[33] Jay Berkenbilt. QPDF: A Content-Preserving PDF Transformation System. https://github.com/qpdf/qpdf, April 2008.

[34] Andrew R Bernat and Barton P Miller. Anywhere, Any-Time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, pages 9–16, 2011. DOI 10.1145/2024569.2024572.

[35] Bharat Bhushan. Why a single pixel on this wallpaper crashes Android phones? https://www.slashgear.com/why-a-single-pixel-on-this-wallpaper-crashes-android-phones-13624653/, June 13, 2020.

[36] Tim Bienz, Richard Cohn, and James Meehan. *Portable Document Format Reference Manual*. Adobe Systems Incorporated, 1993.

[37] Clement Blaudeau and Natarajan Shankar. A Verified Packrat Parser Interpreter for Parsing Expression Grammars. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 3–17, 2020. DOI 10.1145/3372885.3373836.

[38] Andreas Bogk and Marco Schöpl. The Pitfalls of Protocol Design: Attempting to Write a Formally Verified PDF Parser. In *IEEE Security and Privacy Workshops*, pages 198–203. IEEE, 2014. DOI 10.1109/SPW.2014.36.

[39] Jacir L Bordim, Yasuaki Ito, and Koji Nakano. Accelerating the CKY parsing using FPGAs. *IEICE Transactions on Information and Systems*, 86(5) pages 803–810, 2003. DOI 10.1007/3-540-36265-7_5.

[40] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Towards Automated Protocol Reverse Engineering Using Semantic Information. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, page 51–62, New York, NY, USA, 2014. Association for Computing Machinery. DOI 10.1145/2590296.2590346.

[41] Abdelkader Boudih. Creating State Machines for Attributes on any Ruby Class, 2016. Online at https://github.com/state-machines/state_machines.

[42] Sergey Bratus, Adam J. Crain, Sven M. Hallberg, Daniel P. Hirsch, Meredith L. Patterson, Maxwell Koo, and Sean W. Smith. Implementing a Vertically Hardened DNP3 Control Stack for Power Applications. In *Proceedings of the 2nd Annual Industrial Control System Security Workshop*, ICSS '16, pages 45–53, New York, NY, USA, 2016. ACM. DOI 10.1145/3018981.3018985.

[43] Sergey Bratus, Trey Darley, Michael Locasto, Meredith L. Patterson, Rebecca Bx Shapiro, and Anna Shubina. Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier. *IEEE Security & Privacy*, 12(1) pages 83–87, January 2014. DOI 10.1109/MSP.2014.1.

[44] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit Programming: from Buffer Overflows to "Weird Machines" and Theory of Computation. *;login:*, December 2011.

[45] Sergey Bratus, Meredith L Patterson, and Dan Hirsch. From Shotgun Parsers to more Secure Stacks. *Shmoocon, Nov*, 2013.

[46] Joel F. Brenner. Eyes Wide Shut: The Growing Threat of Cyber Attacks on Industrial Control Systems. *Bulletin of the Atomic Scientists*, 69(5) pages 15–20, 2013. DOI 10.1177/0096340213501372.

[47] Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4) page 481–494, oct 1964. DOI 10.1145/321239.321249.

[48] Christian Buck, Kenneth Heafield, and Bas Van Ooyen. N-gram Counts and Language Models from the Common Crawl. In *LREC*, volume 2, page 4. Citeseer, 2014.

[49] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 317–329, New York, NY, USA, 2007. Association for Computing Machinery. DOI 10.1145/1315245.1315286.

[50] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, volume 8, pages 209–224, 2008.

[51] Cristiano Calcagno and Dino Distefano, Springer. Infer: An Automatic Program Verifier for Memory Safety of C Programs, 2011. DOI 10.1007/978-3-642-20398-5_33.

[52] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving Fast with Software Verification. In *Proceedings of the 7th NASA Formal Methods International Symposium (NFM)*, pages 3–6, Pasadena, CA, USA, 2015. Springer. DOI 10.1007/978-3-319-17524-9_1.

[53] Justin Cappos, Yanyan Zhuang, Daniela Oliveira, Marissa Rosenthal, and Kuo-Chuan Yeh. Vulnerabilities as Blind Spots in Developer's Heuristic-Based

Decision-Making Processes. In *Proceedings of the New Security Paradigms Workshop*, pages 53–62, 2014.

[54] Defense Use Case. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, v.388, 2016.

[55] Antonio Celesti, Maria Fazio, and Massimo Villari. Se clever: A secure message oriented middleware for cloud federation. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 35–40. IEEE, 2013.

[56] Changing Minds. Snowball sampling. http://changingminds.org/explanations/research/sampling/snowball_sampling.htm. (Accessed: 10th January, 2022).

[57] Pierre Chifflier and Geoffroy Couprie. Writing parsers like it is 2017. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 80–92. IEEE, 2017. DOI 10.1109/SPW.2017.39.

[58] Noam Chomsky. Three Models for the Description of Language. *IRE Transactions on Information Theory*, 2(3) pages 113–124, 1956. DOI 10.1109/TIT.1956.1056813.

[59] H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*, San Diego, CA, January 1996. USENIX Association. Online at https://www.usenix.org/conference/usenix-1996-annual-technical-conference/zero-copy-tcp-solaris.

[60] Catalin Cimpanu. Ripple20 Vulnerabilities Will Haunt the IoT Landscape for Years to Come. *ZDNet*, June 16, 2020.

[61] Cristian Ciressan, Eduardo Sanchez, Martin Rajman, and Jean-Cédric Chappelier. An FPGA-Based Syntactic Parser for Real-Life Almost Unrestricted

Context-Free Grammars. In *Field-Programmable Logic and Applications*, pages 590–594, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. DOI 10.5555/647928.739738.

[62] James Clark, Steve DeRose, et al. XML path language (XPath). http://www.w3.org/TR/xpath20/, 1999.

[63] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol Specification Extraction. In *30th IEEE Symposium on Security and Privacy*, pages 110–125. IEEE, 2009. DOI 10.1145/3134600.3134630.

[64] D. Conzon, T. Bolognesi, P. Brizzi, A. Lotito, R. Tomasi, and M. A. Spirito. The VIRTUS Middleware: An XMPP Based Architecture for Secure IoT Communications. In *International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6, July 2012. DOI 10.1109/ICCCN.2012.6289309.

[65] Luigi Coppolino, Salvatore D'Antonio, Ivano Alessandro Elia, and Luigi Romano. *Security Analysis of Smart Grid Data Collection Technologies*, pages 143–156. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. DOI 10.1007/978-3-642-24270-0_11.

[66] G. Couprie. Nom, A Byte-oriented, Streaming, Zero copy, Parser Combinators Library in Rust. In *IEEE Security and Privacy Workshops*, pages 142–148, 2015. DOI 10.1109/SPW.2015.31.

[67] Sam Cowger, Yerim Lee, Nichole Schimanski, Mark Tullsen, Walter Woods, Richard Jones, EW Davis, William Harris, Trent Brunson, Carson Harmon, et al. ICARUS: Understanding De Facto Formats by Way of Feathers and

Wax. In *IEEE Security and Privacy Workshops (SPW)*, pages 327–334. IEEE, 2020. DOI 10.1109/SPW50608.2020.00067.

[68] Adam Crain. Aegis Fuzzer. https://stepfunc.io/products/aegis-fuzzer/.

[69] Bonan Cuan, Aliénor Damien, Claire Delaplace, and Mathieu Valois. Malware Detection in PDF Files Using Machine Learning. In *15th International Conference on Security and Cryptography (SECRYPT)*, page 8, 2018.

[70] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, article 14 (14 pages), USA, 2007. USENIX Association.

[71] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, page 391–402, New York, NY, USA, 2008. Association for Computing Machinery. DOI 10.1145/1455770.1455820.

[72] Weidong Cui, Marcus Peinado, Helen J Wang, and Michael E Locasto. ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *IEEE Symposium on Security and Privacy (SP'07)*, pages 252–266. IEEE, 2007. DOI 10.1109/SP.2007.34.

[73] Nils Anders Danielsson. Total Parser Combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 285–296, 2010. DOI 10.1145/1863543.1863585.

[74] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[75] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats. *Proceedings of the ACM Conference on Programming Languages*, 3(ICFP) pages 1–29, 2019. DOI 10.1145/3341686.

[76] Max Derhak. DemoIccMAX. https://github.com/InternationalColorConsortium/DemoIccMAX, 2019.

[77] Solar Designer. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63, August 1997.

[78] Digital Corpora. Govdocs1 — (nearly) 1 million freely-redistributable files. http://downloads.digitalcorpora.org/corpora/files/govdocs1/by_type/files.jpeg.tar, 2021.

[79] Leonardo Vieira dos Santos Reis, Roberto da Silva Bigonha, Vladimir Oliveira Di Iorio, and Luis Eduardo de Souza Amorim. Adaptable Parsing Expression Grammars. In *Brazilian Symposium on Programming Languages*, pages 72–86. Springer, 2012. DOI 10.1007/978-3-642-33182-4_7.

[80] Patrick Dubroy and Alessandro Warth. Incremental Packrat Parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 14–25, 2017. DOI 10.1145/3136014.3136022.

[81] Thomas Dullien. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2) pages 391–403, 2020. DOI 10.1109/TETC.2017.2785299.

[82] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey,

et al. The Matter of Heartbleed. In *Proceedings of the Internet Measurement Conference*, pages 475–488, 2014. DOI 10.1145/2663716.2663755.

[83] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Commun. ACM*, 13(2) page 94–102, feb 1970. DOI 10.1145/362007.362035.

[84] Romain Edelmann, Jad Hamza, and Viktor Kunčak. Zippy LL(1) Parsing with Derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1036–1051, New York, NY, USA, 2020. Association for Computing Machinery. DOI 10.1145/3385412.3385992.

[85] Eric Eide and John Regehr. Volatiles Are Miscompiled, and What to Do about It. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 255–264, 2008. DOI 10.1145/1450058.1450093.

[86] Ali ElShakankiry and Thomas Dean. Context Sensitive and Secure Parser Generation for Deep Packet Inspection of Binary Protocols. In *15th Annual Conference on Privacy, Security and Trust (PST)*, pages 77–7709, 2017. DOI 10.1109/PST.2017.00019.

[87] Guillaume Endignoux, Olivier Levillain, and Jean-Yves Migeon. Caradoc: A Pragmatic Approach to PDF Parsing and Validation. In *IEEE Security and Privacy Workshops (SPW)*, pages 126–139. IEEE, 2016. DOI 10.1109/SPW.2016.39.

[88] Jon Erickson. *Hacking: The Art of Exploitation.* No starch press, 2008.

[89] Hiroaki Etoh and Kunikazu Yoda. Memory device, stack protection system, computer system, compiler, stack protection method, storage medium and program transmission apparatus, September 6 2005. US Patent 6,941,473.

[90] Tom Evans, Dele Olajide, Florian Schmaus, Jason Sipula, and Guus der Kinderen. A Highly Modular and Portable Open Source XMPP Client Library Written in Java, 2015. Online at http://www.igniterealtime.org/projects/smack/.

[91] George Fairbanks. Principle of Least Expressiveness. *IEEE Software*, 36(3) pages 116–119, 2019. DOI 10.1109/MS.2019.2896876.

[92] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet Dossier. *Symantec Security Response*, 1(1) pages 1–64, 2010.

[93] Feliam. CVE-2013-2729 Adobe Reader X 10.1.4.38 - BMP/RLE heap corruption. Available from Vulners. Online at https://vulners.com/exploitdb/EDB-ID:26703.

[94] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[95] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. The Essence of Bidirectional Programming. *Science China Information Sciences*, 58(5) pages 1–21, 2015. DOI 10.1007/s11432-015-5316-8.

[96] S. N. Foley and W. M. Adams. Trust Management of XMPP Federation. In *IFIP/IEEE International Symposium on Integrated Network Management*, pages 1192–1195, May 2011. DOI 10.1109/INM.2011.5990581.

[97] Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. *ACM SIGPLAN Notices*, 37(9) pages 36–47, 2002. DOI 10.1145/583852.581483.

[98] Bryan Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, 2004.

[99] Jay Freeman. Exploit (& Fix) Android "Master Key". Online at http://www.saurik.com/id/17.

[100] Nathan Fritz. SleekXMPP – Python Jabber/XMPP implementation, 2016. Online at https://github.com/fritzy/SleekXMPP.

[101] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing Science to Digital Forensics with Standardized Forensic Corpora. *Digital Investigation*, v.6 pages S2–S11, 2009.

[102] Michael Gleicher, Danielle Albers, Rick Walker, Ilir Jusufi, Charles D Hansen, and Jonathan C Roberts. Visual comparison for information visualization. *Information Visualization*, 10(4) pages 289–309, 2011. DOI 10.1177/1473871611416549.

[103] E Mark Gold. Language identification in the limit. *Information and control*, 10(5) pages 447–474, 1967.

[104] Leo A Goodman. Snowball Sampling. *The Annals of Mathematical Statistics*, pages 148–170, 1961.

[105] David Graham. Vines - XMPP/Jabber chat server for Ruby, 2014. Online at http://www.getvines.org.

[106] R. D. Graham and P. C. Johnson. Finite State Machine Parsing for Internet Protocols: Faster Than You Think. In *IEEE Security and Privacy Workshops*, pages 185–190, May 2014. DOI 10.1109/SPW.2014.34.

[107] Andy Greenberg. Hack Brief: Microsoft Warns of a 17-Year-Old 'Wormable' Bug. *Wired*, July 14, 2020.

[108] Dick Grune and Ceriel JH Jacobs. A Programmer-Friendly LL (1) Parser Generator. *Software: Practice and Experience*, 18(1) pages 29–38, 1988.

[109] Dick Grune and Ceriel JH Jacobs. *Parsing Techniques: A Practical Guide*. Springer Science & Business Media, 2007. DOI 10.1007/978-0-387-68954-8.

[110] Jerwin Mark L Guillermo and Proceso L Fernandez. Towards a Faster Incremental Packrat Parser. In *12th International Conference on Information & Communication Technology and System (ICTS)*, pages 170–175. IEEE, 2019. DOI 10.1109/ICTS.2019.8850934.

[111] William Harris. DaeDaLus and SPARTA: Enabling Format Users to Describe Extant Formats Precisely and Enabling Programmers to Parse Them Safely. https://galois.com/project/daedalus-and-sparta/, 2019.

[112] David G Hays. Automatic Language-data Processing in Sociology. Technical report, Rand Corporation, 1959.

[113] Fritz Henglein and Ulrik Terp Rasmussen. PEG Parsing in Less Space using Progressive Tabling and Dynamic Analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 35–46, 2017. DOI 10.1145/3018882.3018889.

[114] Ian Henriksen, Gianfranco Bilardi, and Keshav Pingali. Derivative Grammars: A Symbolic Approach to Parsing with Derivatives. *Proc. ACM Program. Lang.*, 3(OOPSLA) article 127 (28 pages), oct 2019. DOI 10.1145/3360553.

[115] Lars Hermerschmidt, Stephan Kugelmann, and Bernhard Rumpe. Towards More Security in Data Exchange: Defining Unparsers with Context-Sensitive

Encoders for Context-Free Grammars. In *IEEE Security and Privacy Workshops*, pages 134–141. IEEE, 2015. DOI 10.1109/SPW.2015.29.

[116] Ben Herzberg, Dima Bekerman, and Igal Zeifman. Breaking Down Mirai: An IoT DDoS Botnet Analysis, 2016. Online at https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html.

[117] Hispasec Sistemas. Virustotal. http://www.virustotal.com/, June 2004.

[118] Katie Hockman. Go Fuzzing. https://go.dev/doc/fuzz/, February 2021.

[119] Dominik Honnef. Staticcheck Linter for the Go Programming Language. https://staticcheck.io/, 2019.

[120] Horst Rutter et al. PDFCPU: a Go PDF processor. https://github.com/pdfcpu/pdfcpu, 2017.

[121] Matthias Hoschele and Andreas Zeller. Mining input grammars with AUTOGRAM. In *IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 31–34. IEEE, 2017. DOI 10.1109/ICSE-C.2017.14.

[122] Graham Hutton and Erik Meijer. Monadic Parser Combinators, 1996.

[123] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The p4 -> netfpga workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 1–9, 2019.

[124] IBM. MQTT Version 3.1. *Standard, OASIS*, 2014.

[125] IEEE Standard. IEEE Standard for Synchrophasor Data Transfer for Power Systems. *IEEE Std C37.118.2-2011 (Revision of IEEE Std C37.118-2005)*, pages 1–53, Dec 2011. DOI 10.1109/IEEESTD.2011.6111222.

[126] Industrial Control Systems Cyber Emergency Response Team. OSIsoft PI Interface for IEEE C37.118 Memory Corruption and Consumption through crafted packets. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2800, 2013.

[127] Industrial Control Systems Cyber Emergency Response Team. OSIsoft PI Interface for IEEE C37.118 shutdown and data collection outages through crafted packets. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2801, 2013.

[128] Industrial Control Systems Cyber Emergency Response Team. OSIsoft Multiple Vulnerabilities. https://ics-cert.us-cert.gov/advisories/ICSA-13-225-02, 2015.

[129] Insider Application Security Team. Insider Static Application Security Testing (SAST) engine. https://github.com/insidersec/insider.

[130] International Color Consortium. Specification ICC.2:2019. https://www.color.org/specification/ICC.2-2019.pdf, 2019.

[131] International Color Consortium. Specification ICC.2:2019 Cumulative Errata List. https://www.color.org/iccmax/ICC.2-2019_Cumulative_Errata_List_2021-09-09.pdf, September 2021.

[132] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. Practical, General Parser Combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, page 1–12, New York, NY, USA, 2016. Association for Computing Machinery. DOI 10.1145/2847538.2847539.

[133] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[134] Erwin Janssen, Frits Vaandrager, Joeri de Ruiter, and Erik Poll. Fingerprinting TLS Implementations Using Model Learning. *Radboud University Master's Thesis Computing Science in Cybersecurity*, 2021.

[135] Patrik Jansson and Jean-Philippe Bernardy. Certified context-free parsing: A formalisation of Valiant's algorithm in Agda. *Logical Methods in Computer Science*, v.12, 2016.

[136] Jay Freeman (saurik). Exploit (& Fix) Android "Master Key"; Android Bug Superior to Master Key; Yet Another Android Master Key Bug. http://www.saurik.com/id/17; http://www.saurik.com/id/18; http://www.saurik.com/id/19, August 2013.

[137] Trevor Jim and Yitzhak Mandelbaum. Delayed Semantic Actions in Yakker. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, LDTA '11, article 8 (8 pages), New York, NY, USA, 2011. Association for Computing Machinery. DOI 10.1145/1988783.1988791.

[138] Trevor Jim, Yitzhak Mandelbaum, and David Walker. Semantics and Algorithms for Data-Dependent Grammars. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 417–430, New York, NY, USA, 2010. Association for Computing Machinery. DOI 10.1145/1706299.1706347.

[139] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Evaluating GLR parsing algorithms. *Science of Computer Programming*, 61(3) pages 228–244, 2006.

[140] K. Kain, S. W. Smith, and R. Asokan. Digital Signatures and Electronic Documents: A Cautionary Tale. In *Advanced Communications and Multimedia Se-*

curity: IFIP TC6 / TC11 Sixth Joint Working Conference on Communications and Multimedia Security September 26–27, 2002, Portorož, Slovenia, pages 293–307, Boston, MA, 2002. Springer US. DOI 10.1007/978-0-387-35612-9_22.

[141] Dan Kaminsky. CVE-2009-3555 The Mozilla Network Security Services (NSS) fails to properly validate the domain name in a signed CA certificate, allowing attackers to substitute malicious SSL certificates for trusted ones. Available from Vulners. Online at https://vulners.com/exploitdb/EDB-ID:26703.

[142] Dan Kaminsky, Meredith L Patterson, and Len Sassaman. PKI Layer Cake: New Collision Attacks against the Global X. 509 Infrastructure. In *International Conference on Financial Cryptography and Data Security*, pages 289–303. Springer, 2010.

[143] Prashant Kansal and Anjan Bose. Bandwidth and Latency Requirements for Smart Transmission Grid Applications. *IEEE Transactions on Smart Grid*, 3(3) pages 1344–1352, 2012.

[144] Tadao Kasami and Koji Torii. A Syntax-Analysis Procedure for Unambiguous Context-Free Grammars. *Journal of the ACM (JACM)*, 16(3) pages 423–431, 1969.

[145] Vidita V. Kaushik. Point to Point Protocol Daemon demonized by CVE-2020-8597. https://www.secpod.com/blog/rce-in-pppd-cve-2020-8597/, 2020.

[146] Brian W Kernighan and Dennis M Ritchie. *The M4 macro processor*. Bell Laboratories Murray Hill, NJ, 1977.

[147] Samir Khuller, Anna Moss, and Joseph (Seffi) Naor. The Budgeted Maximum Coverage Problem. *Information Processing Letters*, 70(1) pages 39–45, 1999. DOI https://doi.org/10.1016/S0020-0190(99)00031-9.

[148] Michael Kirsche and Ronny Klauck. Unify to Bridge Gaps: Bringing XMPP into the Internet of Things. In *IEEE International Conference on Pervasive Computing and Communications (PerCom) Workshops*, pages 455–458. IEEE, 2012.

[149] Doron Kletter. Effective System and Method for Visual Document Comparison Using Localized Two-Dimensional Visual Fingerprints, December 6 2016. US Patent 9,514,103.

[150] Moshe Kol and Shlomi Oberman. Ripple20 CVE-2020-11896 RCE and CVE-2020-11898 Info Leak. Technical report, JSOF, June 2020.

[151] Moshe Kol, Ariel Schön, and Shlomi Oberman. Ripple20 CVE-2020-11901. Technical report, JSOF, August 2020.

[152] Adam Koprowski and Henri Binsztok. TRX: A Formally Verified Parser Interpreter. In *European Symposium on Programming*, pages 345–365. Springer, 2010.

[153] Vijay Kothari, Prashant Anantharaman, Sean Smith, Briland Hitaj, Prashanth Mundkur, Natarajan Shankar, Letitia Li, Iavor Diatchki, and William Harris. Capturing the iccMAX calculator Element: A Case Study on Format Design. In *Proceedings of the IEEE Security and Privacy Workshops (SPW)*. IEEE, 2022.

[154] Tammo Krueger, Hugo Gascon, Nicole Krämer, and Konrad Rieck. Learning Stateful Models for Network Honeypots. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, pages 37–48, 2012. DOI 10.1145/2381896.2381904.

[155] Kopperla Ranjith Kumar, Katyayani Sesha Kumar Kavuluri, and Debabrata Das. Novel Algorithm to Recover the Lost CDR Information by Control and

User Planes Separation in 4G and 5G. In *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–6. IEEE, 2021. DOI 10.1109/CONECCT52877.2021.9622598.

[156] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery. DOI 10.1145/2535838.2535841.

[157] Haboob labs. Stack-based Buffer Overflow (CWE-121). https://helpx.adobe.com/security/products/acrobat/apsb21-55.html, November 2021.

[158] Adam Laurie and Grzegorz Wypych. New Vulnerability Could Put IoT Devices at Risk. *Security Intelligence*, August 19, 2020.

[159] Kyle Lavorato and Thomas Dean. Deep Grammar Optimization for Submessagae *(sic)* Structure of Network Protocol Parsers. Technical report, Queens University, 2020.

[160] Kyle Lavorato, Fahim T. Imam, and Thomas R. Dean. LL(k) Optimization of a Network Protocol Parser Generator. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 183–192, USA, 2019. IBM Corporation.

[161] DoHyun Lee. Polaris Office Uninitialized Pointer Vulnerability (Remote Code Execution). https://github.com/dlehgus1023/CVE/tree/master/CVE-2021-34280, November 2021.

[162] Lillian Lee. Learning of Context-Free Languages: A Survey of the Literature. https://dash.harvard.edu/handle/1/25104425, 1996.

[163] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical report, Microsoft Corporation, 2001.

[164] Joop M.I.M. Leo. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoretical Computer Science*, 82(1) pages 165–176, 1991. DOI https://doi.org/10.1016/0304-3975(91)90180-A.

[165] Xavier Leroy. *The CompCert C verified compiler: Documentation and user's manual*. PhD thesis, Inria, 2020.

[166] Olivier Levillain, Sébastien Naud, and Aina Toky Rasoamanana. Work-in-Progress: Towards a Platform to Compare Binary Parser Generators. https://paperstreet.picty.org/yeye/resources/conf-spw-LevillainNR21/document.pdf, May 2021.

[167] Guodong Li, Indradeep Ghosh, and Sreeranga P Rajan. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *International Conference on Computer Aided Verification*, pages 609–615. Springer, 2011. DOI 10.1007/978-3-642-22110-1_49.

[168] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3) pages 1199–1218, 2018. DOI 10.1109/TR.2018.2834476.

[169] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution. In *NDSS*, volume 8, pages 1–15. Citeseer, 2008.

[170] Nuno Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. Automatically verifying reachability and well-formedness in P4 Networks. Technical report, Technical Report, 2016.

[171] Zephyr S Lucas, Joanna Y Liu, Prashant Anantharaman, and Sean W Smith. Parsing PEGs with Length Fields in Software and Hardware. In *IEEE Security and Privacy Workshops (SPW)*, pages 128–133. IEEE, 2021. DOI 10.1109/SPW53761.2021.00026.

[172] Stefan Lucks, Norina Marie Grosch, and Joshua König. Taming the Length Field in Binary Data: Calc-Regular Languages. In *IEEE Security and Privacy Workshops (SPW)*, pages 66–79, 2017. DOI 10.1109/SPW.2017.33.

[173] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Notices*, 40(6) page 190–200, jun 2005. DOI 10.1145/1064978.1065034.

[174] Lucas Lundgren. Light-Weight Protocol! Serious Equipment! Critical Implications! *DEFCON*, 2016.

[175] Greg MacManus. CVE-2013-2028 Nginx HTTP Server 1.3.9-1.4.0 Chunked Encoding Stack Buffer Overflow. Available from Rapid 7. Online at https://www.rapid7.com/db/modules/exploit/linux/http/nginx_chunked_size.

[176] Jonas Magazinius, Billy K. Rios, and Andrei Sabelfeld. Polyglots: Crossing Origins by Crossing Formats. In *Proceedings of the ACM SIGSAC Conference on Computer Communications Security*, CCS '13, page 753–764, New York, NY, USA, 2013. Association for Computing Machinery. DOI 10.1145/2508859.2516685.

[177] Christian Mainka, Vladislav Mladenov, Simon Rohlmann, and Jörg Schwenk. Attacks Bypassing the Signature Validation in PDF. Technical report, Ruhr-Universität Bochum, March 2020. https://www.nds.ruhr-uni-bochum.de/media/ei/veroeffentlichungen/2019/02/12/report.pdf.

[178] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. PADS/ML: A Functional Data Description Language. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 77–83, New York, NY, USA, 2007. Association for Computing Machinery. DOI 10.1145/1190216.1190231.

[179] Daniel Marjamäki. CppCheck: Static Analysis of C/C++ code. https://github.com/danmar/cppcheck, 2007.

[180] C Mary. ShellShock Attack on Linux Systems–Bash. *International Research Journal of Engineering and Technology*, 2(8) pages 1322–1325, 2015.

[181] Robert E McGrath. Data Format Description Language: Lessons Learned, Concepts and Experience. https://core.ac.uk/display/4835576?utm_source=pdf, August 2011.

[182] David Boyd Mercer. *Attributed Parsing Expression Grammars*. University of South Alabama, 2008.

[183] Microsoft. /SAFESEH (Safe Exception Handlers), 2003. [Online; accessed 21-September-2019], Online at http://msdn2.microsoft.com/en-us/library/9a89h429.aspx.

[184] Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: A functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 189–195,

New York, NY, USA, 2011. Association for Computing Machinery. DOI 10.1145/2034773.2034801.

[185] Bart Miller. Fuzzing Creation Assignment, 1988.

[186] Michael Millian, Prashant Anantharaman, Sergey Bratus, Sean Smith, and Michael Locasto. Converting an Electric Power Utility Network to Defend Against Crafted Inputs. In *Proceedings of the International Conference on Critical Infrastructure Protection*, pages 73–85. Springer, 2019. DOI 10.1007/978-3-030-34647-8_4.

[187] Rupendra Nath Mitra, Mohamed M Kassem, Jon Larrea, and Mahesh K Marina. CUPS Hijacking in Mobile RAN Slicing: Modeling, Prototyping, and Analysis. In *2021 IEEE Conference on Communications and Network Security*. Institute of Electrical and Electronics Engineers (IEEE), 2021.

[188] Mitre Corporation. CVE – Common Vulnerabilities and Exposures. Online at https://cve.mitre.org/.

[189] Mitre Corporation. CVE-2004-0597 : Multiple buffer overflows in libpng 1.2.5. Available from MITRE CVE-ID CVE-2004-0597, 2004.

[190] Mitre Corporation. CWE-20: Improper Input Validation, 2006. https://cwe.mitre.org/data/definitions/20.html.

[191] Mitre Corporation. CWE-805: Buffer Access with Incorrect Length Value, 2010. https://cwe.mitre.org/data/definitions/805.html.

[192] Mitre Corporation. Arbiter Power Sentinel Denial of Service Attack. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3012, 2012.

[193] Ingo Molnar. ExecShield, 2004. Online at http://www.redhat.com/f/pdf/rhel/WHP0006USExecshield.pdf.

[194] Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In *IEEE Cybersecurity Development (SecDev)*, pages 45–52, 2016. DOI 10.1109/SecDev.2016.019.

[195] Cameron Morris. CVE-2015-1427 The Groovy scripting engine in Elasticsearch before 1.3.8 and 1.4.x before 1.4.3 allows remote attackers to bypass the sandbox protection mechanism and execute arbitrary shell commands via a crafted script. Available from Vulners. Online at https://vulners.com/cve/CVE-2015-1427.

[196] Noam Moshe, Sharon Brizinov, Raul Onitza-Klugman, and Kirill Efimov. Exploiting URL Parsers: The Good, Bad, and Inconsistent. https://claroty.com/wp-content/uploads/2022/01/Exploiting-URL-Parsing-Confusion.pdf, January 2022.

[197] Jens Müller, Dominik Noss, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. Processing Dangerous Paths. In *Network and Distributed Systems Security (NDSS) Symposium*. NDSS, 2021. DOI 10.14722/ndss.2021.23109.

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

[199] Arvind M Murching, YV Prasad, and YN Srikant. Incremental Recursive Descent Parsing. *Computer Languages*, 15(4) pages 193–204, 1990. DOI 10.1016/0096-0551(90)90020-P.

[200] Ryo Nakamura. Xrc - XMPP Ruby Client, 2014. Online at https://github.com/r7kamura/xrc.

[201] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM Sigplan Notices*, 42(6) pages 89–100, 2007. DOI 10.1145/1273442.1250746.

[202] Mary Nielsen and Michael Stokes. The creation of the sRGB ICC profile. In *Color and Imaging Conference*, volume 1998, pages 253–257. Society for Imaging Science and Technology, 1998.

[203] Aimaschana Niruntasukrat, Chavee Issariyapat, Panita Pongpaibool, Koonlachat Meesublak, Pramrudee Aiumsupucgul, and Anun Panya. Authorization mechanism for MQTT-based Internet of Things. In *IEEE International Conference on Communications Workshops (ICC)*, pages 290–295. IEEE, 2016. DOI 10.1109/ICCW.2016.7503802.

[204] Nir Nissim, Aviad Cohen, Robert Moskovitch, Asaf Shabtai, Matan Edri, Oren BarAd, and Yuval Elovici. Keeping pace with the creation of new malicious PDF files using an active-learning based detection framework. *Security Informatics*, 5(1) pages 1–20, 2016. DOI 10.1186/s13388-016-0026-3.

[205] Derek Noonburg. Poppler, a PDF rendering library. https://github.com/freedesktop/poppler.

[206] Steffen Nurpmeso. Mutool Clean: Endless Loop. https://bugs.ghostscript.com/show_bug.cgi?id=703092, 2020.

[207] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A DeLong, Justin Cappos, and Yuriy Brun. API Blindspots: Why Experienced Developers Write Vulnerable Code. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 315–328, 2018.

[208] Aleph One. Smashing the Stack for Fun and Profit. *Phrack magazine*, 7(49) pages 14–16, 1996.

[209] Alfredo Ortega. OpenBSD's IPv6 mbufs remote kernel buffer overflow. Available from Vulners. Online at https://vulners.com/cert/VU:986425.

[210] Bryan O'Sullivan. AttoParsec: A fast Haskell library for parsing ByteStrings. https://github.com/haskell/attoparsec.

[211] K. Palani, E. Holt, and S. Smith. Invisible and forgotten: Zero-day blooms in the IoT. In *IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6, March 2016. DOI 10.1109/PerComW.2016.7457163.

[212] Terence Parr and Kathleen Fisher. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 425–436, New York, NY, USA, 2011. Association for Computing Machinery. DOI 10.1145/1993498.1993548.

[213] Yevgeny Pats. Pythonfuzz: Coverage-Guided Fuzz Testing for Python. https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/pythonfuzz, 2020.

[214] Meredith L. Patterson. Hammer: Parser Combinators in C, 2016. Online at https://github.com/UpstandingHackers/hammer.

[215] PaX. Address Space Layout Randomization, 2001. [Online; accessed 21-September-2019], Online at http://pax.grsecurity.net/docs/aslr.txt.

[216] PDF Association. ISO 32000-2 (PDF 2.0). Technical report, International Organization for Standardization, 2017.

[217] PDF Association. DARPA's SafeDocs initiative. https://www.pdfa.org/darpas-safedocs-initiative/, August 2018.

[218] Benjamin C Pierce. *Types and Programming Languages*. MIT press, 2002.

[219] Sebastian Poeplau and Aurélien Francillon. Symbolic Execution with SymCC: Don't Interpret, Compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[220] E. Poll, J. D. Ruiter, and A. Schubert. Protocol State Machines and Session Languages: Specification, Implementation, and Security Flaws. In *IEEE Security and Privacy Workshops*, pages 125–133, May 2015. DOI 10.1109/SPW.2015.32.

[221] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, 2019.

[222] Vijay Prabhu. 6 Bugs in Qualcomm allow hackers to remotely take over your Android smartphone. *Android Rookies*, August 8, 2020. https://androidrookies.com/6-bugs-in-qualcomm-allow-hackers-to-remotely-take-over-your-android-smartphone/.

[223] PyCQA. Bandit: Find common security issues in Python code. https://github.com/PyCQA/bandit, 2018.

[224] Terry Quatrani. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley Professional, 2000.

[225] Ranjan Kumar Rahul. *An SCL-based Constraint Representation Language for Intrusion Detection*. PhD thesis, Queen's University (Canada), 2017.

[226] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1465–1482, 2019.

[227] Tobias Reiher, Alexander Senier, Jeronimo Castrillon, and Thorsten Strufe. RecordFlux Github. https://github.com/Componolit/RecordFlux.

[228] Tobias Reiher, Alexander Senier, Jeronimo Castrillon, and Thorsten Strufe. RecordFlux: Formal Message Specification and Generation of Verifiable Binary Parsers. In *Formal Aspects of Component Software*, pages 170–190, Cham, 2020. Springer International Publishing. DOI 10.1007/978-3-030-40914-2_9.

[229] Sasha Rezvina. Rails' Remote Code Execution Vulnerability Explained. Online at https://codeclimate.com/blog/rails-remote-code-execution-vulnerability-explained.

[230] Tom Ridge. Simple, Functional, Sound and Complete Parsing for All Context-Free Grammars. In *Certified Programs and Proofs*, pages 103–118, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. DOI 10.1007/978-3-642-25379-9_10.

[231] Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, and Heinz Züllighoven. Serializer. *Pattern Languages of Program Design*, v.3, 1997.

[232] Marko A Rodriguez. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10, 2015. DOI 10.1145/2815072.2815073.

[233] Simon Rohlmann, Vladislav Mladenov, Christian Mainka, and Jörg Schwenk. Breaking the Specification: PDF Certification. In *IEEE Symposium on Security and Privacy (SP)*, pages 1485–1501, 2021. DOI 10.1109/SP40001.2021.00110.

[234] J. M. Rushby. Design and Verification of Secure Systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, page 12–21, New York, NY, USA, 1981. Association for Computing Machinery. DOI 10.1145/800216.806586.

[235] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Address Format. *RFC 6122*, March 2011.

[236] Peter Saint-Andre. A Public Statement Regarding Ubiquitous Encryption on the XMPP Network, 2014. Online at https://github.com/stpeter/manifesto/blob/master/manifesto.txt.

[237] Jerome H Saltzer and Michael D Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9) pages 1278–1308, 1975.

[238] Len Sassaman, Meredith L Patterson, Sergey Bratus, and Michael E Locasto. Security Applications of Formal Language Theory. *IEEE Systems Journal*, 7(3) pages 489–500, 2013. DOI 10.1109/JSYST.2012.2222000.

[239] Fred B Schneider, Greg Morrisett, and Robert Harper. A Language-based Approach to Security. In *Informatics*, pages 86–101. Springer, 2001. DOI 10.1007/3-540-44577-3_6.

[240] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium*, pages 25–41, San Francisco, CA, 2011. USENIX.

[241] Donn Seeley. A Tour of the Worm. In *Proceedings of 1989 Winter USENIX Conference, Usenix Association, San Diego, CA, February*, 1989.

[242] Géraud Sénizergues. The Equivalence Problem for Deterministic Pushdown Automata is Decidable. In *International Colloquium on Automata, Languages, and Programming*, pages 671–681. Springer, 1997. DOI 10.1007/3-540-63165-8_221.

[243] Kosta Serebryany. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *Proceedings of the IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016. DOI 10.1109/SecDev.2016.043.

[244] Kostya Serebryany. OSS-Fuzz-Google's Continuous Fuzzing Service for Open Source Software. *USENIX Security '17*, 2017.

[245] Kushal Arvind Shah. Fortinet Discovers Corel PDF Fusion Heap Memory Corruption Vulnerability. https://www.fortiguard.com/zeroday/FG-VD-21-027, September 2021.

[246] Yusuke Shinyama. PDFMiner: A Text Extraction Tool for PDF Documents. https://pypi.org/project/pdfminer/, 2020.

[247] Siguza. Psychic Paper. https://siguza.github.io/psychicpaper/, May 2020.

[248] William Simpson. RFC1661: the point-to-point protocol (PPP), 1994.

[249] M. Singh, M. A. Rajan, V. L. Shivraj, and P. Balamuralidhar. Secure MQTT for Internet of Things (IoT). In *International Conference on Communication*

Systems and Network Technologies (CSNT), pages 746–751, April 2015. DOI 10.1109/CSNT.2015.16.

[250] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[251] Chris Sistrunk and Adam Crain. Project Robus: Serial Killer. https://dale-peterson.com/2014/01/23/s4x14-video-crain-sistrunk-project-robus-master-serial-killer/, 2014.

[252] Sean Smith and John Marchesini. *The Craft of System Security*. Pearson Education, 2007.

[253] Sean W Smith, Ross Koppel, Jim Blythe, and Vijay Kothari. Mismorphism: A Semiotic Model of Computer Security Circumvention. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pages 1–2, 2015. DOI 10.1145/2746194.2746219.

[254] Charles Smutz and Angelos Stavrou. Malicious PDF Detection using Metadata and Structural Features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 239–248, 2012. DOI 10.1145/2420950.2420987.

[255] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen K Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. MD5 considered harmful today, creating a rogue CA certificate. In *25th Annual Chaos Communication Congress*, 2008.

[256] Eugene H Spafford. The Internet Worm Program: An Analysis. *ACM SIGCOMM Computer Communication Review*, 19(1) pages 17–57, 1989. DOI 10.1145/66093.66095.

[257] Michele Spagnuolo. Abusing JSONP with Rosetta Flash. Online at https://miki.it/blog/2014/7/8/abusing-jsonp-with-rosetta-flash/.

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

[259] Ryan Speers, Paul Li, Sophia d'Antoine, and Michael Locasto. Analysis Methods and Tooling for Parsers. https://www.riverloopsecurity.com/blog/2020/06/safedocs-pdf-analysis-methods-intro/, June 2020.

[260] Nedim Šrndic and Pavel Laskov. Detection of Malicious PDF Files based on Hierarchical Document Structure. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, pages 1–16. Citeseer, 2013.

[261] William Stallings. *Computer Networking with Internet Protocols and Technology*. Pearson/Prentice Hall Upper Saddle River, NJ, USA, 2004.

[262] OMG Standard. The Real-Time Publish-Subscribe Wire Protocol: DDS Interoperability Wire Protocol Specification Version 2.1. https://www.omg.org/spec/DDSI-RTPS/2.5/PDF, 2014.

[263] Daniel Stenberg. Don't Mix URL Parsers. https://daniel.haxx.se/blog/2022/01/10/dont-mix-url-parsers/, January 2022.

[264] John Stewart, Thomas Maufer, Rhett Smith, Chris Anderson, and Eren Ersonmez. Synchrophasor Security Practices. *Tennessee Valley Authority Report*, 2011.

[265] Michael Stokes. The History of the ICC. In *Color and Imaging Conference*, volume 1997, pages 266–269. Society for Imaging Science and Technology, 1997.

[266] Zhaoxia Sun, Ping Du, Akihiro Nakao, Lei Zhong, and Ryokichi Onishi. Building Dynamic Mapping with CUPS for Next Generation Automotive Edge Computing. In *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, pages 1–6, 2019. DOI 10.1109/CloudNet47604.2019.9064135.

[267] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. Hardening Attack Surfaces with Formally Proven Binary Format Parsers. In *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*, June 2022.

[268] Francisco Tacliad, Thuy D Nguyen, and Mark Gondree. DoS Exploitation of Allen-Bradley's Legacy Protocol through Fuzz Testing. In *Proceedings of the 3rd Annual Industrial Control System Security Workshop*, pages 24–31, 2017. DOI 10.1145/3174776.3174780.

[269] S. Taylor. Protecting Embedded Systems from Zero-Day Attacks. In *NAECON 2018 - IEEE National Aerospace and Electronics Conference*, pages 165–168, 2018. DOI 10.1109/NAECON.2018.8556791.

[270] The Kaggle Team. Leveraging ML to Fuel New Discoveries with the arXiv Dataset. https://medium.com/@kaggleteam/leveraging-ml-to-fuel-new-discoveries-with-the-arxiv-dataset-981a95bfe365, August 2020.

[271] Susan Thomson, Christian Huitema, Vladimir Ksinant, and Mohsen Souissi. DNS extensions to support IP version 6. Technical report, RFC 1886, December, 1995.

[272] Sagi Tzadik. SIGRed – Resolving Your Way into Domain Admin: Exploiting a 17 Year-old Bug in Windows DNS Servers. *Check Point Research*, July 14, 2020.

[273] K. Underwood and M. E. Locasto. In Search of Shotgun Parsers in Android Applications. In *IEEE Security and Privacy Workshops (SPW)*, pages 140–155, May 2016. DOI 10.1109/SPW.2016.41.

[274] Stephen H. Unger. A Global Parser for Context-Free Phrase Structure Grammars. *Commun. ACM*, 11(4) page 240–247, apr 1968. DOI 10.1145/362991.363001.

[275] Y. Upadhyay, A. Borole, and D. Dileepan. MQTT Based Secured Home Automation System. In *Symposium on Colossal Data Analysis and Networking (CDAN)*, pages 1–4, March 2016. DOI 10.1109/CDAN.2016.7570945.

[276] Alfonso Valdes, Richard Macwan, and Matthew Backes. Anomaly Detection in Electrical Substation Circuits via Unsupervised Machine Learning. In *IEEE 17th International Conference on Information Reuse and Integration (IRI)*, pages 500–505, 2016. DOI 10.1109/IRI.2016.74.

[277] Marcell van Geest and Wouter Swierstra. Generic Packet Descriptions: Verified Parsing and Pretty Printing of Low-Level Data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, page 30–40, New York, NY, USA, 2017. Association for Computing Machinery. DOI 10.1145/3122975.3122979.

[278] Jules van Thoor, Joeri de Ruiter, and Erik Poll. Learning State Machines of TLS 1.3 Implementations. *Bachelors Thesis Technical Report*, 2018.

[279] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 122–135, New York, NY, USA, 2017. Association for Computing Machinery. DOI 10.1145/3050220.3050234.

[280] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. *SIGCOMM Comput. Commun. Rev.*, 34(4) page 193–204, aug 2004. DOI 10.1145/1030194.1015489.

[281] Yipeng Wang, Zhibin Zhang, Danfeng Daphne Yao, Buyun Qu, and Li Guo. Inferring Protocol State Machine from Network Traces: A Probabilistic Approach. In *International Conference on Applied Cryptography and Network Security*, pages 1–18. Springer, 2011. DOI 10.1007/978-3-642-21554-4_1.

[282] John W Webster III. Method and apparatus for visually comparing files in a data processing system, August 25 1992. US Patent 5,142,619.

[283] David Wheeler. Flawfinder. https://security.web.cern.ch/recommendations/en/codetools/flawfinder.shtml, 2001.

[284] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular Specification and Verification of Go Programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 367–379, Cham, 2021. Springer International Publishing. DOI 10.1007/978-3-030-81685-8_17.

[285] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In

Proceedings of the Workshop on Hot Topics in Operating Systems, pages 206–212, 2021. DOI 10.1145/3458336.3465283.

[286] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. Automatic Network Protocol Analysis. In *NDSS*, volume 8, pages 1–14. Citeseer, 2008.

[287] Gary R Wright and W Richard Stevens. *TCP/IP Illustrated, Volume 2 (paperback): The Implementation*. Addison-Wesley Professional, 1995.

[288] Peter Wyatt. Cyclic vs acyclic linked list chains via dictionary keys is ambiguous. https://github.com/pdf-association/pdf-issues/issues/102, 2021.

[289] Peter Wyatt. Mandated indirect reference requirements not followed and not enforced - are they really requirements? https://github.com/pdf-association/pdf-issues/issues/106, July 2021.

[290] Peter Wyatt. Table 29 lang key and invalid values. https://github.com/pdf-association/pdf-issues/issues/105, July 2021.

[291] Peter Wyatt. The DemoIccMAX Project. https://github.com/petervwyatt/DemoIccMAX, 2021.

[292] Linda Xiao. Automatic Generation of Input Grammars Using Symbolic Execution. https://digitalcommons.dartmouth.edu/senior_theses/163, 2020.

[293] Y. Yang, K. McLaughlin, S. Sezer, T. Littler, B. Pranggono, P. Brogan, and H.F. Wang. Intrusion Detection System for Network Security in Synchrophasor Systems. *IET Conference Proceedings*, pages 246–252(6), January 2013. Online at http://digital-library.theiet.org/content/conferences/10.1049/cp.2013.0059.

[294] Tim Yardley. Building A Physical Testbed For Blackstart Restoration. https://www.youtube.com/watch?v=lGPDmOjaOO4.

[295] Michael Yashkin, KOLANICH, Petr Pucil, and Stefanos Mandalas. Kaitai Struct. https://kaitai.io/, 2015.

[296] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. NetPlier: Probabilistic Network Protocol Reverse Engineering from Message Traces. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS'21)*, 2021.

[297] Zachary Yedidia and Stephen Chong. Fast Incremental PEG Parsing. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, pages 99–112, 2021. DOI 10.5281/zenodo.5573543.

[298] Daniel H. Younger. Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, 10(2) pages 189–208, 1967. DOI https://doi.org/10.1016/S0019-9958(67)80007-X.

[299] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011.

[300] ZecOps Research Team. You've Got (0-click) Mail! Technical report, ZecOps, April 20, 2020. https://blog.zecops.com/vulnerabilities/youve-got-0-click-mail/.

[301] Liang Zhang, David Choffnes, Dave Levin, Tudor Dumitraş, Alan Mislove, Aaron Schulman, and Christo Wilson. Analysis of SSL certificate reissues and revocations in the wake of Heartbleed. In *Proceedings of the Internet Measurement Conference*, pages 489–502, 2014. DOI 10.1145/2663716.2663758.

[302] Bonnie Zhu, Anthony Joseph, and Shankar Sastry. A Taxonomy of Cyber Attacks on SCADA Systems. In *4th International Conference on Cyber, Physical and Social Computing*, pages 380–388. IEEE, 2011. DOI 10.1109/iThings/CPSCom.2011.34.

[303] Bonnie Zhu and Shankar Sastry. SCADA-specific Intrusion Detection/Prevention Systems: A Survey and Taxonomy. In *Proceedings of the 1st Workshop on Secure Control Systems (SCS)*, volume 11, page 7, 2010.

Index