Automatic Extraction of Domain Name Generation Algorithms from Current Malware

Thomas Barabosch[1], Andre Wichmann[1], Felix Leder[2], and Elmar Gerhards-Padilla[1]

Fraunhofer FKIE
Friedrich-Ebert-Allee 144
53113 Bonn
GERMANY

Norman ASA
P.O. Box 43
N-1324 Lysaker
NORWAY

1 {Thomas.Barabosch, Andre.Wichmann, Elmar.Gerhards-Padilla}@fkie.fraunhofer.de
/ 2 Felix.Leder@norman.com

ABSTRACT

Botnets are a major threat to security on the Internet. Besides espionage and spamming, they are even used for attacking whole countries with DDoS attacks. In the ongoing arms race between law enforcement agencies and bot herders, the bot herders try to armour their botnets against takedowns with several sophisticated techniques.

Many botnets employ a method called domain fluxing for resilience. This technique strengthens the addressing layer of a botnet and allows a bot herder to dynamically provide command and control servers. For the calculation of new domains, a domain name generation algorithm (DGA) is used. In order to take actions against a domain fluxing botnet, the domain name generation algorithm has to be known.

This paper systematically classifies the different classes of DGAs and presents a novel approach for automatically extracting domain name generation algorithms from malware binaries for quickly initiating countermeasures. The approach's feasibility is shown in two case studies on current malware that uses domain fluxing.

1.0 INRODUCTION

Using the Internet has become a daily routine in many people's life. It becomes harder and harder not to be part of it, when even governmental institutions encourage their citizens to do their tax assessments online. On the one hand, the Internet comes with many benefits, like online shopping, online encyclopaedias, and real time news. On the other hand, users are confronted with new threats almost every day. Security played a subordinate role during the enormous expansion of the Internet during the past few years. This lack of security has been noticed by criminals, who have used the Internet to form a whole new area of business. This, among other things, yielded to a massive flood of malware. According to current statistics, the size of the malware population increased exponentially during the last few years and reached its historical high in 2011with 26 million newly created unique malware samples [1].

Large networks of infected machines, termed *botnets*, are formed by cyber criminals for financial gain and espionage. Over the years, botnets have become one of the most severe threats on the Internet. Some botnets have gathered several million members, called bots. For instance in 2007, the North European state Estonia was attacked by several *Distributed Denial of Service (DDoS)* attacks performed by botnets.

Governmental institutions and critical business infrastructures were targeted. This led to an enormous financial damage [2].

Botnets are commanded and controlled by persons termed *bot herders*. A botnet needs an addressing mechanism to identify the command and control entity, termed command and control server (C&C-server). Furthermore, it needs a communication channel to distribute commands to the bots [3]. Those two parameters depend on the chosen topology. It can be distinguished between centralized (e.g. a small fixed number of HTTP servers or an IRC server), decentralized (e.g. peer to peer networks where every bot can be a C&C-server) or locomotive (e.g. the commanding and controlling entities are moving over time) topologies.

It is very common for centralized or locomotive botnets to address the C&C-servers with the help of the domain name system (DNS). In practice this is usually combined with the HTTP protocol for the communication channel, but in principle any kind of protocol can be used.

In case of centralized botnets, there exist only a small, fixed number of C&C-servers. The addresses of those servers are usually hardcoded into the bot's binary. In order to dismantle a centralized botnet, a classical countermeasure is taking down the C&C-servers [3]. This is usually done by a cooperation of law enforcement agencies and the Internet service providers (ISPs), where the C&C-servers are hosted. Once the C&C-servers are taken down, the botnet is headless. Even though the bot machines are still infected, the bot herder has lost his control over those machines and cannot command them anymore. Since it can take some time until law enforcement agencies and ISPs agree on taking down the C&C-servers, network administrators can also blacklist IP addresses or domain names belonging to such a botnet to prevent any further communication of bots which are located in the network administrators' local networks with the C&C-servers. In this case, the bot herder just looses a part of his botnet.

Therefore, cyber criminals have started employing a technique called *domain fluxing*. This method introduces a variable set of C&C-server domain names which prevents a naive blacklisting approach employed by network administrators. In addition, it increases the flexibility of the addressing layer, and thus the resilience, of domain fluxing botnets against takedowns. Domain fluxing bots generate a list of domain names based on a predefined algorithm, called *domain name generation algorithm (DGA).* Each domain name in this list is resolved by a DNS query until there is no domain name left in the list or a domain name resolves to a C&C-server.

One side effect of (pseudo-) randomly computed domain names are collisions with existing ones. For example, the *Conficker.C* botnet generates 50.000 domain names each day and queries 500 of them [4]. Given the enormous amount of generated domain names, this leads to collisions with around 150-200 existent domain names per day. This can lead to DDoS attacks on the existing domain names and thus, possible result in loss of money.

The DGA of a domain fluxing botnet needs to be known in order to take countermeasures like for example preregistering future domain names and sinkholing the botnet traffic [3]. In addition, victims of domain name collisions can be notified early. However, malware is notorious for employing defensive techniques like executable packing for complicating its analysis [5]. In general, the process of manually reverse engineering a malware sample is tedious and time consuming. Therefore, this paper proposes an approach for increasing the efficiency of the analysis of domain fluxing botnets. The proposed method enables the malware analyst to automatically extract a DGA from a malware binary and use this knowledge in order to take countermeasures.

In detail, this paper makes the following contributions.

- It systematically classifies the different classes of DGAs
- It formalizes the problem of extracting DGAs and proposes a solution based on dynamic and static analysis techniques
- It presents the architecture of an automatic DGA extraction framework
- It demonstrates the feasibility of this approach in two case studies on current malware samples

The remainder of the paper is structured as follows. Section 2.0 presents a taxonomy of DGAs. Section 3.0 formalizes the DGA extraction problem and proposes a solution to it. Section 4.0 presents a DGA extraction framework, and in section 5.0, this framework is evaluated. Section 6.0 lists related work. Section 7.0 outlays future work and concludes this paper.

2.0 TAXONOMY OF DOMAIN NAME GENERATION ALGORITHMS

Domain fluxing malware uses domain name generation algorithms (DGA) for generating a set of possible C&C-servers' domain names. In the last couple of years, there have been several domain fluxing botnets like *Kraken* [6], *Conficker.C* [4] or *Torpig* [7]. There exist several approaches for generating domain names. Therefore, this section systematically classifies the different classes of DGAs and introduces a DGA naming convention.

It can be distinguished between four classes of DGAs. There are two possible parameters for a DGA, which are time and causality. The first class is the deterministic and time independent DGA (TID-DGA). Those DGAs generate the same set of domain names every time they are executed due to using a static seed. The early versions of the *Kraken* botnet use a TID-DGA [6]. The next class of DGAs is the time dependent and deterministic DGA (TDD-DGA). Here, the seed of the DGA is changing in a regular fashion. However, the precomputation of the domain names is still easy because of its determinism. The *Conficker* worm uses a TDD-DGA [4]. The third class of DGAs is the non deterministic and time dependent DGA (TDN-DGA). The seed cannot be anticipated and thus precomputation is impossible. This leads to a situation where neither law enforcement agencies nor the bot herder have got any advantage. The *Torpig* botnet uses a TDN-DGA. It uses the popular trending topics of the social networking service Twitter as a seed [7]. The fourth class is the time independent and non deterministic DGA (TIN-DGA). Malware employing TIN-DGAs has not been seen in the wild yet. This might work for small domain names but the probability of meeting a C&C-server, drastically decreases with the increase of the domain name length. Table 1 summarizes the four classes of DGAs.

type	time dependent	deterministic	example
TID	no	yes	Kraken
TDD	yes	yes	Conficker
TDN	yes	no	Torpig
TIN	no	no	Not yet seen

Table 1: The four different DGA types

3.0 METHODOLOGY

The previous sections introduced domain fluxing botnets. Before countermeasures like traffic sinkholing can be carried out against a domain fluxing botnet, the addresses of its C&C-servers have to be known.

Thus, the DGA of the botnet, which is compiled into every bot, has to be understood. The manual extraction of a DGA from a bot's binary through reverse engineering is a tedious and time consuming task. Therefore, this section presents a novel approach for automatically extracting DGAs from malware binaries.

At first the DGA extraction problem is defined in section 3.1. Given a domain fluxing program, the DGA extraction problem is to correctly extract the underlying DGA. This is followed by the presentation of a novel approach for solving this problem in section 3.2.

3.1 The DGA extraction problem

This section formalizes the problem of extracting DGAs from (malicious) binaries. It is called the DGA extraction problem. At first, important notions are introduced. This is followed by the definition of the DGA extraction problem. Finally, assumptions regarding this problem are discussed.

In the following, an algorithm $\mathcal{A}$ is assumed to consist of a non empty list of instructions, denoted by $\mathcal{I}_{\mathcal{A}} \subseteq \mathcal{I}$, where the processor's instruction set is denoted by $\mathcal{I}$. An algorithm has one entry point and it can have several possible exit points. The set of all algorithms is denoted by $\mathcal{A}$. Of course for every DGA $\mathcal{A}_{DGA} \in \mathcal{A}$.

Definition 1:

Given a program *p* consisting of a list of instructions $\mathcal{I}_p \subseteq \mathcal{I}$. Given that *p* uses domain fluxing.

Problem:

Determine the following three unknowns of the DGA $\mathcal{A}_{DGA}$, given the above information.

- The call $I_C \in \mathcal{I}_{\mathcal{A}_{DGA}}$ to a suspicious network related API
- The starting point $I_S \in \mathcal{I}_{\mathcal{A}_{DGA}}$ of $\mathcal{A}_{DGA}$
- The list $\mathcal{I}_{\mathcal{A}_{DGA_*}} \subseteq \mathcal{I}_{\mathcal{A}_{DGA}} \subseteq \mathcal{I}_p$ of instructions contained in all possible execution paths $p_n: I_S \rightarrow I_C, n \in \mathbb{N}$ starting at $\mathcal{A}_{DGA}$'s starting point I_S and ending at I_C.

Three unknowns of a DGA $\mathcal{A}_{DGA}$ must be determined in order to properly extract it from a program *p*. Those three unknowns set the limits of the DGA (I_S and I_C) and define its functionality($\mathcal{I}_{\mathcal{A}_{DGA_*}}$).

First, the precise determination of the starting point of $\mathcal{A}_{DGA}$ is very important for the extraction. The instruction of the starting point is denoted by I_S. Usually the first action taken by a DGA is to query some kind of source for an initial value. Based on this value, the computation of the domain name is done. The initial value can be, for example, an immediate value, the current time or the current trending topic on Twitter. The instruction querying a source $s \in S$ is denoted by I_S, where the set of all sources is denoted by $\mathcal{S}$.

Second, the exit point of the algorithm $\mathcal{A}_{DGA}$ needs to be found. It is assumed that at one point in time the malware uses a network related API provided by the operating system in order to connect to the C&C-servers. This instruction is denoted by $I_C \in \mathcal{I}_{\mathcal{A}_{DGA}}$.

Third, the precise extraction of $\mathcal{A}_{DGA}$'s instructions has to be performed. As stated in the definition, those are all instructions that are included in every execution path starting at I_S and ending at I_C.

Note that it is possible for a DGA to have multiple sources on which the computation can be based. Therefore, the first source asked for a value is assumed to be the primary source *s*. All other sources are called secondary sources and are denoted by a subscripted number. For example, the third secondary

source is denoted by s_3. In case of multiple sources, it is assumed that the variable returned by the primary source s is used for holding the domain name and the values of the secondary sources are merged with this variable during the computations. This is illustrated in figure 1.

In this illustration a DGA is sketched which has two sources. First, it queries its primary source for an initial value. Based on this value, the domain name is computed. This happens on the path presented by the long dotted arrow, which connects the primary source with the network related API. During the computation, a secondary source is queried for a value. This value is used for the domain name generation as well. Finally, the domain name is used in order to contact a C&C-server. Therefore, it is necessary to identify the first source of a DGA in order to extract it completely.

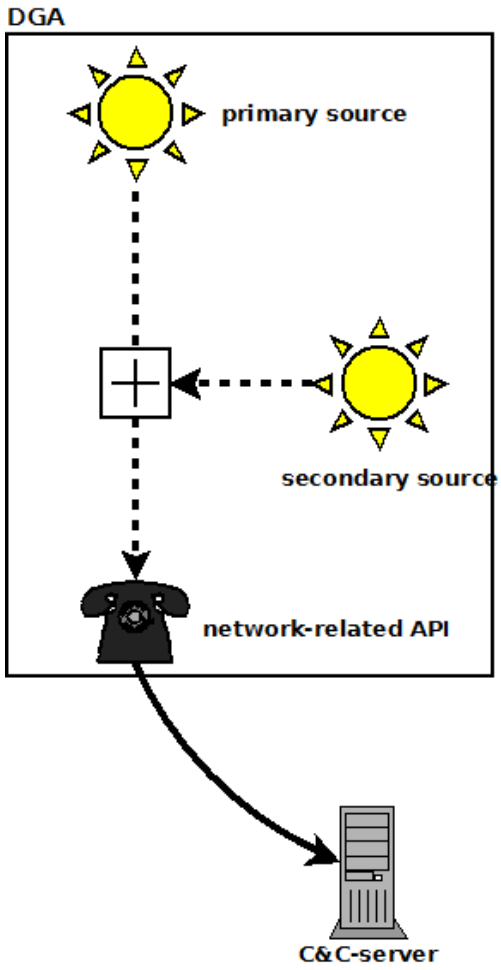

Figure 1: Illustration of a DGA with multiple sources

3.2 Automatic extraction of DGAs

Based on the definition of the DGA extraction problem in the previous section, this section proposes a novel approach for automatically extracting DGAs. At first its fundamental techniques are discussed. Then the approach is explained step by step.

Dynamic and static analysis techniques are used for extracting the DGA. Each type of analysis has its strengths and weaknesses. On the one hand, static analysis techniques can examine the whole code and thus peak into every detail of it. But they are not able to correctly handle programs using dynamic code,

like executable packers. Executable packing is heavily used by current malware and as stated in [5], at least around 80% of all malware samples employ this obfuscation technique. On the other hand, dynamic analysis techniques are able to handle dynamic code. But one major drawback is that during dynamic analysis, only one execution path can be examined [8]. Therefore, both techniques are combined in this approach to benefit from their strengths and eliminate their weaknesses.

Given a program *p* and the knowledge that *p* uses domain fluxing, the three unknowns from Definition 1, the network related API I_C, the DGA's starting point I_S and the DGA's list of instructions $\mathcal{I}_{\mathcal{A}_{DGA_*}}$, need to be determined. Since the query for an initial value is the first action taken by a DGA, the domain name used by $\mathcal{I}_C$ must be followed back to the request of the initial value at I_S.

In figure 2 the extraction approach is illustrated. Each step is explained in detail in the following. Note that the blue annotations next to each step summarize which of the above mentioned unknowns are resolved by it.

At first the program *p* is run until a call I_C to a suspicious network related API is encountered. Then *p* is stopped. Under the assumption that an executable packer unpacks *p* before its main logic is entered, *p* should be unpacked by now. As stated above, the vast majority of malware samples are packed by executable packers. An executable packer unpacks a packed binary into memory, and then jumps to the unpacked code. For this, it does not need network functionality. Therefore, once such a suspicious network related API is called, it is most likely called by the main logic of the unpacked program *p*.

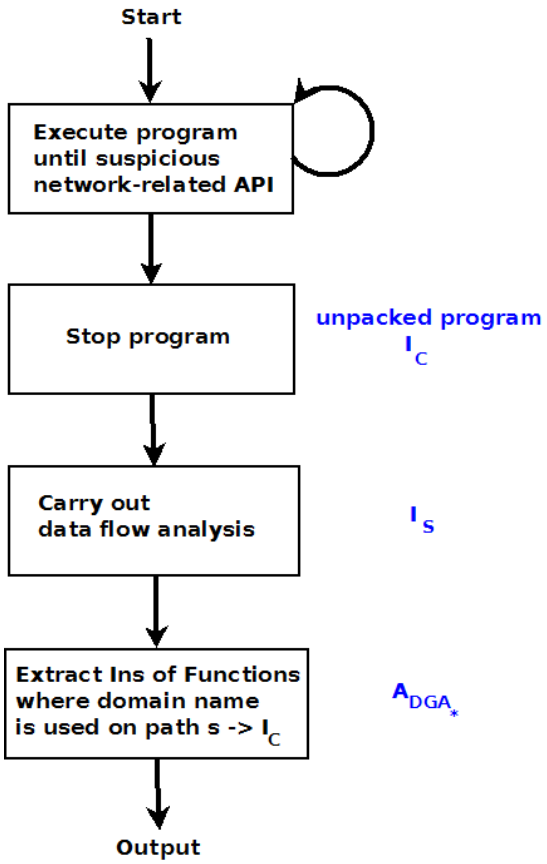

Figure 2: Control flow of the DGA extraction

Given the unpacked version of *p* and the end of the DGA I_C, the static analysis of *p* begins. The static

analysis is based on data flow analysis. Data flow analysis can be used in order to derive knowledge about the data's flow within a program without executing it.

It is assumed that on the execution paths from the main source $s \in S$ to I_C, the domain name is used and defined. Therefore, a reaching definitions analysis is done. This data flow analysis computes the definitions which can possibly reach a point in a program [9]. With the help of this data flow analysis, it is possible to compute *Use-Definition* and *Definition-Use* chains. A Use-Definition chain of a variable *x* consists of a use of this variable and all its definitions, which can reach this use without any redefinition in between. A Definition-Use chain of a variable *x* consists of a definition of this variable and all its usages, which can be reached by this definition without any redefinition in between [9]. Those two data structures are used during the later extraction of the DGA's instructions.

The starting point of the data flow analysis is $I_C \in \mathcal{I}_{\mathcal{A}_{DGA}}$. This is the final use of the domain name before the C&C-server is contacted. The analysis direction is backwards since the domain name has been generated before the C&C-server is contacted. Since a DGA can be split into several low level functions, an interprocedural data flow analysis is performed.

Once the data flow analysis is finished, the primary source of the DGA is determined. Based on the computed Use-Definition and Definition-Use chains, the instructions of the functions in which the domain name has been either used or defined are extracted.

In figure 3 the extraction process is illustrated. This figure shows a call graph of six functions interconnected by black arrows. The data flow analysis starts in the function on the bottom left. Its analysis progress is sketched by the red dotted arrows. The named boxes contain crucial parts of the DGA. The usages or definitions of the domain name in those functions are detected by the data flow analysis. Therefore, those functions are extracted. This is illustrated by the big dotted orange box.

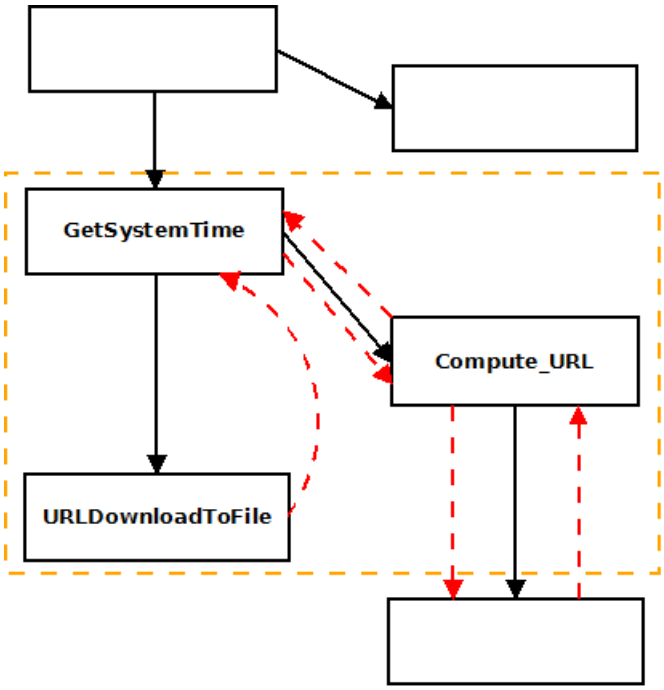

Figure 3: Illustration of the extraction process

4.0 A FRAMEWORK FOR EXTRACTING DGAs FROM MALWARE BINARIES

This section presents a framework for extracting DGAs from malware binaries, which implements the approach described in the previous section. Its general architecture is depicted in figure 4.

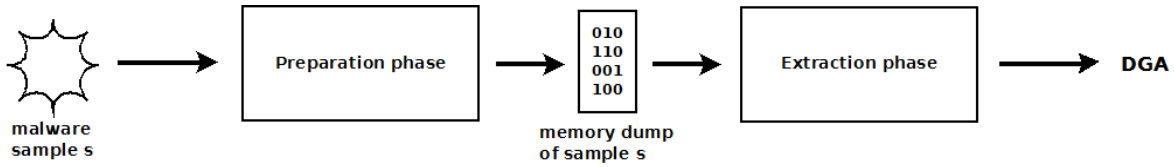

Figure 4: The extraction framework for DGAs

The extraction process consists of two phases, a dynamic analysis phase and a static analysis phase. The dynamic analysis phase prepares the malware binary for the later extraction of the DGA. It is called *preparation phase*. During this phase the malware binary is run in a safe environment and a memory dump is created as soon as a suspicious network related API is called. Based on this memory dump, the DGA is extracted. This is done during the static analysis phase, termed *extraction phase*. This phase carries out the data flow analysis and outputs the extracted DGA. The framework is implemented in IDApython, which is a scripting interface provided by the disassembler IDA Pro [10].

Each phase is described in detail in sections 4.1 and 4.2. This is followed by current limitations of the framework in section 4.3.

4.1 Preparation phase: Creation of the memory dump

The goal of this phase is to prepare the sample for the later extraction of the DGA. The extraction phase expects two parameters as input. First, it expects an unpacked malware sample. Thus, this phase implements a simple unpacking step. The underlying assumption is that once a network related API is called, the sample resides unpacked in memory (see section 3.2).

Second, the extraction phase expects the address of the call I_C to a suspicious network related API. Such a suspicious network related API can be, for example, *URLDownloadToFile*. *URLDownloadToFile* downloads a given URL to a local file [11]. Other examples are *InternetOpen* or *InternetOpenUrl*.

The preparation phase has to be performed within a safe environment. Because the analyzed malware sample is executed, it could potentially infect the environment before its DGA is executed. The execution of the sample is monitored and a dumping heuristic decides before suspicious network related APIs, whether to continue execution of the sample or to create a memory dump of it. The dumping heuristic follows a naive approach. It checks whether the suspicious network related API is called from the main image of the program or one of the process space's heaps. For efficiency reasons, it just dumps the main image or the heap section from which the call to I_C has been initiated. Once a memory dump is created, this memory dump and the last instruction pointer are passed to the Extraction phase.

Today's malware uses several stealth techniques in order to evade its detection [12]. One of these techniques is the injection and execution of code into another process space and is called *remote code injection*. Malware can potentially escape to another process space during the analysis when using this technique. Thus, the preparation phase implements a handling of remote code injection. It monitors the API calls needed for the most common way of remote process injection, *WriteProcessMemory* and *CreateRemoteThread*. *WriteProcessMemory* is used for writing code to the foreign process space. Then a

new thread within the foreign process space is created with *CreateRemoteThread*.

Once this kind of remote process injection is detected, the preparation phase suspends the current process and attaches itself to the process space, where the malware has injected its payload into. This enables it to monitor the newly created thread in the other process space.

4.2 Extraction phase: DGA extraction

The goal of the extraction phase is to extract the DGA from the provided memory dump. The extraction phase expects as input an unpacked malware sample and the address of I_C. It applies a data flow analysis in order to detect I_S. Then the instructions of the DGA are extracted on function granularity.

Before the data flow analysis can be carried out, the variable which holds the generated domain name must be determined. On the Windows NT platform the parameters of an API call are passed via the stack. Therefore, a database is used which holds for every suspicious network related API the stack offset to the generated domain name. Given this offset, the variable which should be used as a starting point of the data flow analysis can be easily determined. Once this variable is determined, the data flow analysis is carried out.

During the data flow analysis, encountered calls to APIs are approximated for efficiency. Every time an API is encountered during the data flow analysis, the influence of this API on the currently tracked variable is approximated with the help of rules stored in a database. Once an API is encountered, the database is queried. In case the database contains knowledge associated with the encountered API, it is applied to the currently tracked variable.

This is illustrated by the following example of the Windows API lpstrcpy. *lpstrcpy* takes two variables *lpString1* and *lpString2* as input and copies *lpString2* to *lpString1* [11]. Suppose the domain name is tracked in backwards direction and the API *lpstrcpy* is encountered. Furthermore, suppose that *lpString2* points to the domain name. In this case, the database is queried for this API. Since the database contains knowledge associated with this API, it returns that lpString2 is defined by lpstrcpy. Since lpString1 is copied to lpString2 this is not assumed as a final definition. Therefore, the data flow analysis continues with lpString1.

Once the data flow analysis is finished, the DGA is extracted on function granularity. Given the extracted DGA, a malware analyst can use this knowledge in order to initiate countermeasures or notify owners of colliding domain names.

4.3 Current limitations

In the following, limitations of the current implementation are described.

The framework's architecture is based on a third-party disassembler. Currently, IDA Pro is used. It is assumed that the underlying disassembler correctly disassembles the code and recognizes all functions. If these assumptions do not hold, the DGA extraction is not possible. It has been shown that even state of the art disassembler can be fooled using simple techniques and creating a precise disassembly can be a severe problem on architectures with variable instruction length like the x86 architecture [13].

Another limitation is a consequence of the preparation phase. Since the malware sample is dynamically analyzed in a safe analysis environment like a virtual machine, it is possible that the sample detects either the analysis software or the analysis environment. Once the malware has detected that it is analyzed, it might execute a non malicious control flow path or attack the analysis environment.

The support for process injecting malware is at the moment limited. Currently, the framework only

handles the *WriteProcessMemory/CreateRemoteThread* technique. However, the vast majority of processes injecting malware samples use the *WriteProcessMemory/CreateRemoteThread* technique.

Note that the limitations described above apply for any static and dynamic analysis techniques in general, and are not limited to the DGA extraction framework presented here.

5.0 EVALUATION

In the following the proposed approach's feasibility is shown by two case studies of current malware samples. The first case study in section 5.1 discusses the *Conficker.B* worm. This is followed by the second case study in section 5.2 which focuses on the file infector *LICAT.A*. The MD5 sums of the case studies' samples can be found in appendix A.1.

5.1 Conficker.B

Conficker is a computer worm. The botnet's first appearance was in mid 2008 and it is still very active. According to [14], in March 2012 there are still around three million machines infected by Conficker on the Internet. Conficker uses a TDD-DGA. A discussion of its DGA can be found in [4].

At first the Conficker.B sample is inputted into the framework. The dumping heuristic detects the use of the suspicious network related API *InternetOpenUrlA*. This API opens a URL on the Windows NT platform [11]. After the detection, the preparation phase finishes its work and passes the resulting memory dump to the extraction phase.

The extraction phase starts its analysis at $\mathcal{I}_C$. It determines the variable that holds the URL on the stack. Given the correct variable, the data flow analysis is carried out. The framework is able to find Conficker's DGA and successfully extracts it. By manually reverse engineering the sample, the framework's output has been verified.

5.2 LICAT.A

LICAT.A is a file infector which contains a download-and-execute routine for which it uses a TDD-DGA. Its DGA generates 1020 different domain names each day. For each domain name, the corresponding server is contacted. In case a server is successfully contacted, the offered file is downloaded and executed. Even though 1020 domain names are generated, it only tries to contact up to 800 of them. A detailed description of its DGA can be found in [15][1]. It has been stated that LICAT.A was used in a spreading campaign for the banking Trojan *Zeus* [16].

The sample is inputted into the framework and executed by the preparation phase. The suspicious network related API *URLDownloadToFile* is encountered and a memory dump is created. The memory dump is passed to the extraction phase. Thereafter, the extraction phase is started. First, the variable which contains the generated URL is determined. Then the data flow analysis is carried out. It encounters LICAT.A DGA's source and successfully extracts the DGA. The validity of the output has been verified by manually reverse engineering the sample.

6.0 Related work

This section presents related work. It covers three areas to which the work in this paper is related. At first related work in the field of algorithm extraction is described in section 6.1. This is followed by work in the field of domain fluxing malware in section 6.2. Finally related work in the field of data flow analysis is presented in section 6.3.

[1] Please note that LICAT.A is also labelled as *Murofet.A* by some antivirus companies.

6.1 Automatic extraction of algorithm from malware binaries

In [17] an approach for algorithm extraction from malware binaries is presented. The presented framework is able to extract proprietary algorithms, which are linked to certain activities of a binary. This includes, for example, domain name generation algorithms or update procedures. The authors of [17] base their framework on dynamic taint tracking for algorithm detection. When using only dynamic analysis techniques like in [17], it is only possible to capture the instructions of one single execution path at a time. This drawback is overcome in this paper by the use of a static analysis technique during the extraction step.

The authors of [18] propose a technique for extracting binary functions. They use a combination of dynamic and static analysis techniques.The approach presented in this paper does not only extract one single binary function, it can extract a whole algorithm which can consist of several binary functions.

6.2 Identifying generated domain names

There has been some work on detecting algorithmically generated malicious domain names in network traffic ([19], [20], [21]). In [19] the authors propose an approach for detecting generated domain names while monitoring DNS traffic. Thus, the presence of domain fluxing bots in a network is detected. While this technique allows the detection of possible infected machines in a network, it does not allow anticipating future malicious domain names for taking global countermeasures against the botnet.

6.3 Data flow analysis

Data flow analysis has been intensively researched during the last decades. It is mainly used during the code optimization step of compilers.

The general technique of data flow analysis has been formulated in 1973 [22].The presented approach for extracting DGAs is based on reaching definitions analysis. There exist several ways to do such an analysis. In [23] those approaches are presented. The paper's approach for extracting DGAs has to find the true origin, termed source, of the generated domain name. Therefore, it uses interprocedural data flow analysis in order to analyze a program on a global scale. Interprocedural data flow analysis is discussed in [24].

7.0 OUTLOOK AND CONCLUSION

This section outlays future work on domain name generation algorithms in section 7.1 and draws conclusions in section 7.2.

7.1 Future work

This section briefly summarizes ideas for future work. At first it addresses questions that are fundamental for future DGA research. This is followed by a summary of future work on the proposed method and framework.

The domain names of non deterministic and time dependent TDN-DGAs cannot be precomputed. The *Torpig* botnet can be named as an example. In *Torpig's* case, the popular trending topics of the social networking service Twitter are used in order to introduce non determinism. Even though those trending topics can not be a priori anticipated, it might be possible to take educated guesses since they are not completely random. In 2012, several big events are scheduled. This includes the Olympic Games in London or the United States presidential election. Those events will most likely influence the trending topics on social networking services like Twitter. Thus, at least for special days it might be possible to anticipate malicious domain names. Consequently this aspect must be researched.

As seen in section 3.2, the proposed approach extracts low level functions. Future work will focus on the

integration of slicing [25] for increasing the extraction precision.

7.2 Conclusion

This paper focuses on domain fluxing malware. This kind of malware employs domain name generation algorithms (DGAs) for generating domain names of possible C&C-servers. Bot herders can make their botnets more robust against takedowns with the help of this technique. In order to take countermeasures against a botnet of this kind, the underlying DGA has to be known.

At first a systematically classification of DGAs is discussed. It can be distinguished between four classes of DGAs. There are two parameters used for the classification of a DGA, namely time and causality. Therefore, a DGA can be either time dependent or time independent and either deterministic or non deterministic.

Furthermore, this paper presents a novel approach for automatically extracting DGAs from malware binaries. At first the DGA extraction problem is formulated and an approach for solving this problem is proposed.

The proposed technique is based on a combination of dynamic and static analysis techniques in order to overcome drawbacks of similar frameworks. An implementation of this approach is presented and its feasibility is shown in two case studies on current malware samples. The proposed approach helps to quickly extract the DGA of a domain fluxing malware sample without any tedious and time consuming reverse engineering. This decreases the time needed to analyze and react to malware. Given todays' amount of malware, this is a crucial step to counter the threat posed by malware.

APPENDIX 1 MD5 SUMS OF THE EVALUATION DATA

The following table contains the MD5 sums of the case studie's samples.

Sample	**MD5 sum**
Conficker	2A6938B042A7A0FD252531D77E409844
LICAT.A	531E84B0894A7496479D186712ACD7D2

Table 2: Samples' MD5 sums

REFERENCES

[1] Panda Security, "Pandalabs annual Report- 2011 summary," 2012.

[2] M. Lesk, "The new front line: Estonia under cyberassault," *Security Privacy*, no. july-aug, p. 76 –79, 2007.

[3] F. Leder, T. Werner and P. Martini, "Proactive Botnet Countermeasures: An Offensive Approach," 2009.

[4] F. Leder and T. Werner, "Know your enemy: Containing conficker," The Honeynet Project, 2009.

[5] F. Guo, P. Ferrie and T.-c. Chiueh, "A study of the packer problem and its solutions," *Recent Advances in Intrusion Detection*, p. 98–115, 2007.

[6] P. Royal, "On the Kraken and Bobax botnets," 2008. [Online]. Available: www.damballa.com/downloads/press/Kraken Response.pdf. [Accessed 06 08 2012].

[7] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel and G. Vigna, "Your Botnet is My Botnet: Analysis of a Botnet Takeover," *Security,* p. 635–647, 2009.

[8] A. Moser, C. Kruegel and E. Kirda, "Limits of static analysis for malware detection," *Computer Security Applications Conference,* p. 421 –430, 2007.

[9] U. P. Khedker, B. Karkare and A. Sanyal, Data Flow Analysis: Theory and Practice, CRC Press, 2009.

[10] "IDA Pro Disassembler," Hex-Rays, 2012. [Online]. Available: http://www.hex-rays.com. [Accessed 06 08 2012].

[11] "MSDN," Microsoft, 2012. [Online]. Available: http://msdn.microsoft.com. [Accessed 06 08 2012].

[12] N. Harbour, "Stealth secrets of the malware ninjas," in *Black Hat USA*, 2007.

[13] C. Linn and S. K. Debray, "Obfuscation of executable code to improve," *Conference on Computer and Communications,* p. 290–299, 2003.

[14] "Conficker Working Group," 2012. [Online]. Available: http://www.confickerworkinggroup.org. [Accessed 06 08 2012].

[15] ThreatExpert Blog, "Domain name generator for murofet," 2010. [Online]. Available: http://blog.threatexpert.com/2010/10/domain-name-generator-for-murofet.html. [Accessed 06 08 2012].

[16] Trend Micro - Trendlabs Malware Blog, "Zeus ups the ante with licat," 2010. [Online]. Available: http://blog.trendmicro.com/links-between-pe licat-and-zeus-confirmed. [Accessed 06 08 2012].

[17] C. Kolbitsch, T. Holz, C. Kruegel and E. Kirda, "Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries," *Security and Privacy,* no. 29–44, 2010.

[18] J. Caballero, N. M. Johnson, S. Mccamant and D. Song, "Binary code extraction and interface identification for security applications," *Electrical Engineering,* 2009.

[19] S. Yadav, A. Reddy and A. Reddy, "Detecting algorithmically generated malicious domain names," in *IMC'10*, 2010.

[20] L. Bilge, E. Kirda, C. Kruegel, M. Balduzzi and S. Antipolis, "Exposure : Finding malicious domains using passive DNS analysis," *18th Annual Network & Distributed System Security Symposium,* p. 1–17, 2011.

[21] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee and N. Feam-, "Building a dynamic reputation system for DNS," *Proceedings of the 19th conference on Security, USENIX Security'10,* p. 18–18, 2010.

[22] G. A. Kildall, "A unified approach to global program optimization," *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, p. 194–206, 1973.

[23] J.-F. Collard and J. Knoop, "A comparative study of reaching-definitions analyses," 1998.

[24] F. E. Allen, "Interprocedural Data Flow Analysis," *World Computer Congress*, p. 398–402, 1974.

[25] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages,* p. 1–65, 1995.