Programming Languages and Target Platforms in Malware - A Trend Analysis

1st
Julian Krieger
IT-Sicherheit
Hochschule München
Munich, Germany
krieger@hm.edu

***Abstract*—The growing field of programming languages influences every software engineer. Never before has there been such an ever increasing number of possible programming language choices when starting a new project. Compared to classic programming languages like C, C++ or even Java, modern choices provide features like simplicity, memory safety or the ability to compile a project for multiple target architectures at once. Indicators like the *TIOBE Programming Community Index* or the *PYPL Popularity of Programming Languages* try to measure popularity trends among programming language choices via surveys or automated statistical analysis on code sharing platforms.**

The availability of a large set of choices has not only affected application engineers but malware authors as well. By researching available malware families and looking at reverse engineering efforts, we hope find trends in programming languages chosen by malware authors. At the same time, we look at malware target platforms in the hope of tracing a possible correlation between the crossplatform compilation feature of modern programming languages like Go and Rust and malware which targets multiple operating systems at once.

***Index Terms*—malware, trend analysis, cross platform malware, programming language trends**

I. Introduction

Malicious software is often not limited to the actual malware binary that ultimately performs the main task for threat actors. In addition to destructive binaries, threat actors may also use open source software intended to aid security researchers for malicious purposes. For example, Command and Control (C2) Frameworks are intended to help red team hackers to aid them in infecting, overtaking and controlling captured systems when penetration testing a company's network. C2s are also maliciously misused by some threat actors to achieve their goals like adding a computer to their botnet or encrypting their victim's system and blackmailing them into paying a ransom to regain access to their files [1]. C2s employ a combination of worker computers and a managing server, from which the former receive commands or payloads (also: *implants*).

To target multiple systems at the same time, malware authors may be interested in tools that are implemented as platform agnostic as possible. A complete set of malicious software may consist of multiple complex systems implemented with varying technologies. Malware implants sometimes also tend to evolve over time. If a malware author finds their efforts at compromising systems thwarted by security researchers or antivirus vendors, they may retrofit their software with a different C2 platform. They may also update their source code and redistribute it, sometimes recompiling their source code to support more aggressive persistence strategies. Alternatively, some opt to rewrite their code entirely in a different programming language, hoping to overcome signature based detection by completely reimplementing core functionality [2] [3]. To assist their efforts in targeting multiple platforms at once, they may choose a modern programming language with the ability to compile binaries for multiple operating systems at a time, further aiding the programmer by supplying wrappers around native operating system libraries.

Malware binaries are almost always found in an obfuscated state, compiled to the binary format of their target architecture. To hide them from operating systems and antivirus defence systems or to defend against reverse engineers, malware authors employ a number of different strategies. They may opt to employ packers to compress their binaries, or even use specialized software to hide textual data, cryptocurrency wallet addresses or command line arguments and configuration parameters used for remotely manipulate their software by embedding these strings in functions that are only evaluated at run time, thus concealing them from static analysis tools.

To successfully analyze a malicious packed binary, their process memory (including data and the program itself) is dumped from a sandboxed environment at run time. Security professionals may then opt to employ reverse engineering suites like Ghidra [4] or IDA [5] to reverse engineer the piece of malware. Tools like these offer a large feature set to simplify reverse engineering work. Among them are features which may depend on knowing a binary's target architecture, compiler or information of the platform the binary was compiled on or debugging symbols. One such example, IDA's *FLIRT*, tries to aid the reverse engineer by recognizing C and C++ standard library functions and to consequently isolate them [6]. This may spare the reverse engineer from doing

unneeded work when reversing a trivial library function that is not part of the malware binary's core functionality.

However, reverse engineering tools and suites rarely adequately support exotic programming languages [2]. Some programming languages may hamper a reverse engineer's efforts. Without a fitting decompiler or disassembler with the ability to understand important compiler metadata, reverse engineering tools lack the ability to gather valuable information about a piece of software. Thus, knowing a malicious binary's original source language before starting the reverse engineering process may provide extremely useful information to tools which aim to achieve functionality similar to FLIRT for programming languages different to C and C++. For example, being able to successfully recognize a piece of malware as being written in Rust, reverse engineering suites may be able to detect a large part of uninteresting core library functions. Those may provide run time functionality like multithreading code, array bound checks or string manipulation procedures which are ultimately uninteresting for the reverse engineer's task. Successfully marking them as library code may aid in reducing a reverse engineer's workload.

This paper aims to determine a statistical trend analysis on programming language choices in malware authorship. If we are able to make an educated guess on the most popular language with malware authors, we may be able to more effectively steer the industry's efforts for providing specialized reverse engineering and analysis tool sets for the most popular upcoming programming languages used for malware.

II. Related Work

There have been a number of past efforts to categorize malware source in correlation with programming languages. Calleja, Tapiador, and Caballero showed an increasing use of C/C++, Visual Basic and Delphi alongside a decrease in the usage of Assembly in the early to mid 2000s [7]. They also showed a growing inclusion of JavaScript, PHP and Python in Botnets and RATS distributed via malware.

Furthermore, there has been past work in trying to find a trend for malware target platforms. By first unpacking their samples and subsequently analyzing the binaries PE headers where available, Plohmann, Clauß, Enders, and Padilla mapped 1792 malware samples categorized into 607 malware families to their target platform [8] . They found that Microsoft's operating systems were by far the most attacked target among their samples: 83.2% of malicious software samples they analyzed were equipped to deal with varying versions of Windows. They then collected their data in an indexing data set with the name *Malpedia*. Malpedia has been bootstrapped at its initialization with data from 2012 to 2017. Further samples are being steadily added to the data set since.

III. Proposed Method

In this paper, we want to achieve two goals: From latest data, we want to gather recent information concerning malware and then check if we can collect that information and subsequently try to draw meaningful trends about the evolution of a) malware target platforms and b) malware architects' programming language choice.

There are multiple approaches to analyzing malware binaries and recognizing their source language. One of them is the open source tool Detect-It-Easy (DIE) [9]. DIE is able to scan a given binary file and will then return information about the packer, the compressing tool or the virtualization software used to obfuscate the binary. Moreover, it can also make a guess based on different criteria about the compiler and linker that was used to translate the original source code into the binary format for the target platform. If the binary is successfully unpacked, DIE is also able to infer the source language by comparing the binary's internal information to a set of rules which may match it to output which is normally generated by a specific programming language compiler. If enough criteria match, DIE will return the most likely source language candidate. For example, DIE matches routines provided by the compiler to safely check and array's bounds on access at run time – among other criteria – to the Rust programming language.

To analyze malware binaries with DIE, one could use a prepared data set and then go on collect information returned by analyzing each binary with DIE's command line tool. We opt to use the *Malpedia* data set, which already contains information of somewhat defused and unpacked malware partially analyzed with DIE. They also combine multiple occurrences of unique malware distributed with different packers into (at the time of writing) 2662 malware families. The data set records information about maliciously acting software from 2017 to 2022, including but not limited to the malware family a specific piece of malicious software might belong to. Currently, they also do provide a description and past analyzation efforts for a subset of their data set. At the time of writing, the binary's source language is missing. The data set features malware implants as well as droppers and command and control frameworks used in malicious ways. Since collecting malware families in the Malpedia data set is an ongoing effort and samples are steadily added to the Malpedia data set, we can have a look at Plohmann, Clauß, Enders, and Padilla's [8] analyses from 2017 and compare them (if applicable) to changes in trends after their paper's publication.

To filter useful information from our data set, we inspect each entry for metadata information, including but not limited to textual information about past analyzing or reverse engineering efforts by third parties. If we can successfully identify for a malware family if a reverse engineer mentions a specific programming language in their past analyzing

efforts, we can try to map an original source programming language to that entry. It is also important to note that some pieces of malicious software employ multiple programming or scripting languages to achieve their target. One such example going by the name of *DarkWatchman* employs the use of JavaScript and C to dynamically compile a C# key logger and will then go on to try to persist the software on the target computer. There is also malware which is first written in one programming language and later is rewritten in a different choice, one example being *IceXLoader*, which was first scripted in AutoIT and later rewritten in Golang [3].

The metadata in the Malpedia data set not only includes information about past reverse engineering efforts, but each malware family's target platform and an *updated* entry, which denotes the date the family was last successfully analyzed. This allows us to try to find interesting correlations between target platforms and programming languages and finally enables finding a trend of attacked target platforms and to subsequently compare these results to the ones mentioned in the original Malpedia paper.

We can also combine date metadata with our gathered matches of malware and source programming languages to analyze the evolution of the popularity of programming languages used for malware in recent years. We will then go on to compare that data to programming language trends in malware in Calleja, Tapiador, and Caballero's [7] paper.

IV. Analysis

To get a first insight into our data set, we extract the target platform from each available piece of malware and look at their total distribution in the data set (see Fig. 1). For a better overview, we combined Malpedia's platforms *win*, *ps1* (PowerShell) and *vbs* (Visual Basic Script) into a single *Windows* platform. The latter two are theoretically usable on Unix or platforms other than Windows, but not without considerable effort or by first installing Windows emulation software like Wine. Since most non-Windows platforms provide scripting language alternatives similar to PowerShell but native to their own operating system and since we can realistically presume that a threat actor would choose an already existing tool to keep as low of a profile as possible, we then go on to assume that if PowerShell is included in a piece of malware, the primary target is most likely a Windows based system. We also combined the Malpedia categories *py*, *asp*, *php* and *jar* into the class *Agnostic*, since their corresponding programming languages Python and PHP or frameworks like .NET and the Java Virtual Machine aim to be cross platform solutions.

Moreover, we can combine rare categories like Symbian (a discontinued mobile device operating system) and NetApp FAS (a data storage system) into the *Other* category. Finally, since malware targeting Windows far outnumbers the combined counts of every other target platform, we display each the number occurrences on malware for each platform

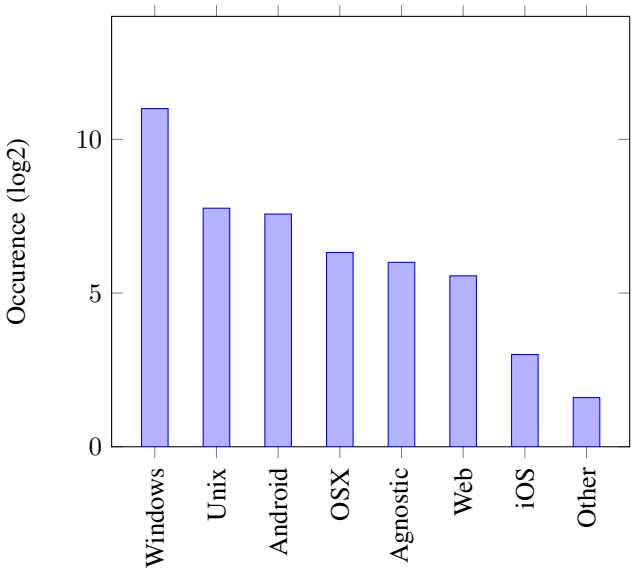

Fig. 1: Platforms Targeted by Malware

in a logarithmic scale to keep them all visible.

We can then go on to have a look at the evolution of the malware attacking the respective platforms over time. This immediately shows a clear picture on how dominating malware targeting Microsoft's operating system is. Even with the growing number of devices with differing operating systems, the number of malware targeting windows systems seems to be growing at a rapid pace. In Fig. 2 we show the evolution of malware targeting platforms different from Windows, by temporarily excluding it from the data set. Devices running OSX or various Android operating systems seem to experience a strong increase in being targeted by malware, though the number of malicious software aiming for OSX or multiple platforms seems to be increasing as well. Malware running on the browser, specifically on websites (for example cryptocurrency miners which try hide in popular open source JavaScript libraries) is also becoming increasingly popular. Interestingly, when comparing the mobile operating systems Android and iOS there seems to be a stark difference: The former is more than 20 times more like likely to be the victim of targeted malware, even though Android devices only outnumber iPhones in a factor of around 3:1. It is unclear where Apple's success in the defense of their mobile operating system comes from, but their effort to only allow verified software from their App Store to run on their devices could be a deciding factor (among others) [10] [11].

Next, we search all malware families' to identify useful metadata within the Malpedia data set and subsequently go on to find entries with existing reverse engineering efforts (as shown in Fig. III). Out of the 2662 samples in the Malpedia data set, 989 (37.5%) have metadata information that can

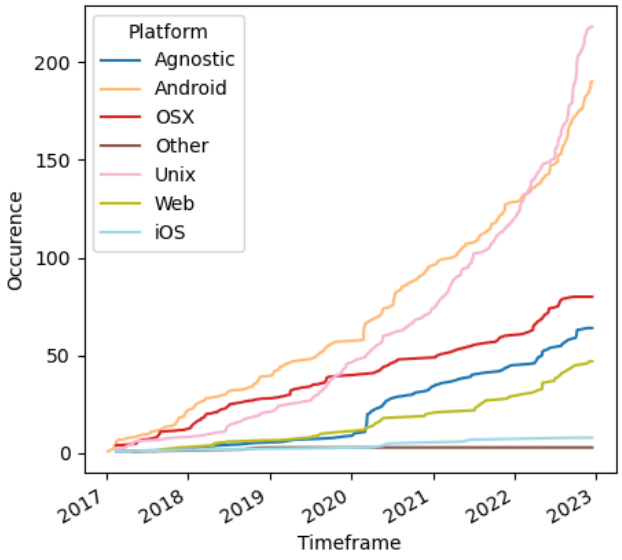

Fig. 2: Evolution of Malware Target Platforms (excluding Windows)

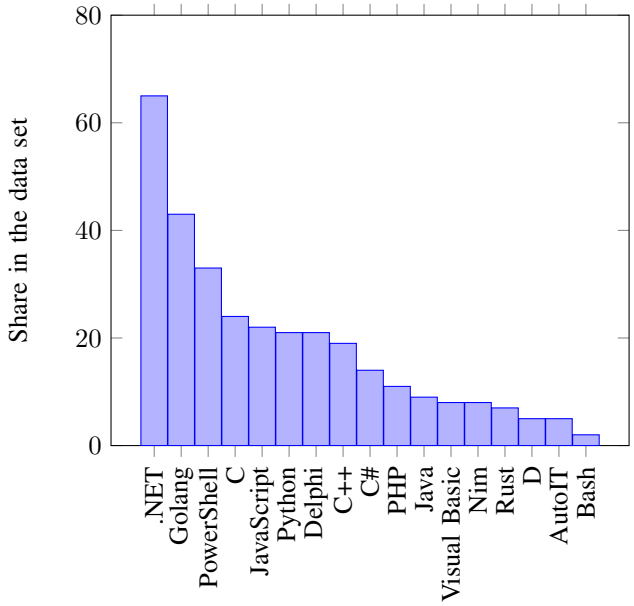

Fig. 3: Programming Languages in Total

be used to discover a possible source language. Next, we need to preprocess the textual information included in the metadata's description field. After modifying our text into lower case, we replace any special characters except the "#" and "+" characters with white spaces. We also keep the term ".net" so we can successfully identify if the description of malware matches languages on the .NET stack. Finally, we can search the description for occurrences of sigils matching programming languages provided we have a large list of the latter. Doing this, we can match possible source languages to 317 families (32% of families with metadata information). We then manually verified each match to check if the programming languages found in the metadata were actually identified as a source language, instead of only being mentioned offhandedly. By doing this, we can filter out all entries where the reverse engineer did not assign a source language to a malware family. In Fig. 3 we can see the distribution of programming languages in the data set by counting the occurrence of every unique source language.

For compiled languages or those shipping with a run time, we can observe that the programming languages .NET and Golang occur most often in the data set. Interestingly, PowerShell scripts are used far more than bash scripts (or any Unix compatible scripting language). This directly correlates to the distribution of target platforms in Fig. 1. The least used languages according to the data set are Rust, D and AutoIT.

By mapping malware family's source languages to the family's update time, we receive a distribution of programming languages used for malware each year (as shown in Fig. 4). Noticeably, we can observe the addition of the previously unrecorded languages Golang, Delphi, PHP, Python in 2018. It also features PowerShell for the first time. In 2019 and 2020 respectively, D and Nim mark their first entries in the data set. Recording of malware written in Visual Basic was initially high, but decreased massively in the following years. Usage of C seems to have had it's highest point in malware that was analyzed in 2018, but after that it has entered a continued decline.

Finally, to get an insight into each programming language's growth from year to year, we calculate how often every language occurs each year and compare that to the previous years (see Fig. 5). In this plot, we can observe a massive increase in the usage of .NET in 2022 compared to previous years. We can also observe that usage of almost all programming languages seems to be increasing, maybe indicating that the distribution of malware is increasing in 2022 altogether. Alternatively, efforts to analyze and add malware to the Malpedia data set could be the reason behind the jump in numbers.

V. Implications

If we take a look at both the distribution and the growth of the different platforms over the time frame from 2017 to 2022 in the Figures 1 and 2 and then go on to compare them to the analyses in the Malpedia paper, we can observe that Windows fell from being the target of 83% of malware to being the destination of 75% of malware in 5 years. There is also a clear trend in malware targeting Unix platforms: The occurrence of malicious software in ELF format increased

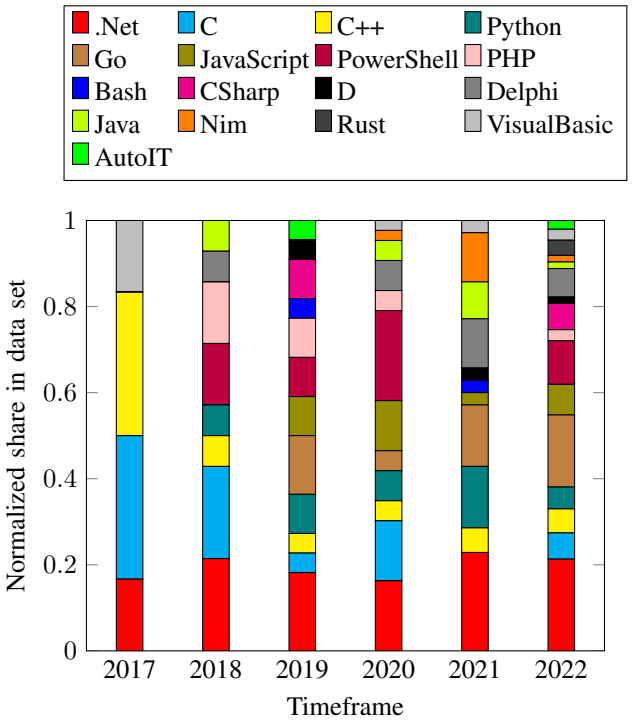

Fig. 4: Programming Language Distribution by Year

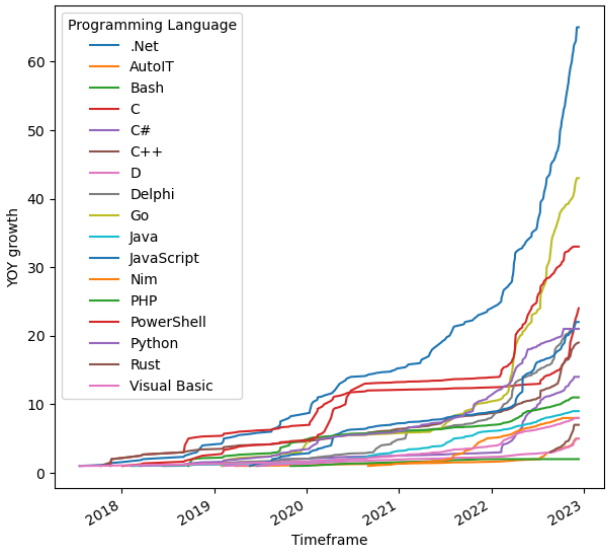

Fig. 5: Evolution of Malware Target Platforms

TABLE I: Comparison of Malpedia's data in 2017 and 2022

Platform	Malpedia 2017	Platforms 2022	Increase
Windows	505	1988	293%
Android	34	190	458%
OSX	29	80	175%
Unix	24	218	808%
iOS	2	8	300%
Agnostic	12	64	433%

TABLE II: Programming Languages used on different Platforms

Programming Language	Windows	Unix 2022	Other
.NET	62	0	2
C	59	8	7
Go	21	14	1
Python	10	2	9
Rust	4	3	0

with a factor of 9 in the 5 year time frame (see Table I). Finally, multi platform and Android malware experiences a strong increase in numbers as well.

We can also compare the original source programming languages we found in our data set to the data in Calleja, Tapiador, and Caballero's [7] paper. They recorded a high number of malware using manual assembly until the mid 2000's. We can further confirm this trend, since there exists only one entry in the Malpedia data set which has been manually written in FASM. They also record a slight increase in the usage of the C programming language, which correlates with the statistics in Fig. 5. Malware authors also seem to have switched from DOS Batch to PowerShell on Windows platforms, since there seems to be no entry in our data in which a batch file could be found. Instead, PowerShell is steadily increasing in popularity. Finally, Golang (which was released in 2012) seems to be picking up speed in 2018 and was virtually unused before. .NET growing as a popular choice is also especially interesting when combined with the fact that virtually every language on the .NET stack is falling in popularity in the benevolent software community [12] [13].

If we look at programming languages that are used for cross platform purposes in Table II, we find that Rust, Python and Golang are especially popular, with half of all malware targeting Unix systems being written in Go. Even though .NET has the capability to be compiled for multiple platforms, virtually every .NET malware explicitly targets Windows platforms. Some malware use the capability of Golang do be easily compiled for multiple systems alike, one example being the RAT *Chaos* [14]. Golang's compiler *gobuild* and its accompanying standard library provide abstractions over system APIs, and the former can compile binaries for a wide array of target platforms without the need of extensive configuration.

VI. Limitations

There are multiple limitations within the data set that need to be mentioned. At a first glance, the chosen Malpedia corpus with (at the time of writing) 2662 families may seem too small to gain insights and to subsequently derive an accurate trend. However, Plohmann, Clauß, Enders, and Padilla argue that the same malware source code is often distributed via a multitude of packers, thus artificially increasing the number of malware in circulation which seem to be unique at first glance. Since the Malpedia corpus is a collection of unpacked pieces of malware, they argue that their corpus is still representative [8]. This effect is further intensified by the fact that not every malware family contained useful metadata.

Moreover, there are some discrepancies between the source language categories C# and .NET. There exist a multitude of languages that ship with the .NET core environment and run time, including but not limited to F#, the *Iron-* versions of Ruby and Python and C# itself, the latter being the most popular language on the .NET stack. It is not known whether researchers could only determine that a piece of malware was using the .NET run time without being able to identify the actual source language.

Finally, since the Malpedia data set only provides us the latest date on which information to a malware family was added or updated, our trend analysis may prove to be unreliable if a large number of data points were updated after a long period of inactivity. Since the Malpedia API does not provide a way to access a malware family's change history, we were unable to identify sample sets which could've been erroneously assigned to the wrong time frame. Instead, we hope to assume that insights on malware arrive relatively close to the malware's appearance.

VII. Conclusion

With the ever increasing number of mobile phones, IoT devices and cloud farms, malware authors are shifting their efforts from targeting Windows to attacking Android and Unix based platforms. However, for the time being Microsoft's Windows seems to still be the most prominent target for malicious software.

References

[1] J. Gardiner, M. Cova, and S. Nagaraja, *Command & Control: Understanding, Denying and Detecting - A review of malware C2 techniques, detection and defences*, arXiv:1408.1136 [cs], Jun. 2015. DOI: 10.48550/arXiv.1408.1136. [Online]. Available: http://arxiv.org/abs/1408.1136 (visited on 12/19/2022).

[2] B. R.
bibinitperiod I. Team, "Old Dogs new Tricks: Attackers Adopt Exotic Programming Languages."

[3] J. S. a. R. Tay, *New IceXLoader 3.0 – Developers Warm Up to Nim — FortiGuard Labs*, en, Section: Threat Research, Jun. 2022. [Online]. Available: https://www.fortinet.com/blog/threat-research/new-icexloader-3-0-developers-warm-up-to-nim (visited on 12/19/2022).

[4] A. N. S. Agency, *GitHub - NationalSecurityAgency/ghidra: Ghidra is a software reverse engineering (SRE) framework*, Mar. 2019. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra (visited on 12/19/2022).

[5] *Hex-Rays Decompiler*. [Online]. Available: https://hex-rays.com/decompiler/ (visited on 12/19/2022).

[6] IDA, *IDA F.L.I.R.T. Technology: In-Depth – Hex Rays*, en. [Online]. Available: https://hex-rays.com/products/ida/tech/flirt/in_depth/ (visited on 12/18/2022).

[7] A. Calleja, J. Tapiador, and J. Caballero, "A Look into 30 Years of Malware Development from a Software Metrics Perspective," en, in *Research in Attacks, Intrusions, and Defenses*, F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 325–345, ISBN: 978-3-319-45719-2. DOI: 10.1007/978-3-319-45719-2_15.

[8] D. Plohmann, M. Clauß, S. Enders, and E. Padilla, "Malpedia: A Collaborative Effort to Inventorize the Malware Landscape," The Journal on Cybercrime & Digital Investigations, vol. 3, 2017.

[9] Horsiq, *Detect It Easy*, original-date: 2014-06-01T21:37:32Z, Dec. 2022. [Online]. Available: https://github.com/horsicq/Detect-It-Easy (visited on 12/17/2022).

[10] Apple, *App security overview*, en. [Online]. Available: https://support.apple.com/guide/security/app-security-overview-sec35dd877d0/1/web/1 (visited on 12/18/2022).

[11] M. S. Ahmad, N. E. Musa, R. Nadarajah, R. Hassan, and N. E. Othman, "Comparison between android and iOS Operating System in terms of security," in *2013 8th International Conference on Information Technology in Asia (CITA)*, Jul. 2013, pp. 1–4. DOI: 10.1109/CITA.2013.6637558.

[12] T. S. EV, *Tiobe Index*, en-US. [Online]. Available: https://www.tiobe.com/ (visited on 12/19/2022).

[13] *PYPL PopularitY of Programming Language index*, en. [Online]. Available: https://pypl.github.io/PYPL.html (visited on 12/19/2022).

[14] *Linux Cryptocurrency Mining Attacks Enhanced via CHAOS RAT*, en-US, Section: research, Dec. 2022. [Online]. Available: https://www.trendmicro.com/en_us/research/22/l/linux-cryptomining-enhanced-via-chaos-rat-.html (visited on 12/19/2022).