Linnea: Detecting blacklist-evading malware with SQL rules

Tobias Ruck and Miranda Mowbray
Hewlett-Packard Enterprise

Abstract

We present a system for detecting malware that uses domain generation algorithms (DGAs) to evade blacklisting. We use SQL rules that identify patterns specific to the malware family in the non-resolving domains queried by infected clients. We have designed a language to describe these rules more easily, which can be compiled to SQL. Using this approach we detected ten DGA families in a day's data from a large enterprise.

Keywords

Network Security; Malware Detection; Domain Generation Algorithms; SQL

1 Introduction

In this paper we introduce a system for detecting clients in enterprise networks that are infected with malware that uses domain generation algorithms (DGAs) to evade blacklisting. Our system uses SQL rules applied to data gathered from Domain Name Service (DNS) servers. We have designed a language that compiles to SQL for easily expressing such rules. Our compiler and rules for 16 families are publically available (see section 2.6). We have deployed our system in a real context. In an evaluation on a day's worth of data from a large enterprise, our system correctly detected the presence of ten different DGA families. This work has not been previously published.

The controllers of botnets and other malware need to communicate with infected machines, to send them instructions and/or to receive data from the machines. Some malware is designed to evade blacklisting of domain names, by including an algorithm that generates a batch of domain names, based on a seed and sometimes also the date. An infected machine periodically generates a new batch of domain names and tries to connect with them all. To communicate with an infected machine, the botnet controller registers one of the domain names in the batch, and uses that domain for command and control.

When an infected machine tries to connect to the domains, it requests their IP addresses from the Domain Name Service. This activity leaves traces in the network's DNS data. The domains in the batch that are not used for command and control will be NXDOMAINs, that is, they will not resolve to an IP address (unless they have been *sinkholed*, i.e. registered by a security expert to prevent them from being used by the malware controller).

In the rest of this section we describe previous work in this area. Section 2 presents our system, and Section 3 gives results from an evaluation and mentions further steps.

1.1 Previous work

There has been previous work done on detecting clients infected by blacklist-evading malware via its traces in DNS data. This work mostly uses machine learning, in various different ways, including clustering, linguistic analysis, and anomaly detection. (Some authors combine two or three of these.)

The clustering approach, for example [Antonakakis et al, 2012; Thomas & Mohoisen, 2014], uses the fact that if more than one client in the network is infected by the same DGA malware using the same seed for domain generation, the infected clients will request IPs for the same DGA-generated set of domains at roughly the same time. This approach looks for sets of domains with similar features (for example syntactical similarity, or similarity of registration history) that are requested at similar times by the same set of clients. This approach has had some success, but it can only detect malware that has infected several different clients in the network. Our solution does not have this limitation.

Another machine-learning approach is to identify clients that request many domains that have linguistic features unusual in human-generated domains; see e.g. [Mouchouz, 2013; Schiavoni 2014]. This works well for some malware, but is not a good approach for detecting malware families such as Suppobox [Geffner, 2013] or Matsnu [Skuratovich, 2015] which use the increasingly common ploy of generating domains based on dictionary words. In our evaluation, our system detected a Matsnu infection.

Another machine-learning approach is to use anomaly detection to identify clients requesting an unusual set of domains: for example, a set with an unusual distribution of domain lengths [Mowbray & Hagen, 2014]. This however does not identify the malware family causing the anomaly. Identification of the malware is important, as it has implications for what actions should be taken.

A general disadvantage of machine learning is that it can lack transparency. It may not be at all clear why a client has been identified as infected, and hence the reason for any false positives or false negatives. Our approach does not use machine learning. The rules that we use to identify infected clients are explicit, and are intended to be easy to understand, and also easy to update if necessary.

[Krishnan et al, 2013] gives an intriguingly simple approach using sequential hypothesis testing. It looks for clients that request non-resolving domains for many DNS zones, and do not make many requests of non-resolving domains for zones that it has already seen. This appears to give good results, however it does not identify which malware a client is infected with. It may also have problems identifying families such as Symmi [Bader, 2015], which generates domains that are all subdomains of `ddns.net`.

2 Our Approach

The basic idea of our system is to detect that a client (identified by its IP address) is infected by a particular blacklist-evading malware family by recognizing a pattern, specific to that family, in the domains that it requests. We write a SQL rule for each family that encodes the pattern, and run it on an enterprise's DNS data. Our system only looks for malware families for which a rule has been written. However, if a new DGA-using family is discovered, it is fairly simple to design a new rule to detect it.

2.1 Architecture Overview

We begin by describing the architecture into which our solution fits. It is illustrated in Fig 1.

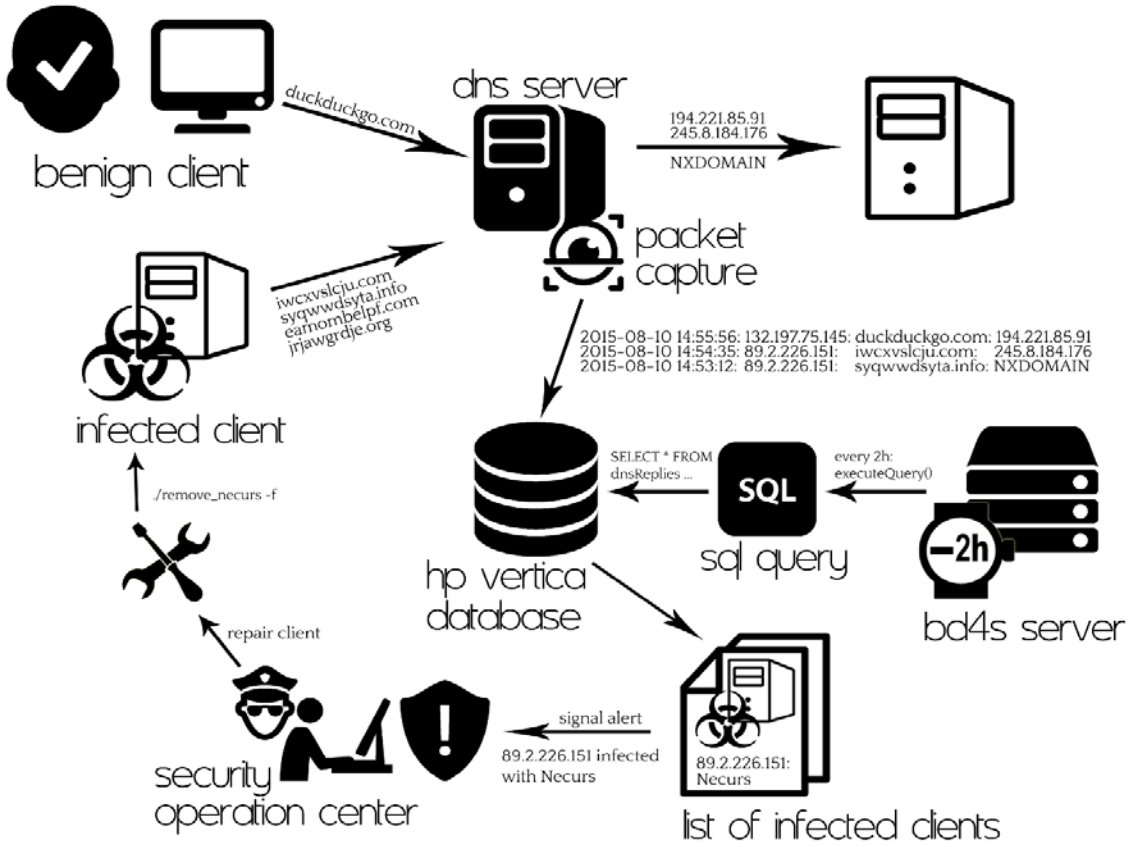

Figure 1: Overall Architecture

We capture packets from enterprise DNS servers, and copy information from them to a HP Vertica database. Requests for IP addresses of very popular domains such as `google.com` and `facebook.com` are ignored, as are requests for malformed domains (e.g. domains containing no dot.) The information that we can retrieve from the database includes `timestamp`, the time of a DNS packet; `client`, the IP address of the requesting client; `domain`, the domain requested; `d0`, the public suffix (for example the public suffix of `example.ac.fr` is `ac.fr` and the public suffix of `2.example.com` is `com`); `d1`, the substring of the domain between the last two dots before the public suffix; `l0` and `l1`, the lengths of `d0` and `d1`; and `nxdomain`, which is true if and only if the domain is an NXDOMAIN.

Every two hours, our server (called the *BD4S server*, where BD4S is short for Big Data for Security) runs a SQL query for each malware family on the last two hours' data. The query returns a list of client IPs in the enterprise detected as having an infection from this family, and the algorithmically-generated NXDOMAINs that they have requested. This is sent as an alert to the security operations center for the enterprise, and mitigating action can be taken. Depending on the malware detected, this might for example be an automated response to the employee using an infected machine with instructions on how to disinfect it, a quarantining of the machine, or further investigation of the machine by experts in the security operations center. The choice of a two-hour interval, rather than a different interval (or streaming detection), fits with the procedures commonly used by enterprise security operations teams.

The infrastructure consisting of packet capture, database and application frontend was presented in [Horne, 2014], and has been used on HP's own network and in trials on other enterprise networks. The additional system is presented for the first time in this paper. It is lightweight, consisting just of a compiler (about 600 lines of Python, excluding comments), and a scheduler. It is thus easy to integrate in an already working system. For a smaller organization, an alternative to using packet capture would be to populate a database from DNS logs.

2.2 SQL queries: an example

In this section we describe how we design SQL queries to detect DGA families.

The DGA families for which we currently have SQL queries are listed below. The Microsoft Malware Encyclopedia [Microsoft, 2015] has entries for all the families for which a specific reference is not given. In one case (New-DGA-v1) the DGA has not yet been identified as far as we are aware as being caused by a known malware family, and so it might be benign.

> Bankpatch [Antonakakis et al., pp.21-22], Bedep, Conficker (we have a single rule that identifies Conficker-A or Conficker-B), DGA10 [Mowbray & Hagen, 2014], Dyre [Chiu & Villegas, 2015], Expiro, Matsnu [Skuratovich, 2015], New-DGA-V1 [Antonakakis et al., p.19], Necurs, Pitou, Pushdo, Pykspa, Ramdo, Runforestrun [Unmask Parasites, 2012a, 2012b; MalwareMustdie, 2012], Shiotob, SillyFDC.

Our SQL queries do not consider individual domains in isolation: rather, they look at the set of NXDOMAINS queried by each client IP in a particular time interval, which allows for more accurate detection of infected client IPs. The query design is intended to be flexible. As a result, the queries contain some redundancy, but as will be shown in the evaluation section, they have reasonable performance.

As an example, here is the SQL query used to detect the Pushdo malware family. Pushdo's DGA generates domains consisting of a string of 9-12 alphabet characters followed by `.kz`, possibly also preceded by `www.`, and vowels (including y) have higher probabilities of occurring in the string than consonants do.

```
SELECT f.client, f.domain, f.timestamp
FROM (
   SELECT d.client, d.domain, d.timestamp, d.d0
      COUNT(d.d0 IN ('com', 'info', 'net', 'in') OR NULL) OVER (PARTITION BY d.client
          ORDER BY d.timestamp RANGE BETWEEN INTERVAL '1 hour' PRECEDING AND INTERVAL
          '1 hour' FOLLOWING) AS number_non_kz,
      COUNT(d.d0 = 'kz' OR NULL) OVER (PARTITION BY d.client ORDER BY
          d.timestamp RANGE BETWEEN INTERVAL '1 hour' PRECEDING AND INTERVAL '1 hour'
          FOLLOWING) AS number_kz,
      COUNT(d.domain) OVER (PARTITION BY d.client ORDER BY d.timestamp
          RANGE BETWEEN INTERVAL '1 hour' PRECEDING AND INTERVAL '1 hour' FOLLOWING)
          AS number_requests
    FROM (
        SELECT domain, client, MAX(timestamp) AS timestamp, d0
        FROM hplDNSReplies
```

```
        WHERE timestamp >= '2015-08-05 00:00:00'
            AND timestamp <= '2015-08-05 23:59:59'
            AND REGEXP_INSTR(request,
                '^(www\.)?[a-z]{9,12}\.(com|in|info|kz|net)$') > 0
            AND nxdomain
            AND REGEXP_COUNT(d1, '[aeiouy]') / l1 > 0.33
        GROUP BY client, domain, d0
    ) d
) f
WHERE f.number_requests >= 20 AND f.number_non_kz < f.number_kz AND f.d0 = 'kz';
```

An informal description of this query is:

1. Select all the NXDOMAINs requested in the timeframe that match the regex `^(www\.)?[a-z]{9,12}\.(com|in|info|kz|net)$` and where `d1` has more than 33% vowels. To remove duplicate domains, group by client and domain and set the timestamp for each (client, domain) pair to be the last timestamp for this pair.
2. For each (client, domain) pair, count the number of selected NXDOMAINs ending `.kz` and not ending `.kz` that were requested by the client, whose timestamps are no more than one hour before or after the timestamp for the (client, domain) pair. If there were least 20 such domains in total, and more than half of them were `.kz` domains, return the client and domain.

The check that more `.kz` domains are requested than domains with public suffix `com`, `in`, `info` and `net` is made in order to distinguish Pushdo domains from domains generated by Flashfake [Gostev, 2012].

After some time spent constructing SQL queries like the one above, we decided that rather than designing them directly it would be easier to specify them in a higher-level language which could then be compiled into SQL. We have written a language for this purpose, which we outline in the next section.

2.3 Linnea, a language for domain-wise predicates

This section describes a simple language called Linnea, which we have written to make it easier to design SQL queries for DGA detection. A summary of how to compile Linnea to SQL is given in section 2.5. The following specifies the elements of the language.

1. `P_0, ..., P_n` defines the program, where each `P_i` is a *predicate set*. Starting with `P_0`, the entirety of the domain data is fed into `P_0`, which yields the remaining domains, which will be fed into `P_1`, and so forth until `P_n`, which will be the output of the program.
2. `{p_0,...,p_n}` defines a predicate set; it is true if and only if all predicates `p_0,...,p_n` are true.
3. For Booleans and numbers, we have the usual arithmetic and logical operators.
4. `a = b` is the string equality predicate.
5. `match(s,r)` is true when the string `s` matches the regular expression `r`.
6. `count(s,r)` counts the number of occurrences of the regular expression `r` in `s`.
7. `| g in h_0,...,h_n: p |` counts how often the predicate `p(g)` holds for each g element of `{h_0,...,h_n}`.
8. `g in h_0,...,h_n` yields true if and only if g is element of {h_0,...,h_n}.
9. `[ a_0,...,a_n:T|p ]` finds in the current set of domain data those domains, that are in the timeframe [timestamp-T;timestamp+T] and have the same properties as the current domain, with respect to the property names `a_0,...,a_n`. It then counts given those how many fulfil the predicate `p`. These expressions are not allowed in `P_0` for performance reasons.
10. Various variables for each domain can be accessed, e.g. `domain, client, timestamp, nxdomain, d0, l1`.

Given these rules, we can define DGA-specific rules to detect infected clients easily. For example the rule for Matsnu [Skuratovich, 2015] is this:

```
{
        timestamp >= t0 - 2h, timestamp <= t0,
        nxdomain,
        match(domain, '^[a-z-]{11,23}\.com$')
```

```
},
{
        [client:1h|true] >= 25,
        [client:1h|count(d1, '-') = 1] +
        [client:1h|count(d1, '-') = 2] >= 20,
        [client:1h|count(d1, '-') = 1] >= 9,
        [client:1h|count(d1, '-') = 2] >= 9
}
```

Which reads as:

1. Select all NXDOMAINS that match a specific regex.
2. Given this set, select the requests by clients that request at least 25 domains, at least 20 of which contain one or two hyphens, at least 9 with exactly one and at least 9 with exactly two.

Below are some of the Matsnu domains detected on 11th August 2015. They were all requested by the same client. It would be hard to tell for example that `birthday-baby.com` was algorithmically-generated, without the context of the other domains the client requested around the same time. Our rule detected the Matsnu domain `professorloose.com` although it does not have a hyphen.

`birthday-baby.com, professorloose.com, term-cow-record.com, blank-operate.com, quantity-apply.com, title-smart-media.com, blindchart-pair.com, reading-persuade.com, reading-sort.com tongue-warm-funeral.com, blacksource-method.com`

Another example is the rule for New-DGA-v1 [Antonakakis et al, 2012]:

```
{
    timestamp >= t0 - 2h, timestamp <= t0,
    nxdomain,
    match(domain, '^[a-f0-9]{8}\.(com|info|net)$')
},
{
    [client:1h| |suffix in 'com','info','net':
        [client,d1:1h|d0=suffix]>=1| >= 2] >= 10
}
```

1. Select all NXDOMAINS that match a specific regex.
2. From this set, select the requests by clients that request at least 10 different domains that each have a companion domain in the set, consisting of the same string followed by a different suffix, requested by the same client: e.g. `abcabcde.com`, `abcabcde.net` are companion domains.

2.4 Building Linnea statements from sample data

In some cases the domain-generation code for a malware family has been reverse-engineered, but this is a highly-skilled task. It is more usual to design a SQL rule based on patterns observed by eye or with some analysis tool, in sets of domains requested by clients infected with a particular DGA-using malware family.

Patterns may be for example, about distributions of `l1` values or `d0` values, a restricted character set for `d1`, limits on how often particular characters appear, or the batch size. These can easily be detected via regular expressions, or counts of matches to them. In our Linnea rules, regular expressions usually appear in the first predicate layer, `P_0`. Then there are properties specific to a batch of requests, for instance how many different domains matching the regular expression are requested within a timeframe, or how many of these contain a hyphen. In Linnea, predicates to detect these properties are in the predicate layers after `P_0`, for performance reasons.

The Linnea rules that we have written so far all use the value of 1 hour for the timeframe parameter `T` used in (9). However we could use a smaller value for malware families that request many algorithmically-generated domains in a short time interval.

2.5 Constructing SQL queries from Linnea

A compiler for Linnea must parse a Linnea file, convert it into an abstract syntax tree, traverse it, and build a SQL query from inner to outer query.

Most of the elements of Linnea can be converted into SQL directly. These elements are the usual arithmetic (rule 3), the *in* syntax (8), the regex functions (5, 6) and the domain variables (10). For double bar notation (7), the compiler just repeats the predicate for each item, replacing the item placeholder with the item, converting the Boolean to a number $\{0, 1\}$, and concatenating them with the addition operator. For predicate sets (2), the compiler concatenates all predicates with 'AND', and it spawns a SQL sub-query for each of the elements in a program (1).

Compiling the bracket notation (9) is a bit more complicated. The whole expression is replaced by a placeholder name. A new sub-query is spawned in which a `COUNT(...) OVER(...)` analytic function with an alias of the placeholder name is inserted in the `SELECT` part of the sub-query. The compiler appends `OR NULL` to the predicate in the `COUNT` expression, because `COUNT` counts non-`NULL` values. The analytic function is partitioned by the parameters `a0, a1..., an,` and is restricted to the timeframe from `timestamp-T` to `timestamp+T` inclusive. Then the compiler returns to the super-query.

2.6 Linnea Compiler Implementation and Query Examples

A working, but not thoroughly tested implementation of a Linnea compiler can be found at https://github.com/EyeOfPython/Linnea. Instructions on running the script are given in the readme file. The implementation uses the *pyparsing* module for the syntax analysis. It offers the ability to directly execute the compiled queries on a SQL database. For this to work, the *pytoml* and *pyodbc* packages have to be present. Examples for some DGAs are in the examples/ folder. At the time of writing they include the 16 DGAs listed in Section 2.2.

3 Evaluation and Further Work

We evaluated our system for performance and accuracy, using a day's worth of DNS data collected from a large enterprise network on 11 August 2015. There were 18.2 million NXDOMAIN entries in the database for that day, recording DNS requests for 308,176 distinct non-resolving domain names by 68,757 clients in the enterprise, where the clients were identified by their IP addresses. (These figures exclude NXDOMAIN requests from IP addresses known to be of servers rather than end clients, for example web servers; we did not count these servers as clients). We ran our queries on a HP Vertica database server with standard configuration, over a standard cluster of HP DL30 servers. HP Vertica is a database with a columnar architecture designed for speed and scalability [Hewlett-Packard Development Company, 2014].

We measured the execution time for each of our SQL queries for each 2-hour timeslot during the day. The longest query execution time was 2.144 seconds, the shortest 0.949 seconds. The average total time for running all our queries consecutively was 19.3 seconds. We only need to run the queries once every two hours, so if the current queries are representative, we could add queries for thousands more DGAs without the server becoming busy. This performance means that when new Linnea rules are being developed, the results of experimental changes to a rule can be displayed within a few seconds.

The data was independently examined (by eye, with the assistance of some simple analytic tools) to identify clients requesting domains from the 16 DGA families for which we currently have Linnea rules. These results were compared with the results from the SQL queries. Strikingly, 10 out of the 16 DGA families were present in the day's data. These were Bedep, Conficker, DGA10, Expiro, Matsnu, New-DGA-V1, Necurs, Pitou, Pushdo and SillyFDC. All 10 were detected by our SQL queries. At the level of identification of individual clients there were two false negatives, one false positive, and 72 clients correctly identified as requesting a batch of domains from one of these families.

The independent examination also found some clients, not detected by our SQL rules, which requested a few domains from one of these families, but not a full batch. These clients did not successfully

connect to a DGA command and control domain on 11 August. They may have been disconnected and/or disinfected before the operation of the DGA could be completed. We did not count them as false negatives.

As a next step for this work, we will run our system on more enterprise data and compare its results with using a machine-learning approach on the same data. Other obvious steps are to design Linnea rules for more DGAs, and to check rules for DGAs that are detected in the new data but were not present in the one-day data set. Both of the false-negative clients in our evaluation requested 9 or more sinkholed DGA domains, so we could also try to improve our approach by treating domains that resolve to known sinkhole IP addresses as though they were NXDOMAINs.

Acknowledgements

Icons used in Fig 1: icons8.com, atyourservice.com.cy. Miranda thanks the BD4S team. Tobias says: I'd like to acknowledge Claudius Wild, Linnèa Wulf, and Neytiri Te Tskaha (yawne lu oer), for helping me to find a good name for the language. Thanks to Tim Stoffel for reviewing help. Many thanks to my compiler construction docent Yannik Hampe. Let me mention Maik Bettka as well, since I forgot that once. Sorry for that. Many thanks to my parents for buying me hundreds of centners of books about programming.

References

[Antonakakis et al, 2012] Manos Antonakakis, Roberto Perdisci, Nikolaus Vasiloglu and Wenke Lee, "Detecting and Tracking the Rise of DGA-Based Malware", USENIX *;login*: vol. 37 no.6 pp.15-24, December 2012 https://www.usenix.org/system/files/login/issues/login_1212.pdf

[Bader, 2015] Johannes Bader, "The DGA of Symmi", blog post, 21 January 2015, http://www.johannesbader.ch/2015/01/the-dga-of-symmi/

[Chiu & Villegas, 2015] Alex Chiu and Angel Villegas, Talos Group, "Threat Spotlight: Dyre/Dyreza: An Analysis to Discover the DGA", Cisco Blog Post, 30 March 2015, http://blogs.cisco.com/security/talos/threat-spotlight-dyre

[Geffner, 2012] Jason Geffner, Crowdstrike, "End-to-end analysis of a domain generating algorithm malware family", Black Hat USA 2013, https://media.blackhat.com/us-13/US-13-Geffner-End-To-End-Analysis-of-a-Domain-Generating-Algorithm-Malware-Family-WP.pdf

[Gostev, 2012] Alexander Gostev, "The anatomy of Flashfake", Securelist blog posting, 19 April 2012 https://securelist.com/analysis/publications/36423/the-anatomy-of-flashfake-part-1/

[Hewlett-Packard Development Company, 2014] HP, "Big Data SQL Analytics", http://www8.hp.com/us/en/software-solutions/advanced-sql-big-data-analytics/

[Horne, 2014] William Horne, HP Discover 2014, "A sneak peek at the future of Big Data for security with HP Labs", 12 June 2014, https://www.youtube.com/watch?v=JwL1gkc9Zxk

[Krishnan et al, 2013] Srinivas Krishnan, Teryl Taylor, Fabian Monrose and John McHugh, "Crossing the Threshold: Detecting Network Malfeasance via Sequential Hypothesis Testing", DSN 2013, http://cs.unc.edu/~fabian/papers/dsn2013.pdf

[MalwareMustDie, 2012] MalwareMustDie NPO Research Group, "DGA/Pseudorandom Case: RunForrestRun/JS", 3 November 2012, https://code.google.com/p/malwaremustdie/wiki/DGA_Research_Tips

[Microsoft, 2015] Microsoft, "Malware Encyclopedia", 2015 http://www.microsoft.com/security/portal/threat/threats.aspx

[Mouchouz, 2013] R. Mouchouz, "DNS Resolution Traffic Analysis Applied to Bot Detection", Botconf 2013, Nancy, France, 5-6 Dec 2013

[Mowbray & Hagen, 2014] Miranda Mowbray and Josiah Hagen, "Finding Domain-Generation Algorithms by Looking at Length Distributions", RSDA 2014, Naples, Nov 3-6 2014 http://www.hpl.hp.com/people/miranda_mowbray/Camera-Ready-RSDA.pdf

[Schiavoni et al, 2014] S. Schiavoni et al., “Phoenix: DGA-based Botnet Tracking and Intelligence”, DIMVA 2014, http://www.syssec-project.eu/m/page-media/3/schiavoni-dimva14-phoenix.pdf

[Skuratovich, 2015] Stanislav Skuratovich, “MATSNU Malware ID”, Check Point blog post, 15 May 2015 http://blog.checkpoint.com/wp-content/uploads/2015/07/matsnu-malwareid-technical-brief.pdf

[Thomas & Mohaisen, 2014] M. Thomas and A. Mohaisen, “Kindred Domains: Detecting and Clustering Botnet Domains Using DNS Traffic”, 23rd World Wide Web (WWW’14) companion publication, IWWWCSC, Geneva, Switzerland, 2014, pp. 707-712. DOI: 10.1134/2567948.2579359

[Unmask Parasites, 2012a] Unmask Parasites, “Runforestrun and Pseudo Random Domains”, blog post, 22 June 2012, http://blog.unmaskparasites.com/2012/06/22/runforestrun-and-pseudo-random-domains/

[Unmask Parasites, 2012b] Unmask Parasites, “RunForestRun Now Encrypts Legitimate JS Files”, blog post, 26 July 2012, http://blog.unmaskparasites.com/2012/07/26/runforestrun-now-encrypts-legitimate-js-files/