Implementation Security, or a language-theoretic security analysis of OpenFlow and what I found there.

a thesis presented
by
Jason Meltzer
to
The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of
Master of Science
in the subject of
Internetworking

University of Alberta
Edmonton, Alberta
November 2013

Thesis advisor: Dr. Mike MacGregor Jason Meltzer

Implementation Security, or a language-theoretic security analysis of OpenFlow and what I found there.

Abstract

Implementation is the most immediate aspect of security in the construction of real-world distributed computing systems; designs have flaws but vulnerabilities themselves manifest in implementations. The challenge of securing systems during implementation is apparent from observing the computing world struggle to manage vulnerabilities. The fact is that the fundamental structure of the implementation security problem is still not well-defined or even easily discernible, and might never be. In spite of the lack of a encompassing problem to solve, I will argue that we can work with the knowledge gained from failures, using the knowledge to develop heuristic approaches that have practical value.

In this paper I introduce language-theoretic security (LANGSEC), an approach to software security that supports reasoning about some types of implementation vulnerabilities as phenomena emergent from formally UNDECIDABLE language recognition problems. As a demonstration I apply the LANGSEC approach to the analysis of OpenFlow, the protocol which underpins the exceedingly popular Software-Defined Networking model of network virtualization. In the analysis I prove that OpenFlow is context-sensitive and discuss the grim consequences of this complexity, for OpenFlow implementation security and for verification schemes in OpenFlow-based networks.

Following from the discussions of implementation security and LANGSEC, I will conclude this paper by introducing a heuristic device for reasoning about implementation security relevant design attributes, a device which I currently refer to as "the Map".

Contents

Listing of figures

Dedicated to the memories of some fine people and a once great organization, stark reminders of impermanence:

- Cédric 'Sid' Blancher;
- Len Sassaman;
- M. A. Padlipsky; and,
- The BlackBerry Security Research Group.

Semper malitiosi in potentia

Acknowledgments

THIS THESIS HAD A LONG AND SOMEWHAT DIFFICULT GESTATION, years of prevarication and stubbornness nearly resulted in it not happening. Now that it is finished I can truly offer my appreciation to those who helped the birthing process; I know I'm a difficult patient. Chief among those due thanks is Josh Ryder, for his dedication to thesis-midwifery and patience with my procrastinations. Dr. Patrick Pilarski, for managing to get me out of the house and keeping me socialized. Alexandra, for making tea and being awesome. Dr. Mike Macgregor, for showing exceptional patience. Last but not least: my parents and brother, for pretty much everything else.

Men are terribly in need of suggestion, and this dangerous need for suggestion is one of my main themes today. My theme is large. I have worked hard but gladly to present it as simply as I can. I fear I have not fully succeeded, and I must ask for your active cooperation.

But I would also ask you not to believe ***anything*** *that I suggest! Do not believe a word! I know that that is asking too much, as I will speak only the truth, as well as I can. But I warn you: I know* ***nothing****, or* ***almost nothing****. We all know nothing or almost nothing. I* ***conjecture*** *that that is a basic fact of life. We know nothing, we can only conjecture: we guess. Our best knowledge is the wonderful scientific knowledge we have built up over 2,500 years. But the natural sciences consist precisely of conjectures or hypotheses.*

– Sir Karl Popper, from "Epistemology and the Problem of Peace"

1 Grab a thread and pull

1.1 Introductions

Implementation is the most immediate aspect of security in the construction of real-world distributed computing systems. Designs have flaws but vulnerabilities themselves manifest in implementations, where repair often has a great cost. The scale of the challenge presented by implementation is certainly apparent from observing the computing world struggle to manage vulnerabilities. It is perhaps arguable that the fundamental structure of the implementation security problem is still not well-defined or even easily discernable, and might never be. In spite of the lack of a encompassing problem to solve, I do think that we can work with the knowledge gained from failures, and the limited understanding failure provides about what things cannot work, to develop heuristic approaches to implementation security that have practical value.

In this paper I will look at some of what cannot work, specifically the security impact of two UNDECIDABLE problems that emerge within the area of communication message processing: The first of these problems occurs when attempting to recognize messages for protocols defined by strong classes of formal languages with an insufficiently strong parser. If a message cannot be fully recognized then it cannot be effectively validated, and malformed messages cannot be decisively rejected. The effect that an arbitrary message has on a given implementation becomes indeterminate without effective validation, this indeterminacy establishes a necessary condition for the existence of many classes of security vulnerability.

The second problem is that validating the computational equivalence of parsers for ambiguous context-free (and stronger) languages is UNDECIDABLE. If the equivalence of the parsers in protocol endpoints can not be determined then it is possible that a receiver will not interpret a message as it was intended by its transmitter. Ambiguous interpretation of arbitrary messages results in indeterminate behaviour in the receiver and this fundamentally undermines the assumed safety[1] (nee security) properties of the involved components, i.e. bad things can happen[2].

Together the input recognition and parser equivalence problems form a substancial basis for LANGSEC, an approach to the security of real-world composed systems based on computational complexity and formal language theory. The approach was initially conceived of by Len Sassaman and Meredith L. Patterson and has quickly attracted many strong contributors from both academia and industry[36] [5]. LANGSEC promises to be a powerful tool for assisting the security analysis of existing protocol implementations, as well as in the design of new implementations and new protocols. In this paper I will be applying LANGSEC concepts in the analysis of a protocol named OpenFlow.

OpenFlow is the protocol which underpins the exceedingly popular Software-Defined Networking (SDN) approach to network virtualization. OpenFlow based networking has received spectacular levels of attention in many ways: from a massive real-world implementation at Google[3] to billion-plus dollar corporate acquisitions[4], and beyond that, the appearance of OpenFlow compatibility as a major feature on equipment from nearly every traditional networking hardware vendor. As easy as it is to be cynical about all that hype, the fact remains that OpenFlow has been effective at enabling a wide variety of novel networking capabilities in the real-world operation of some truly massive-scale systems. Couple its apparent utility with a relatively open standard and OpenFlow becomes an arguabley attractive approache to programmable networking and virtualization; and a worthy target for critical analysis.

1.2 Report Structure

This paper begins with sections introducing Implementation Security, LANGSEC, and network virtualization, respectively, that set up context for a language-theoretic analysis of OpenFlow. In the analysis I will assess the formal language complexity of the OpenFlow protocol, proving that the protocol is at best context-sensitive. The analysis will also review aspects of the protocol that indicate it might belong to a stronger class of formal language.

From the complexity analysis results I will draw implications both for OpenFlow implementations and the use of the protocol in system composition. The implications discussion will specifically address how the complexity of OpenFlow poses a fundamental obstacle to the effectiveness of currently proposed SDN verification schemes, particularly those that rely on inspection of OpenFlow message traffic.

Attempting to break from the inevitable darkness of critical analysis and move forward on a more upbeat note, the concluding sections will introduce a visual heuristic device for reasoning about implementation security and offer a bit of explicitly prescriptive advice directed towards avoiding some of the design flaws observed in OpenFlow, along with other potential avenues for future work.

Notes

[1] In the field of distributed systems there is formal terminology for qualifying system properties: *safety* and *liveness*. For the purpose of maintaining at least some minor pretence of scholarly semantic consistency the following are my working definitions of the terms: a *safety* property is one that asserts something bad does not happen; a *liveness* property is one that asserts something good will eventually happen. Security properties are frequently stated in terms of safety and it is in this sense that the term safety is used. I've also brought in the term safety because with the second problem we are essentially discussing distributed systems.

[2] There is frequently a point in discussions about obscure, complex, or otherwise difficult to understand topics where someone argues that the likelihood of some bad things happening is really really small and therefore said bad things should not be a matter of concern. Quibble all we might about those small probabilities, they can't be accurately estimated and so likelihood is utterly irrelevant to making reasoned decisions about those rarely occurring bad things. It is the consequences of the bad things, and the relative costs of dealing with the consequences, that matter. The way I usually paraphrase the main thrust of this line of reasoning is with the statement, "the only way something **won't** happen is if it **can't** happen".

[3] Google's private "B4" global wide-area network is OpenFlow based. B4's existence was revealed by Urs Hölzle at Open Networking Summit 2012 and the network was recently described in detail at ACM SIGCOMM 2013 [19]

[4] VMware acquired the SDN startup Nicira in July 2012 for US$1.26 Billion. The acquisition can be recognized as a landmark of sorts for SDN, if anything the gargantuan amount of money conveys the level of **ahem** excitement surrounding the technology [42].

"You remember when I said how I was gonna explain about life, buddy? Well, the thing about life is, it gets weird. People are always talking ya about truth. Everybody always knows what the truth is, like it was toilet paper or somethin', and they got a supply in the closet. But what you learn, as you get older, is there ain't no truth. All there is is bullshit, pardon my vulgarity here. Layers of it. One layer of bullshit on top of another. And what you do in life when you get older is, you pick the layer of bullshit that you prefer and that's your bullshit, so to speak."

– Dustin Hoffman as Bernie LaPlante in "Hero"

2

10 Miles High

2.1 Implementation Security

2.1.1 Effing Difficulties

Implementation Security is difficult to discuss perhaps precisely because it has to do with all the assumptions, largely unacknowledged, that are made during the effort between the conceptual moment for a design and its execution in operational software. Thousand page tomes, of well respected quality, have even been written on the topic of Security Engineering but they barely mentioned the subtleties of *implementating* those systems securely let alone providing much specific guiding detail[1]. Having been overlooked in the foregoing context, one might be excused for thinking that Implementation Security is a bit ineffable or even that it might be bullshit.

Now, I'm not trying to leave the impression that there are no treatments of Implementation Security in the Literature, it is just that they are widely distributed and relatively

[1]It's funny in a somewhat disturbing sort of way… Peruse Ross Anderson's *Security Engineering*[1] and observe that the book doesn't ever explicitly discuss fundamental technical origins for vulnerabilities in real systems. Anderson has bits of insight sprinkled throughout his discussions but it's all rather implicit. Security Engineering is still a completely fantastic book though, particularly for addressing the general security *design* problem.

rare. For the sake of the current discussion though, two examples do stand out from the rest: first, Daniel J. Bernstein (DJB) has many useful observations to bring to the topic but he is especially explicit in a retrospective on the security of the "qmail" software package [2]. Second, is an example from the authors of *The Art of Software Security Assessment*, who refer to the concept of 'Accuracy' in their discussion of software design fundamentals. The last paragraph of the Accuracy section begins with the following gem (page 32 of [13]):

> "Discrepancies between a software design and its implementation result in weaknesses in the design abstraction. These weaknesses are fertile ground for a range of bugs to creep in, including security vulnerabilities."

At this point in the discussion the reader would be correct to note that I've described Implementation Security bearing some relation to assumptions, design abstractions, and the contents of a paper from DJB, yet our subject is still in rather vague territory. I might even be accused of spending the first paragraphs of this section prevaricating about the proverbial shrubberies for my own amusement. I'll confess this accusation is partially correct, but only partially. Hopefully I've brought the discussion to a place of intrigue, or some irritated variation of it, and at least partially primed the reader's mind with sufficient skepticism.

Leaving aside any of the professed difficulties in sorting out objective semantics, some semblance of a coherent working definition for Implementation Security would be helpful to make some progress on this topic. I propose the following definition for the current discourse:

> "Implementation Security is the technical Engineering (skillfulness) and Meta-Engineering (artfulness) that has an impact on the emergent security properties of a system."

One can observe that there is still as much concrete structure to even these concepts as there is to any subject in philosophy and I would agree that in some sense this is terribly unsatisfying. From a practical standpoint I think the lack of structure simply means that Implementation Problems writ large are intractable and developing detailed top-down theories to solve them is not only futile but potentially harmful. The endeavour of Implementation Security needs to be conducted from the same perspective as any other real-world engineering activity, namely working from the bottom up by examining failures, identifying what doesn't work, developing heuristics to apply, and then repeating those activities, tinkering as one goes along.

Working with the empirical approach I describe doesn't mean abandoning theory, far from it. The idea I have in mind is to apply the more formal and rigorous methods tactically, in circumstances where it would be instructive to know where "being careful" is not only risky but a practical impossibility, where even absolute care can not manage certain problems. Understanding of these difficult problems and how they manifest in implementations is the specific sort of engineering knowledge that I think can be percolated back towards, and perhaps connected with, the tasks of design and specification. Illustrating the tactical application of formal tools in the described manner is exactly the reason this paper introduces the LANGSEC approach and applies it to the security of OpenFlow.

2.2 LANGSEC

The language-theoretic approach to security (LANGSEC) is based on concepts from formal languages and computational complexity theory. It is rooted in the hypothesis that significant classes of security vulnerabilities emerge from design attributes that implicitly require a system implementer to attempt solving some variation of the Halting Problem (also known as Rice's Theorem) [37]. Computationally intractable (UNDECIDABLE) problems like the Halting Problem cannot be solved through functional approximation with any practical amount of software engineering effort and partial solutions are considered generally unsafe.

Partial solutions are reasoned to be unsafe due to the observation that those solutions will behave indeterminately under some conditions, behaviour which is likely to have security impacts when those conditions are processing of arbitrary malicious data. This point was discussed in the introduction but I restate it here for good measure.

Efforts to apply the LANGSEC approach to software security have largely concentrated on deconstructing how a composed system design can encounter the Halting Problem. The highest level breakdown looks at two contexts, the single component view and the system view. Respectively, this corresponds to input processing and protocol interpreter computational equivalence.

2.2.1 Input Processing

From a formal language perspective input processing can be characterized as the task of recognizing that a received message belongs to a language and then determining, according to the language semantics, whether to accept or reject the message. The implication of this view is that the validation question is UNDECIDABLE if the message cannot be fully recognized. If invalid, and potentially malicious, inputs cannot be deterministically rejected then any assumptions about the safety of using those inputs aren't reasonable and they create the potential for security vulnerabilities. This reasoning leads us to the first admonition from LANGSEC:

- Inputs must be fully recognized before being processed.

Recognizer functionality in real-world software is often ad hoc and distributed throughout the logic of (sometimes sizeable) components, recognition then is effectively an implicit task for the whole system. Testing and debugging recognizer logic implemented in this manner is an extremely hard problem, nearly impossible to do through systematic reasoning. For an accessible example of this phenomenon just look at the implementation of any mainstream operating system's network stack, examine how it generally validates data and then look at its vulnerability history [2].

The first admonition also provokes two significant follow-on questions. How can I ensure full recognition is possible? How is it effective? The corresponding two-fold answer is in what I'll state to be the two major facets of recognition: first, the input parser must be sufficiently powerful to fully recognize the protocol language. Second, the input parser should not be more powerful than is strictly necessary for full recognition, seeing as I don't want my parser to expose more computation resources to an attacker than is absolutely required.

While any software component might have security vulnerabilities due to implementation errors, a formal parser can at least be validated against the algorithm it implements which is in turn supported by formal proofs. An ad hoc parser on the other hand does not have a validation mechanism short of exhaustive testing, which will necessarily always be an intractable task. Attempting to programatically analyze the safety of an ad hoc parser is tantamount to trying to solve the Halting Problem.

The task of recognizing valid or expected inputs for a Turing-complete language is more or less a direct restatement of the Halting Problem. No amount of care, attention, or testing will allow for validation of Turing-complete inputs [3]. The context-sensitive (CS) languages don't present such a definitively impossible challenge for recognizing, it is always *possible* to construct a parser for them. Constructing an efficient parser for CS grammars is nevertheless known to be "very difficult" and the literature on parsers for CS grammars appears extremely sparse compared to that for weaker grammar types (by orders of magnitude in fact), according to §3.4.2 of [15] and gleaned from §18.1.1 of [15] respectively. In contrast, efficient methods for parsing Context-Free grammars (CFG), Regular (Chomsky Type 3) grammars, and Finite-Choice (FC) grammars have been thoroughly studied and the topics are extremely mature.

---

[2]Exempli gratia, the Linux kernel network stack. A quick search of the CVE database returns 23 entries that reference source code locations corresponding to IPv6 components of the network stack [10], using search terms that reference Linux network stack code more generically yield literally hundreds of entries. One must keep in mind that CVE entries only illuminate a subset of the publically known security vulnerabilities in any given component.

[3]The whole notion of formal decidability and the results it provides are quite remarkable. There is a small conceit in simply making the statement that the recognition problem for Turing-complete input languages is UNDECIDABLE though. What the statement misses pointing out is that the fundamental proof

2.2.2 Protocol Interpreter Computational Equivalence

The importance of interpreter equivalence in the LANGSEC approach is based on an information-theoretic modelling of communication from a security perspective. As Shannon describes in [38]:

> "An *information source* selects a desired *message* out of a set of possible messages. The transmitter changes the message into a signal, which is actually sent over a *communication channel* from the *transmitter* to the *receiver*."

The signal transmission process is the subject of what Shannon considered the engineering problem of communications. Now, system designers have some tendency to take an anthropocentric view of communication and protocol design, and especially those aspects related to transmitting semantics of varying sorts[4]. A key subtlety to highlight here is that the engineering problem is not solved completely at the level of raw symbol transmission. As I have discussed with respect to recognition of messages, there are still fundamental technical constraints on communication between computing processes rooted in the formal properties of the automata they implement. Starting from asking the question, "does the destination receive the intended message?", I can formally reason about the recognition properties of the destination processes up to the limit of decidability. This reasoning can be extended to the question of the equivalence of those language recognizers (nee processes).

Modifying Shannon's original diagram slightly, Weaver in [38] introduces the idea of a semantic decoder on the receiving side to deal with what he terms, "injection of semantic noise". Weaver suggests renaming the noise source on the channel to *Engineering Noise* and adding an additional semantic noise source between the information source and the transmitter. I would like to make my own tweak to Weaver's suggestions, adding semantic encoders and decoders to the communication system diagram which essentially correspond to protocol interpreters. The interpreters being automata running on Turing machines. The results of my tweaking are depicted in Figure 2.2.1.

This is where I introduce two more decidability problems to describe the limits for comparing automata: first, is the Equivalence problem. Where G and H are grammars and $\mathcal{L}(x)$ is the language produced by automata implementing those grammars, the Equivalence problem is whether $\mathcal{L}(G) = \mathcal{L}(H)$. The second problem is known as Con-

---

of undecidability is for Type 0 Chomsky grammars and that the Church-Turing thesis provides the connection being claimed for Turing-complete inputs. Practically speaking, this detail doesn't really matter but some pedantic souls might find the absence of its mention deeply unsatisfying. Those who are interested can find a great explanation of decidability and the Church-Turing thesis in Chapter 7 of [35].

[4]All completely understandable, designers being people themselves.

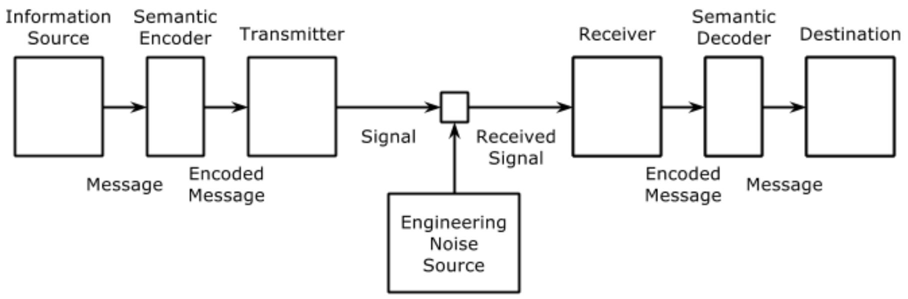


Figure 2.2.1: Diagram of a Semantic Communications System

tainment, or is $\mathcal{L}(G)$ a sublanguage of $\mathcal{L}(H)$, denoted formally by $\mathcal{L}(G) \subseteq \mathcal{L}(H)$. The Equivalence problem is DECIDABLE for deterministic context-free, and weaker, grammars [36]. Where the Equivalence problem is DECIDABLE though, the Containment problem is still UNDECIDABLE. In point of fact, the containment problem is UNDECIDABLE for arbitrary Regular languages as well, as demonstrated by Theorem 7.10 Corollary 2 in [35].

The Equivalence problem is intuitively key to deciding parser equivalence but Containment is particularly relevant to reasoning about equivalence in parsing of protocol sub-formats. While the two problems do create tight constraints, there are some particular automata constructions, with attendant formal language properties, where the Containment and Equivalence problems are DECIDABLE. One such DECIDABLE construction is detailed in Lemma 1 of [16].

Semantic equivalence of parsers has an impact on the more mundane aspects of system composition and interoperability as much as it does on security. In this respect, parser equivalence is simply a more rigorous way of looking at compatibility, which I think often gets taken for granted as being something of an unavoidable challenge in real-world systems. At the very least, if parser equivalence is DECIDABLE then interpreter compatibility should be easier to achieve.

To recapitulate: protocol interpreters must implement computationally equivalent parsers, otherwise the semantics of messages sent between endpoints cannot be preserved. Semantic ambiguity creates the opportunity for indeterminate effects in message processing, therein creating opportunity for insecurity. This draws forward the second, and at the moment last, admonition from the LANGSEC approach:

- Use only regular and deterministic context-free languages in protocol designs to ensure computational equivalence of endpoints.

2.3 Virtualizing Networks

2.3.1 The Traditional (Inter)Network

"Mais qu'est-ce alors cette vérité historique, la plupart du temps? Une fable convenue, ainsi qu'on l'a dit fort ingénieusement.

[But then what is this historical truth, most of the time? A fable agreed upon, as someone ingeniously said.]"

– Napoléon Bonaparte, in de Bradi's Les Misères de Napoléon

Establishing a definitive context for the current Collective Wisdom about network virtualization would be fascinating but it isn't going to get the current topic moving in the right direction. I will have to settle for a brief stereotyping of what I think forms the consensus view. Wikipedia's introductory definition is as good a place as any to start, it insists that a Computer Network is "a telecommunications network that allows computers to exchange data…The best-known computer network is the Internet." For the purposes I have in mind, I will simplify things one slight further and state that our consensus ideal for a network is one with nodes communicating using Internet Protocols (TCP/IP) running over something that is fungible for switched Ethernet.

Application of policy for the control of network traffic in a Traditional Network is orchestrated between routers through application level protocols (such as BGP, OSPF, or IS-IS). Logical network topology in the Traditional Network is generally tightly coupled with the network's physical topology. Routing policy is generally applied in a hierarchical fashion and has fairly low granularity, typically differentiating traffic at a higher level than individual flows [5]. Each router maintains a representation of its view of the network topology in routing look-up tables. The routing tables are in turn used to control forwarding elements that process actual packets. A visual depiction of this conceptualization is illustrated in Figure 2.3.1.

For the sake of drawing distinctions between Traditional Network hardware and that implementing contemporary virtualization schemes, SDN or otherwise, I'll define Traditional Network hardware as that which has routing control and forwarding elements that are integrated into the same physical chassis and which participate in routing protocols as distinct entities.

[5] I define a Network Flow (flow) to be the logically identifiable collection of packets, belonging to communication between distributed processes, transiting through a particular point in a network.

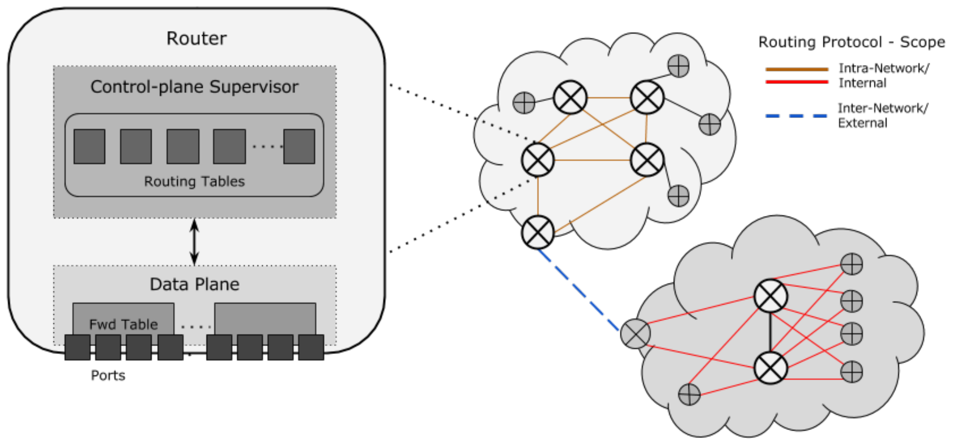


Figure 2.3.1: Depiction of a Traditional Network

2.3.2 Network Virtualization

Much to the chagrin of many telecommunication engineers and service providers, a network on it own does nothing. Intercomputer networking is simply a mechanism for moving data between distributed computing processes, fundamentally just inter-process resource sharing. A simple enough problem to state succinctly but therein, as the Bard will tell us, lies the rub.

Traditional Networking comes with attached conceptual baggage, in pretty much every aspect between design and construction, through to operations. In practical terms the baggage manifests as established implementations, standards, and commonly accepted practises; collections of the abstractions and assumptions created to solve the problems of yesterday. Eventually understanding of those problems changes in some way, new insights might be embraced and old axioms might be forgotten, but suffice it to say that things don't work as well (nee optimally) as they used to. One way of dealing with such exigent circumstances is by creating a new abstraction to encapsulate the old problems. In contemporary computing, a common method of abstracting the past is what is termed Virtualization.

Conceptually, Virtualization entails schemes where the logical organization of resources is decoupled from their physical implementation. Bruce Davie, a luminary in the networking field, describes network virtualization as having the following characteristic functions[11]:

- Decoupling the services provided by a (virtualized) network from the physical network.
- Providing a container of network services (L2-L7) provisioned by software.
- Faithfully reproductioning services provided by a physical network.

I think it should not go unremarked upon that Davie was employed by VMWare when this conceptualization was presented and that the characteristics bear a striking resemblance to contemporary computing hardware virtualization. Nevertheless, his points do form an apt summary of the current Common Wisdom. One reason I reference Davies' presentation is that he takes some effort to stress that while Software-Defined Networks built using OpenFlow are one way of achieving the aforementioned functionality, there are many other ways. A point on which some service providers are in strong agreement, according to Boucadair and Jacquenet in [3]:

> "SDN techniques are not necessary to develop new network services per se. The basic service remains IP connectivity that solicits resources located in the network."

The primary distinction drawn for SDN then is that it exposes direct programmatic access to the forwarding plane of network elements via a remotely accessible API, with OpenFlow currently being the most popular one. In this respect, the general approach of SDN and OpenFlow does make certain assumptions about the existence of some sort of network connectivity between forwarding plane elements and their controller application servers. Connectivity may be an arguable small conceit for many modern networks but it is an important one, at least conceptually, for *Inter*-networking in particular and distributed systems more generally [6].

The corollary to exposing the forwarding plane to remote access via an API is that the control plane can now run on server hardware, with fast processors and expansive storage, and is no longer restricted to the meagre resources of a control-plane supervisor running on embedded-class computing resources. Despite the very astute observation that there are other ways of implementing network services besides SDN, Boucadair and his co-author miss this point entirely when they write [3]:

[6]Now, this may be a fanciful perspective on history but I certainly maintain the understanding that networks were not as homogeneous as they presently appear, the current era being dominated almost exclusively by IP running over something that looks like Ethernet (and is generally pretty reliable).

"By definition, SDN technique activation and operation remain limited to what is supported by embedded software and hardware. One cannot expect SDN techniques to support unlimited customizable features."

SDN techniques are certainly limited to what can be accomplished with programmatic control over (micro-)flow management, but beyond that customizations are only limited by computational complexity and similar constraints. That is to say, features are actually close to being unlimited and that is what makes the technology fundamentally compelling. Interestingly, this appears to be a direct fulfilment of the prediction Michael Padlipsky offered at INFOCOMM in 1983: "with computing power now effectively unrestricted people can do networking any way they please"[31][7].

2.4 OpenFlow

2.4.1 Essential Descriptives

OpenFlow is a switch architecture specification and application-level network protocol designed with the intent of enabling direct programmatic access to the forwarding elements of software and hardware networking components. The architecture is a composition based on a distributed software control plane, the Controller, that interfaces via a wire protocol to an abstraction of the forwarding elements of a generic hardware switch. The switch being modelled after one that employs a TCAM-based Forwarding Information Base (FIB) architecture [8]. In the context of OpenFlow the FIB is represented by a series of flow tables; these tables associate network flow match conditions with packet processing actions and table usage statistic counters. The structure of OpenFlow's architecture is illustrated in Figure2.4.1.

[7]It is worth mentioning that Padlipsky's prediction came with a confession that he feared schemes that did assume end-to-endness at the Network level would foul things up. Some might argue that the connectivity and reliability OpenFlow assumes commits those sins exactly; I am not inclined to disagree.

[8]Ternary Content-Addressable Memory (TCAM) can be summarily described as being a hardware implementation of an associative array that enables lookups based on partial, wildcard like, matching of array keys in addition to binary, i.e. yes/no, matching.

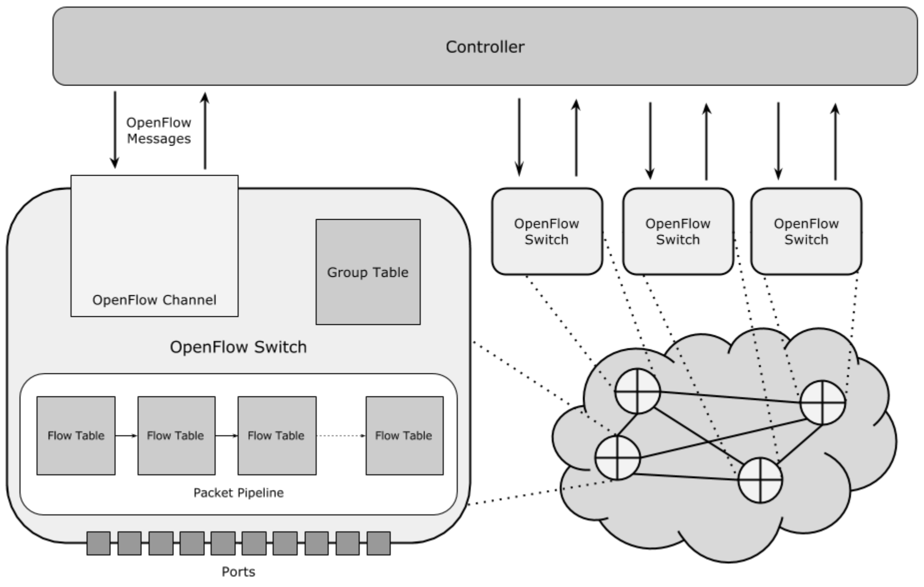


Figure 2.4.1: A Depiction of the OpenFlow Specification Switch Architecture

Referring to the diagram for context I will briefly explain the major functional components, a general understanding of which will make the discussion of complexity analysis targeting clearer. So, a packet will arrive via a Port and then enter the Packet Pipeline starting with the first Flow Table. A Flow Table is a look-up table with entries that consist of the following elements:

- Packet characteristics to match on;
- Actions to apply on matching packets; and
- Entry use statistics, e.g. the number of packets matched and processed by that entry.

The Actions in a flow table entry that can be applied on match conditions correspond to one of six types, the firtst four being fairly general:

1. Forward a flow's packets.
2. Encapsulate a flow's packets and send them to a Controller for processing.
3. Drop a flow's packets.
4. Perform a specified action on a flow's packets.

The fourth Action is rather vague in terms of self-description but in essence it is simply a generic way of referring to the capabilities an OpenFlow switch implements to manipulate or inspect packets in somewhat arbitrary ways. In practice this would be functionality such as address translation at various layers, traffic filtering and firewalls, VLANs, et cetera. All forwarding Actions can target subsequent Flow Tables in the Packet Pipeline as well as Ports.

The specification allows for so-called hybrid switches, which are typically traditional network switches that implement an OpenFlow processing path along side their normal layer 2/3 processing path. These switches would implement an additional Action for processing:

5. Forward a flow's packets through the switches' normal, non-OpenFlow, processing path.

The OpenFlow specification defines one last primary processing Action type and while it is not related to flow table entry matching per se, it is defined in the specification:

6. Decapsulate packets received from the Controller and forward them.

The Group Table is the OpenFlow mechanism for applying actions to collections or groups of flows.

To round out this description it would make some sense to discuss the Controller briefly. Essentially the Controller is just an application server that manages the configuration of the switch tables and otherwise controls and monitors the operations of the switches. Yes, in practise Controllers can actually be very sophisticated software systems, however, the fact remains that at a fundamental level a Controllers is basically indistinguishable from any other socket-based application server that might be found in a client-server architecture system.

From an operational perspective, OpenFlow's default behaviour is for switches to initiate a TCP connection to the Controller specified in their configuration. An initial exchange of OpenFlow messages is used to conduct an application level handshake procedure that in turn establishes a session, which in OpenFlow parlance is referred to as a Channel. Once the primary Channel is set up, operational messages are exchanged between the switch and its Controller.

2.4.2 A few words about a name

An industry group called the Open Networking Foundation (ONF) [9] currently manages the OpenFlow brand, which they've started using to market a collection of protocol specifications beyond the original switch control protocol. The ONF tends to refer to the original OpenFlow protocol interchangeably, and somewhat inconsistently, as either

"OpenFlow" or the "OpenFlow Switch Specification Wire Protocol" (the latter abbreviated as OF-SPEC). For the sake of familiarity, and a passing effort at reducing confusion, I will observe the historical practise of using "OpenFlow" to refer to the switch specification and the messaging protocol it defines.

2.4.3 Imaginarium

Imagine for a moment[10] the software implementation of a network switch that is written (as one would expect for such things) in the C programming language. The internals of this imaginary switch would very likely consist of numerous collections of `struct` defined types, differentiated by function and specified as analogues to the structure of a hypothetical hardware switch. Since this is networking software, and network byte ordering is big endian, the implementation being imagined stores much of its internal state in big endian representations. Native byte ordering of the hardware running the software switch be damned.

When the switch is running, the contents of all the foregoing collections of structures will literally represent the internal state of the switch. If one desired to change the state of the switch in a dynamic fashion then all one has to do is have a mechanism for changing the contents of the internal structures; to accomplish this from a separate controller application, perhaps running on a different host, one would need a protocol of some sort to facilitate the communication of the structure changes. Easy enough.

The quick and dirty approach to a switch control protocol could be to send the contents of the running switch's internal state directly, maybe with some message headers for metadata. The implementation already stores most state in network byte order so crafting messages for this protocol would be a very straight forward operation, the switch could essentially just write the contents of structures into a socket connected to its controller. Writing the specification document for our simple protocol will be a doddle since the `struct` defined in the source code can be copied and pasted directly into the document. A full header file with all the `struct` can be included as an appendix too of course, one wouldn't want to provoke copy and paste errors in other implementations after all.

Careful, I almost forgot that Security is Terribly Important…The switch control protocol should probably run on top of a TLS wrapped connection and maybe X.509 certificates could be used for authentication (optionally).

[9]Like many industry organizations comprised mostly of vendors, and who's name makes mention of "Open", one doesn't get to play with the other SDN kids for free: the ONF's current membership fee is US$30,000.

[10]Indulge me briefly, please.

Believe it or not, this story does have a punch-line: I've just provided a guided visualization of the OpenFlow protocol. Absent a detailed history of the design process, which doesn't appear to exist, the best explanation for the design of the protocol is that its specification was derived directly from headers in the source code of a prototype programmable software switch. A prototype software switch which in turn eventually became the initial reference implementation for OpenFlow[11].

[11]The location of the source code repository for the reference switch has been somewhat obscure for quite some time, at least since before ONF took guardianship of OpenFlow in early 2012. The last commit to the repository was some time between specification versions 1.1.0 and 1.2 so the repository location likely hasn't been widely published due to the fact that the code was quite out of date. The out of date reference code means that no consolidated header file was openly available for OpenFlow versions 1.2 (December 2011) through 1.3.2 (April 25, 2013). Version 1.4.0 (August 14, 2013) of the specification finally includes a copy of the whole header file (albeit as an appendix inside a PDF document, which only some readers can easily extract). The source of the original reference implementation is managed with git and the repository can be cloned from `git://gitosis.stanford.edu/openflow` (or it could at the time of writing).

Raoul Duke: There's a uh, big machine in the sky, some kind of, I dunno, electric snake, coming straight at us.
Dr. Gonzo: Shoot it.
Raoul Duke: Not yet, I want to study its habits.

– Johnny Depp and Benicio Del Toro in "Fear and Loathing in Las Vegas"

3 Complexity Analysis

A brief inspection of the OpenFlow specification can quickly create the impression that the protocol is feature-rich and highly extensible, if not dazzlingly complicated. Complexity certainly can be some cause for concern but discussing it in subjective terms inevitably turns into arguing about aesthetics. What I think would be immediately valuable then, are formal or quantitative assessments of design attributes related to complexity. Toward this more rigorous end, I think one thing that can be done is establish bounds on the formal language complexity of OpenFlow. Analyzed using the concepts in language-theoretic security, the protocol's complexity bounds will directly indicate the potential for implementation problems. In the following analysis I will illustrate the complexity of OpenFlow using proofs based on the properties of formal languages, detailed implications of the analysis are presented in the chapter that follows.

3.1 Pick a Packet

The OpenFlow protocol presents an expansive array of options for complexity analysis and attempting a top-down analysis, let alone an exhaustive one, would not be anywhere close to practical. Instead of an analysis of the whole protocol then, I chose to focus on a single message type. Now, important characteristics of the OpenFlow protocol emerge from its C structure oriented design and chief among these is the fact that messages, and their component subtypes, generally each contain a length field given in bytes. The length fields have the effect of making each message component its own ad hoc Type-Length-Value (TLV) format, often with very idiosyncratic semantics. With all the myriad TLV formats on offer, choosing one to analyze was based on two general criteria:

- The format should be employed in a generally significant role in the messages sent by both Controllers and Switches;
- The format should be self-contained and unambiguous enough to represent using an attribute grammar without too much simplification.

Looking through the specification for a message type that met these criteria seems like it should be a reasonably straight forward task; it was not.

3.1.1 Any packet?

While examples abound, the curious reader can find a prime example of OpenFlow's pervasive complexity in the Hello message, defined in §7.5.1 of [29]. I would have thought Hello would be quite simple and unambiguous but it turns out to be extensible *just in case* something besides protocol version support needs to be communicated (at some future time). Adding a bit of insult to that injury, the version information itself is also extensible. Specifically, version support is encoded as bitmaps in raw 32 bit unsigned integers, where ordinal bits indicate the versions of wire protocol supported: versions 0 to 31 are encoded in the first integer, 32 - 63 in the second, and so on. The number of bitmaps actually present in a given Hello message must be inferred from the length field of its enclosing `ofp_hello_elem_versionbitmap` structure. The current version of the wire protocol, by the way, is `0x05`.

3.2 OXM_OF_MAKEMEDINNER

After reviewing the OpenFlow specification, I noted that it does explicitly define a single generic TLV format named OpenFlow eXtensible Match (OXM) that met the selection criteria I defined. As its name suggests, the primary application for OXM is representing the flow match criteria used to configure the switch flow tables. OXM has also been repurposed elsewhere in the protocol in a number of different ways, among them:

- to specify modification values in Set-Field actions;
- to communicate which fields a flow table supports matching or modifying within table feature messages (using a list of OXM headers only);

- to communicate flow parameters in flow expiration/removal messages, flow statistics messages, flow monitor messages, flow update messages, and Packet-In messages.

In this analysis I will concentrate on OXM use within matching contexts. A fully specified OpenFlow match is described by an `ofp_match` structure, which has the following declaration[1]:

```
struct ofp_match {
    uint16_t type;             /* One of OFPMT_* */
    uint16_t length;           /* Length of ofp_match (excluding padding) */
   /* Followed by:
    *
    * - Exactly (length - 4) (possibly 0) bytes containing OXM TLVs, then
    * - Exactly ((length + 7)/8*8 - length) (between 0 and 7) bytes of
    *   all-zero bytes
    *
    * In summary, ofp_match is padded as needed, to make its overall size
    * a multiple of 8, to preserve alignement in structures using it.
    */
    uint8_t oxm_fields[0];     /* 0 or more OXM match fields */
    uint8_t pad[4];            /* Zero bytes - see above for sizing */
};
OFP_ASSERT(sizeof(struct ofp_match) == 8);
```

The `oxm_fields` array marks the start of a possible list of OXM fields. Each OXM field begins with a header which the data immediately follows, the header format is illustrated as follows:

```
3                              1 1
1                              6 5           9  8  7             0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+---+-+-+-+-+-+-+-+-+
|           oxm_class          |  oxm_field  | h |   oxm_length  |
|                              |             | m |               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+---+-+-+-+-+-+-+-+-+
```

[1]It is pretty difficult not notice the comments that describe padding within the match structure and the simple fact that the arithmetic doesn't calculate padding correctly. Interestingly, this same incorrect padding description can be found pasted verbatim inside many other structures in the specification. Finding this mistake in one revision of the protocol gives rise to a bit of schadenfreude but seeing it go uncorrected in three versions of the specification (v1.3.1 - v1.4.0) released over the course of a year is bad enough that *I'm* starting to feel embarrassed about it.

An example OXM field for `oxm_field == OXM_OF_ARP_SPA`, which describes matching on the IPv4 address of an ARP payload, would be:

```
3                             1 1
1                             6 5           9 8 7             0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 1 0 1 1 0|1|0 0 0 0 1 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             IPv4 address (network byte order)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         arbitrary address mask  (network byte order)          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

An attribute grammar for recognizing an `ofp_match` structure that contains basic OXM fields is defined in Appendix A.

3.3 Pudding

The following is an examination of the formal language properties of OpenFlow. The proofs draw inspiration from [36] and [16] but any mistakes are my own. In regard to any uncertainty about specific claims, I assert the second definition does not violate the Containment problem is based on the fact that the grammars are not arbitrary. Now, without further ado:

Theorem 1. *OpenFlow is at least context-sensitive.*
Definition 2. *If G and H are grammars for* `ofp_match` *and OpenFlow, respectively, then* $\mathcal{L}(G)$ *and* $\mathcal{L}(H)$ *are the languages they generate.*
Definition 3. *The* `ofp_match` *structure is a necessary element of the OpenFlow specification, per §7.2.2.2 of [29], making it a proper sub-grammar of OpenFlow* ($G \subseteq H$)*. Therefore* `ofp_match` *is a sublanguage of OpenFlow:* $\mathcal{L}(G) \subseteq \mathcal{L}(H)$.
Lemma 4. *The upper bound for the complexity of a language has a corresponding minimally-strong mechanism for recognition.*

Generating strings belonging to language $\mathcal{L}$, $S \in \mathcal{L}$, *requires a mechanism* M *that is minimally-strong enough to apply the production rules of* $\mathcal{L}$. *The minimally-strong mechanism* M *is defined by the upper bound for the complexity of* $\mathcal{L}$.

Proof. Recognition is simply an inversion of the production rules so if a mechanism exists to recognize S that is weaker than M then M is not strong enough to apply the production rules of $\mathcal{L}$ and this contradicts the statement that M is minimally-strong. □

Lemma 5. *If language $\mathcal{L}(G)$ is a sublanguage of $\mathcal{L}(H)$ and a minimally-strong mechanism M is required to recognize strings in $\mathcal{L}(G)$ then a string $S \in \mathcal{L}(H)$ must be recognized using a mechanism at least as strong as M.*

Proof. If $\mathcal{L}(G) \subseteq \mathcal{L}(H)$ then there exists an arbitrary string S' such that $S' \in \mathcal{L}(H)$ implies $S' \in \mathcal{L}(G)$. If S' can be validated using a mechanism that is weaker than M then this contradicts the statement that M is the minimally-strong mechanism required to validate $\mathcal{L}(G)$. □

Lemma 6. *`ofp_match` complexity has an upper bound of Context-Sensitive*

The definition of a linear-bounded automaton for `ofp_match` establishes the upper bound for its complexity as Context-Sensitive.

Proof. We define an input word $P = \{w^n \mid w$ is a hexadecimal octet, $0 < n \leq 16536$ is the size of an `ofp_match` structure, $w^n =$ `len '00' '01' length oxm_fields pad`, where len and length are each two octets representing total input length as an unsigned integer, `oxm_fields = '08' '00' field hashmask length data`, where the concatenation of `field` and `hashmask` is a fixed binary string $\in \{0, 1\}$ encoding match data types as specified in [29] and `data` is a variable length binary string $\in \{0, 1\}$ appropriate for a basic class field and hashmask. pad is up to 4 octets of zero padding$\}$.

Let A_P be a Linear-Bounded Automaton that only accepts strings in P. The length of A_P's input tape is n, it has states $q_0 \ldots q_{39}$, q_r is the REJECT state, and q_{39} is the start state.

1. Go to the leftmost cell of the tape.
2. Consume the octet '00' and transition to state q_{38}. If any other octet is present, transition to q_r and halt.
3. Consume the octet '01' and transition to state q_{37}. If any other octet is present, transition to q_r and halt.
4. Simulate a decrementing counter initialized with the value of octets 3 and 4. Subtract 4 from the counter and transition to state q_{36}. If the counter reaches zero, transition $\rightarrow q_4$
5. Consume the octet '80', decrement the counter, and transition to state q_{35}. If any other octet is present or the counter reaches zero, transition to q_r and halt.
6. Consume the octet '00', decrement the counter, and transition to state q_{34}. If any other octet is present or the counter reaches zero, transition to q_r and halt.
7. Simulate regular expression matching of the fixed bit strings encoding the concatenated `oxm_field` + `oxm_hasmask` values over octets 7 and 8 as described

in the attribute grammar defined in Appendix A. Transitions are made according to the following rules:

(a) OXM_OF_MPLS_BOS, OXM_OF_PBB_UCA, OXM_OF_IP_ECN, OXM_OF_MPLS_TC, OXM_OF_VLAN_PCP, OXM_OF_IP_DSCP, OXM_OF_ICMPV4_CODE, OXM_OF_ICMPV4_TYPE, OXM_OF_ICMPV6_CODE, OXM_OF_ICMPV6_TYPE, OXM_OF_IP_PROTO $\rightarrow q_2$

(b) OXM_OF_ARP_OP, OXM_OF_ETH_TYPE, OXM_OF_SCTP_DST, OXM_OF_SCTP_SRC, OXM_OF_TCP_DST, OXM_OF_TCP_SRC, OXM_OF_UDP_DST, OXM_OF_UDP_SRC, OXM_OF_IPV6_EXTHDR, OXM_OF_VLAN_VID $\rightarrow q_3$

(c) OXM_OF_MPLS_LABEL, OXM_OF_IPV6_FLABEL, OXM_OF_PBB_ISID, OXM_OF_IPV6_EXTHDR_mask $\rightarrow q_4$

(d) OXM_OF_IN_PHY_PORT, OXM_OF_IN_PORT, OXM_OF_ARP_SPA, OXM_OF_ARP_TPA, OXM_OF_IPV4_DST, OXM_OF_IPV4_SRC, OXM_OF_VLAN_VID_mask $\rightarrow q_5$

(e) OXM_OF_IPV6_FLABEL_mask $\rightarrow q_6$

(f) OXM_OF_IPV6_ND_SLL, OXM_OF_IPV6_ND_TLL, OXM_OF_ARP_SHA, OXM_OF_ARP_THA, OXM_OF_ETH_DST, OXM_OF_ETH_SRC, OXM_OF_PBB_ISID_mask $\rightarrow q_7$

(g) OXM_OF_METADATA, OXM_OF_TUNNEL_ID, OXM_OF_ARP_SPA_mask, OXM_OF_ARP_TPA_mask, OXM_OF_IPV4_DST_mask, OXM_OF_IPV4_SRC_mask $\rightarrow q_9$

(h) OXM_OF_ARP_SHA_mask, OXM_OF_ARP_THA_mask, OXM_OF_ETH_DST_mask, OXM_OF_ETH_SRC_mask $\rightarrow q_{13}$

(i) OXM_OF_IPV6_ND_TARGET, OXM_OF_IPV6_DST, OXM_OF_IPV6_SRC, OXM_OF_METADATA_mask, OXM_OF_TUNNEL_ID_mask $\rightarrow q_{17}$

(j) OXM_OF_IPV6_DST_mask, OXM_OF_IPV6_SRC_mask $\rightarrow q_{33}$

(k) No match $\rightarrow q_r$

8. Consume two octets, decrement the counter by 4, and transition: $q_n \rightarrow q_{n-1}$. If the counter reaches zero, transition to q_r and halt

9. Until q_0 is reached, the counter reaches zero, or the rightmost end of the tape is encountered, carry out the following operations:

   (a) Consume an octet

   (b) Decrement the counter

   (c) Transition $q_n \rightarrow q_{n-1}$

10. If:

    - in state q_0, the counter is zero, and the tape head is at the rightmost end of the tape, ACCEPT.

    - in state q_0 and the counter is not zero, transition $\rightarrow q_{36}$

    Otherwise, REJECT.

Because OXM can be described by a linear-bounded automaton, it is therefore at most context-sensitive. □

Lemma 7. *OXM is not context-free.*

To show OXM is not context-free we use the uvwxy theorem, also known as the pumping lemma for context-free languages, as described in §2.7.1 of [15] and Theorem 3.6 [35] respectively.

The theorem is as follows: for every context-free language $\mathcal{L}$ we can give two natural number p and q, such that each string $S \in \mathcal{L}$ that is longer than p has the form UVWXY where $|VWX| \leq q$, $VX \neq \lambda$, and $UV^iWX^iY \in \mathcal{L}$ for all $i \geq 0$. Where λ denotes the empty word and p is what is referred to as the pumping length.

Proof. If OXM can be generated by a context-free grammar then we would have strings $UVWXY$ such that UV^iWX^iY for $i \geq 0$ is of the form P defined in Lemma 4. This, however, is impossible since the only way to arbitrarily vary i without exceeding n is if $VX = \lambda$. Therefore OXM is not context-free. □

Proof(Theorem 1). As P is at most context-sensitive (Lemma 6) and must be stronger than context-free (Lemma 7), P (thus OXM) must therefore be context-sensitive. Since QXM is a sublanguage of OpenFlow (Definition 2) and requires at least the same minimally-strong mechanism for recognition (Lemma 5) then *OpenFlow must be at least Context-Sensitive.* □

3.3.1 A Little Bit Stronger?

In previous efforts toward adumbrating this dissertation I mentioned there was potential for OpenFlow to be Turing-complete. Proof of this conjecture is doubtlessly too sizeable to fit within any reasonable length, or time, allowance for this particular project. That said, I think there is potential to prove Turing-completeness in a reasonable straightforward manner by implementing Turing Machine equivalent automata (such as Rule 110) through the mechanisms available to Instructions and Action Sets. The push/pop actions for MPLS labels or TTL increment/decrement operations could be used to implement stacks for instance (§5.10 [29]).

Dr. Peter Venkman: Ray, pretend for a moment that I don't know anything about metallurgy, engineering, or physics, and just tell me what the hell is going on.
Dr. Ray Stantz: You never studied.

– Bill Murray and Dan Aykroyd in "Ghostbusters"

4 Implications

THE COMPLEXITY ANALYSIS IN THE PRECEDING CHAPTER establishes the most critical attribute of OpenFlow for this investigation: its formal complexity is at least context-sensitive. From a LANGSEC perspective there are two chief consequences to be observed from the complexity strength:

Observation 1 Ad hoc parsers are not capable of fully recognizing OpenFlow messages and consequently are not able to safely, and definitively, validate messages.

Observation 2 It is not possible to determine whether OpenFlow implementations have equivalent parsers. This is to say that one cannot determine if a given collection of parsers will derive the same semantics when processing the same arbitrary OpenFlow messages.

The two observations both directly impact the security of OpenFlow implementations, which I will discuss, but the latter observation also has significant second order impacts for systems composed around OpenFlow.

Without definitive validation of OpenFlow messages there is no assurance that any computation performed on the contents of those messages is safe. In particular, an implementation cannot make any safe assumptions about the structure of data that hasn't been processed based on data that has been processed. In these cases the safety of an operation is dictated by data-dependent interactions on the execution paths leading to that operation within a program's call graph.

Determining exact static call graphs is known to be an UNDECIDABLE problem, approximate solutions are nevertheless necessary when auditing the security of a piece of software. The effectiveness of these approximate solutions (nee auditing) depend significantly on the correctness of the reasoning that goes into them, regardless of whether those solutions are programmatic tools or manual call tracing. Reasoning, however, is fragile.

The uncertainty of security auditing, in particular, highlights why I think LANGSEC is a valuable tool for understanding the nature of implementation insecurity. LANGSEC allows for a substantive explanation for why a fragile looking piece of code cannot rely on the safety of the data it is processing. The question of the likelihood that data *might* be unsafe is moot when formal proof is available to show that it cannot be safe.

4.1 Current Interpreter Implementations

The implications of Observation 1 are profound for the security of OpenFlow interpreter implementations: they are *all* likely to be flawed in some way. Whether a particular implementation is going to be prone to having exploitable vulnerabilities is then largely dependent on the implementation mechanism. An implementation is highly likely to have security bugs if it hasn't been built using tools that structurally prevent particular classes of vulnerabilities, tools which include programming languages that are strongly typed, strictly evaluated, and have automatic memory management. Unfortunately, most switch implementations are written in C and Controllers are a rather mixed bag of C, C++, Java, and assorted dynamic languages like Python. A few niche interpreter implementations built in safer languages do exist though, these include OpenMirage (OCaml) [30] and Galois Inc.'s Nettle (Haskell) [28].

4.2 Fingerprinting

One of the primary consequences of inequivalent parsers is knowing that the parsers will respond in discernibly different ways to carefully crafted inputs. Through such behaviour differences it should be possible to fingerprint OpenFlow parser implementations. I make a subtle distinction here between the parsers and the whole OpenFlow interpreter, even though there isn't likely to be a large difference in most ad hoc implementations. The reason for the distinction is that I think that components whose function is effectively front-end parsing are likely to be reused across OpenFlow implementations, even if the reuse is at the level of copy and paste in source code.

The significant potential for vulnerabilities in parsing code that follows from Observation 1 makes fingerprinting extremely valuable for not only attack targeting in exploitation operations but for targeting bug hunting as well; widely reused components are much more valuable targets for practical security research.

4.3 Grander Aspects

Aside from direct access to an OpenFlow channel, there are two main routes for an attacker to influence data within OpenFlow messages or in the runtime state of protocol interpreters: first, through the ability to manipulate the layout and content of network packets processed by an OpenFlow switch. Second, through the ability to manipulate or otherwise affect the configuration of OpenFlow controllers.

For the first route the refrain remains, "avoid parsing" and this applies to packets as much as OpenFlow itself in SDN designs. In particular, this means being extreme mindful of the risks inherent in architectures that make significant use of exchanging packets between switches and controllers via `Packet-In` or `Packet-Out` messages.

The second route particularly affects multi-tenant distributed service architectures (i.e. cloud computing) which implement dynamic networking services using OpenFlow. The software implementation of OpenFlow most likely to be found in such environments is the Open vSwitch software switch, forming the core network for XenServer and most OpenStack systems amongst other things. The nature and degree of influence over the SDN components in a system using Open vSwitch depends on specific operational configurations, naturally, but the fact remains that there is significant opportunity for an attacker to directly influence the system.

4.4 Expansive Vulnerability

The immediate implications of OpenFlow insecurity are quite significant but when the nature of OpenFlow's diffusion into computing infrastructure is considered, the higher order and longitudinal effects are truly staggering. OpenFlow interpreters are being fielded in every context where networking is deployed: from the software switches of virtual machine hosting infrastructure, through data-centre and service provider networks, and on down to customer premises equipment and similar low-cost embedded applications (e.g. wireless access points and ADSL modems). Applications that involve embedded-class hardware, where the protocol interpreters are implemented in the devices' firmware, will be especially problematic to manage over long time scales. Operational experience has proven that updating firmware is generally some combination of difficult, risky, and costly, which is unlikely to change soon.

In terms of time scales, networking hardware tends to have a much longer operational service lifetime than general purpose computing hardware, especially service provider transport infrastructure. The impact of vulnerabilities in core infrastructure can be immense and the full consequences of such impacts over the course of time is complicated; none of this can be ignored. The legacy of OpenFlow might well be the vastness of vulnerable software it will leave scattered across the computing and telecommunication landscapes.

4.5 Network Verification

Much like every dynamic distributed system, Software-Defined Network implementations have to manage intrinsic transaction and consistency issues. In the lingua franca of contemporary distributed systems these issues are usually discussed in terms of ACID [1], BASE [2], and CAP theorem[3] properties. It is arguable that the design of OpenFlow did not initially take these issues into serious consideration, dealing with them has had to come after the fact as the SDN field has gained experience with designing and fielding systems.

One verification related idea described by Reiblatt et al [34] advocates for determining what kinds of configuration operations are safe to apply a priori. By establishing such a set of safe operations, the authors hope to gain assurance over the behaviour of configuration updates. In an apparent effort to address update related concerns, the latest version of the OpenFlow specification (1.4.0) now defines a mechanism, termed *Bundles*, for improving transaction properties of controller-to-switch command and configuration messages. Aside from improving transaction mechanisms though, a significant element of the response to distributed systems issues is the appearance of various different SDN *verification* schemes.

The schemes examine a variety of different aspects of OpenFlow networks, from formal network specification through to operational verification of so-called invariant properties. Network design checking schemes aren't particularly relevant to the current discussion but operational schemes are, and these fall into two basic types: controller-hosted and message inspecting.

[1]Atomicity, Consistency, Isolation, and Durability

[2]Basically Available, Soft state, Eventual consistency

[3]Consistency, Availability, Partition Tolerance

4.5.1 Controller-hosted Applications

Controller-hosted verification schemes are typically implemented as plug-ins to existing controller applications; examples of such schemes include Kinetic[34] and FLOVER[40]. Effectively subcomponents of a Controller, these schemes rely on the interpretation of OpenFlow messages by their host application. The most significant implication of language complexity for these applications then comes from the inability to effectively validate OpenFlow message data: the verification applications might receive malicious data to process. Realistically, this isn't a problem for these types of verification schemes per se, they are simply affected by insecurity in Controller's parser.

Due to the fact that these schemes are designed to be situated inside the system they are monitoring, they do suffer from observer vantage-point problems. I will discuss this problem in more depth in a following section concerning alternate verification approaches.

4.5.2 OpenFlow Message Inspection

Verification schemes that rely on inspecting or intercepting OpenFlow message traffic, such as Veriflow [24], depend on the assumption that the interpretation of messages by verification elements and endpoints is equivalent. Through the language-theoretic analysis of OpenFlow I've proved that OpenFlow parser equivalence is fundamentally UNDECIDABLE and the assumptions supporting these schemes cannot hold. In practical terms the impact of this undecidability is that systems using these verification mechanisms will be a priori vulnerable to classes of attack that include insertion of malicious OpenFlow messages as well outright evasion of the verification mechanisms.

Even if I were to pretend for a brief moment that computational intractability wasn't a limitation, I can still reason that there are very difficult problems with message inspection schemes. Namely, a verification scheme that examines OpenFlow messages directly would need to manage the interpretations idiosyncrasies of every end-point it is monitoring and do this over time across operational environment changes. In practise, this approach to verification would result in systems that are indeterminately effective, which I think amounts to little more than just pretending the verification system works.

4.5.3 Partition by Proxy

One of the oft cited challenges in empirical networking research is the difficulty of accessing realistic experimental environments; short of direct access to production networks, all other approaches make significant compromises in some dimension. Hardware test-bed networks are costly in nearly every aspect over the course of their lifetime. Software simulation, while cost effective and flexible, is always going to be an incomplete approximation to a real network. To address these problems Sherwood [39] proposes using the capabilities of OpenFlow-enabled networking hardware to allow multiple parties to simultaneously access a real production network by partitioning it into isolated "slices" whose resources can be managed dynamically.

There is no major conceptual problem with the premise of using a dynamic mechanism to enable multi-party access to a real network, this is exactly how Service Providers operate day to day; however, the mechanism Sherwood describes [39] for partitioning a network is a transparent proxy for OpenFlow, named Flowvisor, that will:

> "forward a given *[OpenFlow]* message unchanged, translate it to a suitable message and forward, or 'bounce' the message back to its sender in the form of an OpenFlow error message".

Following from the implications of OpenFlow's complexity it is apparent that Flowvisor will not be able to definitively validate messages it receives nor will it be possible to determine if Flowvisor is deriving the same semantics from messages as the switches it is supposed to protect: Flowvisor can not enforce security policy with any certainty. The OpenFlow interpreter of the proxy itself also presents an exceptionally large attack surface.

In a production network that implements partioning through an intercepting OpenFlow proxy scheme (like Flowvisor), the proxy would be the fundamental mechanism for security policy enforcement. In such circumstances, the intercepting proxy would be a highly valuable target and, due to OpenFlow's complexity, an extraordinarily vulnerable one. The failure modes for an OpenFlow intercepting proxy are disastrous and the design simply can not be used to safely partition a single physical network into multiple independently controlled security domains.

4.5.4 Other Approaches

Security issues aside for a brief moment, the SDN verification approaches in the foregoing discussion all have an additional limitation in that they are subjective and only provide a perspective on the internal consistency of the network. The internal perspective cannot be either complete or objective, the information that would be required for either is outside the targeted OpenFlow system and is inaccessible to the verification mechanism. I do think there is operational value in mechanisms for maintaining control over the internal consistency of a network but the limitations of those mechanisms need to be explicitly acknowledged.

If I was trying to verify operational reality instead of just reaffirming a controller's reference point, I would look at verification mechanisms that collect information from sources outside the control and management planes of the target system, such as through out-of-band network instrumentation [4]. A technique that appears to have some promise for such an application is something named Header Space Analysis (HSA) [22]:

> "*[The technique]* treats the entire packet header as a concatenation of bits without any associated meaning. Each packet is mapped to a point in the $0, 1^L$ space, where L is the maximum length of a packet header, and networking *[middle]* boxes transform packets from one point in the space to another point or set of points (multicast)."

The HSA technique has been recently expanded with additional formalism in the form of a tool name NetPlumber [23]. NetPlumber was apparently very effective when applied to Google's SDN WAN, albeit the tool was operating from within the control plane of the network and suffers from the observer issues previously discussed.

[4]I'm attempting to avoid wandering off topic by keeping discussion about measurement and verification as constrained as possible. Instrumenting networks that make heavy use of virtualization, in particular, present an exceptional challenge and is a distinct subject area in its own right.

Special Agent Fox Mulder: Whatever happened to playing a hunch, Scully? The element of surprise, random acts of unpredictability? If we fail to anticipate the unforeseen or expect the unexpected in a universe of infinite possibilities, we may find ourselves at the mercy of anyone or anything that cannot be programmed, categorized or easily referenced.
{Pops a sunflower seed into his mouth}
Special Agent Fox Mulder: What are we doing up here, Scully? It's hotter than hell.

– David Duchovny in "The X Files"

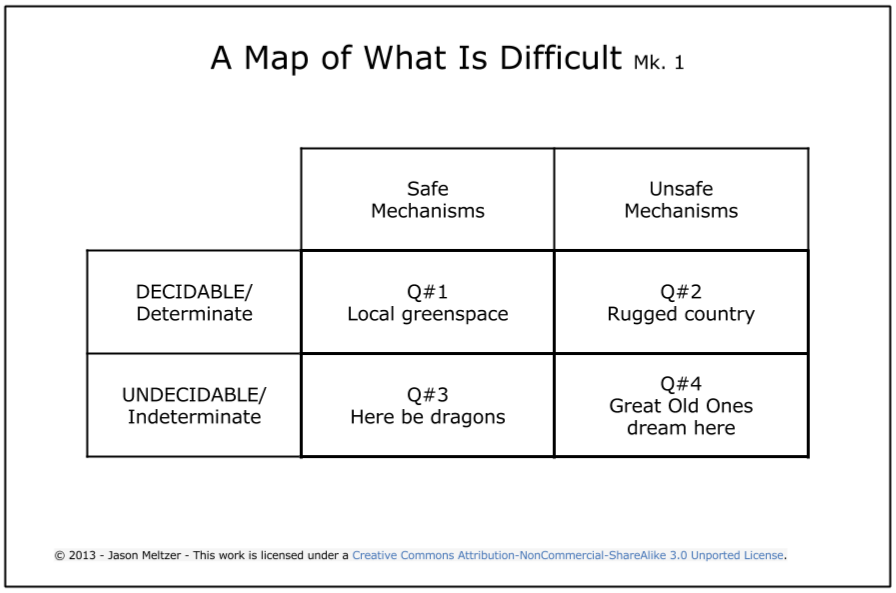
5 A Map of What is Difficult

The Map is a visual reference with an accompanying grand, but hopefully intuitively helpful, analogy to worldly adventure. The intent is to use it to establish a less personally subjective way to frame design discussions with an explicit and multi-dimensional reference for describing some security characteristics of a system, how the characteristics might relate, and how design changes could affect those relationships. To wit, the map is a heuristic tool depicting where experience, expertise, 'being careful', technical mechanisms, et cetera are helpful and where relying on them is likely not possible, perhaps even dangerous. At the very least I think it is a useful representation of important aspects of the implementation security problem in the context of LANGSEC.

Inspiration for the map was drawn in large parts from the areas of mathematical problem solving and risk engineering. Respectively, George Polya's recommendation to, "Draw a hypothetical figure which supposes the conditions of the problem satisfied in all its parts" [33] and Nassim Taleb's discussion of applied statistical knowledge along with his own concept of "The Fourth Quadrant" [41].

After detailing the Map and its analogy I will illustrate its application briefly with OpenFlow and then discuss my perspective on the semantics of implementation mechanism safety.

5.1 The Map and Analogy


A Map of What Is Difficult Mk. 1

	Safe Mechanisms	Unsafe Mechanisms
DECIDABLE/ Determinate	Q#1 Local greenspace	Q#2 Rugged country
UNDECIDABLE/ Indeterminate	Q#3 Here be dragons	Q#4 Great Old Ones dream here

Figure 5.1.1: A Map of What is Difficult

Quadrant #1 Think about the terrain in the first quadrant as being like a stroll through a local municipal park, or similar green-space: it is generally a safe place to let the kids play and any hazards are pretty visible, or arguably non-existent.

Quadrant #2 The second quadrant is an explored wilderness: there are definite hazards, perhaps many, but they are known with some certainty. Experience can guide a traveller around hazards, or otherwise prepare for them, but mistakes are definitely possible, if not inevitable.

Quadrant #3 The third quadrant is likened to areas on the outer edges of an old map where experience is limited and the nature of the terrain is largely hypothetical. It's uncertain whether the adventurous, but prepared, traveller might find relatively benign wilderness or some definitively hostile terrain.

Quadrant #4 Our fourth quadrant is the home of a loose pantheon of ancient, powerful, and malevolent deities; or places of worship for their cults. This is a realm of many-tentacled Lovecraftian horrors replete with unfathomable terror, depthless insanity, and international standards bodies.

5.2 OpenFlow on the Map

An OpenFlow interpreter implemented in C sits squarely in *Quadrant #4* (a genuine horror) and the Map provides some gentle indications of how it might be possible improve the interpreter's circumstances. I'll consider two options that will move the hypothetical OpenFlow interpreter into less arduous terrain by a single step:

1. The first option could be making a move into *Quadrant #2* by removing the implicit requirement to implement solutions to UNDECIDABLE problems. I could remove this requirement for the OpenFlow interpreter by significantly reducing the complexity of the protocol itself, ideally making it at most deterministic context-free.

2. A second option could be to employ safer mechanisms in the implementation and thereby move into *Quadrant #3*. One safer mechanism would be implementing the protocol interpreter in a programming language that is statically typed, strictly evaluated, and uses automatic memory management (such as OCaml) instead of C. Changing the implementation language in this way would eliminate the possibility of most vulnerabilities that rely on abusing memory corruption or data type confusion.

Discussions about my hypothetical implementation now have contextual reference points for the current characteristics of the system design; as well, I have a framework for discussing changes that I might consider for eliminating input handling insecurity. For instance, I can say that, from a practical standpoint, making drastic changes to a widely employed and well entrenched protocol may be slightly fanciful if not entirely impossible. Security vulnerabilities in the new language runtime itself notwithstanding, the second option is entirely under our control and is reasonably practical.

5.3 Implementation Mechanism Safety

If I was to suggest thinking about the vertical partition of the Map as an indication that an exposure exists then the horizontal partition could coarsely articulate the nature, or perhaps even degree, of exposure. Does the implementation mechanism have attributes that make exploiting broad classes of security vulnerabilities fundamentally more difficult? Are there attributes that offer concrete hazard prevention, or otherwise enable risk avoidance? Safer implementation mechanisms are those which embody the aphorism, "The only way to make sure something won't happen, is if it can't happen." It is in this way that *Safety* is being employed as a general descriptor on the Map; formal properties of the mechanisms might contribute to the assessment of safety but it is not about formal properties per se.

When I started writing this section it was initially named “Runtime Environment Safety” but I realised that I’d lost the plot in my own narrative. What I want, is to focus the current discussion on prevention and avoidance. Runtime environment attributes doubtlessly contribute to the overall resilience of a system but they are, by and large, mitigation strategies. Contemporary technical mechanisms with the hinting label of ‘exploit mitigations’ are among the tools that I specifically do not consider immediately relevant, a non-exhaustive list of which include:

- Privilege restrictions and execution containment via structural abstractions, indirections, and architecture (virtualization, sandboxing, ...);
- Operating system/Platform mitigations: ASLR, $W \oplus X$, heap hardening, canaries;
- Compiler-based mitigations: Bounds checking, function replacement.

Unhappy as some people can be with arguments made *via negativa*, I do want to offer a few more specific suggestions for attributes that can affect mechanism safety. I just want to be clear that the intent behind this heuristic is its focus on enabling engineering *discussions* and I’m specifically trying to avoid establishing any assumptions about particular mechanisms. Nevertheless, mechanism safety assessment could consider attributes like the following:

- Memory life-cycle management;
- Data type strength and semantics;
- Computational strength limitations;
- Trusted Computing Base (TCB) minimizations;
- Parser construction tools;
- Explicit grammars and similarly rigorous details in specifications.

...when the weather forecast inevitably becomes RIVERS OF BLOOD ALL DAY EVERY DAY.

– James Mickens in [26]

6 Future Work and similar diversions

6.1 Parsing OpenFlow

The analysis presented in this paper establishes an objective reference point for understanding the significant, and fundamental, obstacles that stand before implementing secure OpenFlow interpreters. Parsing context-sensitive languages isn't trivial in general circumstances let alone those which have strong performance constraints like networking, at least according to the understanding I presently have of the literature. Resigned to live and work with OpenFlow for the near future as we are, I think this is a topic worthy for consideration as Future Work.

In terms of specific suggestions for where the Future Work on securely parsing OpenFlow might proceed, Non-Chomsky grammars and Recognition Systems appear to be promising practical directions. The Attribute Grammar for OXM described in Appendix A is actually an example of a type of Non-Chomsky Grammar that appears to be an effective tool for describing existing protocols in a way that allows parsers to be most easily derived.

As noted in §15.7 of [15], Chomsky Grammars are fundamentally generative descriptions of languages. Just specifying a generative grammar requires significant effort and, even then, deriving practical mechanisms for recognizing those grammars is generally non-trivial. Observing that recognition is usually what actually matters, [15] suggests that a better strategy would be to specify languages, or in our case network protocols, directly with a recognizer system. Considering that the OpenFlow specification development process effectively derived the protocol from an ad hoc recognizer suggests such a strategy might integrate well with real-world development's natural tendencies.

6.2 Hunting OpenFlow Flaws

The main thrust of the analysis in this paper was at the OpenFlow wire protocol itself, however, the specification defines more than just a network protocol. The OpenFlow switch specification defines the structure and behaviour of a hypothetical networking device that the wire protocol is simply an interface to, a device which parses and manipulates other network protocols itself. It would likely be very productive to apply LANGSEC concepts to analysing the way the specification model interacts with other network protocols.

As chance would have it, an example of the type of network protocol parsing inconsistencies that could be illuminated in further such analysis was introduced along with the addition of extensible match support (and consequently OXM) in OpenFlow version 1.2. From the Release Notes section of [29] observe that the following rather significant behavioural change was made in the specification:

```
B.10.7 Removed packet parsing specification

The OpenFlow specification no longer attempts to define how to
parse packets (EXT-3). The match fields are only defined
logically.

    * OpenFlow does not mandate how to parse packets
    * Parsing consistency acheived via OXM pre-requisite
```

When the packet parsing mechanism was specified explicitly it was much more likely, though as we've discussed still not guaranteed, that two OpenFlow endpoints would parse a packet consistently. The specification maintainers did not understand that removing the parsing mechanism requirements is not semantically equivalent to specifying prerequisites for the match contents. It is entirely possible for two OpenFlow endpoints to derive different meaning from the same packets and apply match rules in an inconsistent manner. There are many more similar opportunities to apply the LANGSEC approach to further analysis of the OpenFlow switch specification.

6.2.1 Implementation-defined Behaviour At Best

The OpenFlow switch specification has an unmistakably strong relationship with the C programming language: the specification originated from the C implementation of an experimental software switch and a C header file remains the protocol definition's central feature. The issue here that deserves attention in Future Work is that the definition of message structures in the specification assumes a naive 'hardware' behaviour model that is incorrect for the context of contemporary standards-compliant (C11) development and run-time environment. Using the structures as the specification defines them will result in a number of different implementation-defined and undefined behaviours, some of which are likely to result in security vulnerabilities.

Chiefly, the issues relate to structure memory layouts and assumptions about the nature of alignment and padding. There are numerous instances of these types of problems and a full analysis would be worthy of another thesis, at least. An illustrative example of constructions that lead to problems with undefined behaviour[7] are the numerous message structures, including `ofp_match`, where flexible array members are not declared as the last member of the structure or where, due to nesting, arrays of structures with flexible members are defined inside other structures. The creative use of ad hoc padding to enforce 8 byte alignments in messages also creates all sorts of related issues.

I think a lesson to be found with OpenFlow is that implied implementation mechanisms create signficant risks, which are contributed directly by a protocol designer misunderstanding real-world implementation mechanisms. I would argue that this circumstance aligns well with the reasoning behind the advice from the Internet Engineering Task Force (IETF), stated explicitly in §6 of RFC 2119[4], to make standards descriptive of protocols while avoiding prescription of specific implementation methods.

6.3 Hunting OpenFlow Bugs

A natural corollary to the bug prevention tactic "avoid parsing" is the bug discovery tactic "find parsing". The parser abuse mentioned in the foregoing Implications section of this thesis lays out the context for the strong suspicion that OpenFlow interpreters are likely to have exploitable security bugs. Constructive proof for validating this suspicion could be furnished by constructing exploits for the bugs in OpenFlow implementations predicted by the language-theoretic analysis I have undertaken.

In this vein, translating LANGSEC concepts into novel analysis strategies or even specific tactics is definitely another area for further research. An attack methodology called Parse Tree Differential Analysis, described with application in [36] and [21], is directly derivable from LANGSEC concepts but there are doubtlessly others. It might also be productive to consider existing practical vulnerability discovery techniques from a LANGSEC perspective, such as fuzzing, ideally to derive some improvements if not just to obtain different understanding for their effectiveness.

6.4 Traditional Security Analysis

In this paper I steer quite clear of traditional security analysis of the OpenFlow specification; chiefly for the reason that the typical analysis doesn't account for things such as fundamental computational impediments to fully correct implementation. These impediments being evidence that, I have argued, establishes a prima facie case for insecurity in OpenFlow implementations. Nevertheless, there is always room for more evidence when considering insecurity and the OpenFlow specification provides many opportunities for further analysis. [1].

When the the work on this project began in early summer 2012 no security analyses of OpenFlow had been publicly disclosed. Since that time solid examples of traditional analysis have begun appearing, starting with Wasserman[44] (updated in [43]). Those same authors have also published a draft [17] laying out security requirements for SDN through an IETF working group. The requirements are definitely a good start but it is worth noting that the effort is completely outside the forum responsible for the actual OpenFlow specification. Additional security analyses, arriving most recently at SIGCOMM 2013, likely offer further insight but were not considered in the analysis I have undertaken.

6.5 Tabula Rasa Rasa

> "One of the problems of being a pioneer is you always make mistakes and I never, never want to be a pioneer. It's always best to come second when you can look at the mistakes the pioneers made."
>
> – Seymour Cray, Public lecture at Lawrence Livermore Laboratories on the introduction of the Cray-1 (1976), from Appendix B of [18].

OpenFlow has frequently been discussed as something of a networking analog to the x86 Instruction Set Architecture (ISA). Comparing OpenFlow to x86 isn't entirely fair though: obvious warts notwithstanding, x86 isn't *that* bad. Snottyness aside, there appear to be many parallels between the challenges facing Network Virtualization, particularly OpenFlow, and the evolution of computer processor ISA so I think on the surface

[1]Starting with the absence of any sort of security model... It certainly doesn't bode well that the word *security* has only one relevant appearance in the whole specification, not even warranting its own subsection title. What is there is a very brief subsection (§6.3.3 in [29]) titled "Encryption" that makes (entirely optional) recommendations to run OpenFlow TCP connections over TLS and to authenticate endpoints with certificates. Unfortunately TLS and X.509 certificates are rather prickly mechanisms and their basic integration into any non-trivial system is, at best, a magnificent challenge.

this analogy can be a very apt one. Where the analogy breaks down is in the nature of the interface: OpenFlow is an API, a user interface for programmers, whereas an ISA isn't an API and generally doesn't pretend to be. In terms of the structure of contemporary software an ISA is quite a few levels of abstractions removed from most application software logic [2].

In the implications section I discussed some of the fundamental limitations of verification within OpenFlow systems due to language complexity but I think that even the fact that bolting on verification schemes is necessary speaks to yet deeper problems with the OpenFlow design. In particular, an OpenFlow SDN is a distributed system and yet it makes a priori false assumptions about the reliability, security, capacity of the channels that connect switches and controllers. Sadly, many of these assumptions are so well know that they are collected in a piece of technofolk wisdom known as the "Fallacies of Distributed Computing"[12].

In terms of Future Work here I think there are two related points; first, the design of programmable network flow management interfaces must accept their nature as distributed systems as axiomatic. Second, instead of applying the ISA analogy after the fact, it would be productive to examine designing flow management interfaces starting from an ISA perspective and being mindful of the parsimony that ISA designs require.

6.6 Parsing-aware Design

The LANGSEC approach provides excellent explanations for the fundamental cause of many common classes of software vulnerabilities and it does provide some security engineering advice; however, if there is one aspect that shows the relative newness of the approach it is in the limited availability of more detailed information about applying its principals to real-world protocol design. There is tremendous opportunity for contributing to this aspect of Applied LANGSEC.

Without diving headlong down the rabbit hole of this topic, which is Future Work after all, I can scarcely summarize significant aspects of the situation better than Might and Darais in [27]. A fantastic paper which focuses on parsing tools, those authors astutely observe that nearly every programming language has a well integrated regular expressions package but few have the same level of support for parsers. In particular I think it would be worthwhile investigating the evaluation of parsing approaches especially suited to network protocols (and similar performance concerned applications) as well

[2] I think it's arguable that everything encountered between source code and execution is a substantial layer of abstraction in its own right. This is everything involved in compilation, linking and loading, an operating systems' Application Binary Interfaces…et cetera, et cetera.

as a toolkit that implements them. The Hammer parsing library, from Meredith Patterson no less, has a number of features that make it useful for parsing binary data and is a promising approach in this area [32].

6.7 Reasons Why and Other Approaches

A significant question frequently pops into my mind when I consider LANGSEC and the terrain of existing network protocols, "why do people tend to design protocols that are context-sensitive?". There is a very apparent predisposition towards context-sensitivity in the Internet protocols, in particular, so I think it is worth exploring the nature of this phenomenon and the context related to it. There are questions here that remain very much open.

It is rightly point out by Sassaman [36] that when Claude Shannon was formulating his mathematical theory of communication he was primarily concerned with the problems involved in engineering communication systems; however, it is perhaps a stretch to make a general claim that he regarded the semantics of messages as outside the scope of his model. Shannon *was* aware of the wider implications of his conceptualization of communication and together with Warren Weaver republished his original report together with additional material discussing broader implications of modelling communication mathematically [38].

The original LANGSEC position papers don't spend much time on analyzing the conceptual space between the engineering focus of Shannon's theory the more general problems of communication. In [38] these problems were described by Weaver using a three level taxonomy:

Level A – How accurately can symbols of communication be transmitted? (The technical problem)

Level B – How precisely do the transmitted symbols convey the desired meaning? (The semantic problem)

Level C – How effectively does the received meaning affect conduct in the desired way? (The effectiveness problem)

When discussing modelling communication from a security standpoint in [36] the authors span the conceptual gap between information theoretic communication and semantics with reference to arguments from social science but I don't think that is necessary. In [38] Shannon wrote, "...These semantic aspects of communication are irrelevant to the engineering problem." but he does not discuss the converse. I think the LANGSEC paper misses something when it assumes the foregoing statement about semantics is somehow symmetric, it isn't necessarily. Weaver actually describes in [38] how he, and Shannon, think the problems at levels B and C are related to the statistical characteristics of information at those levels and infers that this makes the level A engineering problems more relevant than is otherwise assumed.

At the time Shannon and Weaver published their work linguistics was in a vastly different paradigmatic era, if only for the reason that Chomsky wouldn't publish his seminal work until a solid decade later. I don't know to what degree the connection described by Weaver remained unexplored but it is intriguing. I think there might be a provable link between the information density of stronger formal languages and information theoretic communications. Some of these ideas about information density are axiomatic to coding theory but the extension to stronger languages doesn't seem to be quite as well explored. Perhaps these ideas are worth further investigation[3].

Intuitively, context-sensitive languages seem more information dense so maybe the predisposition to context-sensitive protocol designs is perhaps due in part to unintentional optimization? By this I mean that it might be that people have a bias to substitute difficult problems at level B and C with related but better understood problems from level A, or otherwise apply techniques from level A. Just to provide some support that the phenomena I hypothesize isn't a completely spurious suggestion, I can offer the fact that this substitution bias is an experimentally reproducible phenomenon, described accessibly by Daniel Kahneman in [20]. I must note that this musing is essentially phenomenological and is by no means an attempt at deriving some sort of reductive grand explanation about protocol design.

6.8 Map Exegesis

The Map heuristic device introduced in this thesis could benefit from further explanation and refinement. Further examples, that is applying it to other implementations and protocols, would likely be helpful. Beyond the Map itself, one of the main aspects of the approach I advocate is developing the concepts and heuristic tools for engineering in the presence of computationally hard problems whose approximate solutions have security implications. The field of distributed computing looks like a particularly compelling source for more such problems.

[3]Indeed, apparently hidden within the overbearing shadow of Chomsky and his conception of formal languages is the field of Mathematical Linguistics, significant aspects of which are statistical and similar quantitative approaches to language. This information was discovered after writing the foregoing paragraph, which I've left as is because it captured my original thoughts.

6.9 Signs and Portents (of Change)

The evolution of OpenFlow has been largely driven by feature extension: capabilities are added to the switch model and the protocol grows accordingly. The first item in the change notes for the v1.4.0 specification was in fact, "More extensible wire protocol". For a protocol that is being widely and rapidly fielded, in non-trivial embedded contexts, the degree to which the syntax and semantics of the protocol continues changing is breathtaking. I seriously wonder at what point OpenFlow caretakers think upgrading implementations will become truly unmanageable, if it isn't already.

The version 1.3.x series of specifications stabilized the wire protocol at version 0×04 between late June 2012 and the middle of August 2013, during which time the use and popularity of OpenFlow has (to put it mildly) experienced exponential growth. No doubt owing to the level of popularity attained during this brief period of lower volatility, the ONF released a package of specifications for optional extensions that can be used to add major 1.4.0 features to an implementation without breaking compatibility with the v.0×04 wire protocol.

Extending older versions of OpenFlow in parallel with defining a new *incompatible* version seems to be a curiously divisive strategy for evolving the protocol and its ecosystem; I am willing to bet hard currency that version 1.3.x will persist for quite some time. Further to that, I think the expansion of the protocol through ever more ad hoc features only serve to embrittle it further and OpenFlow is itself following a trajectory toward the software-defined fate famously described by Alan J. Perlis:

> "In the long run every program becomes rococo – then rubble"

As pessimistic as I am about the future of OpenFlow itself, many of the concepts advanced through the current SDN and network virtualization paradigms do show absolutely tremendous promise as potent tools for improving the use of networking in distributed computing systems.

As I have emphasized in detail already, there are no standard templates or universal solutions... Fundamentals and principles exist, but they require judgement in application, and there is no substitute for studying the environment in detail, developing locally tailored solutions, and being prepared to adjust them in an agile way as the situation develops.

– David Kilcullen, in [25]

7 If you don't stop that you'll go blind...

7.1 Final Words

Language-theoretic security concepts do not define specific solutions so much as they offer a refinement to our understanding of existing problems. Knowing not only where something *does not* work but *cannot* work is the specific sort of implementation relevant engineering knowledge that can be percolated back to inform our doctrine for protocol design and specification. Actionable heuristics developed from this knowledge are especially important because practise demonstrates that the raw attention and care of experts, though necessary, is not sufficient for good engineering.

As a security practitioner I think it is difficult not to notice the wishful thinking of 'expert care' neatly encapsulated in the following quotation from Reitblatt et al[34] in their discussion of determining the safety of configuration updates in an OpenFlow network:

> "We believe that, instead of relying on point solutions for network updates, the networking community needs foundational principles for designing solutions that are applicable to a wide range of protocols and properties. These solutions should come with two parts: (1) an abstract interface that offers strong, precise, and intuitive semantic guarantees, and (2) concrete mechanisms that faithfully implement the semantics specified in the abstract interface. Programmers can use the interface to build robust ap-

> plications on top of a reliable foundation. The mechanisms, while possibly complex, would be implemented once by experts, tuned and optimized, and used over and over, much like register allocation or garbage collection in a high-level programming language. The mechanisms, while possibly complex, would be implemented once by experts, tuned and optimized, and used over and over, much like register allocation or garbage collection in a high-level programming language."

The *what* suggested by Reitblatt is correct I think, but their *how* is a practical impossibility with OpenFlow. Regardless of the good intentions of caring experts, solutions to UNDECIDABLE problems will always be approximate and so anything requiring their implementation will be unavoidably flawed, if only subtly. This isn't to say that time and attention can't beneficially alter the security of a system: the network stacks in many major operating systems today tell just such a story, having had their bugs (metaphorically) hammered out through sheer force of attention over the course of many years. Fixing bugs does not, however, change fundamental attributes of the designs and subtle but devastating vulnerabilities can exist even after years of focus on secure development practises. An existence proof for which is the OpenBSD ICMPv6 `mbuf` handling vulnerability found by Core Security in 2007 [9][1].

So expertise is not a direct solution to intractable problems, but we can inform expertise to improve our ability to identify them. The LANGSEC approach provides such practical means for problem identification, and I have demonstrated how it can be applied to illuminate practical security problems. In this light I think it is not a stretch to claim LANGSEC will have substantial impacts on improving Implementation Security; if not that of OpenFlow, then pretty much everywhere else.

[1]OpenBSD is a free, multi-platform, 4.4BSD-based UNIX-like operating system and is arguable the most secure operating system that is publicly-available. Core elements of the project's philosophy are rigorous application of secure coding practises and a nearly fanatical devotion to implementing correct solutions to problems, despite the time that may be required to develop them. The pace of feature implementation is a common criticism of the project's development process both inside and outside its developer community, but it works.

The interface to an interpreter of a protocol is ***not*** *the protocol.*

– M.A. Padlipsky in [31]

8 Appendix A - Grammar

What follows is an Attribute Grammar for an instance of the OpenFlow `ofp_match` structure containing a sequence of `OFPXMC_OPENFLOW_BASIC` 'OpenFlow eXtensible Match' fields. The grammar follows the notation described by Grune & Jacobs in §15.3.1 of [15] and was chosen, despite being admittedly ad hoc, because it allows for a more straightforward definition of a Linear-Bounded Automata (LBA) recognizer.

Grune & Jacobs note explicitly at the end of §15.3.1 that they are doubtful Attribute Grammars are generative grammars, "since it is next to impossible to use them as a sentence *production* mechanism", and that they should instead be classified as recognition systems. It is in the latter capacity that I employ them here.

Caveat lector, this is a restricted grammar for just the OXM basic field types and I've imposed two, somewhat trivial, constraints in order to be able to write a grammar that isn't absurdly large. The grammar I define:

- does not recognize unknown OXM fields that appear in OXM field sequence;
- does not enforce the prerequisites that exist between OXM fields in sequences, as described in §7.2.2.6 of [29].

8.1 The Grammar

In a further effort to keep the grammar compact it has been made less explicit:

- Sythesized attributes of non-terminals are omitted on the right-hand side of rules as the attributes are already noted where they appear on the left-hand side of grammar rules.
- Inherited attributes, however, *are* noted for non-terminals on the right-hand side of grammar rules in order to make their difference distinct.
- Attribute evaluation rules that encounter undefined attributes can simply defer those rules until the missing attributes have been synthesized in a child node.
- The grammar isn't explicit about what is done with the raw bytes corresponding to the field data because once parsing the larger OXM field is complete they are essentially fixed values.
- `syn` distinguishes left-hand side attributes of grammar rules where the attribute evalution rules of the grammar rule compute (synthesize) the value of that attribute.
- `inh` indicates left-hand side attributes that were computed by the attribute evaluation rules of a parent node, therefore 'inherited'.
- `u16` denotes an unsigned 16 bit integer, likewise `u8` indicates an unsigned 8 bit integer.
- Attribute evaluation statements are contained in `{}` braces.
- `bits(n)` signifies a string of `n` raw bits.
- The length for OXM 'basic' fields is fixed for known type and `hasmask` values, all of which is detailed in the OpenFlow specification. For the sake of tidiness, a long list of the actual values has been omitted and in its place a `lookup_field_length()` function has been defined. Conceptually. the function simply consults a table that contains the length values.

```
ofp_match(inh u16 len, # we always have the external perspective on the size of the structure
          syn u16 plength,
          syn u16 flength): Type Length OXM_fields(flength) Pad(plength)
                            { check(len >= 8); /* min. 8 byte-aligned empty, external check */
                              check(type == 0x0001); /* OXM? */
                              plength:= ((length % 8) ? (8 - (length % 8)) : 0); /* padding? */
                              check(len == (length + plength)); /* consistancy check */
                              flength:= (length - 4);
                              check(flength >= 5); #smallest OXM
                              check( (flength - consumed) == 0); }
```

```
 Type(syn u16 type): bits(16)
                    { type:= value(Type);}

 Length(syn u16 length): bits(16)
                        { length:= value(Length);}

# At this point we know there is at least enough data for OXM_Field to
# contain a single copy of the smallest OXM field, so keep parsing...

 OXM_fields(inh u16 flength,
            syn u16 consumed): OXM
                             { check(flength == oxm_length);
                               consumed:= oxm_length; }

 OXM_fields(inh u16 flength,
            syn u16 consumed): OXM OXM_fields(flen,con)
                             { check( (flength - consumed) >= 5);
                               flen:= flength - oxm_length;
                               consumed:= con + oxm_length; }

 OXM(syn u8 dlength): OXM_Type OXM_Length OXM_Data(dlength)
                     { check(oxm_class == 0x8000);
                       check(oxm_length == spec_length);
                       dlength:= oxm_length; }

 OXM_Type(syn u8 spec_length): OXM_Class Fieldmask
                             { spec_len:= lookup_field_length(oxm_field, oxm_hasmask); }

 OXM_Class(syn u16 oxm_class): bits(16)
                             { oxm_class:= OXM_Class; }

 Fieldmask(syn u8 oxm_field,
           syn u8 oxm_hasmask): bits(8)
                              { oxm_field:= ((Fieldmask >> 1) & 0x7F);
                                oxm_hasmask:= (Fieldmask & 0x01); }

 OXM_Length(syn u8 oxm_length): bits(16)
                              { oxm_length:= value(OXM_Length); }

 OXM_Data(inh u8 dlength): bits(8)
                         { check(dlength == 1); }

 OXM_Data(inh u8 dlength): bits(8) OXM_Data(dlen)
                         { check( dlength > 1);
                           dlen:= (oxm_length - 1); }

 Pad(inh u16 plength):  bits(8)
                       { check(plength == 1); }

 Pad(inh u16 plength):  bits(8) Pad(plen)
                       { check( plength > 1);
                         plen := (plength - 1); }
```

8.2 Slightly more formal notation

The first draft of this thesis featured a note here indicating that the attribute grammar for `ofp_match` could be fleshed out in a more standard formal notation, 'time permitting'. On further reflection, I think adding more notational formalism to this analysis would not have high practical value and, as M. A. Padlipsky might put it, we can all do without me gilding the ragweed any further.

8.3 But there are pictures

Born out of a concern that the attribute grammer I defined for OXM might not be immediately intuitive, I though it would be helpful to provide a visual depiction of the parse tree the grammar implies. The diagram that follows (8.3.1) depicts OXM parsed down to raw byte terminal nodes, where the attributes of non-terminals are annotated using attached boxes and the arrows in the tree indicate the direction of information flow.

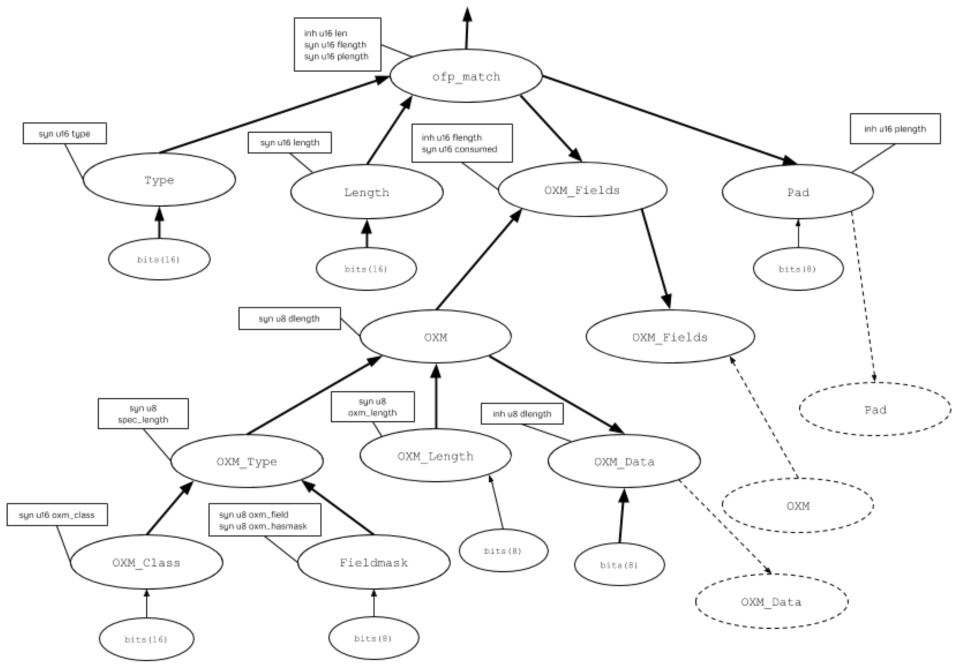


Figure 8.3.1: The tree for an attribute grammar parsing of an OpenFlow `ofp_match` structure.

Raoul Duke: Suddenly, there was a terrible roar all around us, and the sky was full of what looked like huge bats, all swooping and screeching and diving around the car, and a voice was screaming: Holy Jesus. What are these goddamn animals?

– Johnny Depp in Terry Gilliam's Fear and Loathing in Las Vegas

9
Appendix B - Notions

During the course of knitting this thesis together I ended up, metaphorically speaking, with a few extra balls of interesting yarn. The following sections are discussions of topics that are part of a grander examination of OpenFlow and Implementation Security but which turned out to be too awkward to integrate without adding substantially more context, and so length. This thesis being perhaps a bit overlong as it is...

9.1 OpenFlow Spec Complexity Exploding

An independent technology research institute in Brazil, Fundação CPqD, has implemented switches based on the Ericsson TrafficLab software switch for the last three major versions of OpenFlow [8]. To satisfy a matter of curiosity around the software development impact of OpenFlow's version on version feature expansion, I undertook some (very) ad hoc comparison of source code metrics from the three CPqD switches. Specification size increase was calculated from OpenFlow specification page numbers, a very rough metric certainly but directly correlated with feature growth nonetheless. Source Lines of Code (SLOC) values for the switches were obtained using the Scientific Toolworks Understand IDE. Convenient for the purposes of analysis the majority of the switches' OpenFlow-related code is located in a single `oflib` directory. The results are presented in Table 9.1.1.

OpenFlow version	1.1	1.2	1.3	1.4
OpenFlow Spec. size increase	–	25.0%	62.5%	207%
`ofsoftswitch oflib` SLOC	6214	7247	9287	???
Code size increase	–	16.6%	28.1%	???

Table 9.1.1: Obervations of coincident specification and code growth

One should note that the CPqD switches do not implement any backwards compatibility so any growth between successive switch releases should correspond, effectively at least, to handling the expansion of OpenFlow features. A brief examination of the source code for the switches did not indicate that significant refactoring of the `oflib` code had occurred so as to confound the foregoing explanation. No deep interpretation of these observations is particularly necessary, it is simply interesting to observe how substantially the spec and a particular implementation have grown.

The specification for OpenFlow 1.4.0 adds many significant features and consequently expands the protocol enough to necessitate a version bump in the wire protocol. It will be interesting to observe the impact should CPqD continue to evolve their switch implementation. As well, if construction defect rates were to remain relatively constant, or even decline slightly, it is at least reasonable to expect future implementations will have ever greater numbers of bugs.

9.2 Via Negativa

It is often as important to define what something isn't as it is to define what something is, sometimes more so. That in mind, the working definition for Implementation Security I'm proposing does not include the high level secure software development activities such as those that are usually collected under the popular banner of "Security Development Lifecycle (SDL)". The reasoning for this position is that I think SDL activities are executed at a level of abstraction well above the technical engineering I consider to have proximate causal links to the emergent security properties found in implementations. It is entirely possible for bad engineering to be part of a well executed development process.

Leaving aside SDL isn't any sort of attempt to denigrate it. There is reasonably strong real-world evidence establishing correlations between SDL activities and overall software security quality [6] that certainly establishes a measure of value for SDL. In other contexts I will even heap praise on SDL for the simple fact that it brings some focus to software security problems. Nevertheless, confusing the hypothetical outcomes of a process with causality appears to be a frequent conceit of the way entities go about the organization and execution of SDL activities; that is reason enough to make an explicit distinction between Implementation Security and SDL.

9.2.1 Secure Coding

There is a tremendous amount of important and helpful work that has gone into improving software security, particularly over the last twelve or so years[1]. Some of that work can be grouped under the larger banner of SDL, as discussed above, but there are bits that arguably fall under Implementation Security as I've defined it. This situation is precisely my concern though, there are bits and pieces but there is little said about the nature of the incompleteness itself. Essentially, a large part of what I'm arguing for is developing an epistemology of Implementation Security.

9.3 On Models

Ever so briefly, as life is just long enough to make disputes with the milling machinery worthwhile...

Models are the result of abstractions and people[2] generally have at least some intuitive understanding of the fact that a model necessarily contains less information than the thing it describes, the so-called *unnecessary* having been removed. To wit, the modelling process is a lossy one. It is unfortunate then how ephemeral this understanding often is, vanishing especially rapidly as we implement a system based on a design (nee model) which is itself an abstraction of requirements and intents that are, in their own turn, likely incomplete and possibly inconsistent.

That inconsistency is certainly important but I want concentrate on flogging the completeness nag for just a moment or two more... So information is lost during design creation, or just doesn't exist, due to error, omission, or otherwise. So what of this blinding glimpse of the obvious? Three quick observations: first, a realized system contains information beyond what was provided in its design, this extra information is part of what is commonly referred to as Implementation Details. Second, the Details are important because the functionality and behaviour of the realized system ultimately emerges

[1]I'd argue that the time (2001, roughly) around the publication of Howard & LeBlanc's "Writing Secure Code" and Viega & McGraw's "Building Secure Software" is a good reference point for software security properly entering the developer mainstream. Many people were working on software security for a very long time before that, certainly, but in terms of *mainstreaming* I've based my assessment on the question "Can I buy a book from a major publisher about it".

[2]With the exception of most Economists and PHBs.

from those Details, unintended security-impacting functionality (aka. Vulnerabilities) included. Third, the opacity[3] of Implementation Details might be intrisic due to the fundamentally intractable or undecidable nature of problem solutions being attempted in an implementation.

9.3.1 The Teleological Fallacy of Computing Systems Engineering

Riffing on Nassim Nicolas Taleb's more general Teleological Fallacy and applying it to engineering, the illusion that:

- you know exactly what you are building;
- you knew what you were building in the past;
- others have succeeded in the past by knowing what they were building.

9.4 Because API

I've personally argued for the benefits of textual protocols in the past, a la "look I can talk to my server with netcat, testing is easy!". This idea is strongly embraced in the UNIX/Internet programming culture, a fact which reminds me about the *extensive* discussion of protocol design in Eric S. Raymond's Art of UNIX Programming. Incidentally, a book which now needs a thorough critique. Part of the confusion does stem, I think, from anthropocentric misunderstandings of the circumstances of semantic processing and protocol interpreters. Very succinctly, what gets mixed up or otherwise confused is *my* interpretation of semantics versus *endpoint processes'* interpretation. Those viewpoints are not equivalent and they have different requirements for encoding/decoding which don't completely intersect. There are compromises here though and there is *some* overlap: for instance, you can have a protocol be text-based and still be deterministic CFG (or weaker). DJB talks about this in his own way in the 'Avoid Parsing' section of [2] when he describes the concept of 'good interfaces vs. user interfaces'. I might call this situation a grand misunderstanding of a subtle but crucial point. Anyroad, critiquing textual protocol design is a whole other dissertation.

[3] *Opacity* refers to the quality that some particular knowledge is inaccessible ('opaque') to us, and where this inaccessibility leads to illusions of understanding.

9.5 Correct Parser

The following parody was written to introduce the LANGSEC overview section but it just never worked quite the right way with the specific LaTeX formatting I eventually decided on. With all apologies to Adam Brooks and the cast of Frech Kiss…

Kate: Unambiguous context-free grammar — deterministic pushdown automaton. Regular grammar — finite-state automaton. Use the corresponding recognizer for the corresponding language complexity. But no. You want this implementation to be optimized, efficient, extensible…

Luc: Non. No no no. It is not me who wants it. I don't want it.

Kate: Well what do you want?

Luc: I want you… I want you…

Kate: You want me…

Luc: I want you… to… make it secure. To make a potentially malicious user feel like even though you are right there in front of them, they can't exploit you.

References

[1] Ross J. Anderson. *Security engineering - a guide to building dependable distributed systems (2. ed.)*. Wiley, 2008. ISBN 978-0-470-06852-6.

[2] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In Peng Ning and Vijay Atluri, editors, *CSAW*, pages 1–10. ACM, 2007. ISBN 978-1-59593-890-9.

[3] M. Boucadair and C. Jacquenet. Software-defined networking: A service provider's perspective. Internet-Draft draft-sin-sdnrg-sdn-approach-04, Internet Engineering Task Force, October 2013. URL https://datatracker.ietf.org/doc/draft-sin-sdnrg-sdn-approach.

[4] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), March 1997. URL http://www.ietf.org/rfc/rfc2119.txt.

[5] 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:*, 36(6), 2011. URL http://www.cs.dartmouth.edu/~sergey/langsec/papers/Bratus.pdf.

[6] BSIMM. The building security in maturity model, 2013. URL http://bsimm.com.

[7] CERT Secure C Coding Standard. Dcl38-c. use the correct syntax when declaring flexible array members, November 2013. URL https://www.securecoding.cert.org/confluence/display/seccode/DCL38-C.+Use+the+correct+syntax+when+declaring+flexible+array+members.

[8] Fundação CPqD. Github page, 2013. URL https://github.com/CPqD.

[9] CVE-2007-1365. Openbsd ipv6 mbuf remote kernel buffer overflow. National Vulnerability Database, March 2007. URL http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-1365. [description] http://www.coresecurity.com/content/open-bsd-advisorie.

[10] CVE Database. Search of cve database for linux ipv6 network stack vulnerabilities. Common Vulnerabilities and Exposures Database, 2013. URL "https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux%2C+net%2Fipv6".

[11] Bruce Davie. Network virtualization: Delivering on the promise of sdn, 2013. URL http://www.slideshare.net/drbruced/ons-2013nv.

[12] Peter Deutsch. The eight fallacies of distributed computing, 1992. URL https://blogs.oracle.com/jag/resource/Fallacies.html.

[13] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment*. Addison-Wesley Professional, November 2006. ISBN 978-032144-442-4.

[14] Nick Feamster and Jeff Mogul, editors. *NSDI'13: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2013. USENIX Association.

[15] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide (2. ed.)*. Springer-Verlag, 2007. ISBN 978-0-3872-0248-8. URL http://dickgrune.com/Books/PTAPG_2nd_Edition/.

[16] Robert J. Hansen and Meredith L. Patterson. Guns and butter: Toward formal axioms of input validation. In *Black Hat USA 2005*. Black Hat Briefings, 2005. URL https://www.blackhat.com/html/bh-media-archives/bh-multi-media-archives.html#USA-2005.

[17] S. Hartman, M. Wasserman, and D. Zhang. Security requirements in the software defined networking model. Internet-Draft draft-hartman-sdnsec-requirements-01, Internet Engineering Task Force, April 2013. URL https://datatracker.ietf.org/doc/draft-hartman-sdnsec-requirements/.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Edition*. Morgan Kaufmann, 1996. ISBN 1-55860-329-8.

[19] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In Dah Ming Chiu, Jia Wang, Paul Barford, and Srinivasan Seshan, editors, *SIGCOMM*, pages 3–14. ACM, 2013. ISBN 978-1-4503-2056-6.

[20] Daniel Kahneman. *Thinking, Fast and Slow*. Doubleday Canada, 2011. ISBN 978-038567-651-9.

[21] Dan Kaminsky, Meredith L. Patterson, and Len Sassaman. Pki layer cake: New collision attacks against the global x.509 infrastructure. In Radu Sion, editor, *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 2010. ISBN 978-3-642-14576-6.

[22] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association. URL http://cseweb.ucsd.edu/~varghese/PAPERS/headerspace.pdf.

[23] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In Feamster and Mogul [14], pages 99–112. URL http://www.sysnet.ucsd.edu/sysnet/miscpapers/net_plumber-nsdi13.pdf.

[24] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In Feamster and Mogul [14], pages 15–28. URL http://www.sysnet.ucsd.edu/sysnet/miscpapers/net_plumber-nsdi13.pdf.

[25] David Kilcullen. *Counterinsurgency*. Oxford University Press, 2010. ISBN 978-019973-749-9.

[26] James Mickens. The night watch. *;login:logout*, 38, November 2013. URL http://research.microsoft.com/en-us/people/mickens/thenightwatch.pdf .

[27] Matthew Might and David Darais. Yacc is dead. *CoRR*, abs/1010.5023, 2010.

[28] Nettle. Haskell openflow package, 2013. URL http://hackage.haskell.org/package/nettle-openflow.

[29] OpenFlow v1.4.0. Openflow switch specification version 1.4.0 (wire protocol 0x05). Approved-Draft, August 2013. URL https://test.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[30] OpenMirage. Cloud operating system, 2013. URL http://www.openmirage.org/.

[31] M. A. Padlipsky. *The Elements of Networking Style*. iUniverse.com, 2000. ISBN 0-595-08879-1.

[32] Meredith L. Patterson. Hammer parsing library, 2013. URL https://github.com/abiggerhammer/hammer.

[33] G. Polya. *How to Solve It: A New Aspect of Mathematical Method*. Princeton University Press, 1945. ISBN 0-691-02356-5.

[34] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In Lars Eggert, Jörg Ott, Venkata N. Padmanabhan, and George Varghese, editors, *SIGCOMM*, pages 323–334. ACM, 2012. ISBN 978-1-4503-1419-0.

[35] Gyorgy E. Revesz. *Introduction to Formal Languages*. Dover Publications, 2012. ISBN 978-0-4866-6697-6.

[36] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. Security applications of formal language theory. Technical Report TR2011-709, Dartmouth College, 2011. URL http://www.cs.dartmouth.edu/~sergey/langsec/papers/langsec-tr.pdf.

[37] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Anna Shubina. The halting problems of network stack insecurity. *;login:*, 36(6), 2011. URL http://db.usenix.org/publications/login/2011-12/openpdfs/Sassaman.pdf.

[38] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1949. ISBN 978-025272-546-3.

[39] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru M. Parulkar. Can the production network be the testbed? In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *OSDI*, pages 365–378. USENIX Association, 2010. ISBN 978-1-931971-79-9.

[40] Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip A. Porras, and Guofei Gu. Model checking invariant security properties in openflow. In *ICC*, pages 1974–1979. IEEE, 2013.

[41] Nassim Nicholas Taleb. The fourth quadrant: A map of the limits of statistics. *Edge.org*, 2008. URL http://edge.org/conversation/the-fourth-quadrant-a-map-of-the-limits-of-statistics.

[42] VMWare. Nicira acquisition press-release, 2012. URL http://www.vmware.com/company/news/releases/vmw-nicira-07-23-12.html.

[43] M. Wasserman and S. Hartman. Security analysis of the open networking foundation (onf) openflow switch specification. Internet-Draft draft-mrw-sdnsec-openflow-analysis-02, Internet Engineering Task Force, April 2013. URL https://datatracker.ietf.org/doc/draft-mrw-sdnsec-openflow-analysis.

[44] M. Wasserman, S. Hartman, and D. Zhang. Security analysis of the open networking foundation (onf) openflow switch specification. Internet-Draft draft-mrw-sdnsec-openflow-analysis-00, Internet Engineering Task Force, October 2012. URL https://datatracker.ietf.org/doc/draft-mrw-sdnsec-openflow-analysis.

Colophon

THIS THESIS WAS TYPESET using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Arno Pro, designed by Robert Slimbach in the style of book types from the Aldine Press in Venice, and issued by Adobe in 2007. The style this thesis was formatted in is the result of the author's significant modifications to a template created by Jordan Suchow (many thanks!). The original template was released under the permissive MIT (X11) license and can be found online at github.com/suchow/ or from the author at suchow@post.harvard.edu.