Information Security

Domain Generation Algorithms

Unification and Analysis - A Novel Framework

Diploma Thesis

For attainment of the academic degree of

Diplom-Ingenieur/in

submitted by

Georg HEHBERGER, BSc

is201804

in the
University Course Information Security at St. Pölten University of Applied Sciences

The interior of this work has been composed in LaTeX.

Supervision
Advisor: Dipl.-Ing. Patrick Kochberger, BSc
Assistance: -

St. Pölten, May 30, 2022

(Signature author) (Signature advisor)

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

Ort, Datum *Unterschrift*

Kurzfassung

Verlässliche Kommunikation stellt eine wichtige Kernfunktion für verteilte Systeme dar, dies gilt ebenso für Schadsoftware. Diese Diplomarbeit beschäftigt sich mit der Thematik kommunizierender Schadsoftware, insbesondere den Codeteilen zur Generierung von Domainnamen, welche für die Kontaktaufnahme mit zentralen Kommandostrukturdiensten verwendet werden. Die Herkunft und Verfügbarkeit dieser Codeteile, auch "Domain Generator Algorithm" oder kurz "DGA" genannt, wird festgestellt und deren Quellcode analysiert. Zur weiteren Funktionsanalyse wird eine DGA Basisklasse erstellt und die einzelnen DGAs jeweils als abgeleitete Klasse implementiert. Durch die Verwendung von standardisierten Methoden werden diese DGAs sodann auf ihre korrekte Funktionsweise und Vielfalt der erzeugten Domains geprüft.
In weitere Folge wird ein Framework entwickelt um den Quellcode der so erzeugten DGAs weiter zu untersuchen. So werden implementierungs- wie auch programmiersprachenabhängige Elemente erhoben, ausgewertet und entsprechende Metriken generiert. Dies erfolgt durch die automatisierte Analyse des abstrakten Syntaxbaumes der jeweiligen DGAs und gibt Auskunft über die zu erwartende Komplexität bei der Übersetzung des DGAs in eine weitere Programmiersprache. Es werden sämtliche Ergebnisse und Metriken zusammengefasst und diskutiert um die Vorgänge im Inneren der untersuchten DGAs transparent darzulegen.

Abstract

As reliable communication is a vital function of any distributed system, it is also for malware. This thesis examines the different methods of how malware generates domains to enable communications with its central command structures. Repositories containing such a Domain Generation Algorithm (DGA) are researched and their contents are analyzed and extracted for further use in this document. By creating an abstract class and reimplementing existing DGAs as child classes, their source code is brought into a common form which enables functional analysis to verify correct working order. Output is verified by leveraging industry standard webservices to conduct tests of authenticity for each generated domain, while also factors such as unique domains generated per day are observed, compared and saved for additional processing.

Furthermore, this document provides a framework to analyze and evaluate DGA code in respective to language specific elements of their current implementation. An abstract syntax tree analysis implementation is used to extract metrics from each DGA's source code. With regard to the estimated effort of reimplementation in another programming language this system of metrics may be then used as a reference framework. The combination of the created and discussed functionalities brings transparency to the inner workings of domain generation algorithms.

Contents

1. Introduction

Malware is a lucrative business - in recent years many malware campaigns focused on generating revenue, just like any legal software enterprise. The components of the bad actors product, however, drastically differ from the classic business productivity suite found in modern office environments. Different types of malware aim to accomplish *different goals*, botnets for instance run coordinated Distributed Denial of Service (DDoS) attacks, spyware campaigns exfiltrate data from a victims network or ransomware variants encrypt a business' data and demand various kinds of currency on the promise of decryption. Of course, the transition between these is always fluent.

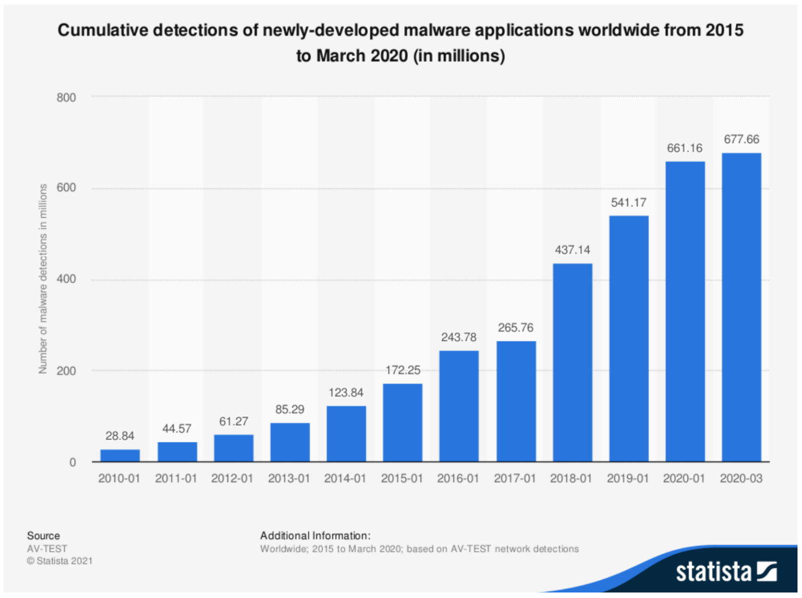

Figure 1.1.: *Malware, a lucrative business:* A statistic [1] by Statista from 2021 shows the detections of newly generated malware variants in the years from 2015-2020. The economy of bad actors is thriving as more and more malicious payloads such as ransomware is deployed via zero-day exploits.

There is one *common denominator* to be observed when examining various modern malware types, the majority needs a facility to communicate with a central instance of authority. These services are so called

”Command and Control (C2) servers” and are vital in issuing orders of action or receiving status updates from deployed malware agents. As with most services reachable on the internet the C2 servers are issued one or more domain names to be contacted at.

It may be considered a bit overly dramatic to classify those generated domain names as an achilleas heel to all sorts of malware, but they undeniably play a significant role [2] in it’s proper operation.

This fact lends great importance to the task of analyzing malware and the domains it generates to operate. In academic or cyber range settings it is imperative to have a toolkit at hand to enable or aid various trainings scenarios. Of what concrete use could a malware extracted, modularized and working DGA be? There are many possible use cases, for example the automated generation of arbitrary training malware samples for reverse engineering in an academic environment. Another purpose would be the creation of an environment to test the capabilities of network security appliances detecting malware traffic. Further, methods to create large training sets for machine learning algorithms [3] aiming to find malware domains can be implemented.

1.1. Problem Description

Interested parties, such as security researchers, are inclined to analyze as many domain generation algorithms as possible for the reasons described above. This however is not a trivial task. There are many factors and circumstances preventing an individual from starting a thorough DGA analysis.

The first obstacle is intrinsic to the matter at hand: a domain generation algorithm is an integral part of an active malware, as such analyzing a domain generation algorithm often bears the risk of being contaminated with live malware. This is something that should be duly avoided in a training or academic scenario. Second, when - or better, if - extraction of a domain generation algorithm from a malware sample is successful it still cannot be analyzed right away. Malware and it’s components are often obscured [4] by the creator to explicitly avoid such scenarios. This creates the significant challenge of reverse engineering complete malware samples and extracting or even recreating the domain generation algorithm in code.

Further, if malware samples are readily available through online sources there are still some precautions to take. The algorithm must be vetted in terms of reliability, a domain generation algorithm that does not generate valid malware domains found in the wild is useless for most training scenarios. This vetting process needs to be standardized to ensure consistent results across the analyzed algorithms. This leads to the next challenge: the implementation process of the extracted domain generation algorithm is completely in the hand of the individual executing that task.

The obtained samples may vary wildly in code complexity, coding language and implementation language specific libraries or concepts. Researchers need to create project-specific standardized methods or frameworks to create comparable domain generation algorithms for analyzation and further use. Re-implementing available domain generation algorithms may thus be considered a complex problem.

1.2. Research Question

As described in section 1.1 there are many pitfalls and problems associated with procuring, analyzing and reimplementing domain generation algorithms. This paper aims to provide solutions to some of the above mentioned challenges and provide mitigation strategies for others. In particular, the following topics shall be examined:

The collection and retrieval of domain generation algorithms: Are there specific online resources which serve as reliable sources of domain generation algorithms? Where may such samples of DGAs be obtained? Are there means an methods to test the functional integrity and reliability of said algorithms? Is there related work that may augment such efforts?

Analyzing domain generation algorithms: What can be done to bring found-in-the-wild domain generation algorithms into a format that increases comparability between them? Which collection of metrics can be designed to render domain generation algorithms fit for efficient reimplementation into other languages?

Evaluating measurements and metrics: Using metrics, may domain generation algorithms be clustered into groups of candidates especially feasible for reimplementation into other languages?

1.3. Contribution

This thesis contributes to the field of academic malware research. Knowledge about the analyzation process of domain generation algorithms is generated and made useable to the reader in multiple forms.
First, a DGA testing suite is introduced to embed DGAs into a common abstraction and check their functional integrity. Secondly the analyzation framework creates metrics to evaluate and classify DGA samples. Both, including the yielded results, may be used to conduct malware research in academic settings more efficiently.

1.4. Thesis Outline

This document is organized in several parts.

Chapter 1 gives a quick introduction and lays out the motivation behind this thesis. Further the research questions are scoped and an outlook of the contribution is provided. Chapter 2 describes certain fundamental concepts and prerequisites to provide basic knowledge about the advanced topics processed later in the document. Chapter 3 lists the related work contributing to the broader scope of the thesis. In chapter 4 the approach to answering the research questions is described and subsequently the results are evaluated in chapter 5. Chapter 6 concludes and sums up the results that were found out during the research work done, an outlook on future work is provided.

2. Basics

This explains basic knowledge required to understand the rest of the work.

2.1. Domain Name System

Each device participating in an Internet Protocol (IP) network is assigned a numerical address. This numerical address is used to establish communication between devices. In most cases these addresses exploit properties which may interrupt or complicate communication attempts, such as:

- the IP address may change [5, p. 12] over time.
- the IP address is hard to remember.
- more than one service [6, p. 44] may be served via a single IP address.
- multiple IP addresses may provide the same service in load balancing [7, p. 44] scenarios.

A better way to identify devices or services on an IP network is the use of domain names [8]. Domain names used on networks for public communication follow the common notation shown in table 2.1

Host Part(s)	**Domain Part**	**Top-level Domain**
www	example	org
gitlab.nwt	fhstp	ac.at

Table 2.1.: *Samples of full qualified domain names:* This table displays the building blocks for common domains used on the public internet. Domains are presented split up in their different parts: *host*, *domain* and *top-level*.

The *Host Part*, *Domain Part* and *Top-level Domain* are referenced as *labels* [9, p. 7] and are combined together via a *"."* to represent the full qualified domain name or FQDN.

The domain name system is a distributed facility to translate domain names into IP addresses. A Domain Name System (DNS) server receives client requests containing a specific domain name and returns the matching IP address from it's database. The single DNS records stored on a DNS server are dynamic and may be altered to reflect IP address changes of devices or services on the network. Further, DNS records may reference a single domain name to many IP addresses, and vice-versa, a single IP address may be referenced by many domain names [9, p. 14].

On the public internet DNS servers form a hierarchy [9, p. 2] to distribute responsibility towards the respecitve domain owners. This ensures that only the designated owners of a domain may change DNS records within this domain or delegate the right to modify sub-domain names to a subordinate entity.

```
georg@AUDEMARS:~$ dig example.com A @8.8.8.8
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26854
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;example.com.                   IN      A

;; ANSWER SECTION:
example.com.            6750    IN      A       93.184.216.34

;; Query time: 17 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
```

Listing 2.1: Querying a designated DNS resolver to receive the IP address of an Full Qualified Domain Name (FQDN) available on the public internet.

A user may query a DNS designated resolver to obtain the IP address assigned to an FQDN. This process is shown in listing 2.1 via the tool *dig*. The command executed orders the *dig* utility to query the public resolver listening on IP ”8.8.8.8” to reply with the IP address of ”example.com”. The resolver then looks up the IP address in it’s internal cache. If the record cannot not be retrieved the resolver queries the authoritative DNS server for the desired FQDN. This process is fully transparent to the user. The DNS server then sends back the result to the client, this may be the IP address received from either cache or query, or, in case of a non-existent FQDN an ”NXDOMAIN” answer, shown in listing 2.2.

```
georg@AUDEMARS:~$ dig does-not-exist.example.com A @8.8.8.8
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 34213
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;does-not-exist.example.com.    IN      A

;; AUTHORITY SECTION:
```

```
example.com.            517     IN      SOA     ns.icann.org. noc.dns.icann.
    org. 2022031411 7200 3600 1209600 3600

;; Query time: 23 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
```

Listing 2.2: Querying a designated DNS resolver to receive the IP address of a non-existent FQDN.

2.2. Command and Control

The general, military, definition of command and control was first established in 1988 by the NATO as follows: "Command and control is the exercise of authority and direction by a properly designated individual over assigned resources in the accomplishment of a common goal" [10]. This concept is transferred to the digital domain as the system known as ”command and control infrastructure” or simply ”C2 infrastructure” to manage malware deployments.

In most scenarios an internet connected server acts in the role of the *properly designated individual* as a central endpoint. The malware clients use this server for multiple purposes during the lifecycle of a malware campaign. When the malware is first deployed it may send commands to the server to ”check-in” and receive additional instructions. These instructions may contain directives to carry out specific commands on the infected system or download additional payloads from the server. Further, the C2 infrastructure may serve as the endpoint to upload stolen data from infected systems. Some malware variants enable their client infrastructure to self-destruct when receiving a certain command from the server, this then concludes the lifecycle of a malware campaign.

2.2.1. Protocols

An early protocol used for command and control by malware campaigns is the ”internet relay chat” [11] service or ”IRC”. This distributed chat platform was originally invented to provide a textual communications platform for humans. Due to it’s simple protocol and scalability it was quickly adopted to serve as the communications channel [12] for many C2 infrastructures. As organic usage of the IRC service declines throughout the years [13] it has become more and more common for businesses to simply drop IRC packets at firewall level to block malicious C2 traffic.

The HTTP protocol [6] is client-server oriented, text based and inherently stateless. These properties make it an ideal candidate to facilitate command and control messages. Further, with HTTP being the top most observed protocol [14] on the public internet it’s a non-trivial problem to reliably detect command and control traffic [15] disguised as benign internet traffic. Thus, HTTP is considered the current state-of-the-art protocol [16] to enable a malware command and control communications infrastructure.

2.3. Domain Generation Algorithm

Resilient communication dependent applications rely on domain name system (DNS) resolution to map the given name of an communications endpoint (DNS entry) to an IP address. This ensures the desired named endpoint is available as long as the DNS entry correctly references the current IP address. If the DNS entry itself is unavailable or incorrectly configured the capability of the application to communicate may be fatally impacted. This scenario naturally also applies to malware applications.

There are many ways in which domains used for communication with a command and control infrastructure may be rendered useless. A common scenario for businesses is to blacklist the domain on firewall level to mitigate the impact malware may have on the IT infrastructure. If domains are used frequently for the distribution of malware or the purpose of enabling a command and control infrastructure they may also show up on public threat feeds [17] [18], which in turn firewalls may ingest to automatically block traffic upon. Further a domain may be seized by the authorities [19] via their respective registrar to redirect any traffic away from the command and control infrastructure, rendering them useless.

For a malware author it is imperative to make the seizure of domain names impossible or impractical and thus preserve the communication between the malware and the command and control infrastructure.

As the name implies a domain generator algorithm creates internet domain names based on certain properties and inputs. Commonly a domain generator algorithm consists of 4 major components. The seed, a list of top-level domains, the algorithm itself and the output.

The seed in a domain generator algorithm provides a similar function as in common pseudo random number generators. The seed is the base from which the output is calculated from. The seed may be provided in many formats, it may be hardcoded into the malware or it may be derived from an external factor such as the current date or the stock value of a publicly traded company.

The top-level domains are selected by the malware author and in most cases hardcoded into the malware. During the domain creation process they are combined with the generated domain name to form a valid routable internet domain name.

The algorithm is considered the main part of the domain generator algorithm. Based on the given input seed one or more domain names are calculated. The number of calculated domains per seed depends on the actual implementation and intention of it's author. For example some algorithms provide a very large number of

Seed	Top-level Domain	Algorithm	Output
MD5 Hash of Datestring	co.cc, cz.cc, .info, .org	Win32/Bamital	37C716B1EF8A468B4301314DCCE830FA.cz.cc
Date values	.online .tech .support	Mirai	vmdefmnsdoj.tech xpknpxmywqsr.online oornsduuwjli.tech
Hardcoded	.com	DirCrypt	pibqzedhzwt.com

Table 2.2.: *Examples of generated domains and their respective inputs [20]:* DGAs generating domains require individual inputs. While the *top-level domain(s)* is usually a hard coded array that is looped through, the seed varies from a simple number to a construct of dates, characters or non-deterministic factors such as twitter posting or stock prices. [21]

domains making it unfeasible for law enforcement to suppress communication by just preregistering all calculated domains.

The output commonly consists of a list of one or more publicly routable internet domains. Depending on implementation the malware tries to initiate communications via one or more of the generated domains. In case of the famous *conficker.c* [22] worm the malware created 50.000 domains per day of which it attempted to use 500 each day.

Table 2.2 provides a small set of domain generation algorithms samples [20] found in the wild along with their respective seed, top-level domains and a subset of the generated domains. Subsequently these, or a subset of these domains are registered by the attacker and used as the endpoint of the command and control infrastructure to enable malware communications. As long as the seed and it's origin remain a secret the domains generated by the malware application seem random or inconspicuously to the observer.

2.4. Abstract Syntax Tree

Computer programs may be written in many different programming languages. The fundamental process of creating an executable file from source code written in an imperative programming language is the process of compiling [23] said source code. Taking Python as an example there are several fundamental steps [24] during compilation.

Tokenization: Source code is parsed and based on language specific rules - or grammar files - transformed into tokens [25]. This can be considered as splitting text into structured chunks of information.

Abstract Syntax Tree creation: Tokens are, based on rules, grouped together into a structured tree consisting of several nodes, representing the source code elements.

Abstract Syntax Tree transformation into a Control Flow Graph: The CFG is a directed graph [24] derived from the abstract syntax tree that models the actual flow of the program.

Creation of bytecode based on the Control Flow Graph: The resulting bytecode is then executed by the CPython virtual machine.

How the binary is executed after compilation is heavily language dependent. In terms of Python this happens via the CPython virtual machine executing the resulting bytecode [25]. The most important step to further understand this document lies within the second step of this process, the creation of the Abstract Syntax Tree (AST).

The AST is a structured abstract representation of the source code, in case of Python it is specified using the Zepyhr [26] language. It consists of hierarchical nodes containing all source code constructs such as expressions, statements, etc. This gives the researcher the ability to inspect, analyze and compare a normalized version of the source code by looping through all nodes and interpreting the results.

```
1 import ast
2
3 # This creates an abstract syntax tree from the given source code.
4
5 myast = ast.parse("print(1 + 2)")
6
7 # This prints the AST. The ast.dump() method retrieves the AST as a formatted
      string from the object
8 print(ast.dump(myast))
```

Listing 2.3: A small Python program generating and outputting an AST of a given piece of code.

Listing 2.3 shows a small Python script generating an AST from a given source code snippet. As seen in line 4, the code prints out the addition of two integers. Further the *ast.dump()* method to output a user readable version of the AST is demonstrated.

In listing 2.4 the corresponding output is shown. The function name - in this case ”print” can easily be identified, as well as the ”add” operation with it’s two inputs as constants, in this case ”1” and ”2”. This is considered the AST representation of the above mentioned printing and addition of the two integers.

```
Module(body=[Expr(value=Call(func=Name(id='print', ctx=Load()), args=[BinOp(
    left=Constant(value=1, kind=None), op=Add(), right=Constant(value=2, kind=
    None))], keywords=[]))], type_ignores=[])
```

Listing 2.4: The output of an AST generated by the dump() method

For the sake of completeness it shall be mentioned that the AST may be modified [27] during compile time to make changes to the final program. While this is a mighty functionality it is well beyond the scope of this document. In summary, the AST can be considered a truly capable tool to analyze Python source code and generate a quantitative analysis of all containing elements comprising the program.

3. Related Work

This chapter provides an overview about work related to this document. The research in the field of malware is highly diverse and this stands also true when looking at the domain of DGAs. First, some of the research in the field of detecting DGA generated domains is given. Secondly papers and scientific projects about the analyzation of DGAs are examined. This chapter includes multiple viewpoints: an external view on the products - the domains - DGAs generate and an internal view on how the DGAs create said domains and which ”ingredients” or seeds are consumed by different DGAs.

3.1. Detection of DGAs

The detection, classification or identification of found-in-the-wild domains as DGA generated is not part of core content of this document. However the verification of domains created by reimplemented DGAs is fundamental to this research, thus a broad overview of related work in this field adds value to the general narrative.

Detecting domains generated by malware to enable C2 communication is a task that already many researchers have picked up. Given the fundamental importance of this topic research is still ongoing and will continue to do so. When looking at the various research efforts it can be determined that detecting new DGAs in the wild is an ongoing game of cat and mouse between malware authors and vendors of information security products.

DGAs tend to become more and more complex while detection mechanism start to utilize modern methods such as machine learning approaches and artificial intelligence. Reviewing research efforts over time documents this trend.

As one of the first research papers about central botnet communication *Detecting Algorithmically Generated Malicious Domain Names* [28] examines the possibility of detection via passive DNS analysis. On the premise of not having a reverse engineered DGA sample available Yadav *et al.* try to algorithmically detect malware generated domains by employing various statistical methods, such as character distribution, Kullback-Leibler divergence or Jaccard index. The methods considered in this paper worked to the 2010 researchers satisfaction, given the rather ordinary methods of domain generation at that time.

2015 In *Detecting DGA malware using NetFlow* [29] the authors try to indirectly detect DGA based malware by analyzing not the generated domains themselves, but certain properties of the network packet stream of individual hosts. Grill *et al.* leverage NetFlow information from networking hardware to identify endpoints trying to resolve an unusual high number of domains in comparison to the actual data transmitted. This paper provides insight to the sheer number of potential malware domains generated and queried each day. The authors succeed in detecting the "Shiz"[30] malware in a corporate network by the techniques described above.

2018 In *A Machine Learning Framework for Studying Domain Generation Algorithm (DGA)-Based Malware: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I* [31] a training set of malware domains is acquired over a period of over six month. These are then fed into the proposed machine-learning framework to create a two-level approach: in the first level test data domains are clustered and classified to detect any DGA created domains. In the second step the machine-learning framework aims to identify the generating DGA for each malware domain. Chin *et al.* provide accuracy numbers of 95.14% in the first level and 92.45% in the second.

3.2. Analysis of DGAs

A slightly similar but yet different discipline is the analyzation of DGAs themselves and the various underlying concepts of actually creating domains. This research did benefit from the various information about DGAs found in the following papers. For example the description about the inner workings of DGAs and their placement into specific functionality clusters based on PRNG input factors.

2017 Vishwakarma [32] describes multiple DGA Families in his thesis *Domain Name Generation Algorithms [online]* [32]. His research assigns DGAs into four different classes based on either deterministic or non-deterministic seed usage and time dependence or independence. Further a concept of C2 infrastructure classification is defined. Vishwakarma clusters them into either *decentralized*, *centralized* or *locomotive* - where only the latter two are DNS dependent and as such relevant for further research.

2016 In *A Comprehensive Measurement Study of Domain Generating Malware* [21] Plohmann *et al.* researched the actual output of DGAs in the wild, given real world seeds and timespans when the specific DGAs were active. Based on this output further metrics per DGA a derived, this also gives insight into domain registration event patterns and the strategy of the malicious actors behind botnets. Being a comprehensive measurement study a very broad range of DGA families are observed and analyzed. One goal of the paper is to make the raw data output from the research available to other researchers. After reaching out to the author, credentials to the DGArchive - Fraunhofer FKIE [20] could be obtained. This software platform enables researchers to run different DGAs on different dates to analyze their output. In this research this functionality is used to test if the DGA is working and their output is valid before commencing with code analysis.

4. Approach

This chapter describes the lifecycle of the DGA code analysis done in this paper. The first section provides a brief quantitative overview of the DGA landscape and then continues with the concept of DGA collection and the criteria on which DGAs were considered for evaluation. The practical implementation section then elaborates on how each DGA's code was brought into a common form to ensure comparability between them. In the main section about DGA analysis an overview about the workflow of testing and examining properties of the DGA's source code used is given. Further an overview about the various metrics used is provided and augmented with samples as deemed necessary.

4.1. DGA Collection

According to industry sources [33] [34] there are approximately 75 DGA families and subfamilies observable in the wild at the time or writing. These DGAs are active in terms of the possibility to have syntactically correct domains generated by them.

- ABCBOT
- ANTAVMU
- BAMITAL
- BANJORI
- BEBLOH/URLZONE
- BEDEP
- BEEBONE
- BIGVIKTOR
- BLACKHOLE
- CCLEANER
- CHINAD
- CONFICKER
- COPPERSTEALER
- COREBOT
- CRYPTOLOCKER
- DIRCRYPT
- DROMEDAN
- DYRE
- EMOTET
- ENVISERV
- FEODO
- FLUBOT
- FOBBER
- G01
- GAMEOVER
- GEODO
- GOZI
- GSPY
- HESPERBOT
- KFOS
- KRAKEN
- LOCKY
- M0YV
- MADMAX
- MATSNU
- MIRAI
- MONEROMINER
- MUROFET
- MYDOOM
- NECRO
- NECURS
- NGIOWEB
- NYMAIM
- OMEXO
- P2P
- PADCRYPT
- PANDABANKER
- PIZD

- PROSLIKEFAN
- PT
- PUSHDO
- PYKSPA
- QADARS
- QAKBOT
- RAMDO
- RAMNIT
- RANBYUS
- ROVNIX
- SHIFU
- SHIOTOB
- SIMDA
- SISRON
- SPHINX
- SUPPOBOX
- SYMMI
- TEMPEDREVE
- TINBA
- TINYNUKE
- TOFSEE
- TORDWM
- UNKNOWNDROPPER
- UNKNOWNJS
- VAWTRAK
- VIDRO
- VIRUT
- VOLATILE
- XSHELLGHOST

For this document, this list of DGAs represents the foundation of the authors DGA procurement effort. For a DGA to be a valid candidate for further analyzation in this work it has to have the following properties:

Availability: The DGA source code is readily available from a free public source. This can either be a software repository, the work of a fellow researcher or derived from write-ups or other analyzation efforts from commercial security vendors.

Condition: The acquired source code is written in a high-level programming language.

Functionality: The DGA cannot make use of external non-deterministic sources of randomness which cannot be reconstructed within a closed lab environment. This includes sources like stock prices or information from public short messaging services.

4.1.1. Sources of DGAs

Researching the given list of possible DGAs leads to certain source code repositories. Reviewing this sources reveals that the programming language *python* is prevalent when reimplementing DGAs for further examination or analysis. After examination, three public git repositories remained as best choice to receive DGAs, while the source code for the implementations used in the private DGArchive [20] is not shared by the author. However, the provided Application Programming Interface (API) to test the validity of DGA generated domains is incredibly useful to any researcher in this field.

These three repositories provide approximately 42 unique DGA algorithms at the time of writing:

- *dga-collection* by pchaigno
- *DGA* by andrewaeva
- *domain_generation_algorithms* by baderj

A more detailed overview of the considered repositories is provided in appendix A, table A.1, the list of unique DGAs is shown below.

- BANJORI
- BAZARBACKDOOR
- CHINAD
- COREBOT
- DIRCRYPT
- DNSCHANGER
- FOBBER
- FOSNIW
- GAMEOVERZEUS
- GOZI
- KRAKEN
- LOCKY
- MATSNU
- MONEROMINER
- MUROFET
- MYDOOM
- NECURS
- NEWGOZ
- NYMAIM
- NYMAIM2
- PADCRYPT
- PITOU
- PIZD
- PROSLIKEFAN
- PUSHDO
- PYKSPA
- QADARS
- QAKBOT
- QSNATCH
- RAMDO
- RAMNIT
- RANBYUS
- RECONYC
- SHIOTOB
- SIMDA
- SISRON
- SUPPOBOX
- SYMMI
- TEMPEDREVE
- TINBA
- VAWTRAK
- ZLOADER

4.1.2. Sampling DGAs

Out of all acquired DGAs a subset is sampled for reimplementation, testing and analysis. All DGAs sampled are reimplemented in Python as a child class of an DGA abstract class, which is further explained in section 4.2.1. Implementing DGAs as a child class of a common abstract class increases the comparability between all given samples. Since the basic building blocks and output functions of each DGA are already implemented within the abstract class this does not count towards the code complexity of each DGA. To further harmonize the internal code structures of the examined DGAs, all static external sources of input, such as wordlists or top-level domains lists, are brought into the DGA child class itself. Finally, all DGAs which are successfully implemented as a child class are equipped with input parameter defaults and the same output variables. This creates the opportunity to implement a common validity testing mechanism with a high grade of automation. This is described in detail in section 4.3.2.

4.2. Practical Implementation

This section describes the process of practical implementation. First an overview of the abstract class and it's properties and methods is given. Then the skeleton code for the child class is shown and the major parts are elaborated upon. Finally, to ensure consistent code quality, the used coding standards enforcement methods are described.

4.2.1. An Abstract DGA Class

The main goal of the implementation of an abstract DGA class is to provide a common interaction interface for all analyzed DGAs. The abstract class also has certain functionalities built in to reduce the amount of redundant code when reimplementing DGAs. Yet, the abstract class is kept as simple and as extendable as possible to provide sufficient flexibility for most DGA implementations.

```python
"""This is the module for an abstract base class DGA."""
from abc import ABC, abstractmethod


class DGA(ABC):
    """An abstract base class for a DGA."""

    def __init__(self):
        """
        State and domain history attributes are initialized here.

        Attributes
        ----------
        _state:
            A dictionary to keep the current state of the running DGA
            Field "generateddomains": this should be increased with every
                domain generated
        domainhistory:
            A list keeping all domains generated during the lifetime of the
                object.
        """
        self._state = dict()
        self._state["generateddomains"] = 0
        self.domainhistory = list()

    def get_nextdomain(self) -> str:
        """Return the next generated domain.

        This is implemented to do state & stats "housekeeping".
        It calls "generatedomain"
        """
        self.domainhistory.append(self.generate_domain())
        self._state["generateddomains"] += 1
        return self.domainhistory[-1]

    @abstractmethod
    def generate_domain(self) -> str:
        """This method shall be implemented by the inheriting class."""
        pass
```

Listing 4.1: The abstract base class for all DGAs analyzed in this document

Listing 4.1 shows the implementation of the abstract class in Python. This class inherits it's properties from the ABC abstract base class object to provide the functionality of defining abstract methods, which are implemented via the respective child DGA classes.

The main building blocks of this class are as follows:

__init__(self) method: This method creates the structure for certain "housekeeping" tasks during the lifetime of an instantiated object. This includes providing a list object to save all generated domains into, additionally a dictionary for variables such as the total number of domains generated is initialized. This may be extend by the child class DGA implemented.

get_nextdomain(self) method: This is the main method to interact with the implemented DGA. Modules utilizing this class call this function to retrieve the next domain generated by the DGA. The counter of created domains is automatically incremented and the generated domain is returned as a string. This method wraps the actual implementation of the DGA.

generate_domain(self) method: Declared as an abstract method, it is the responsibility of the inheriting DGA class to provide an implementation. This method should return one new domain on each call.

Not included in the design of the abstract class are various input parameters. Figure 4.1 shows that DGAs are different in their parameter requirements, some just generate domains without any input, some require a starting seed or a date, some require both. Again, it is in the responsibility of the inheriting DGA class to enhance the *__init__()* method with all parameters necessary for correct operation. To simplify automated testing it is advisable to equip all additional input parameters with default values. In case of a date it is logical to use the current date, in case of a seed it makes sense to use a seed that is also observed in the wild, if available.

4.2.2. A DGA Child Class

When a DGA is acquired from one of the sources named above it is rewritten to fit into the schema the abstract DGA class provides. In some cases this means that most of the source code may be reused, in other cases a lot of functionality has to be rewritten to fit to the abstract DGA class. Depending on the required input parameters of the DGA a snippet of skeleton code is used to simplify implementation. An example of this skeleton for a DGA requiring a seed and a date as input parameter is shown in listing 4.2. In this case a date and a seed is required.

```python
class DGAEmptySeedDate(DGA):
    """Implementation of the "EmptySeedDate" DGA."""

```

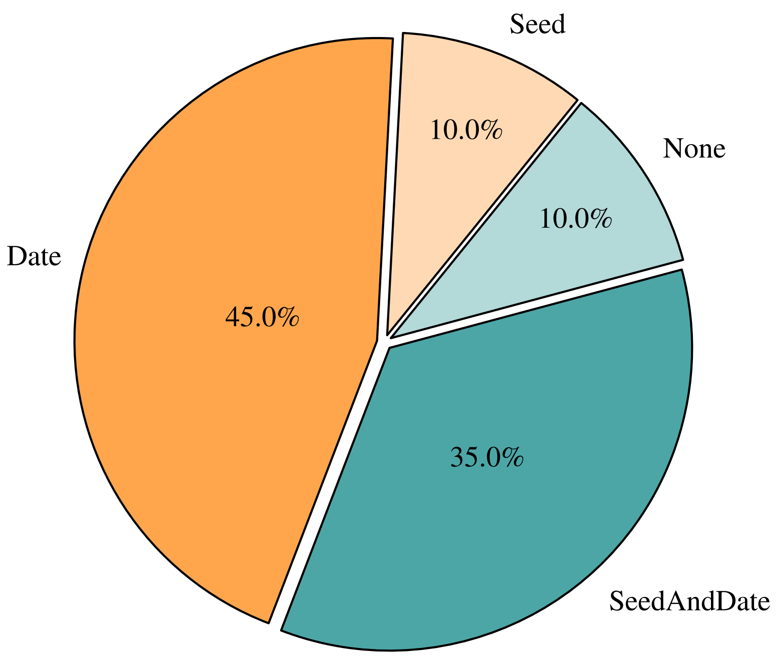

Figure 4.1.: *Required Parameters over all analyzed DGAs:* This pie chart shows the distribution of the required input parameters displayed as percentage over all DGAs. If a date is accepted as a seed the parameter it is counted as seed. Date parameters are only counted in the event of contextual date dependence.

```python
    import datetime

    def __init__(self, seed: any = "", date: datetime.datetime = datetime.
        datetime.now()):
        """Initializing the DGA.

        This DGA is using a seed and a date as parameter.

        Parameters
        ----------
        date : datetime
            The date of which the domains are generated on, default value: now
                ()
        seed : any
            The seed provides the source of randomness, default value: ''
        """
        super().__init__()
        self._seed = seed
        self._date = date

    def generate_domain(self):
        """Generates a new domain name."""
        self._state["lastdomain"] = "Seed: " + self._seed + " Date: " + str(
            self._date)
        return(self._state["lastdomain"])
```

Listing 4.2: Sample of one variant of the skeleton code DGAs may be implemented as.

For completeness all variations of the skeleton code are provided in appendix B, listing B.1.

The main building blocks of this class are as follows:

__init__(self) method: This method first calls the *__init__* method of the parent class via *super().__init__()*. Next the date and the seed for which the domains shall be generated on are copied into class wide available variables. The default value for the date is already set to *datetime.now()* which represents the current date and time. The seed in this example is an empty string, this should be replaced with a value found in the wild at implementation time, if available.

generate_domain(self) method: Now the abstract method from the parent class is implemented by this code. This is the main DGA method that will generate the domain and create or update the state dictionary entry for *lastdomain* of the DGA class. Finally this dictionary entry containing the respective next domain as string is returned to the *get_nextdomain(self)* method.

Adherence to this implementation guidelines yields a common interface for an automated validity testing of reimplemented DGAs. By following this guideline research output is scaled simply by correctly reimplementing DGAs and running the standardized tests, described in section 4.3.2, with minimal administrative overhead.

The list of DGAs, reimplemented as child class and ready for automated testing and analyzation, is shown below.

- DGABANJORI
- DGACHINAD
- DGACOREBOT
- DGADIRCRYPT
- DGAFOBBERV1
- DGAFOBBERV2
- DGAMONERODOWNLOADER
- DGAMUROFETV1
- DGAMUROFETV2
- DGAMUROFETV3
- DGANECURS
- DGANEWGOZ
- DGAPADCRYPT22861
- DGAPADCRYPT22970
- DGAPROSLIKEFAN
- DGAPYKSPA
- DGAQADARS
- DGAQSNATCH
- DGARAMDO
- DGARANBYUS

4.2.3. Coding Standards Enforcement

Code written for this document adheres to the coding standard defined in the *Python Enhancement Proposal 8* [35]. This represents the standard style guide convention for writing Python code when working with the standard libraries. Research proves [36] that enforcing coding guidelines in a project reduce error proneness and increases legibility and consistent code quality.

The following source code parts adhere to the defined standards with one exception: logging of the error code *D401* is disabled. The mechanic throwing this error code is a contextual analysis of the docstring itself in terms of language tone used.

- The abstract DGA class
- All DGA child classes
- All code written for analysis and evaluation of the DGAs

Additionally the Python docstrings inside the source code files adhere to the *numpy* [37] documentation standard. Enforcement of these coding guidelines is achieved via *flake8* [38], a library that wraps multiple tools for static code analysis into one useable package.

4.3. DGA Analysis

This section handles the analysis of the implemented DGAs. It is basically divided in three parts, first the methods and the analyzation workflow is demonstrated. Second each phase of the analyzation workflow is presented in detail to further describe the used methods. The chapter concludes with an overview over the generated metrics and their origin in code.

4.3.1. Analyzation Method

After a new DGA is reimplemented in Python as a child class of the defined DGA parent class, the analyzation phase begins. First, the DGA is tested for functionality. This includes checking validity of the generated domains and the variance of the DGAs output, i.e. the number of domains that may be generated within a single day. Both metrics are recorded for use later in the analyzation phase.

If the DGA proves valid an abstract syntax tree is created and iterated over. Throughout this iteration the occurrence of certain nodes of interest are recorded. The types of these nodes are the inputs for metrics like complexity, parameter dependence or imported modules.

Modules provides an overview of the imported modules the DGAs make use of.

Parameters shows concepts on how deterministic randomness is seeded into the DGAs.

Implementation Specifics are language dependent structures that need to be considered for successful reimplementation.

Complexity measures the number of general occurrences of well known structural elements such as loops and branches.

Table C.1 provides an overview identifying each AST node class corresponding to their respective metric. A detailed overview of which AST node metric is recorded for which purpose is given in section 4.3.6 to section 4.3.9. The results are saved for the next phase of analyzation.

In the last phase the inputs of the two preceding phases are merged and data is generated to successfully analyze, compare and evaluate the given DGAs. Multiple outputs, such as CSV files, are generated for concrete comparison of DGAs or quantitative analysis.

4.3.2. Analyzation Workflow

This section provides an end to end view of how the analyzation phases work together, from the ingestion of a DGA to the conversion of the processed data into usable metrics. First, a workflow of the three phases of analysis is provided, then the recorded metrics during these phases are enumerated and described further.

4.3.3. Phase 1: Validity and Variance

Figure 4.2 provides a visual overview of this phase, further each step shown is explained in more detail below.

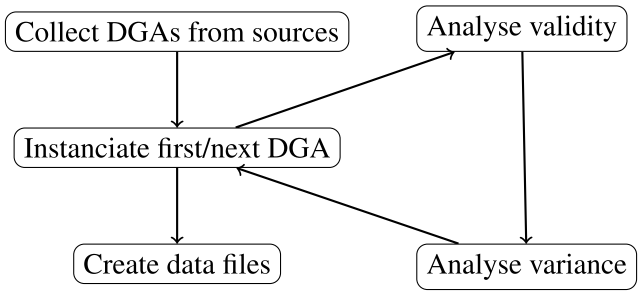

Figure 4.2.: *Diagram of Phase 1 in the analysis workflow:* This diagram shows the ”validity and variance” phase of the analyzation workflow. The main steps are the collection and instantiation of DGAs and the loop of collecting the data from each DGA. The phase finishes with the output of the collected data.

Collect DGAs from sources All DGAs are implemented as classes within their respective module files. These module files are enumerated via the Python *glob* module and then imported at runtime via the *importlib* module.

Instantiate first/next DGA Since all DGAs are subclasses of the DGA abstract class, a Python list may be created by executing the *__subclasses__()* method of the DGA abstract class. The resulting list may then be iterated over and the next two steps executed for each DGA.

Analyze validity This step checks if the domains created by the DGA are in fact ”valid”. If a domain can be found in the standardized DGA Archive [20] it can be considered as found in the wild and thus valid. Checking a domain against this archive happens via sending the generated domain name via a Secure Hypertext Transfer Protocol (HTTPS) Representational State Transfer (REST) request to the authenticated DGA Archive webservice. If successful the response either contains the known name of the DGA, or, if unsuccessful, an error message. The DGA parameters used are always the implemented default parameters for the seed and the current days date. The process is repeated arbitrarily often for each DGA, the exact number is defined in a global variable with the testing source code. The results for all queried domains are saved in a local cache file for further lookups to reduce strain on the webservice and expedite the process.

Analyze variance Each DGA that scores 100% valid domains in the previous step is analyzed for domain variance. This test runs each instantiated DGA a 10.000 times and adds each resulting domain to a Python set. At each power of 10 the length of this set is queried, since Python sets may not contain duplicates the length of the set precisely give the amount of uniquely generated domains up to this point.

Create data files As last step in the first phase the data files of the previous steps are created for further use. The files are dumped Python dictionaries structured in JavaScript Object Notation (JSON).

4.3.4. Phase 2: Collection of Metrics via AST

In this phase the AST tree of each DGA implementation is collected and the metrics are extracted. Figure 4.3 provides a visual overview, further each step shown is explained in more detail below.

Collect DGAs from sources All DGAs are implemented as classes within their respective module files. These module files are enumerated via the Python *glob* module and then imported at runtime via the *importlib* module. This step is similar to the first step in phase 1.

Create AST from first/next DGA Again, a list is created by calling the *__subclasses__()* method in the DGA abstract class object. The resulting list is iterated over an for each subclass DGA the corresponding source file is found via the *__module__* property. This source file is loaded and passed to the AST parser via *ast.parse()*. The resulting AST is passed to the custom AST analyzer class executing the

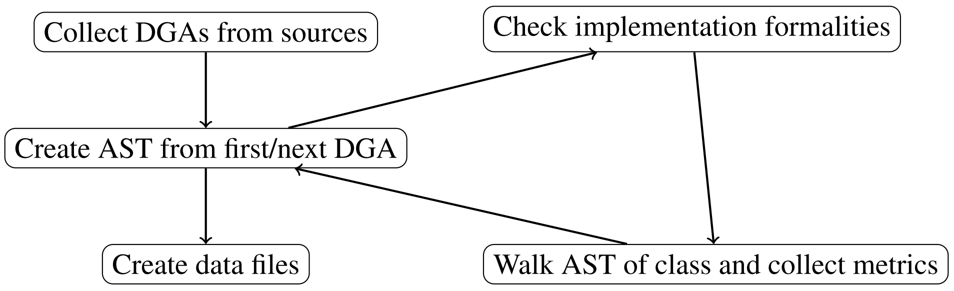
Collect DGAs from sources
Check implementation formalities
Create AST from first/next DGA
Create data files
Walk AST of class and collect metrics

Figure 4.3.: *Diagram of Phase 2 in the analysis workflow:* This diagram shows the "AST analyzation" phase of the analyzation workflow. The main steps are the collection of the implemented DGAs and the subsequent creation of an AST for each. Then the AST is iterated over and its nodes examined. The phase finishes with the output of the collected data.

next two steps.

Check implementation formalities To analyze an AST of a DGA the implementation has to abide to two basic guidelines. First, only one DGA class per module file is allowed. This makes templating new DGAs more reliable and simplifies the automated instantiation executed in phase 1. Second, all DGA module and class names must be unique to ensure coherent metrics without either double counts or omitted classes. If both conditions are met the next step is started, if an error occurs a warning is generated and the next DGA source file is examined.

Walk AST of class and collect metrics The actual inspection of the AST happens in two steps, first the class of the DGA is looked up inside the AST by instantiating a custom *ast.NodeVisitor* class and starting a *visit_ClassDef()* method. Once this custom AST visitor has arrived at the one and only class node of the source file the metric dictionary is initialized and step 2 is initiated: all nodes below the class node are iterated over via the *ast.walk()* method. This is the step where all metrics are collected and written into a single structured dictionary. A detailed overview of which node metric is recorded for which purpose is given in section 4.3.6 to section 4.3.9.

Create data files Same as during the first phase the data file of the previous step is created for further use. The file is a dumped Python dictionary structured in *JavaScript Object Notation* - JSON.

4.3.5. Phase 3: Enrichment and Creation of Data Files

Figure 4.4 provides a visual overview of this phase, further each step shown is explained in more detail below.

Ingest data files The files created in the previous phases are loaded into Python dictionary objects. If

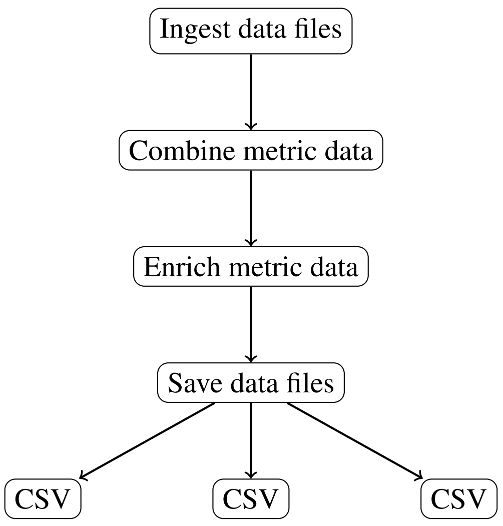

Figure 4.4.: *Diagram of Phase 3 in the analysis workflow:* This diagram shows the "Enrichment and creation of data files" phase of the analyzation workflow. The main steps are the ingestions & combination of the JSON data files from the previous phases and the enrichment of the data. The phase finishes with the output of the final CSV data files for use in the evaluation.

the main data file does not exist a critical error is thrown and the phase is terminated. In case of one or more missing supplemental datafiles from phase 1 a warning is logged into the central metric dictionary, the ingest will continue nonetheless.

Combine metric data Metric data from all ingested files is combined into the central metric dictionary. The properties generated in phase 1 which are added in this step are *DomainsValidParcent* and *DomainsUniqueDay*. All missing properties from non-existent supplemental datafiles are further substituted with the value "-1".

Enrich metric data This step generates additional metrics from already collected data by combining or summing up various datapoints. This includes but is not limited to the following metrics:

- *global_import_counts* - a dictionary containing the import count of each distinct module is created by looping through all individual DGAs and their respective imports.
- *PythonicComplexity* - each DGA is assigned a value representing how often Python language specific elements are used. This is further described in section 4.3.8.
- *LanguageAgnosticComplexity* - in contrast to the previous metric this shows the number of language "agnostic" structural elements. These are elements common in higher programming languages, for example loops, functions, operation, decisions or comparisons.
- *global_dga_params* - a dictionary counting the required input parameters and their respective

kind over all DGAs.

Save data files This final step creates various dataframes from the central metric dictionary. These dataframes are then processed via the *pandas* [39] library, this enables the quick and efficient creation of multiple well-formatted, sorted CSV files for various evaluation use cases. This data is directly referenced in throughout this document, especially in chapter 5.

4.3.6. Metrics: Modules

In software development it is common practice to import functionality into an application that is already implemented by a 3rd party. This practice saves time, effort and reduces redundancy. In case of standard libraries it also brings the convenience of real world proven code into the own application. This is no different for DGAs, therefore the following metrics about the analyzed DGAs are collected.

Imported Modules For each DGA the count and the module names of the individual imports are recorded. The effort of reimplementation into another language is directly proportional dependent to this metric - everything imported must also be imported in the language the DGA is reimplemented in.

Most Used Modules This metric provides an overview of the most imported modules counted over all DGAs analyzed. With the help of this metric, modules especially important in DGA development, may be identified.

```
if (isinstance(node, ast.Import)):
    self.stats["dga"][dga_class]["ImportCount"] = self.stats["dga"][dga_class
       ].get("ImportCount", 0) + 1
    for alias in node.names:
        self.stats["dga"][dga_class]["Imports"].append(alias.name)

if (isinstance(node, ast.ImportFrom)):
    self.stats["dga"][dga_class]["ImportCount"] = self.stats["dga"][dga_class
       ].get("ImportCount", 0) + 1
    for alias in node.names:
        self.stats["dga"][dga_class]["Imports"].append("from " + node.module +
            " import " + alias.name)
```

Listing 4.3: Sample code for collecting module metrics via AST

4.3.7. Metrics: Parameters

Parameters of DGA can be found on two different occasions. One being external input parameters which may be set when the DGA is instantiated, the other kind are internal parameters which may vary when implementing subfamilies of the same DGA but from, for example, different malware campaigns.

External Parameters For each DGA the following external input parameters are enumerated: *Seed*, *Date*. Both are extracted via AST analysis of the *__init__()* method of the class node. These two parameters are defined via the DGA child class templates and thus may be extracted with good confidence.

Hardcoded Parameters Discovering hard-coded parameters is non-trivial in code and heavily dependent on the implementation. There are AST node types which hint at hard coded parameters for domain creation - such as *ast.Constant* which may represent a string, depending on its contents. The classes *ast.Tuple*, *ast.List* and *ast.Set* may also hold parameter values such as a list of top-level or second level domains.

```
if isinstance(node, ast.FunctionDef) and (node.name == "__init__"):
    for arg in node.args.args:
        if (arg.arg == "seed"):
            self.stats["dga"][dga_class]["ParamSeed"] = 1
        if (arg.arg == "date"):
            self.stats["dga"][dga_class]["ParamDate"] = 1
```

Listing 4.4: Sample code for collecting external parameters of an implemented DGA via AST

As described, obtaining specific AST nodes for hardcoded parameters depends on generic nodes, a code sample is therefor omitted.

4.3.8. Metrics: Implementation Specifics

As described in section 4.1.1 most DGAs discovered for further analysis are implemented in Python. The following metrics show how dependent a DGA is on specific types or constructs found in the Python programming language. This section focuses on features not available out of the box in classic higher languages such as C.

Data Structures This metric includes language constructs which are very convenient to use in Python, but may be challenging in other languages. Python provides in-memory data structures with little to no requirements on the users part when it comes to memory allocation or garbage collection. The structures considered as such in this document are: *ast.Tuple*, *ast.List*, *ast.Set* and *ast.Dict*.

```
if isinstance(node, ast.List) or isinstance(node, ast.Tuple) or
    isinstance(node, ast.Set) or isinstance(node, ast.Dict):
```

Listing 4.5: Sample code for collecting the occurrence of Python specific in-memory data structures implemented DGA via AST

Formatting Elements Python provides a mechanic called *literal string interpolation*[40], more commonly known as *f-strings*. This allows the user to easily concatenate multiple variables into a single output string while applying transformation such as case modification or formatting of non-string variables. This is a powerful tool, however very specific to Python, and may thus be considered non-trivial to reimplement in another programming language.

```
if isinstance(node, ast.FormattedValue) or isinstance(node, ast.JoinedStr
    ):
```

Listing 4.6: Sample code for occurrences of literal string interpolation in an implemented DGA via AST

4.3.9. Metrics: Complexity

The overall complexity is a quantitative analysis of the source code given. These metrics provide an overview of the DGA in terms of code volume. Additionally the number of AST nodes is counted an put into context.

Lines of code The metric counts the total lines of code used to implement the DGA. This is counted from the first line of the implemented child class to the last line. All code or comments outside this area is omited. In general lines of code provide an overview about the expected complexity, variations due to coding style or implementation specifics are to be expected.

Average line complexity The average line complexity is a calculated value to put the previously measured line of code metric into context. The calculation is the amount of all AST nodes comprising the DGA divided by the number of lines of code. This metric then shows how many AST nodes are used on average in each line of code, calculation follows equation (1):

$$AverageLineComplexity = \frac{NumberofNodes}{LinesofCode} * 100 \tag{1}$$

Elements

This section provides a quantitative overview about different elements spanning a single line of code. This covers the existence of literals and the assignment of values to certain describes datatypes.

Nodes This metric counts each node in the AST tree of a DGA child class.

Constants In Python a node of class *constant* may represent different elements, for example: number, string, byte, ellipsis or namedconstant. This metric helps to identify the number of constants, variables or, more generally, atomic data types needed in other programming languages to reimplement an analogue functionality.

Assignments There are three subtypes of assignments in Python.

- *ast.Assign* This node class represents a common assignment of a value to one or more targets. The type is implied at runtime.
- *ast.AnnAssign* is detected when a node is assigned a certain class, such as ”int” or ”str”. Only one target can be assigned a class at once.
- *ast.AugAssign* represents an augmented assignment. For example an integer ”a” may be increased by 1 using this syntax: *a += 1*

Fundamental Control Structures

Loops In Python two different styles of loops are considered: the *for* loop and the *while* loop. Both add to the count of this loop count metric.

Branches Decisions are implemented in different AST node classes. The common *if statement* is a basic control structure represented by a single AST node class. In Python the *elif statement* is not represented by a unique AST node class but by nesting an *if statement* into the *orelse clause* of the surrounding *if statement*. Further, Python allows for expressions such as *a if b else c*, this is also considered a branch and is counted towards to final number of this metric.

Functions This metric counts all function definitions within the AST of the implemented DGA child class. A higher function count may hint at a more complex source code implementation.

Lambda Functions Anonymous functions which fit inside a single AST node are known as *lambda functions* in Python. While many programming languages support such anonymous functions, they are still missing in the standard C programming language. Thus this metric is helpful in estimating the reimplementation effort.

Operations

This section covers the four different types of operations which may occur with one or two operands. Further, table 4.1 provides an overview of the actual operations counted towards each type.

Binary Operations Additions, Subtractions, Multiplications, Divisions, Bitwise operations, Modulus, etc.

Boolean Operations AND, OR

Unary Operation Unary operations work on a single operand, prepending an integer with a - is commonly observed.

Comparisons Comparators check for value equality, reference equality or value difference of the operands.

Operation Type Metric	**Operation**
Binary (ast.BinOp)	ast.Add, ast.Sub, ast.Mult, ast.Div, ast.FloorDiv, ast.Mod, ast.Pow, ast.LShift, ast.RShift, ast.BitOr, ast.BitXor, ast.BitAnd, ast.MatMult
Boolean (ast.BoolOp)	ast.And, ast.Or
Unary (ast.UnaryOp)	ast.UAdd, ast.USub, ast.Not, ast.Invert
Comparison (ast.Compare)	ast.Eq, ast.NotEq, ast.Lt, ast.LtE, ast.Gt, ast.GtE, ast.Is, ast.IsNot, ast.In, ast.NotIn

Table 4.1.: *Operation Types in Python:* This table provides an overview of the four analyzed operation types and the operations belonging to each type. The metrics generated for each type are the sum of all occurrences of the individual operations.

5. Evaluation

This evaluation aims to create results from the metrics collected in section 4.3. As described in section 1.2 this document aims to identify DGAs or groups of DGAs which may be suitable candidates for reimplementation in other programming languages while retaining an efficient administrative effort. Further, DGAs shall be evaluated based on various key facts such as variance in domains generated or the use of mandatory input parameters. These may serve as so called ”knock-out” criteria to enable or deny reimplementation based on project use case.

To satisfy these goals the evaluation is done from three different perspectives, each covering unique aspects in terms of complexion of the given DGAs and their analyzed properties.

- *3rd Party Dependency:* DGAs are often implemented referencing third party libraries to ease implementation and reduce coding effort. This however may have detrimental effects on the ability to efficiently reimplement a DGA into another language.
- *Proprietary or Complex Code Elements:* As described in section 4.1.1 the most prevalent implementation language of available DGAs is *Python*. This perspective evaluates the implications on the to-be-expected DGA complexity based on language specific metrics.
- *General Classification:* In contrast to the more code centered approach of the preceding two perspectives the focus here lies on the functional criteria of the analyzed DGAs. This perspective works on the premise that when a DGA does not fulfill certain criteria it may be unsuitable for further use, thus eliminating the need for reimplemenation altogether.

The following three sections describe each of the selected perspectives in more detail, providing tables and visual aides to create visibility and insights into the evaluation process of the analyzed DGAs. Each section starts with a general reasoning why evaluation of the respective metrics was chosen and then shows outcomes and statistics generated from the analyzation metrics. They conclude with statements about the results and implications of the data generated.

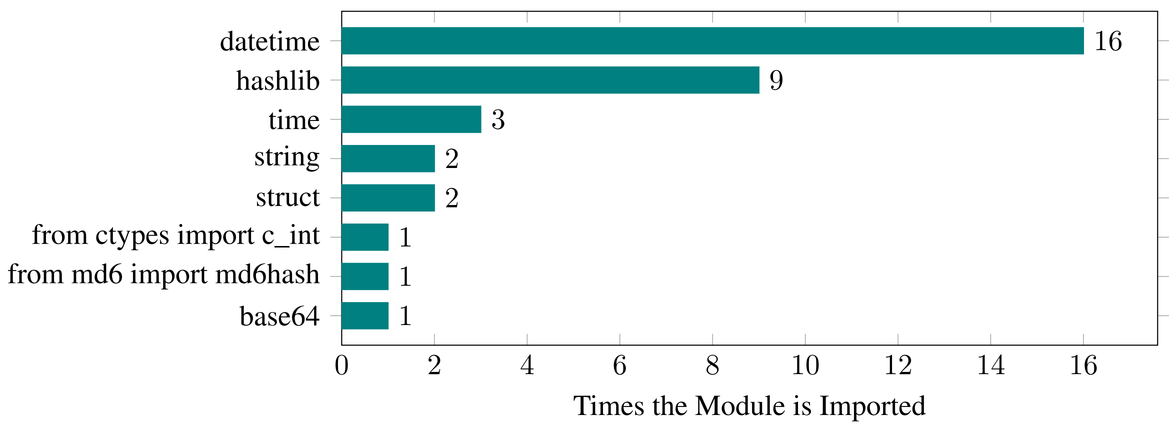

Figure 5.1.: *Most Imported Modules:* Some DGAs use standard Python modules within their reference implementation, this table shows which module are used most often to provide functionality.

5.1. 3rd Party Dependency

Dependency on third party libraries is common in the world of programming, especially in the Python ecosystem where more than 200 modules [41] are readily available - within the standard libraries alone - to provide many commonly used functionalities. To successfully port a Python script into another programming language the used methods of the library also needs to be recreated in some way. This may happen either via manual recreation of the functionality or via available libraries in the target language providing similar functionality. Figure 5.1 shows the top most imported modules in all DGAs analyzed.

In contrast to observing the imported modules themselves, the analyzed DGAs are evaluated based on the total number of imported modules in each. By following the established logic about reimplementation effort being dependent on the number of 3rd party dependency, DGAs with a low module import count may be more suitable for reimplementation with effective administrative effort. Table 5.1 shows how many modules are imported by which DGA in ascending order.

Based on the combined metrics of fig. 5.1 and table 5.1 a third result may be observed. There is a number of DGAs which only import modules from the pool of the top-most referenced modules. This may identify the possibility of a so called "quick-win": when reimplementing the referenced modules of one of the DGAs in this list the prerequisites for, in best case, many other DGAs may also already be fulfilled. Table 5.2 provides this information. In this example all DGAs only importing from a common pool of top three modules are shown in the table. It shall be noted that in case of an ex aequo placement in the top-three pool all possible modules are contained in this table. This can be observed by cross referencing fig. 5.1.

Summary of findings and results of the *3rd Party Dependency* perspective:

- The overview of the number of modules imported is an invaluable tool for further evaluation.
- DGAs with no imported modules are identified and evaluated as feasible candidates for reimplementation.
- DGAs which import only modules commonly also leveraged by other DGAs may create synergy effects for the reimplementation of other DGAs.

5.2. Proprietary or Complex Code Elements

In this section the source code metrics of the analyzed DGAs are examined. Python is a scripting language where many complex processes are abstracted away from the user, this enables rapid development of various complex functionalities with relatively low effort. However, if code functionality written in such manner is to be translated into another language an increased difficulty level when recreating said functionalities is to be expected. Further, Python provides easily approachable data structures to work with, here the same principle applies. If a DGA makes use of many different Python specific code elements, again a heightened reimplementation effort is to be expected.
At first the code elements which are very typical to Python and contribute to the aforementioned difficulties are collected as the number of ”Python Specific Elements”. This is done by applying formula (2) to all metrics collected for each DGA.

$$\begin{aligned} PythonSpecificElements = & ImportCount + LambdaFunctions + PythonFstring + \\ & PythonStructureList + PythonStructureTuple + PythonStructureSet + \\ & PythonStructureDict \end{aligned} \tag{2}$$

The second step is to collect code and structure elements common in most higher, touring complete programming languages. This number of of ”Language Agnostic Elements” is also collected from all metrics of each DGA. After applying formula (3) the two numbers may be put into context for each DGA, providing a hint whether there might be correlation between the two.

$$\begin{aligned} LanguageAgnosticElements = & Functions + Loops + BinaryOperations + \\ & BooleanOperations + UnaryOperations + Comparisons + \\ & Decisions \end{aligned} \tag{3}$$

It shall be noted that section 4.3 provides insights on how the various components of both formulas are collected while in appendix C all code collected metrics are documented.

When these two numbers are combined for each DGA a plot may be drawn to visualize the correlation between the ”Python Specific Elements” and the ”Language Agnostic Elements”. This is provided in fig. 5.2. The linear regression line shows that as the values of the ”Python Specific Elements” increase the ”Language Agnostic Elements” decrease slightly. To augment the data provided in fig. 5.2, table 5.3 shows the numbers behind the figure. This hints at a rather mild correlation, as such it cannot be said with absolute certainty that a higher count of ”Language Agnostic Elements” prevents a DGA from also leveraging code structures specific to the Python language.

Thus, another set of source code specific metrics shall be evaluated to identify DGAs with reduced administrative reimplementation effort.

The next step is to quantitatively evaluate the complexity by looking at the number of lines of code for each DGA in relation to the average complexity of each line of code within each DGA. The needed metric ”Average Line Complexity” is already well known and described in this document in section 4.3.9, the metric ”Lines Of Code” is collected during phase 2 of the analyzation process in section 4.3.4. Thus no further calculations need to be made to further evaluate these dimensions. Figure 5.3 shows the plot for each analyzed DGA.

This time the very weak correlation between the two values is well expected, the complexity of each line of code is in complete dependence on how the user chose to implement the required functionality. However, a DGA with a very high number of lines of code *and* a high value of the average line complexity can be classified as a rather complex and hard to reimplement DGA.

This is exactly the opposite of what we aim to identify, as such it can be conversely inferred that DGAs with the following properties may fit our search better:

- a *high* number of lines of code and a *low* value of average line complexity
- a *low* number of lines of code and a *high* value of average line complexity

Additionally, the obvious case of a low number of lines of code and a low value of average line complexity shall be considered as well.

Table 5.4 shows data of all three cases, sorted by lines of code. While the most useful DGAs cannot be identified at the first glance it is made obvious that there is a high variance in the number of lines of code and the value of average line complexity. This indicates that there is great potential of finding good DGA candidates for reimplementation.

To yield the final results of the complexity and proprietarity evaluation the previous two metrics shall be

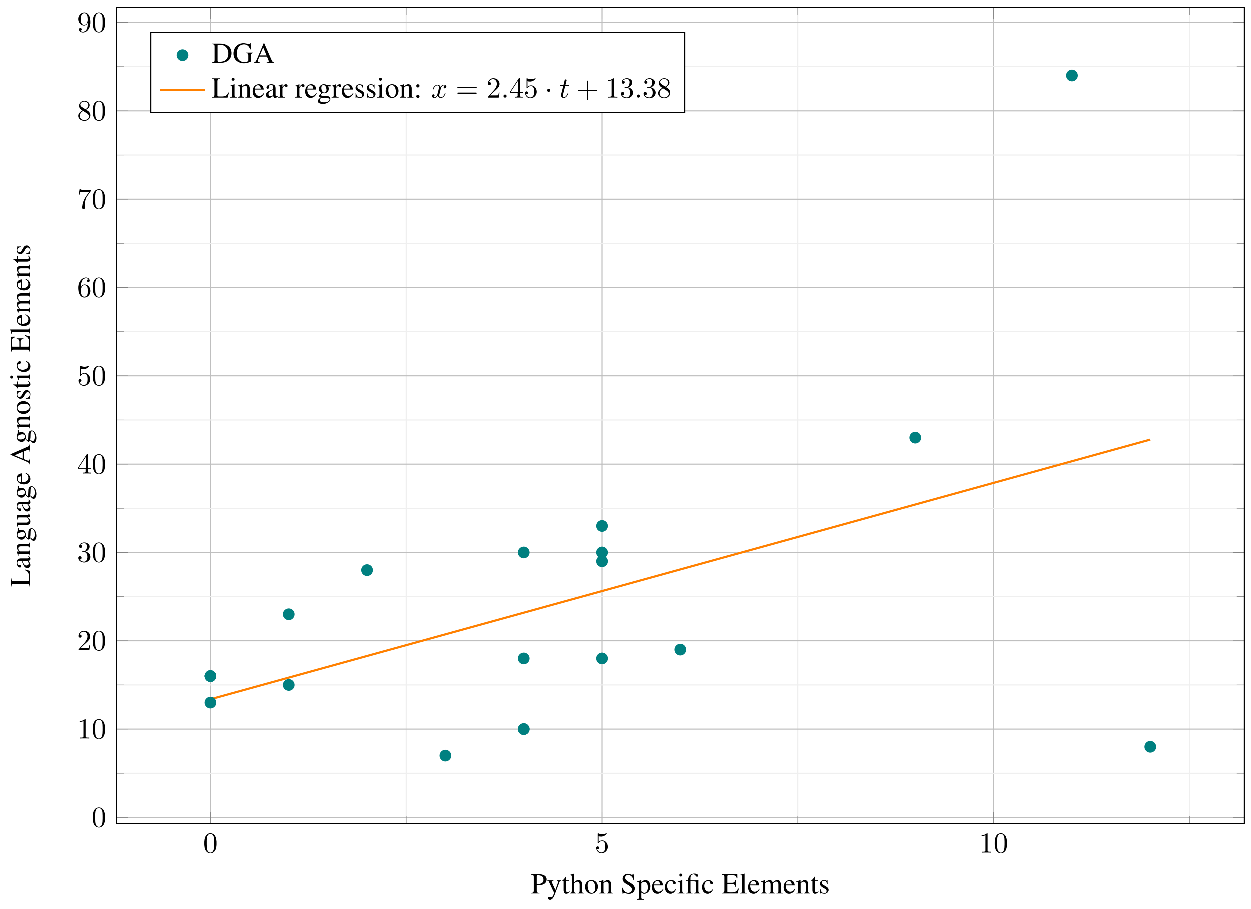

Figure 5.2.: *Python Specific Elements in Relation to Language Agnostic Elements:* This diagram visualizes two different sources of complexity in relation to each other. The value of the ”Python Specific Elements” is the sum of all Python specific AST nodes found in the source code of the displayed DGAs, the ”Language Agnostic Elements” is the result of the sum of common structures observed in most turing-complete programming languages such as functions, loops, unary operations, etc.

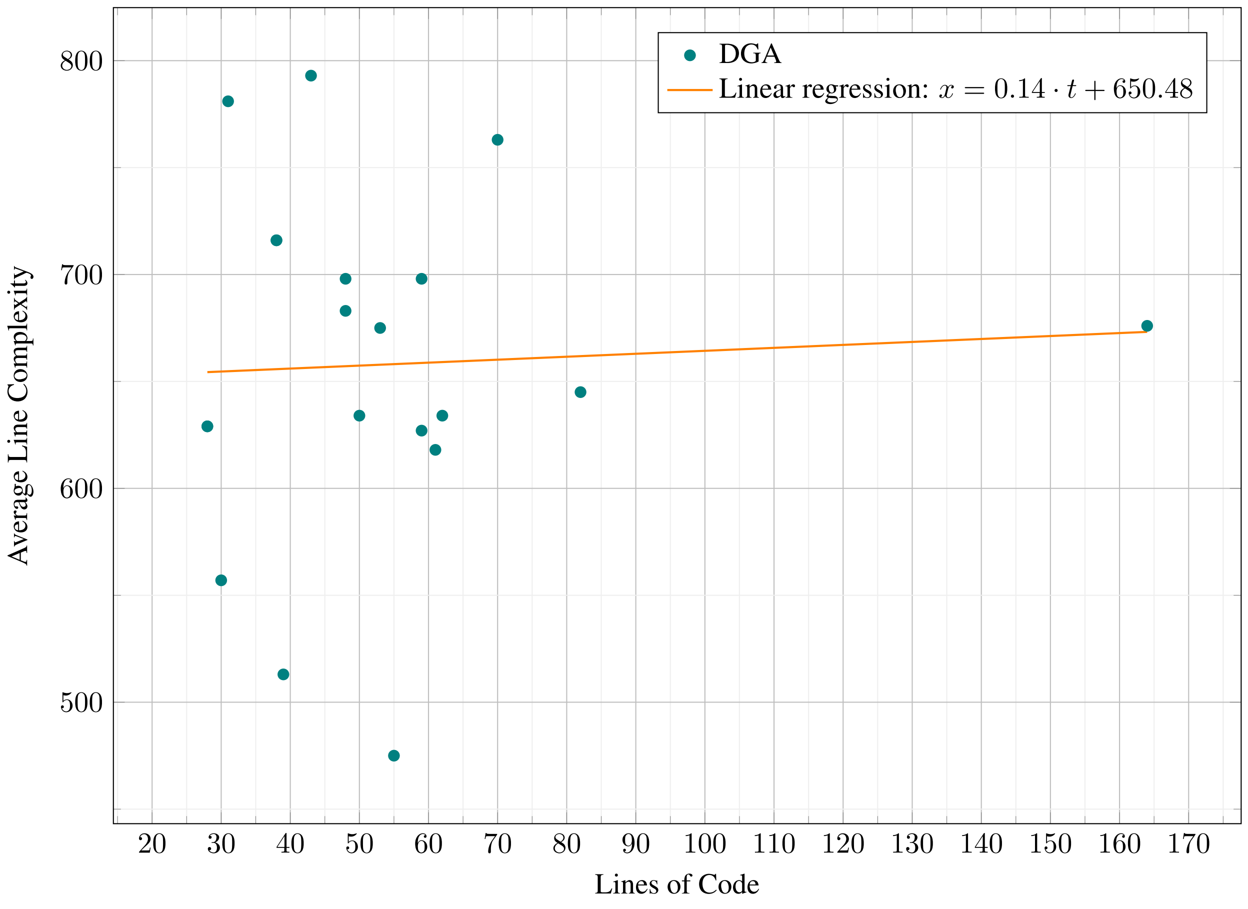

Figure 5.3.: *Diagram Displaying Lines of Code in Relation to the Average Line Complexity:* This diagram puts two fairly similar DGA metrics in context. On the x-axis the "Lines of Code" are displayed, the y-axis represents the "Average Line Complexity". This is the fraction quotient ratio of all AST nodes of a DGA divided by the lines of code, as further described in section 4.3.9. For simple trend analysis the linear regression is calculated and represented in orange.

combined with the first metric - ”Python Specific Elements” - of this section. This intersection delivers a group of DGAs which all exhibit the following property:

The value of ”Python Specific Elements” shall be in the lower third of all analyzed DGAs, with additionally the ”Average Line Complexity” or the ”Lines Of Code” being in the lower third of all analyzed DGAs as well.

This data is generated in section 4.3.5 and shown in table 5.5. The shown DGAs fulfill the aforementioned requirements and can thus be considered as well suited candidates for reimplementation in another programming language.

Summary of findings and results of the *Proprietary or Complex Code Elements* perspective:

- A high number of ”Language Agnostic Elements” does not necessarily prevent a high count of ”Python Specific Elements”.
- To find non-complex DGAs high values of ”Lines of Code” and ”Average Line Complexity” are mutually exclusive.
- The intersection of metrics as done in table 5.5 yields the most promising results.

5.3. General Classifications

Apart from evaluating dependence on external libraries as in section 5.1 or criteria within the source code as seen in section 5.2 it is also important to take into account functional properties of the analyzed DGAs. Depending on the planned future use of a DGA certain properties may be required while others make DGAs infeasible for certain applications. For example, in repeatable training scenarios a time dependent DGA may not be ideal to generate the desired learning effects. On the other hand, if a DGA is be used in machine learning or detection scenarios the uniqueness of the generated domains should be within an expected range. Based on the collected metric in section 4.3 a number of evaluations regarding the feasibility of DGAs can be made.

As an example table 5.6 provides the number of unique domains generated by each DGA for each given day. This also includes time independent DGAs which yield - unsurprisingly - the same domains on each given day. In this example the maximum number of generated domains per day is capped at 10.000 (ten thousand) to ensure execution of the ”Validity and Variance” analysis as described in section 4.3.2 in a timely manner.

Further, the parameter requirements of each DGA are described in table 5.7. It can be observed that some DGAs require only a seed or a date, while others require both. Some DGAs might require no parameters at all, while their usability in real world malware applications might be limited they pose a good start for reimplementation or transpiling efforts. Usually these DGAs are the least complex one might encounter in the wild.

Summary of findings and results of the *General Classifications* perspective:

- A project-specific functional requirement may block reimplementation of a DGA before code or dependency analysis occurs.
- Domain uniqueness and parameter dependency vary throughout all analyzed DGAs.
- There is a small number of DGAs requiring no external parameters. While of limited use in the real world they may provide a good starting point for future work.

DGA	**Import Count**
DGABanjori	0
DGARamdo	0
DGAFobberV2	0
DGAFobberV1	0
DGANecurs	1
DGARanbyus	1
DGACorebot	1
DGADircrypt	1
DGAMonerodownloader	2
DGAMurofetV1	2
DGAMurofetV2	2
DGAPadcrypt22861	2
DGAPadcrypt22970	2
DGAProslikefan	2
DGAMurofetV3	2
DGAChinad	3
DGANewgoz	3
DGAQadars	3
DGAPykspa	4
DGAQsnatch	4

Table 5.1.: *Table showing the number of imported modules by each DGA:* This table shows a sorted overview of the number of Python modules imported by each DGA implemented as a child class. This table gives hints at the effort needed to reimplement a DGA in another language. However, if a DGA imports only common modules often referenced by other DGAs, a low number of imports may not increase the difficulty level of reimplementation.

DGA	**Imports**
DGACorebot	datetime
DGADircrypt	datetime
DGAMonerodownloader	datetime hashlib
DGAMurofetV1	datetime hashlib
DGAMurofetV2	datetime hashlib
DGAMurofetV3	datetime hashlib
DGANecurs	datetime
DGAPadcrypt22861	datetime hashlib
DGAPadcrypt22970	datetime hashlib
DGARanbyus	datetime

Table 5.2.: *Table showing only DGAs which import the top 3 commonly imported modules:* Of all analyzed DGAs this table only shows the ones which import modules from the pool of the top 3 most imported modules. If a module is imported by many DGAs it may be identified as an important prerequisite to enable reimplementation of a DGA. Thus, making the functions of a common module available in the target language eases the reimplementation of many DGAs.

DGA	**Python Specific Elements**	**Language Agnostic Elements**
DGABanjori	0	13
DGAFobberV1	0	16
DGAFobberV2	0	16
DGARamdo	0	16
DGACorebot	1	23
DGADircrypt	1	15
DGANecurs	2	28
DGAMonerodownloader	3	7
DGAProslikefan	4	18
DGAPadcrypt22970	4	10
DGAPadcrypt22861	4	10
DGANewgoz	4	30
DGAMurofetV3	5	29
DGAMurofetV1	5	30
DGAChinad	5	18
DGAMurofetV2	5	33
DGAQadars	6	19
DGARanbyus	9	43
DGAPykspa	11	84
DGAQsnatch	12	8

Table 5.3.: *Python Specific Elements and Language Agnostic Elements:* This table displays Python specific elements and language agnostic elements for each analyzed DGA. A small pythonic complexity may hint at a decreased reimplementation effort.

DGA	**Lines of Code**	**Average Line Complexity**
DGAFobberV1	28	629
DGAFobberV2	28	629
DGARamdo	30	557
DGABanjori	31	781
DGAPadcrypt22861	38	716
DGAPadcrypt22970	38	716
DGADircrypt	39	513
DGACorebot	43	793
DGAQsnatch	48	683
DGAProslikefan	48	698
DGAQadars	50	634
DGAChinad	53	675
DGAMonerodownloader	55	475
DGAMurofetV1	59	627
DGANecurs	59	698
DGAMurofetV3	61	618
DGAMurofetV2	62	634
DGARanbyus	70	763
DGANewgoz	82	645
DGAPykspa	164	676

Table 5.4.: *Lines of Code and Average Line Complexity:* This table shows each DGA and the metrics ”Lines of Code” and ”Average Line Complexity” of it’s respective implemented child class. The composition of the metric ”Average Line Complexity” is further described in section 4.3.9

DGA	**Python Specific Elements**	**Lines of Code**	**Average Line Complexity**
DGAFobberV1	0	28	629
DGAFobberV2	0	28	629
DGARamdo	0	30	557
DGABanjori	0	31	781
DGADircrypt	1	39	513

Table 5.5.: *Promising DGAs for efficient reimplementation:* This table only displays the lower third of all DGAs in the ”Python Specific Elements” metric which are additionally in the lower third of either the ”Lines of Code” or ”Average Line Complexity” metric.

DGA	**Unique Domains per 10000**
DGAQsnatch	30
DGAChinad	256
DGAMurofetV3	1000
DGAMurofetV1	1020
DGAMurofetV2	1020
DGAProslikefan	9501
DGAQadars	10000
DGAPykspa	10000
DGAPadcrypt22970	10000
DGAPadcrypt22861	10000
DGANewgoz	10000
DGABanjori	10000
DGARamdo	10000
DGAMonerodownloader	10000
DGAFobberV2	10000
DGAFobberV1	10000
DGADircrypt	10000
DGACorebot	10000
DGANecurs	10000
DGARanbyus	10000

Table 5.6.: *Unique domains generated per day per DGA:* This table shows the number of unique domains generated for each DGA per day. To effectively count the number of unique domains the *get_nextdomain()* method of each implemented child class DGA is executed 10.000 times, subsequently the number of distinct domains is determined.

DGA	Date Parameter required?	Seed Parameter required?
DGAFobberV1	no	no
DGAFobberV2	no	no
DGABanjori	no	yes
DGARamdo	no	yes
DGAChinad	yes	no
DGAMonerodownloader	yes	no
DGAMurofetV1	yes	no
DGAMurofetV3	yes	no
DGANewgoz	yes	no
DGAPadcrypt22861	yes	no
DGAPadcrypt22970	yes	no
DGAPykspa	yes	no
DGAQsnatch	yes	no
DGACorebot	yes	yes
DGADircrypt	yes	yes
DGAMurofetV2	yes	yes
DGANecurs	yes	yes
DGAProslikefan	yes	yes
DGAQadars	yes	yes
DGARanbyus	yes	yes

Table 5.7.: *Parameter requirements for each implemented DGA:* This table displays the input parameters required by each implemented DGA split by parameter type. A ”yes” in the respective ”Date Parameter required?” or ”Seed Parameter required?” column indicates a mandatory parameter which has to be supplied to the DGA at instantiation time.

6. Conclusion

Research shows that domain generation algorithms can be found readily available on the internet. They may be found in public source code repositories, security blogs or are distributed as samples by the antimalware industry. On the other hand, there are also projects by fellow researchers analyzing the functionality of certain types of DGAs, some specialize in detection, some in generation of said domains.
The important part which could be achieved throughout this research was to bring those angles together. Putting the different approaches not only into perspective, but also creating a common framework for the analyzation of DGAs yields repeatable, useable results for every interested researcher.
The evaluation clearly shows, apart from creating an abstract class and unifying different DGAs into one seamless framework, it was also accomplished to create a system of metrices to successfully evaluate and classify DGAs for further use in the field.

6.1. Future Work

Both, the functional DGA testing suite, and the analyzation framework, are a sound platform for further research. For instance, there are projects exploring language-to-language transpiling efforts to create source code in some true compiler based language from Python scripts. Currently, these endeavors mostly only support a subset of Python elements to be transpiled into another language native structures. An adoption of this analyzation framework may aide the process of identifying DGAs already fit for automatic transpiling.
Another idea would be a plug-in system for malware research. Users may create a piece of harmless "malware" for academic purposes and have the DGA effortless inserted via the already in-place abstract class. Further, the to-be-inserted DGA could be automatically or randomly selected by using any number of metrics created during the phase of functional analysis.
The possibilities in this field are broad and creativity knows no bounds.

A. Appendix: Repositories of DGAs

Author	URL	# of DGAs	License
pchaigno	https://github.com/pchaigno/dga-collection	8	MIT
andrewaeva	https://github.com/andrewaeva/DGA	8	unknown
baderj	https://github.com/baderj/domain_generation_algorithms	44	GPL-2.0

Table A.1.: *Public software repositories containing DGAs:* This table shows the URLs to the public software repositories used in this document. Additionally the number of DGAs and the license under which the source code is published is provided.

B. Appendix: Source Code

B.1. DGA Child Class Skeleton

This listing shows the complete skeleton code classes DGAs may be implemented as.

```python
"""Module to showcase varios empty DGA classes.

They may be used to scaffold real world DGA implemenatations.
A real world DGA should only consist of one classe per module.

"""

from dga_classes.dga_abstract_class import DGA


class DGAEmpty(DGA):
    """Implementation of the "Empty" DGA."""

    def __init__(self):
        """Initializing the DGA.

        This implementation does not require any parameters.

        """
        super().__init__()

    def generate_domain(self):
        """Generates a new domain name."""
        self._state["lastdomain"] = "DGAEmpty"
        return(self._state["lastdomain"])


class DGAEmptySeed(DGA):
    """Implementation of the "EmptySeed" DGA."""

    def __init__(self, seed: any = ""):
        """Initializing the DGA.

        This DGA is using a seed as parameter.

        Parameters
        ----------
        seed : any
            The seed provides the source of randomness, default value: ''
        """
        super().__init__()
        self._seed = seed

    def generate_domain(self):
        """Generates a new domain name."""
        self._state["lastdomain"] = "Seed: " + self._seed
        return(self._state["lastdomain"])


class DGAEmptyDate(DGA):
    """Implementation of the "EmptyDate" DGA."""

    import datetime

    def __init__(self, date: datetime.datetime = datetime.datetime.now()):
        """Initializing the DGA.

        This DGA is using a date as parameter.

        Parameters
        ----------
        date : datetime
            The date of which the domains are generated on, default value: now
                ()
        """
        super().__init__()
        self._date = date

    def generate_domain(self):
        """Generates a new domain name."""
        self._state["lastdomain"] = " Date: " + str(self._date)
        return(self._state["lastdomain"])


class DGAEmptySeedDate(DGA):
    """Implementation of the "EmptySeedDate" DGA."""

    import datetime

    def __init__(self, seed: any = "", date: datetime.datetime = datetime.
        datetime.now()):
        """Initializing the DGA.

        This DGA is using a seed and a date as parameter.

        Parameters
        ----------
        date : datetime
            The date of which the domains are generated on, default value: now
                ()
        seed : any
            The seed provides the source of randomness, default value: ''
        """
        super().__init__()
        self._seed = seed
        self._date = date

    def generate_domain(self):
        """Generates a new domain name."""
        self._state["lastdomain"] = "Seed: " + self._seed + " Date: " + str(
            self._date)
        return(self._state["lastdomain"])
```

Listing B.1: The base skeleton code DGAs may be implemented as.

B.2. DGA Analysis Phase 1: Validity and Variance

```
import requests
import csv
import time
import importlib
import glob
import json

from dga_classes.dga_abstract_class import DGA

txt_dga_not_matched = "!!!_DGA_not_found_!!!"

chkdm_api_url = "https://dgarchive.caad.fkie.fraunhofer.de/r/"
chkdm_cache_filename = "_dga_cache.csv"
chkdm_cache_dict = {}

stats_amount_domains_valid_test = 10
stats_amount_domains_valid_filename = "_dga_stats_valid_domains.json"
stats_amount_domains_unique_test = 10000
stats_amount_domains_unique_filename = "_dga_stats_unique_domains.json"
stats_score_dga = {}
stats_domains_dga = {}


def stats_increase_dga_score(dga: str):
    stats_score_dga[dga] = (stats_score_dga.get(dga, 0) + 1)
```

```


def checkdomain_api(domain: str):
    r = requests.get(chkdm_api_url + domain, auth=requests.auth.HTTPBasicAuth(
        chkdm_api_user, chkdm_api_pass))
    print("Cache miss, checking API, HTTP " + str(r.status_code))
    time.sleep(1)
    result = ""
    if ((r.json().get(["hits"][0]))):
        result = str(r.json()["hits"][0]["family"])
    else:
        result = txt_dga_not_matched
    return result


def writedomain_cache(domain: str, family: str):
    chkdm_cache_csv = csv.writer(open(chkdm_cache_filename, "a"), delimiter=",
        ")
    chkdm_cache_csv.writerow([domain, family])
    return


def checkdomain_full(domain: str):
    if (domain in chkdm_cache_dict):
        return chkdm_cache_dict[domain]
    result = checkdomain_api(domain)
    writedomain_cache(domain=domain, family=result)
    return result


chkdm_cache_file = open(chkdm_cache_filename, "r")
chkdm_cache_csv = csv.reader(chkdm_cache_file, delimiter=",")
for row in chkdm_cache_csv:
    chkdm_cache_dict[row[0]] = row[1]
chkdm_cache_file.close()

module = glob.glob("dga_classes/*.py")
module.sort()

for m in module:
    importlib.import_module(m.replace("/", ".")[:-3])

dgalist = DGA.__subclasses__()

for currentdga in dgalist:
    testdga = currentdga()
    testdga_name = str(type(testdga).__name__)
    if (not(testdga_name.startswith("DGAEmpty"))):
        stats_score_dga[testdga_name] = 0
        for _ in range(stats_amount_domains_valid_test):
            cresult = checkdomain_full(testdga.get_nextdomain())
            if (cresult != txt_dga_not_matched):
                stats_increase_dga_score(testdga_name)

print("DGA reliabilty:")
for dgaentry in stats_score_dga:
    print(dgaentry + ": " + str(stats_score_dga[dgaentry]) + "/" + str(
        stats_amount_domains_valid_test))
    stats_score_dga[dgaentry] = round((stats_score_dga[dgaentry] /
        stats_amount_domains_valid_test) * 100)

for currentdga in dgalist:
    testdga = currentdga()
    testdga_name = str(type(testdga).__name__)
    testdga_domains = set()
    testdga_domain_count = 0
    if (not(testdga_name.startswith("DGAEmpty"))):
        if stats_score_dga[testdga_name] == 100:
            for _ in range(stats_amount_domains_unique_test):
                testdga_domains.add(testdga.get_nextdomain())
            stats_domains_dga[testdga_name] = len(testdga_domains)

print("DGA unique domains generated:")
for dgaentry in stats_domains_dga:
    print(dgaentry + ": " + str(stats_domains_dga[dgaentry]) + "/" + str(
        stats_amount_domains_unique_test))

print("Writing results of integrity and unique domain tests to: " +
    stats_amount_domains_valid_filename + " and " +
    stats_amount_domains_unique_filename)
with open(stats_amount_domains_unique_filename, "w") as outfile:
    json.dump(stats_domains_dga, outfile)

with open(stats_amount_domains_valid_filename, "w") as outfile:
    json.dump(stats_score_dga, outfile)
```

Listing B.2: Python code for phase 1 of the analysis workflow

Notice: For security purposes the credentials to authenticate against the webservice referenced in this code are omitted.

B.3. DGA Analysis Phase 2: Collection of metrics via AST

```
"""
This python file contains a collection of tools to perform static code
    analyzation for multiple DGAs.

All subclasses of the abstract DGA class are enumerated and analyzed, the
```

```
    results are saved into a JSON file to be processed further.
"""

import ast
import json
import sys
import importlib
import glob

from dga_classes.dga_abstract_class import DGA


class DGAAnalyzer(ast.NodeVisitor):
    """The DGAAnalyzer class provides methods to loop through an AST
        representation of sourcecode."""

    def __init__(self, statsdict):
        """Initializing the Analyzer.

        Parameters
        ----------
        statsdict : dict
            The dictionary object into which the data will be saved.
        """
        self.stats = statsdict
        self.classnr = 0

    def visit_ClassDef(self, node: ast.AST):
        """This visitor method stops at class AST nodes and starts the
            analyzation of each.

        If it encounters multiple classes within a single file or multiple
            definitions of the same class in different files an alert is
            logged.
        """
        self.classnr += 1
        if (self.classnr > 1):
            self.stats["alerts"].append("Ignored " + node.name + ": multiple
                classes in same file detected")
            return
        if (node.name in self.stats["dga"]):
            self.stats["alerts"].append("Ignored " + node.name + ": class
                already analyzed")
            return
        self.stats["dga"][node.name] = {}
        self.stats["dga"][node.name]["LinesOfCode"] = (node.end_lineno - node.
            lineno) + 1
        self.stats["dga"][node.name]["AvgLineComplexity"] = 0
        self.stats["dga"][node.name]["SyntaxNodes"] = 0
        self.stats["dga"][node.name]["Imports"] = []
        self.stats["dga"][node.name]["ImportCount"] = 0
        self.stats["dga"][node.name]["Functions"] = 0
        self.stats["dga"][node.name]["LambdaFunctions"] = 0
        self.stats["dga"][node.name]["Constants"] = 0
        self.stats["dga"][node.name]["Assignments"] = 0
        self.stats["dga"][node.name]["ParamSeed"] = 0
        self.stats["dga"][node.name]["ParamDate"] = 0
        self.stats["dga"][node.name]["ParamCount"] = 0
        self.stats["dga"][node.name]["PythonFstring"] = 0
        self.stats["dga"][node.name]["PythonStructures"] = 0
        self.stats["dga"][node.name]["PythonStructureList"] = 0
        self.stats["dga"][node.name]["PythonStructureTuple"] = 0
        self.stats["dga"][node.name]["PythonStructureSet"] = 0
        self.stats["dga"][node.name]["PythonStructureDict"] = 0
        self.stats["dga"][node.name]["Loops"] = 0
        self.stats["dga"][node.name]["Decisions"] = 0
        self.stats["dga"][node.name]["BinaryOperations"] = 0
        self.stats["dga"][node.name]["BinaryOperationModulo"] = 0
        self.stats["dga"][node.name]["BooleanOperations"] = 0
        self.stats["dga"][node.name]["UnaryOperations"] = 0
        self.stats["dga"][node.name]["Comparisons"] = 0
        self.analyze_dgaclass(node, node.name)
        self.stats["dga"][node.name]["ParamCount"] = self.stats["dga"][node.
            name]["ParamSeed"] + self.stats["dga"][node.name]["ParamDate"]
        self.stats["dga"][node.name]["AvgLineComplexity"] = round((int(self.
            stats["dga"][node.name]["SyntaxNodes"]) / int(self.stats["dga"][
            node.name]["LinesOfCode"])) * 100)

    def analyze_dgaclass(self, tree, dga_class: str):
        """Given a node within an AST tree this method walks through the
            remaining AST tree.

        Each node is analyzed and the results are added to the stats
            dictionary.
        """
        for node in ast.walk(tree):
            self.stats["dga"][dga_class]["SyntaxNodes"] = self.stats["dga"][
                dga_class].get("SyntaxNodes", 0) + 1

            if (isinstance(node, ast.Import)):
                self.stats["dga"][dga_class]["ImportCount"] = self.stats["dga"
                    ][dga_class].get("ImportCount", 0) + 1
                for alias in node.names:
                    self.stats["dga"][dga_class]["Imports"].append(alias.name)

            if (isinstance(node, ast.ImportFrom)):
                self.stats["dga"][dga_class]["ImportCount"] = self.stats["dga"
                    ][dga_class].get("ImportCount", 0) + 1
                for alias in node.names:
                    self.stats["dga"][dga_class]["Imports"].append("from " + node.module + " import " + alias.name)

            if isinstance(node, ast.FunctionDef):
                self.stats["dga"][dga_class]["Functions"] = self.stats["dga"][dga_class].get("Functions", 0) + 1

            if isinstance(node, ast.Lambda):
                self.stats["dga"][dga_class]["LambdaFunctions"] = self.stats["dga"][dga_class].get("LambdaFunctions", 0) + 1

            if isinstance(node, ast.Constant):
                self.stats["dga"][dga_class]["Constants"] = self.stats["dga"][dga_class].get("Constants", 0) + 1

            if isinstance(node, ast.Assign) or isinstance(node, ast.AugAssign) or isinstance(node, ast.AnnAssign):
                self.stats["dga"][dga_class]["Assignments"] = self.stats["dga"][dga_class].get("Assignments", 0) + 1

            if isinstance(node, ast.FunctionDef) and (node.name == "__init__"):
                for arg in node.args.args:
                    if (arg.arg == "seed"):
                        self.stats["dga"][dga_class]["ParamSeed"] = 1
                    if (arg.arg == "date"):
                        self.stats["dga"][dga_class]["ParamDate"] = 1

            if isinstance(node, ast.FormattedValue) or isinstance(node, ast.JoinedStr):
                self.stats["dga"][dga_class]["PythonFstring"] = self.stats["dga"][dga_class].get("PythonFstring", 0) + 1

            if isinstance(node, ast.List) or isinstance(node, ast.Tuple) or isinstance(node, ast.Set) or isinstance(node, ast.Dict):
                self.stats["dga"][dga_class]["PythonStructures"] = self.stats["dga"][dga_class].get("PythonStructures", 0) + 1

            if isinstance(node, ast.List):
                self.stats["dga"][dga_class]["PythonStructureList"] = self.stats["dga"][dga_class].get("PythonStructureList", 0) + 1

            if isinstance(node, ast.Tuple):
                self.stats["dga"][dga_class]["PythonStructureTuple"] = self.stats["dga"][dga_class].get("PythonStructureTuple", 0) + 1

            if isinstance(node, ast.Set):
                self.stats["dga"][dga_class]["PythonStructureSet"] = self.
```

```
                    stats["dga"][dga_class].get("PythonStructureSet", 0) + 1

            if isinstance(node, ast.Dict):
                self.stats["dga"][dga_class]["PythonStructureDict"] = self.
                    stats["dga"][dga_class].get("PythonStructureDict", 0) + 1

            if isinstance(node, ast.For) or isinstance(node, ast.While):
                self.stats["dga"][dga_class]["Loops"] = self.stats["dga"][
                    dga_class].get("Loops", 0) + 1

            if isinstance(node, ast.If) or isinstance(node, ast.IfExp):
                self.stats["dga"][dga_class]["Decisions"] = self.stats["dga"][
                    dga_class].get("Decisions", 0) + 1

            if isinstance(node, ast.BinOp):
                self.stats["dga"][dga_class]["BinaryOperations"] = self.stats[
                    "dga"][dga_class].get("BinaryOperations", 0) + 1
                if (isinstance(node.op, ast.Mod)):
                    self.stats["dga"][dga_class]["BinaryOperationModulo"] =
                        self.stats["dga"][dga_class].get("
                        BinaryOperationModulo", 0) + 1

            if isinstance(node, ast.BoolOp):
                self.stats["dga"][dga_class]["BooleanOperations"] = self.stats
                    ["dga"][dga_class].get("BooleanOperations", 0) + 1

            if isinstance(node, ast.UnaryOp):
                self.stats["dga"][dga_class]["UnaryOperations"] = self.stats["
                    dga"][dga_class].get("UnaryOperations", 0) + 1

            if isinstance(node, ast.Compare):
                self.stats["dga"][dga_class]["Comparisons"] = self.stats["dga"
                    ][dga_class].get("Comparisons", 0) + 1


def save_json_stats(dgadict, jsonfile):
    """."""
    with open(jsonfile, 'w') as outfile:
        json.dump(dgadict, outfile)


def main():
    if sys.version_info < (3, 8, 10):
        print("Please run ver. 3.8.10 or above")
        return

    dga_all_stats = {}
    dga_all_stats["dga"] = {}
    dga_all_stats_file = "_dga_stats_ast.json"
```

```python

    module = glob.glob("dga_classes/DGA*.py")
    module.sort()

    for m in module:
        importlib.import_module(m.replace("/", ".")[:-3])

    dgalist = DGA.__subclasses__()

    for currentdga in dgalist:
        dga_file = str(currentdga.__module__.replace(".", "/") + ".py")
        with open(dga_file, "r") as source:
            tree = ast.parse(source.read())
        analyzer = DGAAnalyzer(dga_all_stats)
        analyzer.visit(tree)

    save_json_stats(dga_all_stats, dga_all_stats_file)


if (__name__ == "__main__"):
    main()
```

Listing B.3: Python code for phase 2 of the analysis workflow

B.4. DGA Analysis Phase 3: Enrichment and creation of data files

```python
import json
import csv
import pandas as pd


def enrich_stats(statsdict, uniquedomainsfile: str, validdomainsfile: str):
    """This method expands the collected stats into different dimensions.

    Data from the DGA integrity testing is also imported.
    """
    dga_unique_domains = {}
    dga_valid_domains = {}
    statsdict["global_dga_params"] = {}
    statsdict["global_dga_params"]["None"] = 0
    statsdict["global_dga_params"]["Seed"] = 0
    statsdict["global_dga_params"]["Date"] = 0
    statsdict["global_dga_params"]["SeedAndDate"] = 0
    statsdict["global_counts"] = {}
    statsdict["global_counts"]["DGAs"] = 0

    try:
        with open(uniquedomainsfile) as infile:
            dga_unique_domains = json.load(infile)
    except FileNotFoundError:
        statsdict["alerts"].append("WARN: Unique domains JSON file " + uniquedomainsfile + " not found. You may want to run the domain integrity test first.")

    try:
        with open(validdomainsfile) as infile:
            dga_valid_domains = json.load(infile)
    except FileNotFoundError:
        statsdict["alerts"].append("WARN: Valid domains JSON file " + validdomainsfile + " not found. You may want to run the domain integrity test first.")

    for x in statsdict["dga"]:
        for imp in statsdict["dga"][x]["Imports"]:
            statsdict["global_import_counts"][imp] = statsdict["global_import_counts"].get(imp, 0) + 1
        statsdict["dga"][x]["DomainsValidPercent"] = dga_valid_domains.get(x, -1)
        statsdict["dga"][x]["DomainsUniqueDay"] = dga_unique_domains.get(x, -1)
        statsdict["dga"][x]["PythonicComplexity"] = statsdict["dga"][x]["ImportCount"] + statsdict["dga"][x]["LambdaFunctions"] + statsdict["dga"][x]["PythonFstring"] + statsdict["dga"][x]["PythonStructures"]
        statsdict["dga"][x]["LanguageAgnosticComplexity"] = statsdict["dga"][x]["Functions"] + statsdict["dga"][x]["Loops"] + statsdict["dga"][x]["BinaryOperations"] + statsdict["dga"][x]["BooleanOperations"] + statsdict["dga"][x]["UnaryOperations"] + statsdict["dga"][x]["Comparisons"] + statsdict["dga"][x]["Decisions"]
        statsdict["dga"][x]["TotalComplexity"] = statsdict["dga"][x]["PythonicComplexity"] + statsdict["dga"][x]["LanguageAgnosticComplexity"]
        if ((statsdict["dga"][x]["ParamSeed"] == 0) and (statsdict["dga"][x]["ParamDate"] == 0)):
            statsdict["global_dga_params"]["None"] += 1
        if ((statsdict["dga"][x]["ParamSeed"] == 1) and (statsdict["dga"][x]["ParamDate"] == 0)):
            statsdict["global_dga_params"]["Seed"] += 1
        if ((statsdict["dga"][x]["ParamSeed"] == 0) and (statsdict["dga"][x]["ParamDate"] == 1)):
            statsdict["global_dga_params"]["Date"] += 1
        if ((statsdict["dga"][x]["ParamSeed"] == 1) and (statsdict["dga"][x]["ParamDate"] == 1)):
            statsdict["global_dga_params"]["SeedAndDate"] += 1
        statsdict["global_counts"]["DGAs"] += 1
    return


```

```
def save_stats(dgadict, out_files: dict):
    """This method creates CSV files based on the statistics collected."""
    with open(out_files["csv_common_stats_file"], "w") as csvfile:
        first = 1
        for data in dgadict['dga']:
            temp_data = dgadict['dga'][data]
            temp_data["DGA"] = data
            if (first == 1):
                csvcolumns = temp_data.keys()
                writer = csv.DictWriter(csvfile, fieldnames=csvcolumns)
                writer.writeheader()
                first = 0
            writer.writerow(temp_data)

    # APPROACH: PARAMETER TYPE STATISTICS --- START
    df_paramstats = pd.DataFrame(list(dgadict["global_dga_params"].items()),
        columns=['ParameterType', 'Count'])
    df_paramstats['CountPercent'] = round(((df_paramstats['Count'] /
        df_paramstats['Count'].sum()) * 100), 1)
    df_paramstats.to_csv(out_files["csv_param_counts_file"], index=False)
    # APPROACH: PARAMETER TYPE STATISTICS --- END

    # EVALUATION: MOST USED MODULES --- START
    with open(out_files["csv_imports_file"], "w") as csvfile:
        sorted_dgadict = sorted(dgadict["global_import_counts"].items(), key=
            lambda x: x[1], reverse=True)
        header = ['imported', 'module']
        writer = csv.writer(csvfile)
        writer.writerow(header)
        for imp in sorted_dgadict:
            san_import = str(imp[0]).replace("_", "\_")
            # san_import = san_import.replace(" ", "\ ")
            data = [imp[1], san_import]
            writer.writerow(data)
    # EVALUATION: MOST USED MODULES --- END

    # EVALUATION: DGAS WITH MOST IMPORTED MODULES --- START
    df = pd.read_csv(out_files["csv_common_stats_file"])
    header = ["DGA", "ImportCount"]
    df = df.sort_values(by=["ImportCount"])
    df.to_csv(out_files["csv_dga_import_count_file"], columns=header, index=
        False)
    # EVALUATION: DGAS WITH MOST IMPORTED MODULES --- END

    # EVALUATION: DGAS WITH TOP 3 COMMON IMPORTS --- START
    df_top_imports = pd.read_csv(out_files["csv_imports_file"])
    header = ["DGA", "imports"]

    df_top_imports = df_top_imports.nlargest(3, 'imported', keep="all")
    list_top_imports = df_top_imports["module"].tolist()

    df_dgas_top_imports = pd.DataFrame(columns=header)

    for data in dgadict['dga']:
        c_dga = data
        c_imports = str()
        c_imports_only_top3 = 1
        for c_imp in dgadict['dga'][data]["Imports"]:
            if c_imp in list_top_imports:
                c_imports = c_imports + c_imp + " "
            else:
                c_imports_only_top3 = 0
        if (c_imports_only_top3 and len(c_imports) > 0):
            df_dgas_top_imports = df_dgas_top_imports.append({"DGA": c_dga, "imports": c_imports}, ignore_index=True)

    df_dgas_top_imports.to_csv(out_files["csv_dga_top_imports_file"], columns=header, index=False)
    # EVALUATION: DGAS WITH TOP 3 COMMON IMPORTS --- END

    # EVALUATION: DGAS WITH LOW PYTHONIC COMPLEXITY --- START
    df = pd.read_csv(out_files["csv_common_stats_file"])
    header = ["DGA", "PythonicComplexity", "LanguageAgnosticComplexity"]

    df = df.sort_values(by=["PythonicComplexity"])
    df.to_csv(out_files["csv_pycomp_lacomp_file"], columns=header, index=False)
    # EVALUATION: DGAS WITH LOW PYTHONIC COMPLEXITY --- END

    # EVALUATION: DGAS WITH LOW LOC OR LOW AVERAGE COMPLEXITY --- START
    df = pd.read_csv(out_files["csv_common_stats_file"])
    header = ["DGA", "LinesOfCode", "AvgLineComplexity"]

    df = df.sort_values(by=["LinesOfCode"])
    df.to_csv(out_files["csv_loc_avgcomp_file"], columns=header, index=False)
    # EVALUATION: DGAS WITH LOW LOC OR LOW AVERAGE COMPLEXITY --- END

    # EVALUATION: DGAS WITH (LOW LOC OR LOW AVERAGE COMPLEXITY) AND LOW PYTHONIC COMPLEXITY --- START
    df = pd.read_csv(out_files["csv_common_stats_file"])
    header = ["DGA", "LinesOfCode", "AvgLineComplexity", "PythonicComplexity"]

    amount_divisor = 3

    df_lowest_loc = df.nsmallest(round(len(df.index) / amount_divisor), "LinesOfCode", keep="all")
    df_lowest_alc = df.nsmallest(round(len(df.index) / amount_divisor), "AvgLineComplexity", keep="all")
```

```python
    df_lowest_pyc = df.nsmallest(round(len(df.index) / amount_divisor), "PythonicComplexity", keep="all")
    df_lowest_alc_or_loc = pd.concat([df_lowest_loc, df_lowest_alc]).drop_duplicates().reset_index(drop=True)
    df_lowest_alc_or_loc_and_pyc = pd.merge(df_lowest_alc_or_loc, df_lowest_pyc, how="inner")
    df_lowest_alc_or_loc_and_pyc = df_lowest_alc_or_loc_and_pyc.sort_values(by=["PythonicComplexity"])
    df_lowest_alc_or_loc_and_pyc.to_csv(out_files["csv_loc_alc_pyc_file"], columns=header, index=False)
    # EVALUATION: DGAS WITH (LOW LOC OR LOW AVERAGE COMPLEXITY) AND LOW PYTHONIC COMPLEXITY --- END

    # EVALUATION: UNIQUE DOMAINS PER DGA PER DAY --- START
    df = pd.read_csv(out_files["csv_common_stats_file"])
    header = ["DGA", "DomainsUniqueDay"]

    df = df.sort_values(by=["DomainsUniqueDay"])
    df.to_csv(out_files["csv_unqiue_domains_file"], columns=header, index=False)
    # EVALUATION: UNIQUE DOMAINS PER DGA PER DAY --- END

    # EVALUATION: PARAMETER REQUIREMENTS PER DGA --- START
    df = pd.read_csv(out_files["csv_common_stats_file"])
    header = ["DGA", "ParamDate", "ParamSeed"]

    df = df.replace(to_replace=1, value="yes")
    df = df.replace(to_replace=0, value="no")

    df = df.sort_values(by=["ParamDate", "ParamSeed"])
    df.to_csv(out_files["csv_param_details_file"], columns=header, index=False)
    # EVALUATION: PARAMETER REQUIREMENTS PER DGA --- END


def main():

    dga_all_stats = {}

    dga_all_stats_file = "_dga_stats_ast.json"
    dga_unique_domains_stats_file = "_dga_stats_unique_domains.json"
    dga_valid_domains_stats_file = "_dga_stats_valid_domains.json"

    out_files = {
        "csv_common_stats_file": "dga_stats_common.csv",
        "csv_imports_file": "dga_stats_imports.csv",
        "csv_dga_import_count_file": "dga_stats_dga_import_count.csv",
        "csv_dga_top_imports_file": "dga_stats_dga_top_imports.csv",
        "csv_loc_avgcomp_file": "dga_stats_loc_avgcomp.csv",
        "csv_pycomp_lacomp_file": "dga_stats_pycomp_lacomp.csv",
        "csv_loc_alc_pyc_file": "dga_stats_loc_alc_pyc.csv",
        "csv_param_counts_file": "dga_stats_param_counts.csv",
        "csv_param_details_file": "dga_stats_param_details.csv",
        "csv_unqiue_domains_file": "dga_stats_unique_domains.csv"
    }

    try:
        with open(dga_all_stats_file) as infile:
            dga_all_stats = json.load(infile)
    except FileNotFoundError:
        print("CRIT: No DGA statistic file found, please run AST analysis
            first.")
        exit()

    print(dga_all_stats)

    dga_all_stats["global_import_counts"] = {}
    dga_all_stats["alerts"] = []

    enrich_stats(dga_all_stats, dga_unique_domains_stats_file,
       dga_valid_domains_stats_file)
    save_stats(dga_all_stats, out_files)


if (__name__ == "__main__"):
    main()
```

Listing B.4: Python code for phase 3 of the analysis workflow

C. Appendix: Metrics Collected via AST

Phase 2 Metric	AST Node Class
ImportCount & Imports	ast.Import, ast.ImportFrom
Functions	ast.FunctionDef
LambdaFunctions	ast.Lambda
Constants	ast.Constant
Assignments	ast.Assign, ast.AugAssign, ast.AnnAssign
PythonFstring	ast.FormattedValue, ast.JoinedStr
PythonStructures	ast.List, ast.Tuple, ast.Set, ast.Dict
PythonStructureList	ast.List
PythonStructureTuple	ast.Tuple
PythonStructureSet	ast.Set
PythonStructureDict	ast.Dict
Loops	ast.For, ast.While
Decisions	ast.If, ast.IfExp
BinaryOperations	ast.BinOp
BinaryOperationModulo	ast.Mod
BooleanOperations	ast.BoolOp
UnaryOperations	ast.UnaryOp
Comparisons	ast.Compare
BinaryOperationModulo	ast.Mod

Table C.1.: *Metrics collected via AST:* This table provides an overview of the metrics collected in Phase 2 and which corresponding AST node classes are summed up to result in the final value of their respective metric.

List of Figures

List of Tables

Listings

Glossary

AES	Advanced Encryption Standard
API	Application Programming Interface
AST	Abstract Syntax Tree
C2	Command and Control
CSV	Comma Separated Value
DDoS	Distributed Denial of Service
DES	Data Encryption Standard
DGA	Domain Generation Algorithm
DNS	Domain Name System
FQDN	Full Qualified Domain Name
GCC	GNU Compiler Collection
HTTP	Hypertext Transfer Protocol
HTTPS	Secure Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation

Malware	Malicious computer software
OSS	Open Source Software
REST	Representational State Transfer

Bibliography

[1] • *global new malware volume 2020 | statista*, https://www-statista-com.ezproxy.fhstp.ac.at:2443/statistics/680953/global-malware-volume/, (Accessed on 02/14/2022).

[2] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna, "Your botnet is my botnet: Analysis of a botnet takeover," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, Chicago, Illinois, USA: Association for Computing Machinery, 2009, pp. 635–647, ISBN: 9781605588940. DOI: 10.1145/1653662.1653738. [Online]. Available: https://doi.org/10.1145/1653662.1653738.

[3] Vinayakumar Ravi, Mamoun Alazab, Sriram Srinivasan, Ajay Arunachalam, and K. P. Soman, "Adversarial defense: Dga-based botnets and dns homographs detection through integrated deep learning," *IEEE Transactions on Engineering Management*, pp. 1–18, 2021. DOI: 10.1109/TEM.2021.3059664.

[4] *Obfuscation: Malware's best friend | malwarebytes labs*, https://blog.malwarebytes.com/threat-analysis/2013/03/obfuscation-malwares-best-friend/, (Accessed on 02/15/2022).

[5] Ralph Droms, "Dynamic host configuration protocol," RFC Editor, RFC 2131, Mar. 1997, http://www.rfc-editor.org/rfc/rfc2131.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2131.txt.

[6] R. Fielding and J. Reschke, "Hypertext transfer protocol (http/1.1): Message syntax and routing," RFC Editor, RFC 7230, Jun. 2014, http://www.rfc-editor.org/rfc/rfc7230.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7230.txt.

[7] T. Brisco, "Dns support for load balancing," RFC Editor, RFC 1794, Apr. 1995, http://www.rfc-editor.org/rfc/rfc1794.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1794.txt.

[8] P. Mockapetris and K. J. Dunlap, "Development of the domain name system," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88, Stanford, California, USA: Association for Computing Machinery, 1988, pp. 123–133, ISBN: 0897912799. DOI: 10.1145/52324.52338. [Online]. Available: https://doi.org/10.1145/52324.52338.

[9] P. Mockapetris, "Domain names - concepts and facilities," RFC Editor, STD 13, Nov. 1987, http://www.rfc-editor.org/rfc/rfc1034.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1034.txt.

[10] Neville Stanton, Chris Baber, and Donald Harris, *Modelling Command and Control: Event Analysis of Systemic Teamwork*. Oct. 2017, ISBN: 9781315595825. DOI: 10.1201/9781315595825.

[11] Carol Simpson, "Internet relay chat," *Teacher Librarian*, vol. 28, no. 1, p. 18, 2000.

[12] Simon Heron, "Botnet command and control techniques," *Network Security*, vol. 2007, pp. 13–16, Apr. 2007. DOI: 10.1016/S1353-4858(07)70045-4.

[13] *Irc is dead, long live irc | pingdom*, https://www.pingdom.com/blog/irc-is-dead-long-live-irc/, (Accessed on 03/07/2021).

[14] Karikari Abina Mary, "Analysis of web protocols evolution on internet traffic," PhD thesis, Universidade da Beira Interior (Portugal), 2014.

[15] Martin Warmer, "Detection of web based command & control channels," Master's thesis, University of Twente, 2011.

[16] *Application layer protocol: Web protocols, sub-technique t1071.001 - enterprise | mitre att&ck®*, https://attack.mitre.org/techniques/T1071/001/, (Accessed on 03/07/2021).

[17] *Urlhaus | malware url exchange*, https://urlhaus.abuse.ch/, (Accessed on 03/07/2021).

[18] *A list of the best open source threat intelligence feeds | logz.io*, https://logz.io/blog/open-source-threat-intelligence-feeds/, (Accessed on 03/07/2021).

[19] Eihal Alowaisheq, Peng Wang, Sumayah Alrwais, Xiaojing Liao, Xiaofeng Wang, Tasneem Alowaisheq, Xianghang mi, Siyuan Tang, and Baojun Liu, "Cracking the wall of confinement: Understanding and analyzing malicious domain take-downs," Jan. 2019. DOI: 10.14722/ndss.2019.23243.

[20] Daniel Plohmann, *Dgarchive - fraunhofer fkie*, `https://dgarchive.caad.fkie.fraunhofer.de/site/families.html`, (Accessed on 02/20/2021), Feb. 21.

[21] Daniel Plohmann, Khaled Yakdan, Michael Klatt, Johannes Bader, and Elmar Gerhards-Padilla, "A comprehensive measurement study of domain generating malware," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16, Austin, TX, USA: USENIX Association, 2016, pp. 263–278, ISBN: 9781931971324.

[22] Niall Fitzgibbon and Mike Wood, "Conficker. c: A technical analysis," *Sophos Labs, Sophos Inc*, vol. 1, 2009.

[23] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2nd Edition. Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811. DOI: `10.5555/1177220`.

[24] *Design of cpython's compiler - python developer's guide*, `https://devguide.python.org/compiler/`, (Accessed on 02/16/2022).

[25] *Glossary — python 3.10.2 documentation*, `https://docs.python.org/3/glossary.html`, (Accessed on 02/16/2022).

[26] Daniel Wang, Andrew Appel, Jeffrey Korn, and Christopher Serra, "The zephyr abstract syntax description language.," Jan. 1997, pp. 213–228.

[27] *Examples of working with asts — green tree snakes 1.0 documentation*, `https://greentreesnakes.readthedocs.io/en/latest/examples.html`, (Accessed on 02/16/2022).

[28] Sandeep Yadav, Ashwath Reddy, A. Reddy, and Supranamaya Ranjan, "Detecting algorithmically generated malicious domain names," Jan. 2010, pp. 48–61. DOI: `10.1145/1879141.1879148`.

[29] Martin Grill, Ivan Nikolaev, Veronica Valeros, and Martin Rehak, "Detecting dga malware using netflow," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015, pp. 1304–1309. DOI: `10.1109/INM.2015.7140486`.

[30] *Kaspersky threats — shiz*, `https://threats.kaspersky.com/en/threat/Backdoor.Win32.Shiz/`, (Accessed on 03/22/2022).

[31] Tommy Chin, Kaiqi Xiong, Chengbin hu, and Yi Li, "A machine learning framework for studying domain generation algorithm (dga)-based malware: 14th international conference, securecomm 2018, singapore, singapore, august 8-10, 2018, proceedings, part i," in. Aug. 2018, pp. 433–448, ISBN: 978-3-030-01700-2. DOI: `10.1007/978-3-030-01701-9_24`.

[32] Deepak Kumar Vishwakarma, “Domain name generation algorithms [online],” SUPERVISOR: Ing. Mgr. et Mgr. Zdeněk Říha, Ph.D., Diplomová práce, Masarykova univerzita, Fakulta informatiky-Brno, 2017. [Online]. Available: `https://theses.cz/id/6c0o95/`.

[33] *Osint feeds from bambenek consulting*, `https://osint.bambenekconsulting.com/feeds/`, (Accessed on 02/20/2022).

[34] *Dga - netlab opendata project*, `https://data.netlab.360.com/dga/`, (Accessed on 02/20/2022).

[35] *Pep 8 – style guide for python code | python.org*, `https://www.python.org/dev/peps/pep-0008/`, (Accessed on 02/22/2022).

[36] Srdan Popić, Gordana Velikić, Hlavač Jaroslav, Zvjezdan Spasić, and Marko Vulić, “The benefits of the coding standards enforcement and it’s influence on the developers’ coding behaviour: A case study on two small projects,” in *2018 26th Telecommunications Forum (TELFOR)*, IEEE, 2018, pp. 420–425. DOI: `10.1109/TELFOR.2018.8612149`.

[37] *Error codes — pydocstyle 6.1.1 documentation*, `http://www.pydocstyle.org/en/6.1.1/error_codes.html`, (Accessed on 02/22/2022).

[38] *Flake8: Your tool for style guide enforcement — flake8 4.0.1 documentation*, `https://flake8.pycqa.org/en/latest/`, (Accessed on 02/22/2022).

[39] *Pandas - python data analysis library*, `https://pandas.pydata.org/`, (Accessed on 02/22/2022).

[40] *Pep 498 – literal string interpolation | python.org*, `https://www.python.org/dev/peps/pep-0498/`, (Accessed on 02/25/2022).

[41] *The python standard library — python 3.10.3 documentation*, `https://docs.python.org/3/library/`, (Accessed on 03/20/2022).