Spring 3-2024

A Cross-Architecture Framework for Anti-Debugging Techniques

Austin Norby

A CROSS-ARCHITECTURE FRAMEWORK FOR ANTI-DEBUGGING TECHNIQUES

A dissertation submitted to Dakota State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Cyber Operations

March, 2024

By
Austin Norby

Dissertation Committee:

Tyler Flaagan, Ph.D.
Yong Wang, Ph.D.
Mark Spanier, Ph.D.

DAKOTA STATE UNIVERSITY

DISSERTATION APPROVAL FORM

This dissertation is approved as a credible and independent investigation by a candidate for the Doctor of Philosophy degree and is acceptable for meeting the dissertation requirements for this degree. Acceptance of this dissertation does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department or university.

Student Name: Austin Norby Student ID: 101070802

Dissertation Title:
A Cross-Architecture Framework for Anti-Debugging Techniques

Graduate Office Verification: DocuSigned by: Abby Chowning F44C8D9E621C417... Date: 04/29/2024

Dissertation Chair/Co-Chair: DocuSigned by: Tyler Flaagan 0DA9F3A62ABD484... Date: 04/30/2024
Print Name: Tyler Flaagan

Dissertation Chair/Co-Chair: _____ Date: _____
Print Name: _____

Committee Member: DocuSigned by: Yong Wang 70AB505BC7B649E... Date: 04/30/2024
Print Name: Yong Wang

Committee Member: DocuSigned by: Mark Spanier 55905BBEC9A7414... Date: 04/30/2024
Print Name: Mark Spanier

Committee Member: _____ Date: _____
Print Name: _____

Committee Member: _____ Date: _____
Print Name: _____

Submit Form Through Docusign Only
or to Office of Graduate Studies
Dakota State University

ACKNOWLEDGMENT

I would like to acknowledge at least the following people for their unwavering support in this endeavor and without them I would never have made it this far:

- I would like to thank my wife for her dedicated support during the many instances of late nights, travel, and time spent focusing on this research. You are my foundation, and I could not have done this without you.
- I would like to thank my committee for helping me take this idea and refine and review until it became the paper that it is today.
- I would like to thank all the cited authors for their work in this area. I am humbly aware that I am standing on the shoulders of giants and hope to provide a foundation for future researchers down the road.

ABSTRACT

Anti-debugging techniques are often used in malware samples and software protection frameworks. From the malware perspective, the malware author includes these checks to hinder the analysis of the sample to increase the time to affect their target. Software protection frameworks on the other hand do this to protect sensitive information such as intellectual property, music, movies, and other proprietary information. The anti-debugging techniques serve to deter reverse engineers from compromising the sensitive data in this case. Modern anti-debugging research focuses on creating new techniques or defeating techniques during analysis. Additionally, modern anti-debugging research is paired with other anti-analysis techniques and used as a component in novel software protection frameworks. This dissertation seeks to better understand anti-debugging techniques in terms of CPU execution time and provide an artifact for anti-debugging identification within target executables based on performance data. This research will use a design science methodology to investigate performance data anti-debugging techniques and generate an artifact that can inform cybersecurity defenders to the presence of anti-debugging techniques based on performance data and statistical tests. The results of this research will be implementations of the anti-debugging techniques, many datasets representing the performance data of Anti-Debugging techniques, statistical results regarding the difference in performance of the anti-debugging techniques compared to a control group, and an artifact that can identify the presence of anti-debugging techniques by measuring the similarity of data variances in performance datasets. In conclusion, the generated artifact was able to detect 27 out of the 58 total techniques across operating systems and architectures with any degree of sensitivity. Out of those 27, seven were significantly less sensitive to changes in execution duration and detected every time in the artifact experiment.

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

_____

<Student name>

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Background of the Problem

When someone is learning to write code, it is often incomplete and contains errors or bugs. These problems are recognized during compilation or runtime execution and then are subsequently fixed. Often, many bugs are non-trivial to diagnose and fix and thus require the help of a program called a debugger. A debugger is a program that will execute another program step by step such that someone, or something, can intuit about the underlying execution environment incrementally rather than only receiving Boolean results; the application runs correctly, or not.

A debugger often has a user-interface that a human, often a developer, tester, or reverse engineer, will interact with to better understand the incremental execution of the program under test, or debuggee. This user-interface could be in the terminal, like GDB (the GDB developers, 2023), or a desktop application, like WinDbg or IDA Pro (DOMARS, 2023; Hex-Rays, 2023), or even a web-based application like the Compiler Explorer (Godbolt, 2023). These types of debuggers are separated into two distinct categories: static and dynamic.

Static debuggers use well-understood structures applied to the binary to analyze and structure the program information in a more visually appealing, human-understandable format. It can break the executable up into programmatic sections, it can resolve function calls to names or symbols, it can apply well-known structures to make program instructions easier to understand, and more. The key feature of a static debugger is that it does not execute the code. This means whatever bytes are in the executable file on disk are loaded into the static debugger. Some modern debuggers can apply programmatic transforms to the data for improved analysis, but this is additional analysis code from the debugger and not any execution of the debuggee.

Dynamic debuggers are different from static debuggers because they load the program into memory and begin executing the debuggee. For the dynamic debugger to halt execution

for analysis, the debuggee must execute a breakpoint instruction. This instruction tells the CPU to interrupt execution and pass the breakpoint interrupt signal to the program to handle it. The debugger will capture this breakpoint interrupt signal and pause the execution of the debuggee. Once the debuggee is paused, the debugger will populate as much information as possible about the execution context of the debuggee to inform the user about the current execution state of the program. This information commonly includes data from memory, register values, the instruction pointer, and instructions near the current instruction. While this is the common information populated by the debugger, at this point the debugger typically has access to all memory within the debuggee process to both read and write data or instructions.

Another consideration for dynamic debugging is that some software authors do not want anyone to debug their code. This can lead to negative consequences for things like malware but, desirable outcomes for copyright protection and digital rights management (DRM) (Bahaa-Eldin & Sobh, 2014). Malware has been known to do almost anything after detecting it is being debugged. Some malware simply displays an error message and does not execute to avoid being detected (Onofri, 2022), but other malware has been known to be destructive upon detecting that it is being debugged (*Update*, 2022). This kind of destruction ranges from simply shutting down the computer to destroying data on the system. From the reverse engineering perspective, protection mechanisms related to copyright protection and DRM are supported via laws that attempt to dissuade anyone from looking at, not only the code, but also the data. In the United States, copyright infringement can carry up to a 5-year felony punishment (*1847. Criminal Copyright Infringement -- 17 U.S.C. 506(a) And 18 U.S.C. 2319*, 2015; *Title 17 - Copyrights*, 2021, p. 17; *Title 18 - Crimes and Criminal Procedure*, 2021, p. 18).

So how does a program determine if it is in a debugger? The way the debuggee can detect a debugger is by the software author implementing an anti-debugging check. Anti-debugging is a category of techniques in and of itself but also is an umbrella for techniques including anti-virtualization, anti-disassembly, anti-forensics, anti-emulation, and generic detection evasion (Roccia & Lesueur, 2023; Shi & Mirkovic, 2017). Anti-debugging techniques are varied and can be unique to operating systems or architectures. In general, anti-debugging techniques look for discrepancies in the execution environment when the debuggee is running as a debuggee versus as a regular process. Windows has many data structures

within process memory that contain different values when a debugger is being used compared to when it is executing the program normally. These values are available to the debuggee to be accessed and read. In Linux, the ptrace() functionality is commonly used and a simple attempt to use it can sometimes be sufficient to determine if the process is being debugged. Attempting to use this function is successful because it can only be called once per process so, if the debugger already called it then when the debuggee calls ptrace() for a second time, it will fail. This failure will alert the debuggee that it is likely being debugged (kirschju, 2018; Miller, 2005; Wan et al., 2018).

Anti-debugging research and documentation was published as early as 1999 for the Linux operating system and 2005 (Eilam, 2005) for Windows. Despite these techniques having been around for almost twenty years as of this writing, there is still a lack of research in anti-debugging performance. Some anti-debugging techniques have more computational or memory impact on the execution of the protected program than others while also providing differing levels of obfuscation, hinderance, and protection for the protected information.

Statement of the problem with motivation

There is minimal research on the performance aspects of anti-debugging techniques. Some related cybersecurity research includes performance measurement of software protection frameworks (Bahaa-Eldin & Sobh, 2014), and executable packing with virtualization (M.-J. Kim et al., 2010). Another paper, (Apostolopoulos et al., 2020), included some statistical measurements of the differences between their packed sample and the original including memory usage, time overhead, file entropy, and increased amount of code in the executable. Closer still, in one cybersecurity paper, the author uses benchmark frameworks to measure the performance of their rootkit (Spisak, 2016).

This paper seeks to measure a wide range of anti-debugging techniques' performance overhead for CPU execution time for Windows on the Intel architecture and Linux operating systems on the Intel and ARM architectures. In addition, the research will create an artifact that will use performance results to identify the presence of anti-debugging techniques within a binary. The ability to detect anti-debugging capabilities using only performance features will provide insight into the binary for malware analysts and cybersecurity defenders regarding the binary sample's anti-debugging capabilities. In addition, the measurement of

performance is a non-invasive way to measure the impacts of Anti-Debugging techniques on a sample. Having a non-invasive strategy to detect Anti-Debugging techniques decreases the likelihood that Anti-Debugging techniques will detect it and change their behavior.

Performance measures provide concrete data on the impacts of the anti-debugging techniques on their protected program that is not subject to human bias, or reverse engineering skills. In addition, the artifact will provide insight into the functionality of the binary related to its anti-debugging capabilities. This research will answer the following questions related to the impact of anti-debugging techniques:

- What is the performance impact of an anti-debugging technique on program execution time across the Windows and Linux operating system for Intel and ARM architectures?
- Can an artifact be produced to identify the presence and capabilities of anti-debugging techniques using performance data?

Objectives of the project

The objective of this research is to investigate 58 anti-debugging techniques on both the Windows and Linux operating systems and on both the Intel and ARM architectures. Windows provides different user-mode API functions to access different information than the Linux operating system which leads to slightly different anti-debugging techniques between operating systems. For instance, Windows includes a function, IsDebuggerPresent(), which retrieves a value from process memory in order to determine if a debugger is present ("IsDebuggerPresent Function," 2021; Shi & Mirkovic, 2017). IsDebuggerPresent() does not exist on Linux as they are different operating systems and instead additional steps must be taken to find and query similar information and answer the same question in the same manner.

Some techniques are similar algorithmically between the operating systems but require different system API functions to complete. One example of this type of anti-debugging technique is querying the parent process name. Both Windows and Linux allow a process to query its parent's process name and to complete the technique the process can check the retrieved name against known debugging or reverse engineering programs (Xu et al., 2016).

While there are differences between operating system techniques like the aforementioned examples, there are operating system independent checks like checking the

value of the Trap Flag (TF) which is commonly turned on during single step debugging. Techniques like checking for parent processes and reading the Trap Flag provide interesting differences and quantitative analysis opportunities for better understanding anti-debugging techniques across operating systems (*Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023; Shi & Mirkovic, 2017; Vasudevan & Terraballi, 2005).

There will be architectural differences and similarities between the ARM and the Intel anti-debugging techniques such as different instruction sizes, timings, and available registers. These components will provide different information within different times while measuring anti-debugging checks. Similarities exist when using architecture level instructions like timing checks and reading the Trap Flag. The Trap Flag is the name for the Intel architecture implementation but on ARM it uses the Single Step (SS) flag for the same purpose. Similarly, instead of RDTSC on Intel, we can read from the PMSCR register, for example (*ARM Developer Documentation*, 2023).

Each of these techniques are implemented in C, C++, and Assembly as needed to minimize any overhead. The timing tests will use the CPU Time-Stamp counter to minimize overhead (Paoloni, 2010). Once the timing values have been captured, the values will be written to a log for later processing and statistical analysis. The memory tests will wrap memory allocation routines for the technique and write details to a log which will be used to calculate the memory overhead and compute statistical analysis, as well.

By implementing code as close to the CPU as possible, the test will eliminate as much overhead as possible while timing execution which give us the most accurate results. Once the artifact captures the results, the timing no longer needs to be optimal for writing data to a log file. Similarly, once the memory tests are complete and data captured, writing data to the log no longer needs to be optimal from a memory management perspective.

Implementing the artifact will be done in Python. The artifact will process the performance data from the Anti-Debugging techniques and test the provided sample performance against each of the models. Using the Levene test, the artifact will measure similarity between distributions of data based on the variance (Brown & Forsythe, 1974; Levene, 1960; “Levene Test for Equality of Variances,” n.d.). The results will be used to determine the likelihood of anti-debugging techniques causing the performance discrepancy based on the results of the original performance data distribution. The statistics will be

calculated and displayed to the user including the p-value which measures the likelihood that an anti-debugging technique is present given the similar distribution of performance data.

This paper shows quantitative analysis on the performance data of the anti-debugging techniques and group them by technique, measurement, operating system, and architecture. Each of these categories will measure the impact of each of the anti-debugging techniques against a base sample of code to show the performance impact. In addition, the results and code will be contributed into the open-source community after publication for further research, awareness, improvements on implementation, and detection, of anti-debugging techniques. Furthermore, the artifact will the performance data of the target executable to identify anti-debugging presence within the target executable based on performance data.

Limitations

This research is limited to two different operating systems with different kernels and system APIs, namely Ubuntu Linux and Microsoft Windows, and two different architectures, namely Intel and ARM. It is beyond the scope of this research to consider other architectures such as PowerPC, MIPS, and SPARC. It is also outside the scope of this research to consider alternate operating systems and kernels such as the community developed ReactOS, the multiple BSD kernels (FreeBSD, OpenBSD, and NetBSD), Solaris, and MacOS (XNU kernel) (“Comparison of Operating System Kernels,” 2023). Lastly, it is also outside the scope of this research to measure kernel-mode, or privileged, Anti-Debugging techniques. Similarly, no measurements were made using kernel-mode, or privileged, code. This has the consequence of being unable to reserve a CPU for exclusive execution of the code as described in Paoloni (Paoloni, 2010).

In addition, this research has a limitation on the size and execution duration of the programs able to be analyzed. While Anti-Debugging techniques do leave a performance fingerprint on the execution time, the natural variance of executions times for larger or longer running programs will overwhelm the variance introduced by the Anti-Debugging techniques and the artifact fails to identify Anti-Debugging presence in those types of samples.

CHAPTER 2

LITERATURE REVIEW

Foundations of Anti-debugging

The furthest back results for anti-debugging research are cited around the 1999 timeframe for Linux with an almost lost manuscript (original hosted content is offline), “LINUX ANTI-DEBUGGING TECHNIQUES (FOOLING THE DEBUGGER)” (Cesare, 1999). Only a few years later in 2005, (Eilam, 2005) was published discussing techniques for Windows. These seminal works discuss the foundations of anti-debugging research for the Windows and Linux operating systems. The original PTRACE_ME technique is discussed in (Cesare, 1999) along with anti-disassembly techniques to fool disassemblers, scanning code for breakpoints, and using signals to handle false breakpoints to detect the presence of a debugger. Similarly, in (Eilam, 2005) the IsDebuggerPresent API is discussed along with NtQuerySystemInformation calls for debugging information, single-byte interrupts, checking the Trap Flag, and code checksums. All these techniques are being measured as part of this research. Years after the original anti-debugging paper for the Linux operating system, a follow-up paper on anti-anti-debugging is published in CodeBreakers magazine using the same anti-debugging techniques cited previously but also including circumventions for the published anti-debugging techniques (Schallner, 2006). This kind of back-and-forth research continues to the present day leading to the concept of stealth debuggers and the pursuit of new anti-debugging techniques (Lee et al., 2013; *ScyllaHide*, 2023; Shi & Mirkovic, 2017). These techniques and historical papers build the foundation for anti-debugging research but unlike this research do not focus on the performance of anti-debugging techniques.

Anti-analysis as a super-set of Anti-debugging

Anti-debugging is only one type of a broader range of techniques called anti-analysis techniques. This broader range of techniques includes anti-disassembly, anti-virtualization,

anti-forensics, anti-dumping, and anti-monitoring (Li et al., 2015; Roccia & Lesueur, 2023). Often these techniques span multiple types and are included in research, malware, and software protection together. (Apostolopoulos et al., 2020; Aycock et al., 2009; Bahaa-Eldin & Sobh, 2014; Collberg & Thomborson, 2002; Guoyuan et al., 2010; Li et al., 2015; Onofri, 2022; Wan et al., 2018; Xu et al., 2016). One of the most common types of anti-analysis is anti-disassembly. This commonality comes from the ease and flexibility of implementation. Using techniques that hinder disassemblers by exploiting either the linear sweep or the recursive descent algorithms will hinder understanding using those tools depending on which algorithm it uses (Eilam, 2005; Sikorski & Honig, 2012). In (Aycock et al., 2009), the authors suggest a Pseudo-Random Number Generation (PRNG) scheme that hinges on a secret seed for the generator that greatly hinders the ability to disassemble the code while the secret is not known. The next anti-analysis technique used more often within malware is anti-virtualization. These techniques check if the malware sample is running inside of a virtual machine rather than a debugger. Anti-virtualization checks are often derived from the original blue-pill research (Rutkowska, 2016) which compares the results of certain virtualization assembly instructions that differ in results when executed on a physical CPU vs. a virtual CPU. These differences can be due to different implementation, lack of implementation, or simply inconsistencies on the virtualization platform compared to a physical system (X. Chen et al., 2008). While these techniques will not directly be measured in this paper, they are tightly linked to anti-debugging when studying cybersecurity, malware, and software protection mechanisms, as seen in the cited research. Papers like the blue-pill research and other anti-analysis techniques are novel and introduce new techniques. In this research we are not discovering or releasing new techniques and instead focusing on the performance and detections of these techniques.

One key component of dynamic analysis on malware, packers, and Anti-Debugging techniques in general is the use of the Pin framework provided by Intel. This framework provides Dynamic Binary Instrumentation (DBI) which provides the capability to intercept instructions, function calls, and analyze execution at an extremely low level. In Park et al., they analyzed 29 different Anti-Debugging techniques and their ability to detect them during execution using the Pin framework and were successful implementing detections for 21 of the techniques. This research specifically was focused on the unpacking of packed executable by

17 common executable packers (Park et al., 2019). Yet another framework, PINvader, was focused on detecting and handling Anti-Debugging techniques also using instrumentation provided by the Pin framework. The PINvader framework was designed to classify and detect Anti-Debugging and Anti-Virtualization techniques on the Windows 64-bit operating system. The PINvader framework uses function and instruction hooks to bypass the Anti-Debugging and Anti-Virtualization techniques so analysis can be conducted rather than hindered by the techniques. Their research focuses on classification and performance overhead for all types of Windows 64-bit applications rather than specifically malware analysis or software protection (Gozzini, 2022).

Anti-debugging does not only need to be implemented in user-mode (Ring 3), it can also be implemented in kernel-mode (Ring 0) for additional protections. These additional protections come with significantly higher costs compared to user-mode implementations. Specifically, within the Windows environment, developers cannot simply install kernel drivers without either disabling security features or having the driver signed by Microsoft at the time of this writing. Once either of those criteria is met, one must still have administrative permissions to install a kernel driver. This creates significant hurdles for both malware authors and software protection developers which may explain why kernel drivers are seen less often in the wild. There has been research on this from (Yi et al., n.d.) where a driver was implemented to handle exceptions that reached the kernel and not return the required information to the debugger thereby bypassing the debugger. This research was published in 2009 during which Windows had fewer security mechanisms in place to prevent this kind of installation of a driver however, the overall technique remains a valid pursuit of anti-debugging techniques but, will not be discussed in this paper.

Anti-debugging techniques

Anti-debugging techniques specifically are studied to understand malware and software protection mechanisms. For this research, anti-debugging techniques are sorted into the following categories for Windows: exception-based, API calls, scanning, timing, and miscellaneous. The miscellaneous category includes self-debugging, an instance of anti-disassembly via obfuscation, and memory encryption.

Exception-based Anti-debugging Techniques

Exception-based anti-debugging techniques are often used because while a program is being debugged on the Windows operating system, the debugger will receive the exception before the program. If the debugger handles the exception before the program can, this is a sure-fire way to detect that a debugger is present on the system. There are many ways to trigger an exception within Windows. There exists a single-byte assembly instruction on the Intel architecture that will result in an exception being thrown like INT3. There also exist multi-byte instructions that result in an exception being thrown like INT 0x2d or setting the Trap Flag. Moving past the architecture specific instructions that cause exceptions to be thrown, there are operating system mechanisms that will cause exceptions to be thrown as well. One such technique is using the Windows API CloseHandle on an invalid handle. This will generate an exception that will first be handled by the debugger and then by the program alerting the program to the debugger's presence (Bing et al., 2010; Roccia & Lesueur, 2023; Saad & Taseer, 2022; Shi & Mirkovic, 2017; Vasudevan & Terraballi, 2005). Causing exceptions to occur and identifying if a debugger is present to handle the exceptions is a well-established methodology for anti-debugging techniques. Using these techniques in addition to others, this research will measure the timing and memory performance impacts of these exception handling routines to determine the impact of each technique on the running program.

API-based Anti-debugging Techniques

Windows also has a class of anti-debugging techniques that use API calls that return values or structures regarding the debugging state of the program. The most well-known is IsDebuggerPresent which simply returns true or false if the program is being debugged. There also exists less common API calls that will also return the debugger status such as NtQueryInformationProcess, OutputDebugString, and CheckRemoteDebuggerPresent (Apostolopoulos et al., 2020; X. Chen et al., 2008; "IsDebuggerPresent Function," 2021; J.-W. Kim et al., 2019, 2020; Roccia & Lesueur, 2023; Saad & Taseer, 2022; Shi & Mirkovic, 2017; Shields, 2010; Sikorski & Honig, 2012; Xu et al., 2016; Yoshizaki & Yamauchi, 2014). These techniques are used in modern malware and software protection frameworks to deter analysis of the executables. Despite the ubiquity of these checks, the performance impact is

not measured for these API calls to determine if the resource usage should be considered for implementation of these techniques. This research seeks to quantify the impact of these API calls, and others, to determine the timing performance overhead within an application.

Timing-based Anti-debugging Techniques

Timing analysis is another common anti-debugging technique that takes advantage of the speed of CPU execution compared to the amount of time it takes a human to analyze executable code. These time differences are stark with modern CPUs executing in the GHz operations per second range compared to seconds, minutes, or hours it might take a human to analyze the same code. Different implementations of the timing check exist, some execute the timing check back-to-back and measure the difference looking for single-stepping execution (Shields, 2010) while others execute instructions between timing calls looking for debugging behavior across a larger section of the code (M.-J. Kim et al., 2010). These techniques can use various API calls or even the RDTSC instruction on Intel architectures to measure the timing of code execution. (Apostolopoulos et al., 2020; M.-J. Kim et al., 2010; Saad & Taseer, 2022; Shi et al., 2017; Shi & Mirkovic, 2017; Shields, 2010; Yoshizaki & Yamauchi, 2014) While the technique involves measuring the timing of execution, the performance of the timing checks algorithms has not been considered. These timing techniques including API usage and direct assembly instruction usage will be measured as part of this research.

Scanning-based Anti-debugging Techniques

Another common category of anti-debugging techniques is scanning for information related to a debugging environment. This includes the usage of APIs looking for known processes that are associated with debuggers and scanning for breakpoints, or any modifications, inserted in the code by a debugger. Other modifications can be caught by calculating a checksum on a function or series of bytes and if the checksum does not match, the code has been modified. (Bahaa-Eldin & Sobh, 2014; Giffin et al., 2005; Shields, 2010; Stone et al., 1998) Given that these techniques process many bytes and instructions, it can lead to a measurable performance decrease depending on the usage and size of the scan. Other scans looking for well-known debugging processes also require processing a lot of information like process names or window titles (Bahaa-Eldin & Sobh, 2014; M.-J. Kim et al., 2010; Saad & Taseer, 2022; Shi & Mirkovic, 2017; Shields, 2010; Yoshizaki & Yamauchi,

2014). It is expected that scanning techniques will take longer than other exception-based or API call techniques, but it is unknown to what extent the scanning techniques will affect performance. This research seeks to understand the timing performance impacts of scanning techniques, while using them for anti-debugging purposes.

Additional Anti-debugging Techniques

The last category of anti-debugging techniques for this research will contain three miscellaneous techniques that don't fall into the aforementioned categories. These are anti-disassembly techniques, data encryption, and self-debugging. The only anti-disassembly technique considered for this research is code obfuscation. This includes function name mangling and including necessary instructions to hinder debugging or reverse engineering (M.-J. Kim et al., 2010; Miller, 2005; Wan et al., 2018). Other research used programmatic methods to generate obfuscation that is easier to implement but more difficult to debug. In Aycock et al, a Pseudo-Random Number Generator (PRNG) is used to map instructions from one value to another to hide the actual logic of the code behind the random number. This random number acts like a key to undo the mapping between unknown instructions to the original instructions (Aycock et al., 2009). The last technique mentioned is using pointer arithmetic within the code to deceive reverse engineers and hinder debugging by computing the desired function pointer from another one. This will confuse debuggers and reverse engineers because the expected function is not called but is used as part of the computation to create the correct function address and call it (Gan et al., 2015). These techniques can vary widely in implementation and for this research we are looking at a single implementation of them to establish a baseline for performance impact as anti-debugging techniques.

Anti-debugging Techniques from a Malware Perspective

Malware takes advantage of anti-debugging techniques to hinder analysis and increase its longevity while attacking its targets. This naturally follows from the analysis of the malware, once caught, takes longer than samples without anti-debugging techniques to understand and develop signatures for (Guttman, 2017). Malware will even go a step further and attempt to be destructive to the system or debugger attempting to analyze the sample by exploiting known issues within debuggers which cause the debugger to crash (Sikorski &

Honig, 2012; *Update*, 2022). Analyzing malware is where the stealth debuggers come into play to bypass the anti-debugging, or generally anti-analysis, techniques. When the stealth debugger can pre-empt or dynamically handle the anti-analysis technique and return execution to a benign state the analysis of the malware sample can continue more quickly (*ScyllaHide*, 2023; Shi & Mirkovic, 2017; Vasudevan & Terraballi, 2005; Vasudevan & Yerraballi, 2006).

Some novel research looked to avoid the possibility of malware detecting its analyzing capabilities by building it outside of the environment and using the virtualization/hypervisor layer to conduct analysis. The "ether" capability uses Intel VT and the Xen hypervisor to monitor system calls, memory writes, and more to analyze malware samples in their virtualization solution. This solution was used to analyze 25,000 malware samples and demonstrated great effectiveness for analysis and not being detected by the malware. As one might expect, this did come with a performance penalty as all additional execution time must (Dinaburg et al., 2008). Related to virtualization, Qi et al. proposed a hardware virtualization artifact based on BluePill to act as a thin hypervisor layer to detect specific activity as opposed to debugging the entire system. It intercepted potential debugging and Anti-Debugging events such as the INT1, INT3, and CPUID instructions and hooked function entry and exit points. Lastly, it uses additional memory management privileges while operating at the VMM layer to hide itself from programs that attempt to detect it, including using kernel privileges, so the guest operating system cannot detect it (Pék et al., 2011; Qi et al., 2012). This research takes a similar non-invasive approach to measuring the impact of the Anti-Debugging technique without interacting directly with the sample. While these solutions use virtualization and hypervisors, the non-invasive component is similar between the aforementioned research and this research.

Not all research focuses on defeating or bypassing Anti-Debugging techniques, there is still research on malware that measures the presence and changes of these techniques in malware. Based on research done in P. Chen et al. between 41% and 89% of samples in the malware families analyzed had Anti-Debugging techniques implemented in the malware samples. In addition, samples have on average 1-3 techniques implemented and have a lower detection rate against Anti-Virus solutions compared to malware without these capabilities. Lastly, from this research, according to their statistical analysis of the data, more Anti-

Debugging techniques are being used over time consequentially increasing the difficulty in analyzing these samples over time without additional research (P. Chen et al., 2016).

Anti-debugging Techniques from a Software Protection Perspective

Malware is not the only use for anti-debugging techniques, many software protection mechanisms also make use of these techniques to safeguard intellectual property and digital rights such as music, movies, and video games. These frameworks and applications use anti-debugging techniques to hinder and thwart the attempt to steal the information from the program itself or the management program, such as a music or movie player. One example of a framework that seeks to armor an executable is ANTI. The ANTI framework makes uses of anti-debugging and anti-virtualization checks to prevent analysis of the armored code (Apostolopoulos et al., 2020). Common tactics for software and intellectual property protection include code obfuscation, anti-debugging techniques, cryptography, and remote code distribution. While it is impossible to completely hide the information, as it must be read or executed by the system to have any value at all, significant hurdles, cryptography, and remote code distribution raise the bar significantly to thwart theft of the information. In addition to the technical solutions, a policy and rights management platform is included in Bahaa-Eldin and Sobh's 2014 paper to add an authorization and authentication component to the intellectual property protection beyond just technical hinderances (Bahaa-Eldin & Sobh, 2014; Collberg & Thomborson, 2002; Gagnon et al., 2007; Guoyuan et al., 2010).

Cybersecurity Performance Measurement

This research focuses on the performance of these anti-debugging techniques, but performance is not often studied in cybersecurity except when adding a burden to the original execution and measuring a minimal overhead is a positive attribute of the research or artifact. One example of this is the Bahaa-Eldin and Sobh paper that proposes a software protection mechanism. They use dynamic infection and code and code redirection to automatically protect the Portable Executable files. They measure the overhead of their artifact during initial execution while it unpacks and then compare algorithm tweaks for different obfuscation and performance attributes. The results are measured as a percentage increase of the original

execution time as opposed to absolute values. The results were an acceptable blend of security and performance with a 6.5% increase in execution time over the unprotected version (AbdelHameed et al., 2009; Bahaa-Eldin & Sobh, 2014).

During the measurement of their stealth debugger, Cobra, Vasudevan and Yerraballi considered the performance impacts of modifications while analyzing their malware sample, Win32/Ratos. They break down their overarching artifact, Cobra, into specific components and measure the execution overhead, or latency, of those components using clock cycles. They used four samples with different components tuned for analysis but ultimately the execution overhead, or latency, is highly dependent on the samples being analyzed due to the requirements in their stealth debugger to handle these cases (Vasudevan & Yerraballi, 2006). Other malware artifacts have minimal to no performance considerations with anecdotes such as, "…overhead to the system is negligible" when considering the impact of executing malware under the presence of debuggers (X. Chen et al., 2008).

Another paper measures the overhead of packing an executable with different algorithms such as UPX and AES. The UPX tool uses compression and manipulates the PE file to obfuscate and shrink the original executable. It also has a smaller performance overhead than AES. The AES packing simply encrypts the original file and does not extend beyond the length of the executable resulting in no change in size but a large increase in performance (M.-J. Kim et al., 2010). Another similar measurement was done by the authors of the ANTI framework that found minimal overhead on their armored binary. The measured a mean memory increase of 1.22MB, a mean timing increase of 2.21ms, and only a 1.32% mean increase in code size (Apostolopoulos et al., 2020).

The PINvader paper describes an artifact that's used for dynamic analysis of binary applications for measuring Anti-Debugging and Anti-Analysis techniques on the Windows 64-bit operating system. They built their PINvader artifact on top of the Pin capability which provides Dynamic Binary Instrumentation (DBI) from Intel. For their performance measuring, they use the GetTickCount Windows API. The authors of PINvader considered three different scenarios for performance measurement, normal execution, PINvader attached but with disabled hooks, and full analysis. The PINvader capability added, on average, 26.85 seconds of execution time to the normal execution with an additional 2 seconds, on average, for full analysis (Gozzini, 2022).

Separate from cybersecurity specifically, performance can be hard to measure as modern architectures are extremely complex. Intel put out a white paper on the best way to measure performance of code at the lowest levels of the system. This involves turning off some optimizations, forcing execution on a single CPU, and preventing speculative execution before or after reading the timing counter (Paoloni, 2010).

CHAPTER 3

RESEARCH METHODOLOGY

Overview

This research builds on previous research with additional techniques, better performance measurement, and additional architectures. Each technique is implemented in either C, C++, or Assembly as needed. Some techniques make use of operating system APIs which lend themselves to being implemented in higher level languages like C or C++. Some techniques require Assembly-language level control over the instructions and data to accurately measure the anti-debugging technique.

The tests will be run 1010 times to generate sufficient data for analysis. The first ten measurements will be discarded as they are used to populate the instruction and data cache (Paoloni, 2010). These measurements will be logged and accumulated for statistical analysis and compared to the baseline implementation per operating system and per architecture. This quantitative analysis will numerically answer the research question, “What is the impact of an anti-debugging technique on program execution time across the Windows and Linux operating system with Intel and ARM architectures?”

Research Methodology Design

The methodology for this research follows design science research process as described in (Peffers et al., 2007). This research conducted a survey of many design science research processes and assimilated them into one, six section model. The six steps are Problem Identification and Motivation, Objectives of a solution, Design and Development, Demonstration, Evaluation, and Communication. The Problem Identification and Motivation step sets up the research and provides background on the state of the problem and the likely impact conducting this research will have. The Objectives of a solution step explicitly discusses the outcomes and impact of this research being conducted. These objectives are

influenced by the Problem Identification and Motivation step. The third step, Design and Development, creates the artifact. This step is intentionally left ambiguous to support many different implementations across Information Systems research. The fourth step, Demonstration, is the application of the artifact to the problem space. Penultimately, the fifth step, Evaluation, involves measuring and validating the results of the Design and Development and the Demonstration steps. In addition, researchers must consider if the application of the artifact meets the original objectives of the solution. It is also possible as part of the Evaluation step that additional iteration on the artifact is necessary and researchers may return to Design and Development to improve the solution. Lastly, Communication, is the final output of the results into the body of knowledge. This could include delivery to customers, academic publishing, or other applicable communication methods (Peffers et al., 2007). This methodology will be applied to this research to support its validity and rigor. The details of the application of this methodology to this research are captured in Figure 1 below.

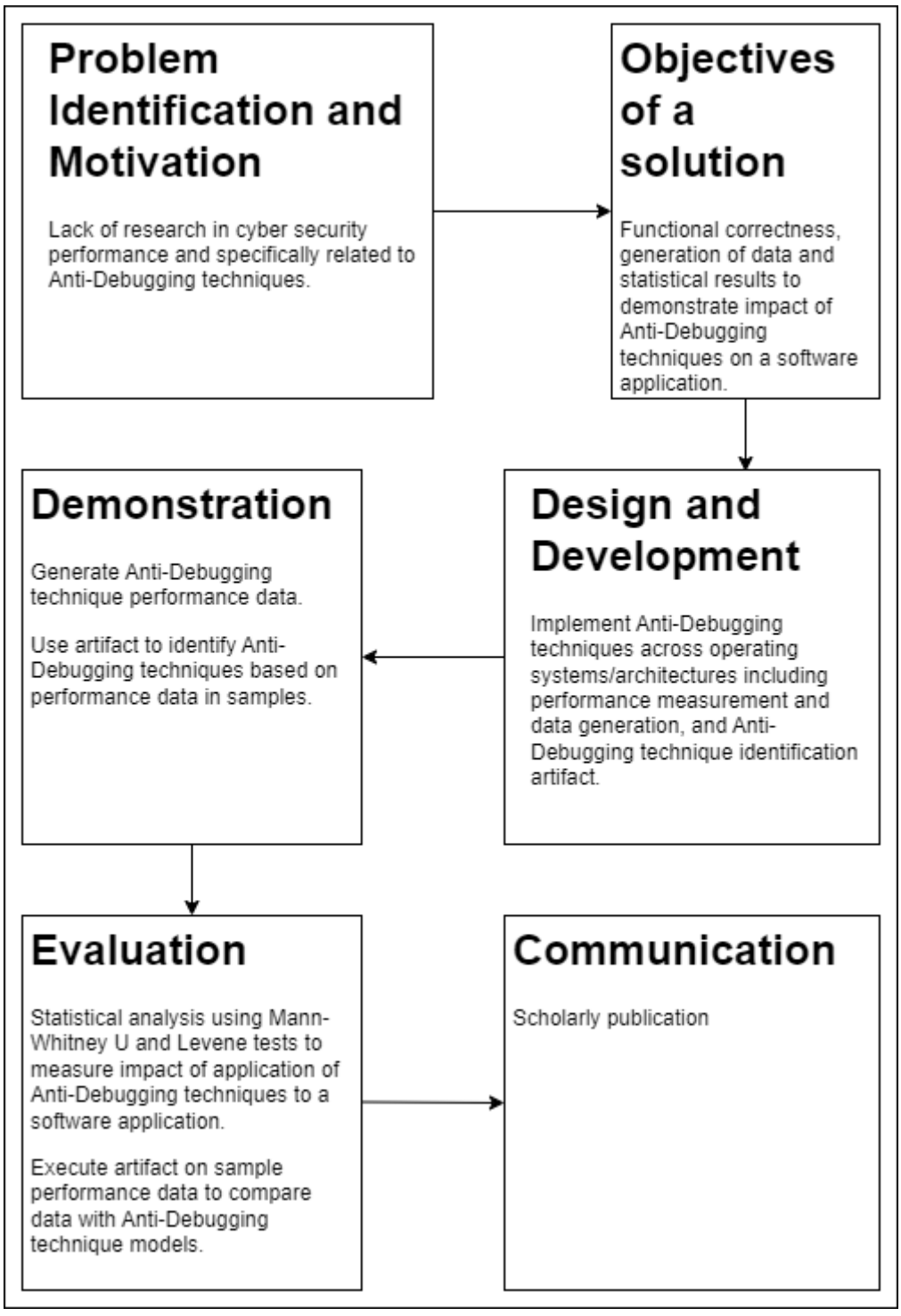

Figure 1 – Research Methodology Visualization

Problem Motivation and Identification

The research question for this research was identified during an initial literature review by its lack of research on the subject. Anti-debugging techniques are not new and have published papers dating back to 1999 (Cesare, 1999), but there is minimal measurement of performance of any of the techniques. The research literature consists mostly of creation of techniques, defeating techniques, and the application of techniques in malware and software protection frameworks. This lack of research necessitates further investigation into the topic.

Objectives of a solution

The objectives of the solution must provide quantitative data for analysis of execution time related to the application of Anti-Debugging techniques to a software application. This data does not currently exist, and the solution must produce this data to improve the body of knowledge related to the performance of Anti-Debugging techniques.

The second objective of the solution is to create an artifact that will use the performance data to measure the similarity between a sample and the model Anti-Debugging techniques using timing data. The similarity according to the Levene test provides information about the similarity of variances for the data distributions and higher similarity implies a higher likelihood of Anti-Debugging techniques being present (Brown & Forsythe, 1974; Levene, 1960; “Levene Test for Equality of Variances,” n.d.).

Design and Development

The design and development of the artifact for this research must include the development of Anti-Debugging techniques for the Windows operating system on the Intel architecture, the Linux operating system on the Intel architecture, and the Linux architecture on the ARM architecture. For each of these three areas of development, performance measurement must be included to generate the data. The performance measurement code must match the running computer architecture. Some of the code already exists in the open-source community while other code must be developed for this research. The Intel-based performance code will take best practices from (Paoloni, 2010) as published by the Intel corporation. These best practices maintain the level or rigor necessary for performance analysis due to their expertise on the subject. Similar best practices will be taken from (*ARM*

Developer Documentation, 2023) for the same reasons. The specific details of the implementations follow in the below sections, “Performance”, “Windows”, and “Linux (Intel)”, and “Linux (ARM)”.

The artifact design must include key features such as processing performance information, statistical analysis, production of information regarding the similarity of the sample data distribution and the model data distributions for the of Anti-Debugging techniques, and communication of the results.

Demonstration

The demonstration of the performance measurement code will be two-fold. The first demonstration is execution of the tests to gather performance data for data analysis. Graphs of the data distribution, quartile values of the data distribution, and statistical test values comparing the Base Test distribution to the Anti-Debugging technique distribution will be generated from the demonstration.

The second demonstration will use the generated data models to test the presence of anti-debugging techniques within sample programs using the Levene test. Based on the results of the Levene test, rejecting, or failing to reject, the null hypothesis gives information on the similarity of the variances of the data sets which, based on the model data, are based on the presence of the Anti-Debugging techniques (Brown & Forsythe, 1974; Levene, 1960; “Levene Test for Equality of Variances,” n.d.). If a sample data distribution has a similar variance to the models generated, based on the p-value, we have numerical representation of the similarity and likelihood there are Anti-Debugging techniques present in the sample.

Evaluation

The generated data from the Demonstration step will be used during the execution of the artifact. This data will be processed and analyzed using statistical tests the Mann-Whitney U test and the Levene test (Levene, 1960; “Levene Test for Equality of Variances,” n.d.; Mann & Whitney, 1947; Nahm, 2016). The results from the statistical tests will measure the changes of applying Anti-Debugging techniques to a software application and quantify the impact over the data distribution from numerous runs of a sample. In all reasonable cases, the data will be reported to three significant figures except for the U statistic, which is often in the hundreds of thousands for this research, and zeros, which will be represented 0.0.

The Mann-Whitney U test is used for measuring the difference in medians between samples, or in our case the performance time datasets. Using this test to compare the Base Test to each of the Anti-Debugging techniques' datasets provides a new test statistic and p-value. Importantly, this test does not assume normally distributed data as it is nonparametric (Nahm, 2016). Not assuming these conditions of the test provide a more accurate answer for our results. We use the SciPy.Stats module for Python 3, but for this test the mannwhitneyu() method is executed on the datasets (Virtanen et al., 2020). Additionally, since adding an Anti-Debugging technique will increase the performance time compared to the BaseTest due to the increase in amount of code executed, the alternative hypothesis is set to assume that the mean of the Base Test dataset is less than the Anti-Debugging technique dataset. Given the ubiquity of $p=0.05$ as the default significance level, for this research we reject the null hypothesis for all Mann-Whitney U tests that have a p-value less than 0.05. When we reject the null hypothesis, we can conclude that there are statistically significant differences between the medians of the performance data distributions. In other words, the dependent variable, performance time, is affected by the independent variable, application of an Anti-Debugging technique, and the statistically significant different means are a consequence of the Anti-Debugging technique (Mann & Whitney, 1947).

Moving onto our last statistical test, the Levene test for equal variances between datasets. This is different from the previous two tests because it measures the difference in variance of the dataset rather than the difference in the mean of the dataset. Due to the highly skewed distribution of data (heavily skewed right) for most data distributions, the Levene test is selected over the Bartlett test. The Levene test is implemented in the scapy.stats package for Python 3 and is executed on the datasets using levene(). While using the levene() function, due to the highly skewed nature of the data distribution, use the median as the center for the test (Brown & Forsythe, 1974; Levene, 1960; "Levene Test for Equality of Variances," n.d.; Virtanen et al., 2020). Given the ubiquity of $p=0.05$ as the default significance level, for this research we reject the null hypothesis for all Levene tests that have a p-value less than 0.05. When we reject the null hypothesis, we can conclude that there are statistically significant differences between the variances of the performance data distributions. In other words, the dependent variable, performance time, is affected by the independent variable, application of

an Anti-Debugging technique, and the statistically significant different variances are a consequence of the Anti-Debugging technique.

Communication

The artifact, generated data, and statistical results will be accumulated into the dissertation associated with this research and published as part of the degree requirements for a Doctor of Cyber Operations from Dakota State University (DSU).

CHAPTER 4

SYSTEM DESIGN

Performance

The performance measurement is focused on execution timing. Execution timing is often the primary measurement of performance for computers as it aligns with implementing speedy algorithms and optimized assembly instructions. For this research, we are not interested in doing the optimization itself but measuring the overall performance of anti-debugging techniques. Despite not implementing optimizations, it's important to understand the mechanics of execution timing to accurately measure and discuss the results.

Execution timing is a valuable measurement of performance because ultimately the code needs to be executed and that will take some amount of time. Even if memory is optimized or specialized hardware is used, the algorithms will be executed over some period and in general, minimizing that time is efficient and desirable. The measurement can be done quite easily on Intel architectures with the RDTSC instruction. This single instruction captures the timestamp counter, and we can compare those results from the atomic timestamp reads and expect accurate timing measurements. There are other timing mechanisms for the ARM architecture and even less precise measurements including measuring seconds or clock differences on Windows and Linux operating systems. One obvious downside is that capturing the timing of execution requires executing instructions so the result can never be a perfect measurement of only the code being tested; there will always exist delays due to the

measurement instructions. The implementation will be as minimal as possible and therefore will make use of the RDTSC instruction for Intel architectures. This provides the minimal amount of overhead for execution timing available to that architecture. For the ARM architecture, there exists a special register, CNTVCT_EL0, that stores the count of clock cycles for the processor. This can also be read at the assembly level to minimize overhead (*ARM Developer Documentation*, 2023; *Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023; Paoloni, 2010).

Performance Data Collection

For each of the tests, performance data must be collected that minimizes any extraneous execution to isolate the performance of the Anti-Debugging technique. This research used a custom implementation based on Paoloni to take advantage of assembly instructions and CPU side-effects to mitigate any speculative execution which might alter the performance measurement of a technique.

To mitigate speculative execution before execution of the Anti-Debugging technique begins, the code must call the CPUID instruction using the __cpuid compiler intrinsic which forces serialization of execution. Immediately following that we capture the first RDTSC value and begin execution of the technique. Similarly, when ending the measurement, we need to serialize execution to prevent speculative execution so the __rdtscp compiler intrinsic is called. This instruction is functionally equivalent to RDTSC but provides the opportunity to return additional information. This additional information isn't what's valuable but is necessary for serialization of execution. Lastly, when stopping measurement, the code calls __cpuid again to prevent any effects of speculative execution.

This data is stored in the C++ AntiDebuggingTimer class implemented for this research. This class stores the values which allows the computation to be done at a later time so it doesn't affect the measurement of the Anti-Debugging technique. Once the RDSTC results are captured, the timing must be computed based on the CPU processor speed. RDTSC returns the number of clock ticks instead of a time measurement. Thus, we need a frequency to convert clock ticks into seconds. For Windows, this conversion value can be returned using QueryPerformanceFrequency. Lastly, the execution time is multiplied by 1,000,000 to compute microseconds and divided by frequency to return the actual number of microseconds

for the duration of execution (*Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023; Paoloni, 2010).

The computation of durations is slightly different for all operating systems and architectures. While Windows uses QueryPerformanceFrequency, this API does not exist on Linux systems. To get the frequency for Linux (Intel) we read the MHz value out of the /proc/cpuinfo file which returns the same information. This frequency value is applied in the same way to the difference between starting and ending clock ticks while being converted to microseconds.

The Linux (ARM) duration calculations are the most different from the previous two due to the change in architecture. There is no direct equivalent of RDTSC on ARM, so a new instruction is used to read the cycle count value from a register, this register is called, CNTVCT_EL0. This register holds the same clock cycle counter value, and we can read it using the, MRS %0, CNTVCT_EL0, instruction. Similarly, we need to prevent speculative execution on this architecture as well and that can be accomplished with the ISB instruction. In totality, the start of measurement executes the ISB instruction followed by the MSR instruction. The end of the measurement executes ISB and the MRS instruction again. Lastly, in order to get the frequency, a different register must be read using the, MRS %0, CNTFRQ_EL0, instruction. As with the previous calculations, the execution duration in clock cycles is divided by frequency and multiplied by 1,000,000 to convert to microseconds for execution duration.

For each operating system and architecture, once these values are computed post execution, they are written to a log file for later processing by data munging scripts and ultimately the artifact. The data munging is for cleaning up and processing the data into a workable comma-separated value (CSV) format.

Windows

For the windows operating system, there are many APIs that return information about the current state of debugging for a program. Many of these and some aspects of implementation come from The Unprotect Project (Roccia & Lesueur, 2023) which has lists of techniques and some implementations for anti-debugging techniques. In addition, the project has information and implementations for anti-virtualization, anti-disassembly, anti-

forensics, and more. Other anti-debugging techniques fall into categories such as: exception-based, scanning, timing, and miscellaneous. The miscellaneous category includes self-debugging, an instance of anti-disassembly, and memory encryption. All the techniques are implemented in C, C++, or Assembly as needed for the technique.

Most often, the APIs are well-documented and simple usage with the right parameters is sufficient to determine the debugging status of the program. In some cases, the description from the Unprotect Project is enough to figure out what's necessary for implementation. There are no anti-debugging techniques in the API category that do not have sufficient details for an implementation. These APIs are NtGlobalFlag, Heap Flag, IsDebugged Flag/IsDebuggerPresent(), CsrGetProcessID, EventPairHandles, OutputDebugString, NtQueryObject, NtSetInformationThread, ProcessDebugPort/NtQueryInformationProcess, CheckRemoteDebuggerPresent, NtSetDebugFilterState, and Debug registers (*Aadp*, n.d.; Apostolopoulos et al., 2020; Branco et al., 2012; P. Chen et al., 2016; X. Chen et al., 2008; Ferrie, 2011; Gao & Lin, 2012; Gozzini, 2022; J.-W. Kim et al., 2019, 2020; Kulchytskyy, 2019; Park et al., 2019; Roccia & Lesueur, 2023; Saad & Taseer, 2022; Shi & Mirkovic, 2017; Sikorski & Honig, 2012; Vasudevan & Terraballi, 2005; Yoshizaki & Yamauchi, 2014).

The second category is the exception-based anti-debugging checks. These types of checks are like API-based checks but rely on the program throwing an exception. The important component here is that if the program catches the thrown exception, it's not being debugged whereas if the program does not catch the exception, then there must have been a debugger that did. These are only slightly more difficult to implement but, like the API-based anti-debugging checks, there are no techniques that do not have sufficient information for implementation. The exception-based techniques are Guard Pages, Exception-based interrupts, Unhandled Exception Filter, CloseHandle, INT 0x2d, and Trap Flag (*Aadp*, n.d.; Bing et al., 2010; Branco et al., 2012; P. Chen et al., 2016; X. Chen et al., 2008; Ferrie, 2011; Gao & Lin, 2012; Gozzini, 2022; Kulchytskyy, 2019; Park et al., 2019; Roccia & Lesueur, 2023; Saad & Taseer, 2022; Shi & Mirkovic, 2017; Vasudevan & Terraballi, 2005).

The third category of anti-debugging techniques are based on scanning the process or system environment. This type of check is looking for signs of a debugger being present by scanning through process memory, a process list, or looking for a sign of a debugger being

present. These anti-debugging techniques are also like API-based but require more data structures and memory to hold information and take longer due to repeated calls for some APIs and scanning lots of data. The scanning anti-debugging techniques are, INT3 Instruction Scanning, Code checksum, Enum Processes, and Parent Process Detection (Bing et al., 2010; Branco et al., 2012; Ferrie, 2011; Giffin et al., 2005; Roccia & Lesueur, 2023; Saad & Taseer, 2022; Shi & Mirkovic, 2017; Stone et al., 1998).

The penultimate category of anti-debugging techniques are the timing checks. These all have the same structure but use different APIs on the Windows operating system for capturing time. Each of these timing APIs will have different overhead and likely require different thresholds for determining if debugging behavior is present. The timing anti-debugging techniques work by assuming that if one exists, the amount of time it would take to execute some instructions will be larger than if the program continued executing without being interrupted by a debugger. This difference can be measured and compared to a threshold based on the resolution of the timer and the number of instructions being measured. The different APIs are GetLocalTime, GetSystemTime, timeGetTime, NtQueryPerformanceCounter, GetTickCount, and RDTSC (*Aadp*, n.d.; Apostolopoulos et al., 2020; Branco et al., 2012; P. Chen et al., 2016; Ferrie, 2011; Gozzini, 2022; M.-J. Kim et al., 2010; Park et al., 2019; Saad & Taseer, 2022; Shi et al., 2017; Shi & Mirkovic, 2017; Shields, 2010; Vasudevan & Terraballi, 2005, 2005; Yoshizaki & Yamauchi, 2014).

Lastly, we have our miscellaneous anti-debugging techniques. These techniques don't fall into the aforementioned categories well and can be considered individually on their own anti-debugging merit. Anti-disassembly techniques tend to focus on static analysis and defeating the recursive or linear sweep disassemblers within the static analysis products. The second miscellaneous anti-debugging technique is commonly used for software protection and encrypts the memory before use. This research will only focus on one instance of this, but it can be a versatile technique. The data could be encrypted before use, the instructions could be encrypted before use, the memory could be decrypted all at once for use or decrypted piecemeal and re-encrypted after use for additional security and intellectual property protection. The more encryption operations that are done the more memory and execution time the technique will take for the benefit of ensuring code or data is difficult to recover. Lastly, the final miscellaneous anti-debugging technique is self-debugging. This technique

takes advantage of the fact that operating systems only allow one debugger to debug a process at a time. Therefore, if the program has its own debugger, then another one cannot be added to it (Abrath et al., 2016; Bahaa-Eldin & Sobh, 2014; Branco et al., 2012; Ferrie, 2011; Gozzini, 2022; M.-J. Kim et al., 2010; Saad & Taseer, 2022; Shi & Mirkovic, 2017; Shields, 2010; Wan et al., 2018; Yoshizaki & Yamauchi, 2014).

API Anti-Debugging Techniques

NtGlobalFlag

The NtGlobalFlag exists as a member within the Process Environment Block (PEB) structure. Each Windows process has a PEB, and it contains information about and for the process' execution. For 64-bit Intel architecture-based Windows 10 operating systems, the PEB can be located by reading offset 0x60 of the GS segment register. The GS segment register is used to point to a unique segment in memory that the PEB is stored in for this particular architecture/operating system combination. At offset 0x60 of the GS segment, we can access the PEB and at offset 0xBC for this architecture/operating system combination we get a DWORD value called NtGlobalFlag. This value is altered when a process is created by a debugger and can be used to detect the presence of a debugger. If the FLG_HEAP_ENABLE_TAIL_CHECK, FLG_HEAP_ENABLE_FREE_CHECK, and the FLG_HEAP_VALIDATE_PARAMETERS flags are turned on, we can be sure that the process was started with a debugger and react accordingly. (*Aadp*, n.d.; *Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023; Branco et al., 2012; P. Chen et al., 2016; Ferrie, 2011; Kulchytskyy, 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017)

Heap Flag

The Heap Flags are located within the process heap structure that is part of the PEB structure. Once again, the PEB can be accessed by reading the address at offset 0x60 of the GS segment register. The address of the process heap is located at offset 0x30 of the PEB which points to the process heap. The process heap contains information about how the process was created including different flags being enabled if the process was created by a debugger. Two metadata fields within the heap, Flags and ForceFlags, contain different values but are understood to have specific flags enabled when a debugger starts the process. If a

debugger started the process the Flags value does not have the HEAP_GROWABLE flag enabled and the ForceFlags value has HEAP_TAIL_CHECKING_ENABLED, HEAP_FREE_CHECKING_ENABLED, and HEAP_VALIDATE_PARAMETERS_ENABLED flags enabled. By checking the value of these two fields within the process heap, we can determine if the process was started with a debugger and act accordingly. (*Aadp*, n.d.; Branco et al., 2012; P. Chen et al., 2016; Ferrie, 2011; Kulchytskyy, 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017)

IsDebuggerPresent()

This Anti-Debugging technique is likely the best known and least complex to implement. The Windows operating system provides an API call to check if a debugger is present and returns a Boolean true or false value. This value is queried from the PEB and exists at offset 0x2 within the PEB structure. For this research, we used the exported API call rather than inline assembly. Based on the result of the API call, we can determine if a debugger is present and act accordingly.(*Aadp*, n.d.; "IsDebuggerPresent Function," 2021; Branco et al., 2012; P. Chen et al., 2016; X. Chen et al., 2008; Ferrie, 2011; Gao & Lin, 2012; Gozzini, 2022; Kulchytskyy, 2019; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017)

CsrGetProcessId()

The CsrGetProcessId() Anti-Debugging technique takes advantage of additional privileges given to processes that are started by a debugger. When a process is started by a debugger, it is given the SeDebugPrivilege which gives more access permissions to a process than normal. In addition, there exists an undocumented function, CsrGetProcessId(), that will return the Process ID (PID) of CSRSS.exe. A normal process is unable to open a handle to the CSRSS.exe process, but this is not the case with SeDebugPrivilege enabled. With the SeDebugPrivilege, a process started by a debugger can open a handle to the CSRSS.exe process by retrieving its PID and opening a handle to the process based on its PID. If this succeeds, it is likely the process was started by a debugger and given the SeDebugPrivilege and can be used for debugging detection. (Branco et al., 2012; Ferrie, 2011; Gao & Lin, 2012; Roccia & Lesueur, 2023)

EventPairHandles

For the EventPairHandles technique, when a debugger starts, or is attached to, a process, a DebugEvent object is created. This object is created with an internal Debug Message object that is used as part of the WaitForDebugEvent loop implemented by all debuggers. This technique looks for differing values from a normal execution by creating a debug buffer using RtlCreateQueryDebugBuffer. Then, using RtlQueryProcessDebugInformation, the buffer will be populated and will have a specific value for RemoteSectionBase member of the debug buffer. Also, the debug bugger will have an EventPairHandle member that will also have a specific value. By checking for these values, an Anti-Debugging check can be implemented due to the difference between values for debugged and non-debugged processes (Bonfa, n.d.; Roccia & Lesueur, 2023).

OutputDebugString

The OutputDebugString() function can be used to display information to a debugger that is executing a program. When there is no debugger attached, Windows sets the LastError value to a non-zero value. The GetLastError() and SetLastError() functions are commonly used to indicate when Windows APIs fail and give more information about the failure. Since OutputDebugString will set the LastError value when no debugger is present, it can be used as a check for debuggers by looking for a non-zero value after OutputDebugString returns using GetLastError(). If GetLastError() is non-zero it is likely a debugger is present and the code can act accordingly. (*Aadp*, n.d.; Branco et al., 2012; Ferrie, 2011; Gao & Lin, 2012; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017)

NtQueryObject

The NtQueryObject function is a semi-documented function that returns operating system objects. These objects are not well documented but one has been understood enough to become an Anti-Debugging technique. To use the NtQueryObject function, it must be dynamically resolved during run-time. Once the function has been resolved, the amount of memory needed to retrieve the operating system objects must be retrieved by a first call to NtQueryObject that intentionally fails. When NtQueryObject fails due to insufficient buffer size, the amount of memory needed is returned in the optional in/out parameter,

ReturnLength. This value is used to allocate sufficient memory to store the objects returned by NtQueryObject. The successful NtQueryObject call will return many objects, but one of the objects returned has a member with a string value, “DebugObject” and if this object has a positive value for the TotalNumberOfObjects attribute in the OBJECT_TYPE_INFORMATION structure, we can assume a debugger is present. (*Aadp*, n.d.; Ferrie, 2011; ReactOS Team, 2023; Roccia & Lesueur, 2023)

NtSetInformationThread

The NtSetInformationThread function can be used to modify the properties of a thread. There exists a ThreadInformationClass enumeration value called ThreadHideFromDebugger that can be applied to a thread. This technique opens a handle to its current thread and then applies this property to the thread using NtSetInformationThread. When this property is applied, debuggers are no longer alerted to events from this thread which keeps the thread executing without debugger interference. This technique does not return a Boolean value of debugger presence but avoids the debugger altogether (*Aadp*, n.d.; P. Chen et al., 2016; Ferrie, 2011; Gozzini, 2022; Kulchytskyy, 2019; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

NtQueryInformationProcess

NtQueryInformationProcess returns process information about a process. The information returned is selected via the ProcessInformationClass enumeration value that is passed into the function. Specifically, there is a less well-known enumeration value called ProcessDebugPort. When this value is passed into the function, the debug port value is returned. The debug port is only non-zero when the process is being debugged. Using this non-zero value as the check for debugging can cause this function to act as an Anti-Debugging check and determine if a debugger is present. (*Aadp*, n.d.; Branco et al., 2012; P. Chen et al., 2016; Ferrie, 2011; Gozzini, 2022; Kulchytskyy, 2019; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017)

CheckRemoteDebuggerPresent

The Windows operating system has another API available to developers called CheckRemoteDebuggerPresent. This is like the IsDebuggerPresent() technique but requires a process to check. By passing in the current process as the first argument, the Boolean argument passed in as the second argument is populated with the value of a debugger being present. This value can be checked and used for Anti-Debugging checks. This will determine if a debugger is present in this or other processes (Branco et al., 2012; P. Chen et al., 2016; X. Chen et al., 2008; Ferrie, 2011; Gao & Lin, 2012; Kulchytskyy, 2019; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

NtSetDebugFilterState

This function is used to check for kernel debuggers while the program is executing. This function returns values based on the presence of the SeDebugPrivilege, like the CsrGetProcessId check. It is unlikely that a regular process will have SeDebugPrivilege while running normally so checking this function call can return information about the privileges the process has. While this does not directly query an attached debugger, it can provide information about privileges and a developer can determine what to do with that information. It is likely that a normal process will not have SeDebugPrivilege and therefore can be used to decide if it is likely a debugger is attached (Branco et al., 2012; Ferrie, 2011; Kulchytskyy, 2019; Roccia & Lesueur, 2023).

Checking Debug Registers

This Anti-Debugging technique takes advantage of additional information being provided by the Windows operating system; it’s not present in Linux. Each thread has a lot of information including the execution context available. This thread context is made up of all the values of each register including debug, segment, instruction pointer, and floating-point registers to name a few. The debug registers are used for hardware breakpoints and are often populated by debuggers using SetThreadContext for breakpoints. Due to the infrequency that debug registers are used, it is likely that if these registers are populated, they were populated by a debugger indicating its presence. The check uses GetThreadContext on the current thread. The debug registers that are set are D0 through D3, if any of these four registers is

non-zero, it is likely a debugger is present and can be used as an Anti-Debugging check (Branco et al., 2012; Ferrie, 2011; Gagnon et al., 2007; *Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023; Kulchytskyy, 2019; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017; Shields, 2010; Vasudevan & Terraballi, 2005; Yi et al., n.d.).

Exception-based Anti-Debugging Techniques

Guard Pages

The Windows operating system allows setting memory protections on pages of memory that are used as Guard Pages. Guard Pages are used to detect memory overruns and invalid use of memory. The technique is implemented using VirtuaAlloc to allocate a page of memory. In addition, it can use VirtualProtect to add the PAGE_GUARD protection to the allocated memory. If the Guard Page is accessed during normal execution, something went wrong in the code and an exception is thrown to represent that abnormal execution. This technique takes advantage of throwing an exception to check the presence of a debugger by attempting to access the allocated Guard Page. If a debugger is attached, the exception is passed to the debugger first to be handled. After the exception is handled, the exception handler in the code is skipped and the Anti-Debugging check will return that a debugger is present. If the exception is not handled, the exception handler in the code will catch the STATUS_GUARD_PAGE_VIOLATION exception and continue execution. Having the code handle its own exception likely means there is no debugger attached (Branco et al., 2012; Ferrie, 2011; Gozzini, 2022; Roccia & Lesueur, 2023).

Int3 Interrupt

Using the Intel architecture INT3 instruction generates an interrupt to be handled by the debug exception handler. Like other exception-based techniques, the interrupt is passed directly to the debugger if one is present. If a debugger is present and it handles the interrupt then the exception handler in the code will not be executed and therefore alert the program to a debugger being present. However, if a debugger is not present, the exception handler in the code will be executed which will return information that a debugger is not present which encompasses the entire Anti-Debugging check functionality (P. Chen et al., 2016; Ferrie,

2011; Gao & Lin, 2012; Gozzini, 2022; Kulchytskyy, 2019; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

Int2d Interrupt

Like the Int3 interrupt, Int2d will also generate an exception but execute the handler at offset 0x2d in the interrupt handler table. This offset is for the breakpoint exception handler. This handler will raise the exception to the debugger and if it's handled by the debugger, the exception handler in the code will not be executed returning information about the presence of the debugger. Like other exception-based techniques, if the code's handler is executed then the debugger is not present and the Anti-Debugging check technique is complete (Ferrie, 2011; Gao & Lin, 2012; Gozzini, 2022; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

Unhandled Exception Filter

The Windows operating system provides an API to set one's own exception handling routine using the SetUnhandledExceptionFilter function. This function takes a pointer to a new exception handling routine and returns the previous top exception handling routine. Using this function, one can set their own exception handling routine for the process to provide information on the presence of a debugger. The code causes an exception which is either passed to the debugger or the new exception handling routine. If the debugger handles the exception, the exception handling routine is not executed and therefore information is passed through the program that a debugger is present. If the exception handling routine is executed the information is altered to demonstrate no debugger is present (*Aadp*, n.d.; Branco et al., 2012; P. Chen et al., 2016; Ferrie, 2011, 2011; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

CloseHandle

When a program is being debugged, attempting to call CloseHandle on an invalid handle will generate an exception. If a debugger is present, it will handle the exception and the information that a debugger is present will be passed through to the code. In the other case, when an invalid use of CloseHandle does generates an

EXCEPTION_INVALID_HANDLE exception, the exception is handled by the author's exception handling code and will continue execution providing information that a debugger is not present (Gao & Lin, 2012; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

Trap Flag

The Trap Flag is a part of the EFLAGS register that exists in the Intel architecture. This flag allows for single-stepping execution of a process at the CPU level. When a debugger is in use, this flag can be set using the __readeflags() and __writeeflags() compiler intrinsics with the value 0x100 to turn on the Trap Flag in the EFLAGS register to facilitate single stepping through a process instruction by instruction. Since this flag is often used by a debugger, checking for the flag can be used for determining the presence of a debugger. However, it is equally as valuable to set the Trap Flag to generate a single-step exception that will be passed to a debugger if present. This technique registers an exception handler that will only be executed if a debugger is not present and catches the single-step exception. If a debugger is present, the single-step exception is passed to the debugger to be handled and execution is continued. These conditions provide enough information to the program to determine the presence of a debugger (Branco et al., 2012; Ferrie, 2011; Gao & Lin, 2012; Gozzini, 2022; *Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023; Kulchytskyy, 2019; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017; Shields, 2010; Vasudevan & Terraballi, 2005).

Scanning Anti-Debugging Techniques

INT3 Instruction Scanning

The INT3 instruction scanning Anti-Debugging check works by looking through the code trying to identify INT3 instructions in the code. This instruction is represented by a single byte, 0xCC. Most production-like programs do not contain any 0xCC bytes in it due to the instruction only being used for debugging purposes. Therefore, if the code detects an INT3 instruction in the code, it was likely put there by a debugger which would indicate the presence of a debugger. Thus, scanning for the 0xCC byte in code where there are no 0xCC

bytes expected will provide indications that a debugger is likely present and operate as an Anti-Debugging check (Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

Code Checksum

To detect a debugger by computing a checksum of the code the program needs to select a starting point and ending point, or length, to compute a checksum over. This will detect changes to the code from the expected code. The checksum can be computed over all the code, important parts of the code, or whichever parts the author prefers. By calculating a checksum, if any bytes are changed the checksum will not equal the original value and the program will be aware of changes to the code. Since debuggers implement software breakpoints by overwriting the byte at a particular address with 0xCC (INT3 instruction) the original checksum will not equal the new checksum and it is likely a debugger is present. This will also work for identifying hooks or malicious software that overwrites or modifies code. If the checksum is equal to the original value, it is highly likely that no debugger is present and if the checksum is not equal, it is likely that a debugger made changes to the code. For this research, the Fletcher-16 algorithm was selected based on ease of implementation with a sufficiently sized result to limit collisions (Giffin et al., 2005; Roccia & Lesueur, 2023; Stone et al., 1998).

Process Enumeration

For this scanning technique, the code looks at the environment the program is being executed in rather than the program itself. In some cases, analysis programs will execute in the background or in parallel looking for changes to the system or may be able to evade some Anti-Debugging checks. The technique iterates over all the running processes on the system and looks for known, undesired programs that are currently executing. The implementation uses CreateToolhelp32Snapshot to get a list of all the processes currently running. For each process, the process object has a process name member which is compared to a list of known bad process names (deny list). This could be a debugger that isn’t attached or some kind of analysis program that the author is aware of. If one of the programs on the deny list is present, the code can use that information as an Anti-Debugging check and decide about the continued execution of the program (Ferrie, 2011; Roccia & Lesueur, 2023).

Parent Process Scanning

This technique is included in the scanning group due to the need to scan all processes to find the parent process. While the Parent Process ID (PPID) is connected to the process, the program name is not. This technique acquires the PPID of the current process using GetParentProcessId and resolves the name of the parent process using CreateToolhelp32Snapshot. Like Process Enumeration, if the parent process is a program on the deny list, this can act as information for the running program to make future execution decisions. If the parent process is on an allow list, or not on a deny list, execution can continue as normal (Branco et al., 2012; Ferrie, 2011; Gao & Lin, 2012; Roccia & Lesueur, 2023).

Timing Anti-Debugging Techniques

GetLocalTime

The GetLocalTime function returns the system time in the current time zone of the system. This time measurement can be used to measure the time of execution between two parts of a program. To use this for debugging, the time snapshot is taken before the measured part of the code and after the measured part of the code. This difference would be very small if a computer is running without a user debugging the program. Based on the author preferences, if the time exceeds the limit, then it is likely that an analyst is debugging the program and this information can be used as an Anti-Debugging check (Branco et al., 2012; Ferrie, 2011; Gozzini, 2022; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

GetSystemTime

The GetSystemTime function returns the system time in Coordinate Universal Time (UTC). This time measurement can be used to measure the time of execution between two parts of a program. To use this for debugging, the time snapshot is taken before the measured part of the code and after the measured part of the code. This difference would be very small if a computer is running without a user debugging the program. Based on the author preferences, if the time exceeds the limit, then it is likely that an analyst is debugging the

program and this information can be used as an Anti-Debugging check (Ferrie, 2011; Gozzini, 2022; Roccia & Lesueur, 2023).

timeGetTime

The timeGetTime function returns the number of milliseconds since the system was started. The function returns a DWORD which has 2^32 bits of information which can cause the timer to wrap around approximately every 49 days. For this research, a DWORD is sufficient space as we are measuring the execution of programs which should not exceed 49 days. The timeGetTime function is called before and after the measured code to compute the difference, or execution time. As with other timing techniques, if the execution of the program or sections of the program exceed the author's threshold this can be used as evidence that the program is being debugged and can act like an Anti-Debugging check (Branco et al., 2012; Ferrie, 2011; Gozzini, 2022; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

NtQueryPerformanceCounter

The Windows API offers a high-resolution performance counter that returns "ticks" from the operating system. These ticks are sub-microsecond measurements, typically around 100 nanoseconds, and useful for measuring small intervals of time. The results of NtQueryPerformanceCounter are stored in a larger structure to handle the increase in values accompanied by an increase in accuracy of measurement. While the same strategy applies for timing techniques using this API function, the threshold is larger as it's measuring units of 100 nanoseconds as opposed to milliseconds or larger. NtQueryPerformanceCounter is used before and after the measured code to compute the execution duration. Like other timing techniques, if the author's threshold is exceeded that information can be used as part of the program as an Anti-Debugging check (Branco et al., 2012; P. Chen et al., 2016; Ferrie, 2011; Gozzini, 2022; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

GetTickCount

The technique using GetTickCount is almost identical to timeGetTime as GetTickCount returns millisecond precision and the results are stored in a DWORD. Once

again, the maximum time to be measured is approximately 49 days which is still not a problem for this research as programs measured here will not exceed 49 days of execution time. The GetTickCount function is called before and after the measured code to compute the execution duration. If the author's threshold is exceeded this information can be used as an Anti-Debugging check because it is likely that the increase in execution time results from a user debugging the program (*Aadp*, n.d.; Branco et al., 2012; P. Chen et al., 2016; Ferrie, 2011; Gozzini, 2022; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

RDTSC

The RDTSC instruction is part of the Intel architecture and returns the time-stamp counter; the instruction stands for "Read Time-stamp Counter". This counter is incremented every clock cycle and can be used to measure elapsed time between executed instructions. While the counter does not map directly to a unit of time without the CPU frequency, on the same system the increment of the counter is consistent and provides a precise way to measure time between executing instructions. Like other timing techniques, an author must set a threshold and the measurement of the duration of execution between the beginning and ending of the measured instructions can provide information about the presence of a debugger. If the threshold is exceeded, the program can use that information to determine the presence of a debugger as an Anti-Debugging check (Apostolopoulos et al., 2020; Branco et al., 2012; P. Chen et al., 2016; Ferrie, 2011; Gozzini, 2022; *Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023; Paoloni, 2010; Park et al., 2019; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017; Shields, 2010; Vasudevan & Terraballi, 2005).

Miscellaneous Anti-Debugging Techniques

Anti-Disassembly Techniques

While Anti-Disassembly techniques are not always congruent to Anti-Debugging techniques, they are included in this research as an experiment to demonstrate the performance overhead as it relates to increasing the difficulty of debugging, which is one of the purposes of Anti-Debugging. These techniques can be implemented in various ways and for this research the developed technique combined moving function pointers to obfuscate which function is being called, loop unrolling, and bogus calls unrelated to primary purpose

of execution. Each of these techniques by themselves contribute to the performance overhead of the execution, however combining them for this research gives a good introduction to the performance overhead from an Anti-Debugging perspective while using Anti-Disassembly primitives.

For the first technique, function pointer movement is simply the obfuscation of which calls will be made by calculating the function address based on the original and calling the new function instead of the old one. The second technique, loop unrolling, can sometimes even have performance gains depending on the architecture, code implementation, and compiler optimizations. For this research no compiler optimizations are used thus increasing the performance overhead by loop unrolling. Lastly, bogus function calls distract analysts and reverse engineers from understanding the primary purpose of the routine. This increases time spent understanding the code and therefore hinders debugging making it an Anti-Debugging technique (Roccia & Lesueur, 2023; Shi & Mirkovic, 2017).

Memory Encryption

Memory encryption is another way to implement Anti-Debugging as the information is encrypted and cannot be analyzed until after it is decrypted. This technique can be used to protect sensitive code or data and, aside from exploiting cryptography which will not be discussed in this research, the information cannot be recovered until it is executed or used in memory. As one might expect, memory encryption carries a large performance overhead due to the increased number of functions, memory management, and encryption/decryption routines needed for use of the code or data (Vrba et al., 2010).

For this research, the implementation used is Window's built-in BCrypt functionality using the AES encryption algorithm. Before the experiment can even be run, the initial setup must be completed instantiating the algorithm, Initialization Vector (IV), Symmetric Key, and initially encrypting the data. This requires using functions such as, BCryptOpenAlgorithmProvider, BCryptGetProperty, BCryptSetProperty, and BCryptGenerateSymmetricKey for setup. This setup is not included in the measurement as it is a one-time expense that can be done at the beginning of the program and used for its duration. The measurement occurs when the information is decrypted, processed, and then re-encrypted using BCryptEncrypt and BCryptDecerypt. This behavior is what one would expect

to impact performance most while the sensitive code or data is being used and would increase the performance overhead of the program. While this is still an Anti-Debugging technique, there is no Boolean check for the presence of a debugger as the debugger is pre-emptively defeated by the encryption and can only be ascertained once the code or data is decrypted in memory (Roccia & Lesueur, 2023; Shi & Mirkovic, 2017; Vrba et al., 2010).

Self-Debugging

Self-Debugging for this research against the Windows operating system uses part of the debugging design to hinder debuggers from viewing the program at all. The Windows operating system only allows one program to debug another at a time. Therefore, for this technique we implemented a child program that is started by the Anti-Debugging technique that is executed by the parent which attaches itself to the parent as the sole debugger. Since the parent is already being debugged, a debugger cannot be attached to the parent preemptively denying a debugger the opportunity to analyze the program. This also has a large performance overhead as the parent must start a child process which runs the debugger loop and intercepts every debug event generated by the parent. For the child process to do this, it needs to call DebugActiveProcess on the parent PID and disable the parent being killed using DebugSetProcessKillOnExit. For simplicity, it continues execution in each case but there are many events that must be processed by the child and each time execution must halt, be processed, and resume. It handles these debug events using WaitForDebugEvent, but unlike a real debugger, all events are handled the same way with ContinueDebugEvent. Like other miscellaneous techniques, this technique does not have a Boolean check for a debugger and instead prevents a debugger from attaching to it in the first place (Abrath et al., 2016, 2020; Branco et al., 2012; Ferrie, 2011; Gozzini, 2022; Roccia & Lesueur, 2023; Shi & Mirkovic, 2017; Shields, 2010; Wan et al., 2018; Xu et al., 2016).

Linux (Intel)

For the Linux anti-debugging techniques on the intel architecture, there are no API-based techniques, a few exception-based techniques, one new scanning technique, and overall fewer anti-debugging techniques compared to the Windows operating system. All these techniques are implemented in C, C++, or Assembly as needed for the technique.

The exception-based anti-debugging techniques mirror the Windows operating system due to the same underlying architecture and instructions causing the exceptions. The unique exception-based anti-debugging technique makes use of the Linux signal functionality. This could arguably also be API-based but focuses more on catching signals based on exceptions which is why it was not included in that group. Writing signal handlers is not complex and focuses on handling exceptions like a try/catch block in code. There is ample source code and explanation to draw from for implementation.

The timing-based anti-debugging techniques are also like the Windows implementation but use Linux operating system functionality to capture timestamps and calculate time differences. The three major timing mechanisms, in addition to the RDTSC instruction, are clock, time, and clock_gettime. These different implementations of acquiring CPU time and returning different types will present interesting findings for the performance measurements.

The scanning-based anti-debugging techniques are the most plentiful within the Linux operating system. Similar functionality exists to Windows such as computing a code checksum and scanning for INT3 instructions (Giffin et al., 2005; Miller, 2005). However, the new scanning technique is reading the /proc/self/status file and looking for the TracerPid variable. This variable will have a non-zero value if a debugger is present. This technique is also very similar to an API-based check but requires scanning through a file and not a single API call for the information. The TracerPid will hold the PID of the debugger process which, if non-zero, means a debugger is present. Additional scanning techniques are like Windows' scanning techniques for things like enumerating parent processes or other running processes that might indicate a debugger is present.

Lastly, memory encryption, generic anti-disassembly code, and self-debugging have been implemented in prior research for the Linux operating system and can be used here with minimal modifications. For the self-debugging technique, the Linux operating system has a feature implemented as ptrace() which is used for debugging within the Linux operating system. This ptrace functionality is used to start, stop, get, set, and modify debugging behavior. In this research, the self-debugging technique is implemented similarly to the Windows self-debugging technique where the parent starts a child process that attaches to the parent as a debugger. Once the parent process is done executing, it kills the child process

which ends the debugging session started by the child. During the time the child is debugging the parent process, no other debuggers can be attached (Abrath et al., 2016, 2020; kirschju, 2018; Miller, 2005; Vrba et al., 2010).

Exception-based Anti-Debugging Techniques

INT3 Interrupt

As seen in the Windows Anti-Debugging techniques, using the Intel architecture INT3 instruction generates an interrupt to be handled by the debug exception handler. However, unlike the Windows operating system, Linux uses a signal handler to catch debug exceptions in addition to other signals. For this specific test, the implemented handler catches the SIGTRAP signal as part of the Anti-Debugging technique detection. If the signal handler catches the signal and handles it, then there is no debugger attached to the process. On the other hand, if the SIGTRAP signal is raised to the debugger, the signal handler will not be executed and therefore alert the program to a debugger being present. By understanding what component of the operating system or processes handles the SIGTRAP signal, we can implement an Anti-Debugging check (*Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023; Roccia & Lesueur, 2023; Voisin, 2016).

Trap Flag

The Trap Flag is a part of the EFLAGS register that exists in the Intel architecture. This flag allows for single-stepping execution of a process at the CPU level. When a debugger is in use, this flag can be set to facilitate single stepping through a process instruction by instruction. Since this flag is often used by a debugger, checking for the flag can be used for determining the presence of a debugger. However, it is equally as valuable to set the Trap Flag to generate a single-step exception that will be passed to a debugger if present. For Linux, this exception is sent as the SIGTRAP signal. This technique registers a signal handler that will only be executed if a debugger is not present and catches the SIGTRAP signal. If a debugger is present, the single-step exception is passed to the debugger to be handled and execution is continued. These conditions provide enough information to the program to determine the presence of a debugger (*Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023; Roccia & Lesueur, 2023; Shields, 2010; Vasudevan & Terraballi, 2005).

Scanning-based Anti-Debugging Techniques

IsTracerPidZero

As part of starting a process on the Linux operating system, an entire process structure is set up in the /proc filesystem. This allows a process to reference information about itself by looking in the /proc/self directory. Within the /proc/self directory exists many files, but for this technique we can examine the /proc/self/status file which contains information about the status of the process. In this file exists a line, akin to a key-value pair, that lists the Process ID (PID) of a tracer process. In other words, the status file will give the PID of the debugging process if it exists. If there is no tracer program, the value of TracerPID will be zero. By checking this value we can easily see if a debugger, or tracer, is attached to this process based on the value of TracerPID and use this as an Anti-Debugging Technique (Roccia & Lesueur, 2023).

INT3 Instruction Scanning

The INT3 instruction scanning Anti-Debugging check works in the same way on Linux as it does on Windows by looking through the code trying to identify INT3 instructions in the code. This instruction is represented by a single byte, 0xCC. Most production-like programs do not contain any 0xCC bytes in it due to the instruction only being used for debugging purposes. Therefore, if the code detects an INT3 instruction in the code, it was likely put there by a debugger which would indicate the presence of a debugger. Thus, scanning for the 0xCC byte in code where there are no 0xCC bytes expected will provide indications that a debugger is likely present and operate as an Anti-Debugging check (Miller, 2005; Roccia & Lesueur, 2023).

Code Checksum

To detect a debugger by computing a checksum of the code the program needs to select a starting point and ending point, or length, to compute a checksum over. This will detect changes to the code from the expected code. The checksum can be computed over all the code, important parts of the code, or whichever parts the author prefers. By calculating a checksum, if any bytes are changed the checksum will not equal the original value and the

program will be aware of changes to the code. Since debuggers implement software breakpoints by overwriting the byte at a particular address with 0xCC (INT3 instruction) the original checksum will not equal the new checksum and it is likely a debugger is present. This will also work for identifying hooks or malicious software that overwrites or modifies code. If the checksum is equal to the original value, it is highly likely that no debugger is present and if the checksum is not equal, it is likely that a debugger made changes to the code. This technique works in the same way on Linux as it does on Windows. For this research, the Fletcher-16 algorithm was selected based on ease of implementation with a sufficiently sized result to limit collisions (Giffin et al., 2005; Roccia & Lesueur, 2023; Stone et al., 1998).

Process Enumeration

For this scanning technique, the code looks at the environment the program is being executed in rather than the program itself. In some cases, analysis programs will execute in the background or in parallel looking for changes to the system or may be able to evade some Anti-Debugging checks. The technique iterates over all the running processes on the system and looks for known, undesired programs that are currently executing. This technique is implemented using C++ std::filesystem::directory_iterator on the /proc folder. The /proc folder has a directory for every process running on the system. For each process running, the status file is queried for the process name in a similar fashion to the IsTracerPidZero technique, but looking for the cmdline key instead of tracerpid. This could be a debugger that isn't attached or some kind of analysis program that the author is aware of. If one of the programs on the deny list is present, the code can use that information as an Anti-Debugging check and decide about the continued execution of the program. While implemented using different API calls, the Linux technique is functionally equivalent to the Windows technique (Roccia & Lesueur, 2023).

Parent Process Scanning

This technique is included in the scanning group due to the need to scan all processes to find the parent process. For Linux, the getppid() function is available to return the Process ID (PID) of the parent process. Using this PID, we can once again query the /proc directory

using the Parent PID (PPID) as the second directory. Lastly, the technique reads the cmdline file which has the command-line execution of the process. We can check that no bad or denied process names are in that file or if the parent process is on an allow list, or not on a deny list, execution can continue as normal. This check can be used as the implementation for the Anti-Debugging technique (Roccia & Lesueur, 2023; Voisin, 2016).

Timing-based Anti-Debugging Techniques

Time

The Linux operating system provides a function, time(), that returns the number of seconds since the Epoch, 1970-01-01 00:00:00. Since this is such a coarse-grained measurement of time for programs executing so fast, the difference between the beginning and ending will have a small limit value. The limit value is determined by the author to decide when program execution has been too slow and is likely being debugged instead. Despite being a less precise measurement than other timing functions, measuring seconds for an analyst debugging a program is more than sufficient as analysts will operate significantly slower than a computer and even 1 to 2 seconds of delay is easily measurable and could pass the limit. By selecting a good limit value, the technique can use the measurement returned from time() to decide if a debugger is likely present and implement this Anti-Debugging technique (Roccia & Lesueur, 2023).

Clock

Using a different measurement than time(), clock() returns the amount of time used by a program to approximately microsecond precision. The microsecond precision is not guaranteed but does align with the POSIX standard which uses a CLOCKS_PER_SECOND value of 1,000,000. This level of precision does come with a trade-off in long-term measurements. The clock_t type that is returned uses a 32-bit integer which will cause an overflow approximately every 72 minutes according to the man page for clock(). Using this function in an identical manner to other timing tests, the initial value is subtracted from the after-execution value to give the duration. This duration is then compared to the limit given by the programmer. With a good limit, comparing the elapsed time to the limit gives sufficient implementation details for an Anti-Debugging Check (Roccia & Lesueur, 2023).

ClockGetTime

This technique aligns more closely with clock() than time() but uses more configurable parameters to extract specific timing information. The clock_gettime() function expects a specific clock ID and a resolution to return the timing values. This allows the programmer to return specific timing information at a resolution of their choosing. For this research, the technique uses the CLOCK_REALTIME ID and does not specify a resolution which will return the most accurate resolution available. The resolution does support nanosecond precision, however the actual measurement at that resolution is based on the CPU processing speed. Once again, the time value is captured before and after code execution and subtracted to calculate the duration. Once a good limit is selected, the comparison between duration and the limit will determine the likely presence of a debugger and can be used as an Anti-Debugging Technique (Roccia & Lesueur, 2023).

RDTSC

The RDTSC instruction is part of the Intel architecture and returns the time-stamp counter; the instruction stands for “Read Time-stamp Counter”. This counter is incremented every clock cycle and can be used to measure elapsed time between executed instructions. While the counter does not map directly to a unit of time without the CPU frequency, on the same system the increment of the counter is consistent and provides a precise way to measure time between executing instructions. Like other timing techniques, an author must set a threshold and the measurement of the duration of execution between the beginning and ending of the measured instructions can provide information about the presence of a debugger. If the threshold is exceeded, the program can use that information to determine the presence of a debugger as an Anti-Debugging check. Since this is an architecture construct and not operating system specific, the Windows and Linux implementations are functionally equivalent (*Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023; Roccia & Lesueur, 2023; Voisin, 2016).

Miscellaneous Anti-Debugging Techniques

Memory Encryption

Memory encryption is another way to implement Anti-Debugging as the information is encrypted and cannot be analyzed until after it is decrypted. This technique can be used to protect sensitive code or data and, aside from exploiting cryptography which will not be discussed in this research, the information cannot be recovered until it is executed or used in memory. As one might expect, memory encryption carries a large performance overhead due to the increased number of functions, memory management, and encryption/decryption routines needed for use of the code or data (Vrba et al., 2010).

For this research, the implementation used on Linux is OpenSSL's AES cipher (*OpenSSL*, 2024). Before the experiment can even be run, the initial setup must be completed instantiating the algorithm, Initialization Vector (IV), Symmetric Key, and initially encrypting the data. The initial setup is done using gen_params to create the key and IV values. Once that is complete, the initial encryption is done with aes_encrypt. This setup is not included in the measurement as it is a one-time expense that can be done at the beginning of the program and used for its duration. The measurement occurs when the information is decrypted, processed, and then re-encrypted using aes_encrypt and aes_decrypt functions. This behavior is what one would expect to impact performance most while the sensitive code or data is being used and would increase the performance overhead of the program. While this is still an Anti-Debugging technique, there is no Boolean check for the presence of a debugger as the debugger is pre-emptively defeated by the encryption and can only be ascertained once the code or data is decrypted in memory (Roccia & Lesueur, 2023).

Anti-Disassembly

While Anti-Dissasembly techniques are not always congruent to Anti-Debugging techniques, they are included in this research as an experiment to demonstrate the performance overhead as it relates to increasing the difficulty of debugging, which is one of the purposes of Anti-Debugging. These techniques can be implemented in various ways and for this research the developed technique combined moving function pointers, loop unrolling, and bogus calls unrelated to primary purpose. Each of these techniques by themselves

contribute to the performance overhead of the execution, however combining them for this research gives a good introduction to the performance overhead from an Anti-Debugging perspective while using Anti-Disassembly primitives.

For the first technique, function pointer movement is simply the obfuscation of which calls will be made by calculating the function address based on the original and calling the new function instead of the old one. The second technique, loop unrolling, can sometimes even have performance gains depending on the architecture, code implementation, and compiler optimizations. For this research no compiler optimizations are used thus increasing the performance overhead by loop unrolling. Lastly, bogus function calls distract analysts and reverse engineers from understanding the primary purpose of the routine. This increases time spent understanding the code and therefore hinders debugging, making it an Anti-Debugging technique. This implementation is functionally equivalent to the Windows implementation (Miller, 2005; Roccia & Lesueur, 2023).

Self-Debugging

Self-Debugging for this research against the Linux operating system uses ptrace to hinder other debuggers from viewing the program at all. The Linux operating system only allows one program to debug another at a time. Therefore, for this technique we implemented a child program that is started by the Anti-Debugging technique that is executed by the parent which attaches itself to the parent as the sole debugger. Since the parent is already being debugged, a debugger cannot be attached to the parent preemptively denying a debugger the opportunity to analyze the program. The parent starts this process by fork-ing itself and calling system on the child process. The parent continues to execute the desired functionality while the child process attaches to the parent using ptrace. Additionally, the child also wants to ignore debug events and automatically continues execution if any debug events are raised. This technique has a large performance overhead as the parent must start a child process which attaches and waits to be killed by the parent after the parent has finished execution. Like other miscellaneous techniques, this technique does not have a Boolean check for a debugger and instead prevents a debugger from attaching to it in the first place (Abrath et al., 2016, 2020; kirschju, 2018; Miller, 2005; Roccia & Lesueur, 2023; Voisin, 2016).

Linux (ARM)

There exist some techniques that will require writing ARM assembly to implement the anti-debugging technique. First, the INT3 instruction does not exist and due to architectural differences, using the corresponding instruction, BRK #F000, does not result in a viable Anti-Debugging technique so we will raise a SIGTRAP signal instead of relying on an assembly instruction to raise it for us (*ARM Developer Documentation*, 2023). The next difference will be the assembly level timer for ARM. There exist built-in registers to the ARM architecture that can be read using the MSR instruction to capture the cycle count and then compute the difference between the beginning and end of the technique. For this research, the CNTVCT_EL0 register is used for this purpose (*ARM Developer Documentation*, 2023; Cownie, 2021). The third and final difference for ARM will be the Trap Flag exception. There exists a single step debugging flag in the ARM architecture within the PSTATE register that can be set. However, the Single-Step (SS) flag cannot be set in user-mode based on the ARM architecture construction and therefore the Trap Flag technique is not viable for this research (*ARM Developer Documentation*, 2023). Other techniques that are developed at the operating system API level will not require specific assembly instructions and the compiler will handle the transformation from C++ to assembly on the correct architecture.

Exception-based Anti-Debugging Techniques

Raise SIGTRAP Signal

As seen in the Intel Anti-Debugging techniques, the INT3 instruction generates an interrupt to be handled by the debug exception handler. However, unlike the Windows operating system, Linux uses a signal handler to catch debug exceptions in addition to other signals. In addition, unlike the Linux operating system on the Intel architecture, the breakpoint instruction, BRK #F000, execution does not continue after hitting the breakpoint (*ARM Developer Documentation*, 2023; scottt, 2021). To account for this issue, the SIGTRAP signal must be raised using the raise() function instead of an assembly instruction that raises it for us. For this specific test, the implemented handler catches the SIGTRAP signal as part of the Anti-Debugging technique detection. If the signal handler catches the signal and handles it, then there is no debugger attached to the process. On the other hand, if the SIGTRAP

signal is raised to the debugger, the signal handler will not be executed and therefore alert the program to a debugger being present. By understanding what component of the operating system or processes handles the SIGTRAP signal, we can implement an Anti-Debugging check (Voisin, 2016).

Scanning-based Anti-Debugging Techniques

IsTracerPidZero

As part of starting a process on the Linux operating system, an entire process structure is setup in the /proc filesystem. This allows a process to reference information about itself by looking in the /proc/self directory. Within the /proc/self directory exists many files, but for this technique we can examine the status file which contains information about the status of the process. In this file exists a line, akin to a key-value pair, that lists the Process ID (PID) of a tracer process. In other words, the status file will give the PID of the debugging process if it exists. If there is no tracer program, the value of TracerPID will be zero. By checking this value we can easily see if a debugger, or tracer, is attached to this process based on the value of TracerPID and use this as an Anti-Debugging Technique (Roccia & Lesueur, 2023).

BRK Instruction Scanning

The BRK instruction scanning Anti-Debugging check works in the same way on ARM as it does on Intel architectures by looking through the code trying to identify the BRK instructions in the code. This instruction is represented by a single 32-bit value, 0xd4200000 (*ARM Developer Documentation*, 2023; scottt, 2021). Most production-like programs do not contain any BRK instructions in it due to the instruction only being used for debugging purposes. Therefore, if the code detects a BRK instruction in the code, it was likely put there by a debugger which would indicate the presence of a debugger. Thus, scanning for the BRK instruction in code where there are no BRK instructions expected will provide indications that a debugger is likely present and operate as an Anti-Debugging check (Miller, 2005; Roccia & Lesueur, 2023).

Code Checksum

To detect a debugger by computing a checksum of the code the program needs to select a starting point and ending point, or length, to compute a checksum over. This will detect changes to the code from the expected code. The checksum can be computed over all the code, important parts of the code, or whichever parts the author prefers. By calculating a checksum, if any bytes are changed the checksum will not equal the original value and the program will be aware of changes to the code. Since debuggers implement software breakpoints by overwriting bytes with new instructions, the original checksum will not equal the new checksum and it is likely a debugger is present. This will also work for identifying hooks or malicious software that overwrites or modifies code. If the checksum is equal to the original value, it is highly likely that no debugger is present and if the checksum is not equal, it is likely that a debugger made changes to the code. This technique works in the same way on ARM as it does on Intel architectures. For this research, the Fletcher-16 algorithm was selected based on ease of implementation with a sufficiently sized result to limit collisions (Giffin et al., 2005; Roccia & Lesueur, 2023; Stone et al., 1998).

Process Enumeration

For this scanning technique, the code looks at the environment the program is being executed in rather than the program itself. In some cases, analysis programs will execute in the background or in parallel looking for changes to the system or may be able to evade some Anti-Debugging checks. The technique iterates over all the running processes on the system and looks for known, undesired programs that are currently executing. This could be a debugger that isn't attached or some kind of analysis program that the author is aware of. If one of the programs on the deny list is present, the code can use that information as an Anti-Debugging check and decide about the continued execution of the program. The technique is implemented using the same C++ code as the Linux Intel version (Roccia & Lesueur, 2023).

Parent Process Scanning

This technique is included in the scanning group due to the need to scan all processes to find the parent process. For Linux, the getppid() function is available to return the Process ID (PID) of the parent process. Using this PID, we can once again query the /proc directory

using the Parent PID (PPID) as the second directory. Lastly, the technique reads the cmdline file which has the command-line execution of the process. We can check that no bad or denied process names are in that file or if the parent process is on an allow list, or not on a deny list, execution can continue as normal. This check can be used as the implementation for the Anti-Debugging technique (Roccia & Lesueur, 2023; Voisin, 2016).

Timing-based Anti-Debugging Techniques

Time

The Linux operating system provides a function, time(), that returns the number of seconds since the Epoch, 1970-01-01 00:00:00. Since this is such a coarse-grained measurement of time for programs executing so fast, the difference between the beginning and ending will have a small limit value. The limit value is determined by the author to decide when program execution has been too slow and is likely being debugged instead. Despite being a less precise measurement than other timing functions, measuring seconds for an analyst debugging a program is more than sufficient as analysts will operate significantly slower than a computer and even 1 to 2 seconds of delay is easily measurable and could pass the limit. By selecting a good limit value, the technique can use the measurement returned from time() to decide if a debugger is likely present and implement this Anti-Debugging technique (Roccia & Lesueur, 2023).

Clock

Using a different measurement than time(), clock() returns the amount of time used by a program to approximately microsecond precision. The microsecond precision is not guaranteed but does align with the POSIX standard which uses a CLOCKS_PER_SECOND value of 1,000,000. This level of precision does come with a trade-off in long-term measurements. The clock_t type that is returned uses a 32-bit integer which will cause an overflow approximately every 72 minutes according to the man page for clock(). Using this function in an identical manner to other timing tests, the initial value is subtracted from the after-execution value to give the duration. This duration is then compared to the limit given by the programmer. With a good limit, comparing the elapsed time to the limit gives sufficient implementation details for an Anti-Debugging Check (Roccia & Lesueur, 2023).

ClockGetTime

This technique aligns more closely with clock() than time() but uses more configurable parameters to extract specific timing information. The clock_gettime() function expects a specific clock ID and a resolution to return the timing values. This allows the programmer to return specific timing information at a resolution of their choosing. For this research, the technique uses the CLOCK_REALTIME ID and does not specify a resolution which will return the most accurate resolution available. The resolution does support nanosecond precision, however the actual measurement at that resolution is based on the CPU processing speed. Once again, the time value is captured before and after code execution and subtracted to calculate the duration. Once a good limit is selected, the comparison between duration and the limit will determine the likely presence of a debugger and can be used as an Anti-Debugging Technique (Roccia & Lesueur, 2023).

CNTVCT_EL0 (RDTSC)

The CNTVCT_EL0 register is part of the ARM architecture and, using the MRS instruction, code can read the return value of the virtual time-stamp counter. This counter is incremented every clock cycle and can be used to measure elapsed time between executed instructions. While the counter does not map directly to a unit of time without the CPU frequency, on the same system the increment of the counter is consistent and provides a precise way to measure time between executing instructions. Like other timing techniques, an author must set a threshold and the measurement of the duration of execution between the beginning and ending of the measured instructions can provide information about the presence of a debugger. If the threshold is exceeded, the program can use that information to determine the presence of a debugger as an Anti-Debugging check. Due to a single instruction not existing in ARM like RDTSC on the Intel architecture, reading from this register using the MRS instruction is functionally equivalent to the RDTSC instruction on ARM (*ARM Developer Documentation*, 2023; Cownie, 2021).

Miscellaneous Anti-Debugging Techniques

Memory Encryption

Memory encryption is another way to implement Anti-Debugging as the information is encrypted and cannot be analyzed until after it is decrypted. This technique can be used to protect sensitive code or data and, aside from exploiting cryptography which will not be discussed in this research, the information cannot be recovered until it is executed or used in memory. As one might expect, memory encryption carries a large performance overhead due to the increased number of functions, memory management, and encryption/decryption routines needed for use of the code or data (Vrba et al., 2010).

For this research, the implementation used on Linux is OpenSSL's AES cipher (*OpenSSL*, 2024). Before the experiment can even be run, the initial setup must be completed instantiating the algorithm, Initialization Vector (IV), Symmetric Key, and initially encrypting the data. This setup is also implemented using gen_params and aes_encrypt in an identical manner to the Linux Intel version. This setup is not included in the measurement as it is a one-time expense that can be done at the beginning of the program and used for its duration. The measurement occurs when the information is decrypted, processed, and then re-encrypted using aes_encrypt and aes_decrypt. This behavior is what one would expect to impact performance most while the sensitive code or data is being used and would increase the performance overhead of the program. While this is still an Anti-Debugging technique, there is no Boolean check for the presence of a debugger as the debugger is pre-emptively defeated by the encryption and can only be ascertained once the code or data is decrypted in memory (Roccia & Lesueur, 2023).

Anti-Disassembly

While Anti-Dissasembly techniques are not always congruent to Anti-Debugging techniques, they are included in this research as an experiment to demonstrate the performance overhead as it relates to increasing the difficulty of debugging, which is one of the purposes of Anti-Debugging. These techniques can be implemented in various ways and for this research the developed technique combined moving function pointers, loop unrolling, and bogus calls unrelated to primary purpose. Each of these techniques by themselves

contribute to the performance overhead of the execution, however combining them for this research gives a good introduction to the performance overhead from an Anti-Debugging perspective while using Anti-Disassembly primitives.

For the first technique, function pointer movement is simply the obfuscation of which calls will be made by calculating the function address based on the original and calling the new function instead of the old one. The second technique, loop unrolling, can sometimes even have performance gains depending on the architecture, code implementation, and compiler optimizations. For this research no compiler optimizations are used thus increasing the performance overhead by loop unrolling. Lastly, bogus function calls distract analysts and reverse engineers from understanding the primary purpose of the routine. This increases time spent understanding the code and therefore hinders debugging, making it an Anti-Debugging technique. This implementation is functionally equivalent to the Windows implementation (Roccia & Lesueur, 2023).

Self-Debugging

Self-Debugging for this research against the Linux operating system uses ptrace to hinder other debuggers from viewing the program at all. The Linux operating system only allows one program to debug another at a time. Therefore, for this technique we implemented a child program that is started by the Anti-Debugging technique that is executed by the parent which attaches itself to the parent as the sole debugger. Since the parent is already being debugged, a debugger cannot be attached to the parent preemptively denying a debugger the opportunity to analyze the program. The parent starts this process by fork-ing itself and calling system on the child process. The parent continues to execute the desired functionality while the child process attaches to the parent using ptrace. Additionally, the child also wants to ignore debug events and automatically continues execution if any debug events are raised. This is identical to the Linux Intel version of Self-Debugging. This technique has a large performance overhead as the parent must start a child process which attaches and waits to be killed by the parent after the parent has finished execution. Like other miscellaneous techniques, this technique does not have a Boolean check for a debugger and instead prevents a debugger from attaching to it in the first place (Abrath et al., 2016, 2020; kirschju, 2018; Miller, 2005; Voisin, 2016).

Artifact Creation

The artifact created for this research relied on Python3 to implement the data processing, statistical tests, and data visualization. The artifact consists of two different python scripts, the statistics script, and the artifact script.

The statistics script uses the argparse package to provide a help menu and familiar user interface to users of the script and allows for specification of architecture to determine which data should be processed. In addition, a user can set the directory to look for logs of information to process. After the data has been processed, a user has the option to export the data to CSV files that can be read into the artifact script. By default, the script will only export logs that rejected the null hypothesis at the alpha=0.05 level. There exists an optional flag to export all data regardless of the p-value. Lastly, if a user is not interested in graphs being generated an option to skip the graph generation is available (The pandas development team, 2020; Virtanen et al., 2020).

The data is loaded from the specified directory or a default of the current working directory with file extensions of .log. The logs are parsed according to comma separated values but had additional information from the Anti-Debugging techniques implementation that needed to be removed. Post data processing, the data is loaded into pandas DataFrame objects that facilitate numpy and scipy routines to operate seamlessly on the data. To make the data plots, the python package, seaborn, was used which wraps the matplotlib package to make data visualization easier for developers (The pandas development team, 2020; Virtanen et al., 2020).

The script then creates the data visualization for the data models, the Mann-Whitney U statistic, the Levene test statistic, and the quartile data. All the statistics are computed using the scipy.stats package implementations and this data populates the statistics tables in Chapter 4 and Appendix C (The pandas development team, 2020; Virtanen et al., 2020).

The artifact script also uses the argparse package to provide two flags for usage, the file to read test data from and the architecture of data to check against debugging models. If no architecture is provided, the artifact compares the testing data to all available debugging models. The debugging model data are in sub-folders of the artifact directory and are read into memory based on the value of the architecture flag. All the data, test data and debugging models, are read in from disk and are outputs of the exported data from the statistics script,

which is in CSV format. Every debugging model is compared with the test data using the levene test from the scipy.stats package and any model that has a p-value score greater than 0.05 is added to the detection list. This detection list is then sorted from highest to lowest p-value and displayed to the user as identification of the anti-debugging technique detected. The higher the p-value, the more similar the test data and the debugging model are based on the Levene test. In all cases, the p-value being larger than 0.05 means we fail to reject the null hypothesis (Levene, 1960; The pandas development team, 2020; Virtanen et al., 2020).

Novelty and Justifications

This research is novel due to the measurement and creation of data models measuring the performance of Anti-Debugging techniques, the statistical tests applied to measure the differences between a control group, and the creation of an artifact using statistical tests to measure similarity between samples and Anti-Debugging technique models of performance.

The measurement of the performance of Anti-Debugging techniques is novel as this type of research has not been done before for Anti-Debugging techniques. However, there are minimal non-invasive capabilities to assess the presence of Anti-Debugging techniques where the Anti-Debugging technique could not detect the measurement and change its behavior. The other research that avoided detection while detecting Anti-Debugging techniques required virtualization or hypervisors (Pék et al., 2011; Qi et al., 2012; Vasudevan & Terraballi, 2005; Vasudevan & Yerraballi, 2006). In addition, using the industry standard for measurements (Paoloni, 2010) provided accurate data with minimal overhead using assembly level instructions that are forced to be synchronized and compiled inline to the greatest extent possible in user-mode.

While using statistical tests in research is not novel by itself, the application of the Mann-Whitney U and Levene tests are novel for testing differences in data distributions of performance data while considering the non-normal data distributions that require less common statistical tests. These tests are all non-parametric which means the information is more accurate than parametric tests that require the data be normally distributed. Using the correct tests increases the accuracy of the results while determining whether one can reject the null hypotheses for these tests (Levene, 1960; “Levene Test for Equality of Variances,” n.d.; Mann & Whitney, 1947; Nahm, 2016).

Lastly, the collection of the data models and implementation of an artifact to collect, process, and test sample performance datasets against the Anti-Debugging performance models is also novel in that it has never been done before. To provide accurate results, the results from the Levene test are used rather than the Mann-Whitney U tests. The measurement of median is only unique for these specific data sets. A longer running program will always have different performance values for median values even if the sample is benign. However, the fluctuation in execution times due to Anti-Debugging techniques can be captured in the variance of the data set and if it is similarly found in another data set, it is possible the Anti-Debugging technique is present there too. By failing to reject the null hypothesis from the results of the sample data set and an Anti-Debugging technique data set, which is the information provided by the artifact, it is reasonable to say the variances are sufficiently similar and an Anti-Debugging technique is likely present.

CHAPTER 5

EXPERIMENTS (CASE STUDY)

Intel Windows Experiments

Experimental Set Up

The Intel Windows experiments were run on a virtual Windows 10 Build 17763 x64 machine. The virtualization was provided by VMWare Workstation 16.2.5 as a type-2 hypervisor. The Windows 10 VM had 4 vCPUs, 16GB of memory, and 40GB of hard drive space.

Each experiment for each of the Anti-Debugging techniques was run in serial to prevent any performance impacts between tests. Each test was run 1010 times which allowed for the first ten results to be thrown out while the data and instruction cache were populated with relevant information based on Paoloni. (Paoloni, 2010)

Each test executed the same code base code while measuring the performance impact of the anti-debugging technique in addition to the baseline. Only the base test experiment did not have any anti-debugging techniques to create the baseline. The base test was also subjected to 1010 test executions where the first ten results were thrown out as well.

Due to the limitation of being unable to acquire exclusive execution due to a lack of kernel-level implementation, the multiple vCPUs could cause changes in execution duration due to context switching, however this was minimized by exclusively executing the experiment on the VM with no other running user-mode programs.

Experiment Implementation

The implementations of each Anti-Debugging technique are in the Windows source code. The code is compiled with MSVC version 19.33 as part of the Microsoft Visual Studio 17.3.4 package. Once the data has been collected, the logs are run through a python script that converts the data log to a Pandas DataFrame object (The pandas development team, 2020).

This object is combined with data visualization customizations based on each test to generate the graphs of the distribution of data. In addition, the Base Test and Anti-Debugging technique data are passed into SciPy implementations of the Mann-Whitney U test and the Levene test (Levene, 1960; "Levene Test for Equality of Variances," n.d.; Mann & Whitney, 1947; Virtanen et al., 2020). These values, in addition to the quartile values, populate the tables in the following section. The quartile values are measured at the 0, 25, 50, 75, and 100% percentile values which are equivalent to the minimum, first quartile, median, third quartile, and maximum values of the data distribution.

Experiment Results

For each of the experiments, distribution graphs are generated, and three statistical tests are run to measure different aspects of the data in addition to quantifying the minimum, 25^{th} percentile, median, 75^{th} percentile, and maximum data values for each test.

The graph visualizations do not display all data points as the results for many tests are heavily skewed right. The graph has been trimmed down to give a better visualization of the data distribution than comprehensive inclusion of all data points. The data was not modified to generate this graph and no data was excluded for statistical tests beyond the initial ten results removed discussed in Chapter 3 and the experimental setup.

Base Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	500000	0.5
Levene Test	0	1

Table 1 – Statistical Test Results for Base Test vs. Base Test

The Base Test is used as the control group whereas the other tests are run with Anti-Debugging Techniques acting as the applied treatment. The differences in the measurement of the performance data indicate the impact that the Anti-Debugging technique has on the program's performance. For the Base Test, no treatment is applied, and the results are equivalent. This is confirmed with the statistical results that have a p-value of 1.0 for the Levene test. This makes sense as we fail to reject the null hypothesis in each of these cases as the data sets are the same and therefore the median and variance are the same.

CheckRemoteDebuggerPresent Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	239900	1.01e-96
Levene Test	1.34	0.247

Table 2 - Statistical Test Results for CheckRemoteDebugger Test vs. Base Test

The Mann-Whitney U test results are U=239,900 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the CheckRemoteDebuggerPresent data set median.

Second, the Levene test results are W=1.34 and p=0.25. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and CheckRemoteDebuggerPresent Test data sets have different variances.

Overall, the inability to differentiate the CheckRemoteDebuggerPresent dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant difference between the Base Test and the CheckRemoteDebuggerPresent dataset, must limit our expectations for the artifact as the Levene test will likely introduce false positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

CloseHandle Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	405556	8.06e-16
Levene Test	7.55	0.006

Table 3 - Statistical Test Results for CloseHandle Test vs. Base Test

The Mann-Whitney U test results are U=405,556 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the CloseHandle data set median.

Additionally, the Levene test results are W=7.55 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and CheckRemoteDebuggerPresent Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The CloseHandle data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the CloseHandle data set model when used with the artifact.

Code Checksum Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	389016	6.54e-20
Levene Test	10.1	0.001

Table 4 – Statistical Test Results for Code Checksum Test vs. Base Test

The Mann-Whitney U test results are U=389,016 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis

and can say that the Base Test data set median is different from the Code Checksum data set median.

Additionally, the Levene test results are W=10.2 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Code Checksum Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Code Checksum data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Code Checksum data set model when used with the artifact.

Code Obfuscation Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	304611	8.85e-57
Levene Test	3.48	0.062

Table 5 – Statistical Test Results for Code Obfuscation Test vs. Base Test

The Mann-Whitney U test results are U=304,611 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Code Obfuscation data set median.

Lastly, the Levene test results are W=3.48 and p=0.06. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and Code Obfuscation Test data sets have different variances.

Overall, the inability to differentiate the Code Obfuscation dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant difference between the Base Test and the Code Obfuscation dataset, must limit our expectations for the artifact as the Levene test will likely introduce false positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

CsrGetProcessID Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	566019	1.00
Levene Test	3.09	0.079

Table 6 – Statistical Test Results for CsrGetProcessID Test vs. Base Test

The Mann-Whitney U test results are U=566,019 and p=1.0 which is above the conventional threshold for statistical significance. For this test, we fail to reject the null hypothesis and cannot say that the Base Test data set median is different from the CsrGetProcessID data set median.

Second, the Levene test results are W=3.09 and p=0.08. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and CsrGetProcessID Test data sets have different variances.

Overall, the inability to differentiate the CsrGetProcessID dataset from the Base Test data using the Mann-Whitney U test and the Levene test means that it is unlikely to successfully detect this Anti-Debugging technique using the artifact. The high p-value for the Mann-Whitney U test identifies a similarity in the median of the performance data distribution compared to the Base test. In addition, the high p-value for the Levene test means the

difference in variances between the Base Test data and CsrGetProcessID data is minimal. Therefore, the distributions of the Base Test and CsrGetProcessID data set are similar in both median and variance and not only difficult to detect in this research but also for the artifact. Since the artifact uses the Levene test to measure variance, this technique, if used as part of the artifact, will have a high false positive rate as it's like the control group data.

Debug Registers Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	213705	6.39e-114
Levene Test	23.3	1.51e-6

Table 7 – Statistical Test Results for Debug Registers Test vs. Base Test

The Mann-Whitney U test results are U=213,705 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Debug Registers data set median.

Additionally, the Levene test results are W=23.3 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Debug Registers Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Debug Registers data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Debug Registers data set model when used with the artifact.

Process Enumeration Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	92678	2.57e-227
Levene Test	48.3	4.90e-12

Table 8 – Statistical Test Results for Process Enumeration Test vs. Base Test

The Mann-Whitney U test results are U=92,678 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Process Enumeration data set median.

Additionally, the Levene test results are W=48.3 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Process Enumeration Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Process Enumeration data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Process Enumeration data set model when used with the artifact.

Event Pair Handle Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	307361	4.96e-57
Levene Test	3.32	0.069

Table 9 – Statistical Test Results for Event Pair Handle Test vs. Base Test

The Mann-Whitney U test results are U=307,361 and $p < 0.05$ which is below the conventional threshold for statistical significance. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Event Pair Handle data set median.

Second, the Levene test results are W=3.32 and p=0.07. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and Event Pair Handle Test data sets have different variances.

Overall, the inability to differentiate the Event Pair Handle dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant difference between the Base Test and the Event Pair Handle dataset, must limit our expectations for the artifact as the Levene test will likely introduce false positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

GetLocalTime Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	559936	1.00
Levene Test	3.76	0.053

Table 10 – Statistical Test Results for GetLocalTime vs. Base Test

The Mann-Whitney U test results are U=559,936 and p=1.0 which is above the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we fail to reject the null hypothesis and cannot say that the Base Test data set median is different from the GetLocalTime data set median.

Second, the Levene test results are W=3.76 and p=0.053. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and GetLocalTime Test data sets have different variances.

Overall, the inability to differentiate the GetLocalTime dataset from the Base Test data using the Mann-Whitney U test and the Levene test means that it is unlikely to successfully detect this Anti-Debugging technique using the artifact. The high p-value for the Mann-Whitney U test identifies a similarity in the median of the performance data distribution compared to the Base test. In addition, the high p-value for the Levene test means the difference in variances between the Base Test data and GetLocalTime data is minimal. Therefore, the distributions of the Base Test and GetLocalTime data set are similar in both median and variance and not only difficult to detect in this research but also for the artifact. Since the artifact uses the Levene test to measure variance, this technique, if used as part of the artifact, will have a high false positive rate as it's like the control group data.

GetSystemTime Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	454268	3.65e-5
Levene Test	5.81	0.016

Table 11 – Statistical Test Results for GetSystemTime vs. Base Test

The Mann-Whitney U test results are U=454,269 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the GetSystemTime data set median.

Additionally, the Levene test results are W=5.81 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and GetSystemTime Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The GetSystemTime data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the GetSystemTime data set model when used with the artifact.

GetTickCount Test Result

Test Name	Statistic Value	p-value
Mann-Whitney U Test	445869	1.45e-6
Levene Test	3.90	0.048

Table 12 – Statistical Test Results for GetTickCount vs. Base Test

The Mann-Whitney U test results are U=445,869 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the GetTickCount data set median.

Additionally, the Levene test results are W=3.90 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and GetTickCount Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The GetTickCount data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-

value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the GetTickCount data set model when used with the artifact.

Guard Page Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	157403	3.94e-161
Levene Test	19.1	1.33e-5

Table 13 – Statistical Test Results for Guard Page vs Base Test

The Mann-Whitney U test results are U=157,404 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Guard Page data set median.

Additionally, the Levene test results are W=19.1 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Guard Page Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Guard Page data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Guard Page data set model when used with the artifact.

Heap Flag Test Results

Test Name	Statistic Value	p-value

Mann-Whitney U Test	570565	1.00
Levene Test	0.06	0.801

Table 14 – Statistical Test Results for Heap Flag vs. Base Test

The Mann-Whitney U test results are U=570565 and p=1.0 which is above the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we fail to reject the null hypothesis and cannot say that the Base Test data set median is different from the Heap Flag data set median.

Second, the Levene test results are W=0.06 and p=0.80. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and Heap Flag Test data sets have different variances.

Overall, the inability to differentiate the Heap Flag dataset from the Base Test data using the Mann-Whitney U test and the Levene test means that it is unlikely to successfully detect this Anti-Debugging technique using the artifact. The high p-value for the Mann-Whitney U test identifies a similarity in the median of the performance data distribution compared to the Base test. In addition, the high p-value for the Levene test means the difference in variances between the Base Test data and Heap Flag data is minimal. Therefore, the distributions of the Base Test and Heap Flag data set are similar in both median and variance and not only difficult to detect in this research but also for the artifact. Since the artifact uses the Levene test to measure variance, this technique, if used as part of the artifact, will have a high false positive rate as it's like the control group data.

INT 0x2D Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	434190	1.53e-8
Levene Test	16.5	4.97e-5

Table 15 – Statistical Test Results for INT 0x2d vs. Base Test

The Mann-Whitney U test results are U=434,190 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the INT 0x2D data set median.

Additionally, the Levene test results are W=16.5 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and INT 0x2D Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The INT 0x2D data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the INT 0x2D data set model when used with the artifact.

INT3 Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	415310	3.41e-13
Levene Test	1.92	0.166

Table 16 – Statistical Test Results for INT3 vs. Base Test

The Mann-Whitney U test results are U=415,310 and $p < 0.05$ which is below the conventional threshold for statistical significance. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the INT3 data set median.

Second, the Levene test results are W=1.92 and p=0.17. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in

this test. As a result, we cannot say that the Base Test and INT3 Test data sets have different variances.

Overall, the inability to differentiate the INT3 dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant difference between the Base Test and the INT3 dataset, must limit our expectations for the artifact as the Levene test will likely introduce false positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

INT3 Scan Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	371568	9.59e-26
Levene Test	5.77	0.016

Table 17 – Statistical Test Results for INT3 Scan vs. Base Test

The Mann-Whitney U test results are U=371,568 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the INT3 Scanning data set median.

Additionally, the Levene test results are W=5.77 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and INT3 Scanning Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The INT3 Scanning data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique

on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the INT3 Scanning data set model when used with the artifact.

IsDebuggerPresent Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	471662	0.006
Levene Test	1.71	0.191

Table 18 – Statistical Test Results for IsDebuggerPresent vs. Base Test

The Mann-Whitney U test results are U=471,662 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the IsDebuggerPresent data set median.

Second, the Levene test results are W=1.71 and p=0.19. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and IsDebuggerPresent Test data sets have different variances.

Overall, the inability to differentiate the IsDebuggerPresent dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant difference between the Base Test and the IsDebuggerPresent dataset, must limit our expectations for the artifact as the Levene test will likely introduce false

positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

Memory Encryption Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	103997	8.03e-214
Levene Test	8.88	0.029

Table 19 – Statistical Test Results for Memory Encryption vs. Base Test

The Mann-Whitney U test results are U=103997 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Memory Encryption data set median.

Additionally, the Levene test results are W=8.88 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Memory Encryption Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Memory Encryption data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Memory Encryption data set model when used with the artifact.

NtGlobalFlag Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	541079	1.00

Levene Test	3.72	0.054

Table 20 – Statistical Test Results for NtGlobalFlag vs. Base Test

The Mann-Whitney U test results are U=541079 and p=1.0 which is above the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we fail to reject the null hypothesis and cannot say that the Base Test data set median is different from the NtGlobalFlag data set median.

Second, the Levene test results are W=3.72 and p=0.053. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and NtGlobalFlag Test data sets have different variances.

Overall, the inability to differentiate the NtGlobalFlag dataset from the Base Test data using the Mann-Whitney U test and the Levene test means that it is unlikely to successfully detect this Anti-Debugging technique using the artifact. The high p-value for the Mann-Whitney U test identifies a similarity in the median of the performance data distribution compared to the Base test. In addition, the high p-value for the Levene test means the difference in variances between the Base Test data and NtGlobalFlag data is minimal. Therefore, the distributions of the Base Test and NtGlobalFlag data set are similar in both median and variance and not only difficult to detect in this research but also for the artifact. Since the artifact uses the Levene test to measure variance, this technique, if used as part of the artifact, will have a high false positive rate as it's like the control group data.

NtQueryInformationProcess Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	411976	6.72e-14
Levene Test	12	0.0006

Table 21 – Statistical Test Results for NtQueryInformationProcess vs. Base Test

The Mann-Whitney U test results are U=411,976 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the

differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the NtQueryInformationProcess data set median.

Additionally, the Levene test results are W=12.0 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and NtQueryInformationProcess Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The NtQueryInformationProcess data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the NtQueryInformationProcess data set model when used with the artifact.

NtQueryObject Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	397929	1.72e-18
Levene Test	16.6	4.78e-5

Table 22 – Statistical Test Results for NtQueryObject vs. Base Test

The Mann-Whitney U test results are U=397,929 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the NtQueryObject data set median.

Additionally, the Levene test results are W=16.6 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in

this test. As a result, we can say that the Base Test and NtQueryObject Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The NtQueryObject data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the NtQueryObject data set model when used with the artifact.

NtQueryPerformanceCounter Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	119424	4.90e-197
Levene Test	91.6	3.077e-21

Table 23 – Statistical Test Results for NtQueryPerformanceCounter vs. Base Test

The Mann-Whitney U test results are U=119,424 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the NtQueryPerformanceCounter data set median.

Additionally, the Levene test results are W=91.6 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and NtQueryPerformanceCounter Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The NtQueryPerformanceCounter data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-

Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the NtQueryPerformanceCounter data set model when used with the artifact.

NtSetDebugFilterState Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	603221	1.0
Levene Test	1.62	0.203

Table 24 – Statistical Test Results for NtSetDebugFilterState vs. Base Test

The Mann-Whitney U test results are U=603,221 and p=1.0 which is above the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we fail to reject the null hypothesis and cannot say that the Base Test data set median is different from the NtSetDebugFilterState data set median.

Second, the Levene test results are W=1.62 and p=0.20. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and NtSetDebugFilterState Test data sets have different variances.

Overall, the inability to differentiate the NtSetDebugFilterState dataset from the Base Test data using the Mann-Whitney U test and the Levene test means that it is unlikely to successfully detect this Anti-Debugging technique using the artifact. The high p-value for the Mann-Whitney U test identifies a similarity in the median of the performance data distribution compared to the Base test. In addition, the high p-value for the Levene test means the difference in variances between the Base Test data and NtSetDebugFilterState data is minimal. Therefore, the distributions of the Base Test and NtSetDebugFilterState data set are similar in both median and variance and not only difficult to detect in this research but also for the artifact. Since the artifact uses the Levene test to measure variance, this technique, if

used as part of the artifact, will have a high false positive rate as it's like the control group data.

NtSetInformationThread Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	199105	9.26e-126
Levene Test	2.53	0.112

Table 25 – Statistical Test Results for NtSetInformationThread vs. Base Test

The Mann-Whitney U test results are U=199,105 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the NtSetInformationThread data set median.

Second, the Levene test results are W=2.53 and p=0.11. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and NtSetInformationThread Test data sets have different variances.

Overall, the inability to differentiate the NtSetInformationThread dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant difference between the Base Test and the NtSetInformationThread dataset, must limit our expectations for the artifact as the Levene test will likely introduce false positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

OutputDebugString Test Results

Test Name	Statistic Value	p-value

Mann-Whitney U Test	545392	1.0
Levene Test	1.01	0.315

Table 26 – Statistical Test Results for OutputDebugString vs. Base Test

The Mann-Whitney U test results are U=545,392 and p=1.0 which is above the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we fail to reject the null hypothesis and cannot say that the Base Test data set median is different from the OutputDebugString data set median.

Second, the Levene test results are W=1.01 and p=0.32. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and OutputDebugString Test data sets have different variances.

Overall, the inability to differentiate the OutputDebugString dataset from the Base Test data using the Mann-Whitney U test and the Levene test means that it is unlikely to successfully detect this Anti-Debugging technique using the artifact. The high p-value for the Mann-Whitney U test identifies a similarity in the median of the performance data distribution compared to the Base test. In addition, the high p-value for the Levene test means the difference in variances between the Base Test data and OutputDebugString data is minimal. Therefore, the distributions of the Base Test and OutputDebugString data set are similar in both median and variance and not only difficult to detect in this research but also for the artifact. Since the artifact uses the Levene test to measure variance, this technique, if used as part of the artifact, will have a high false positive rate as it's like the control group data.

Parent Process Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	0	0
Levene Test	291	5.10e-61

Table 27 – Statistical Test Results for Parent Process vs. Base Test

The Mann-Whitney U test results are U=0.0 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set second is different from the Parent Process data set second.

Additionally, the Levene test results are W=291 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Parent Process Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Parent Process data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Parent Process data set model when used with the artifact.

RDTSC Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	599717	1.0
Levene Test	3.04	0.082

Table 28 – Statistical Test Results for RDTSC vs. Base Test

The Mann-Whitney U test results are U=599,717 and p=1.0 which is above the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we fail to reject the null hypothesis and cannot say that the Base Test data set median is different from the RDTSC data set median.

Second, the Levene test results are W=3.04 and p=0.08. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and RDTSC Test data sets have different variances.

Overall, the inability to differentiate the RDTSC dataset from the Base Test data using the Mann-Whitney U test and the Levene test means that it is unlikely to successfully detect this Anti-Debugging technique using the artifact. The high p-value for the Mann-Whitney U test identifies a similarity in the median of the performance data distribution compared to the Base test. In addition, the high p-value for the Levene test means the difference in variances between the Base Test data and RDTSC data is minimal. Therefore, the distributions of the Base Test and RDTSC data set are similar in both median and variance and not only difficult to detect in this research but also for the artifact. Since the artifact uses the Levene test to measure variance, this technique, if used as part of the artifact, will have a high false positive rate as it's like the control group data.

Self-Debugging Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	1647	0.0
Levene Test	235	3.00e-50

Table 29 – Statistical Test Results for Self-Debugging vs. Base Test

The Mann-Whitney U test results are U=1647 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Self-Debugging data set median.

Additionally, the Levene test results are W=235 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Self-Debugging Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Self-Debugging data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Self-Debugging data set model when used with the artifact.

timeGetTime Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	212491	1.83e-117
Levene Test	3.75	0.053

Table 30 – Statistical Test Results for timeGetTime vs. Base Test

The Mann-Whitney U test results are U=212,491 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the timeGetTime data set median.

Lastly, the Levene test results are W=3.75 and p=0.053. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and timeGetTime Test data sets have different variances.

Overall, the inability to differentiate the timeGetTime dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant

difference between the Base Test and the timeGetTime dataset, must limit our expectations for the artifact as the Levene test will likely introduce false positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

Trap Flag Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	591570	1.0
Levene Test	1.98	0.159

Table 31 – Statistical Test Results for Trap Flag vs. Base Test

The Mann-Whitney U test results are U=591,570 and p=1.0 which is above the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we fail to reject the null hypothesis and cannot say that the Base Test data set median is different from the Trap Flag data set median.

Second, the Levene test results are W=1.98 and p=0.16. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and Trap Flag Test data sets have different variances.

Overall, the inability to differentiate the Trap Flag dataset from the Base Test data using the Mann-Whitney U test and the Levene test means that it is unlikely to successfully detect this Anti-Debugging technique using the artifact. The high p-value for the Mann-Whitney U test identifies a similarity in the median of the performance data distribution compared to the Base test. In addition, the high p-value for the Levene test means the difference in variances between the Base Test data and Trap Flag data is minimal. Therefore, the distributions of the Base Test and Trap Flag data set are similar in both median and variance and not only difficult to detect in this research but also for the artifact. Since the artifact uses the Levene test to measure variance, this technique, if used as part of the artifact, will have a high false positive rate as it's like the control group data.

UnhandledExceptionFilter Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	172094	1.77e-149
Levene Test	65.0	1.29e-15

Table 32 – Statistical Test Results for UnhandledExceptionFilter vs. Base Test

The Mann-Whitney U test results are U=172,094 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the UnhandledExceptionFilter data set median.

Additionally, the Levene test results are W=65.0 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and UnhandledExceptionFilter Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The UnhandledExceptionFilter data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the UnhandledExceptionFilter data set model when used with the artifact.

Intel Linux Experiments

Experimental Set Up

The Intel Linux experiments were run on a virtual Ubuntu 22.04.1 LTS x64 machine. The Linux kernel is version 5.15.0-91-generic. The virtualization was provided by VMWare

Workstation 16.2.5 as a type-2 hypervisor. The Ubuntu VM had 2 vCPUs, 4GB of memory, and 100GB of hard drive space.

Each experiment for each of the Anti-Debugging techniques was run in serial to prevent any performance impacts between tests. Each test was run 1010 times which allowed for the first ten results to be thrown out while the data and instruction cache were populated with relevant information based on Paoloni. (Paoloni, 2010)

Each test executed the same code base code while measuring the performance impact of the anti-debugging technique in addition to the baseline. Only the base test experiment did not have any anti-debugging techniques to create the baseline. The base test was also subjected to 1010 test executions where the first ten results were thrown out as well.

Due to the limitation of being unable to acquire exclusive execution due to a lack of kernel-level implementation, the multiple vCPUs could cause changes in execution duration due to context switching, however this was minimized by exclusively executing the experiment on the VM with no other running user-mode programs.

Experiment Implementation

The implementations of each Anti-Debugging technique are in the Linux source code. The code is compiled with gcc version 11.4.0. Once the data has been collected, the logs are run through a python script that converts the data log to a Pandas DataFrame object (The pandas development team, 2020). This object is combined with data visualization customizations based on each test to generate the graphs of the distribution of data. In addition, the Base Test and Anti-Debugging technique data are passed into SciPy implementations of the Kruskal test, Mann-Whitney U test, and the Levene test (Levene, 1960; “Levene Test for Equality of Variances,” n.d.; Mann & Whitney, 1947; Virtanen et al., 2020). These values, in addition to the quartile values, populate the tables in the following section. The quartile values are measured at the 0, 25, 50, 75, and 100% percentile values which are equivalent to the minimum, first quartile, median, third quartile, and maximum values of the data distribution.

Experiment Results

Base Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	500000	0.5
Levene Test	0	1

Table 33 – Statistical Test Results for Base Test vs. Base Test

The base test is used as the control group whereas the other tests are run with Anti-Debugging Techniques acting as the applied treatment. The differences in the measurement of the performance data indicate the impact that the Anti-Debugging technique has on the program's performance. For the Base Test, no treatment is applied, and the results are equivalent. This is confirmed with the statistical results that have a p-value of 1.0 for Kruskal and Levene tests. This makes sense as we fail to reject the null hypothesis in each of these cases as the data sets are the same and therefore the median and variance are the same.

ClockGetTime Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	683457	1.0
Levene Test	9.57	0.002

Table 34 – Statistical Test Results for ClockGetTime Test vs. Base Test

The Mann-Whitney U test results are U=683,457 and p=1.0 which is above the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we fail to reject the null hypothesis and cannot say that the Base Test data set median is different from the ClockGetTime data set median.

Second, the Levene test results are W=9.57 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and ClockGetTime Test data sets have different variances.

Overall, the Levene test returned statistically significant results and can be differentiated from the Base Test data. The ClockGetTime data set is not measurably different from the Base Test using the Mann-Whitney U test based on the p-value being greater than 0.05. This means that there is low correlation between the treatment of the Anti-Debugging technique on the Base Test for this research regarding the median. However, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the ClockGetTime data set model when used with the artifact.

Clock Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	384085	9.25e-23
Levene Test	36.6	1.76e-9

Table 35 – Statistical Test Results for Clock Test vs. Base Test

The Mann-Whitney U test results are U=384,086 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Clock data set median.

Lastly, the Levene test results are W=36.56 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Clock Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Clock data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation

between the presence of an Anti-Debugging technique in the sample and the Clock data set model when used with the artifact.

Code Checksum Test

Test Name	Statistic Value	p-value
Mann-Whitney U Test	20513	0.0
Levene Test	4.05	0.044

Table 36 – Statistical Test Results for Code Checksum Test vs. Base Test

The Mann-Whitney U test results are $U=20513$ and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Code Checksum data set median.

Additionally, the Levene test results are $W=4.05$ and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Code Checksum Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Code Checksum data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Code Checksum data set model when used with the artifact.

Code Obfuscation Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	0	0

Levene Test	469	1.31e-93

Table 37 – Statistical Rest Results for Code Obfuscation Test vs. Base Test

The Mann-Whitney U test results are U=0.0 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Code Obfuscation data set median.

Additionally, the Levene test results are W=469 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Code Obfuscation Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Code Obfuscation data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Code Obfuscation data set model when used with the artifact.

Process Enumeration Test

Test Name	Statistic Value	p-value
Mann-Whitney U Test	0	0
Levene Test	67.8	3.24e-16

Table 38 – Statistical Test Results for Process Enumeration Test vs. Base Test

The Mann-Whitney U test results are U=0.0 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis

and can say that the Base Test data set median is different from the Process Enumeration data set median.

Additionally, the Levene test results are W=67.8 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Process Enumeration Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Process Enumeration data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Process Enumeration data set model when used with the artifact.

INT3 Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	149127	4.48e-171
Levene Test	97.4	1.82e-22

Table 39 – Statistical Test Results for INT3 Test vs. Base Test

The Mann-Whitney U test results are U=149,127 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the INT3 data set median.

Additionally, the Levene test results are W=97.4 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and INT3 Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The INT3 data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the INT3 data set model when used with the artifact.

INT3 Scanning Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	185455	1.93e-182
Levene Test	10.56	0.001

Table 40 – Statistical Test Results for INT3 Scanning Test vs. Base Test

The Mann-Whitney U test results are U=185,455 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the INT3 Scanning data set median.

Additionally, the Levene test results are W=10.6 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and INT3 Scanning Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The INT3 Scanning data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-

value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the INT3 Scanning data set model when used with the artifact.

IsTracerPidZero Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	1705	0.0
Levene Test	133	8.75e-30

Table 41 – Statistical Test Results for IsTracerPidZero Test vs. Base Test

The Mann-Whitney U test results are $U=1705$ and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the IsTracerPidZero data set median.

Additionally, the Levene test results are $W=133$ and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and IsTracerPidZero Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The IsTracerPidZero data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the IsTracerPidZero data set model when used with the artifact.

Memory Encryption Test Results

Test Name	Statistic Value	p-value

Mann-Whitney U Test	0	0
Levene Test	445	2.11e-89

Table 42 – Statistical Test Results for Memory Encryption Test vs. Base Test

The Mann-Whitney U test results are U=0.0 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Memory Encryption data set median.

Additionally, the Levene test results are W=445 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Memory Encryption Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Memory Encryption data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Memory Encryption data set model when used with the artifact.

Parent Process Test

Test Name	Statistic Value	p-value
Mann-Whitney U Test	1957	0.0
Levene Test	0.564	0.453

Table 43 – Statistical Test Results for Parent Process Test vs. Base Test

The Mann-Whitney U test results are U=1957 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the

differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Parent Process data set median.

Lastly, the Levene test results are W=0.56 and p=0.45. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and Parent Process Test data sets have different variances.

Overall, the inability to differentiate the Parent Process dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant difference between the Base Test and the Parent Process dataset, must limit our expectations for the artifact as the Levene test will likely introduce false positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

RDTSC Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	691386	1
Levene Test	70.7	7.84e-17

Table 44 – Statistical Test Results for RDTSC Test vs. Base Test

The Mann-Whitney U test results are U=691,386 and p=1.0 which is above the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we fail to reject the null hypothesis and cannot say that the Base Test data set median is different from the RDTSC data set median.

Second, the Levene test results are W=70.7 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this

test. As a result, we can say that the Base Test and RDTSC Test data sets have different variances.

Overall, the Levene test returned statistically significant results and can be differentiated from the Base Test data. The RDTSC data set is not measurably different from the Base Test using the Mann-Whitney U test based on the p-value being greater than 0.05. This means that there is low correlation between the treatment of the Anti-Debugging technique on the Base Test for this research regarding the median. However, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the RDTSC data set model when used with the artifact.

Self-Debugging Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	840	0
Levene Test	16.2	6.00e-5

Table 45 – Statistical Test Results for Self-Debugging Test vs. Base Test

The Mann-Whitney U test results are $U=840$ and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Self-Debugging data set median.

Additionally, the Levene test results are $W=16.2$ and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Self-Debugging Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Self-Debugging data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique

on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Self-Debugging data set model when used with the artifact.

Time Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	327774	9.93e-53
Levene Test	1.54	0.215

Table 46 – Statistical Test Results for Time Test vs. Base Test

The Mann-Whitney U test results are U=327,774 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Time data set median.

Second, the Levene test results are W=1.54 and p=0.22. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and Time Test data sets have different variances.

Overall, the inability to differentiate the Time dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant difference between the Base Test and the Time dataset, must limit our expectations for the artifact as the Levene test will likely introduce false positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

Trap Flag Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	1969	0.0
Levene Test	0.288	0.592

Table 47 – Statistical Test Results for Trap Flag Test vs. Base Test

The Mann-Whitney U test results are U=1969 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Trap Flag data set median.

Lastly, the Levene test results are W=0.28 and p=0.59. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and Trap Flag Test data sets have different variances.

Overall, the inability to differentiate the Trap Flag dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant difference between the Base Test and the Trap Flag dataset, must limit our expectations for the artifact as the Levene test will likely introduce false positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

ARM Linux Experiments

Experimental Set Up

The ARM Linux experiments were run on a virtual Ubuntu 22.04 LTS arm64 machine. The Linux kernel is version 6.1. The virtualization was hosted in the cloud by AWS

and was paid for by the author. The Ubuntu VM had 2 vCPUs and 4GB of memory due to the t4g.medium node, and 30GB of EBS General Purpose (SSD) drive space.

Each experiment for each of the Anti-Debugging techniques was run in serial to prevent any performance impacts between tests. Each test was run 1010 times which allowed for the first ten results to be thrown out while the data and instruction cache were populated with relevant information based on Paoloni. (Paoloni, 2010)

Each test executed the same code base code while measuring the performance impact of the anti-debugging technique in addition to the baseline. Only the base test experiment did not have any anti-debugging techniques to create the baseline. The base test was also subjected to 1010 test executions where the first ten results were thrown out as well.

Due to the limitation of being unable to acquire exclusive execution due to a lack of kernel-level implementation, the multiple vCPUs could cause changes in execution duration due to context switching, however this was minimized by exclusively executing the experiment on the VM with no other running user-mode programs.

Experiment Implementation

The implementations of each Anti-Debugging technique are in the Linux source code. The code is compiled with aarch64-linux-gnu-g++-11 version 11.4.0. Once the data has been collected, the logs are run through a python script that converts the data log to a Pandas DataFrame object (The pandas development team, 2020). This object is combined with data visualization customizations based on each test to generate the graphs of the distribution of data. In addition, the Base Test and Anti-Debugging technique data are passed into SciPy implementations of the Kruskal test, Mann-Whitney U test, and the Levene test (Levene, 1960; “Levene Test for Equality of Variances,” n.d.; Mann & Whitney, 1947; Virtanen et al., 2020). These values, in addition to the quartile values, populate the tables in the following section. The quartile values are measured at the 0, 25, 50, 75, and 100% percentile values which are equivalent to the minimum, first quartile, median, third quartile, and maximum values of the data distribution.

In addition, some differences are implemented for the specific nuances of aarch64 execution. The first instance is for the INT3 test. The equivalent to the INT3 instruction, BRK, causes execution to halt and not resume. This causes the Anti-Debugging technique to

be non-functional and unable to be measured. For this test, we simply raise the SIGTRAP signal, which is what happens on Intel architectures when the INT3 instruction is executed (*ARM Developer Documentation*, 2023; scottt, 2021). The second major difference is that the Trap Flag equivalent, the Single Step (SS) flag, is unable to be modified in user-mode. This removes the Trap Flag/Single Step Anti-Debugging technique from the possibility of measurement for this research (*ARM Developer Documentation*, 2023). Lastly, the assembly-level time measurement for both the time measurement and the RDTSC equivalent test requires a different implementation. In order to get the timing measurement, we serialize the execution using the ISB instruction followed by reading from the CNTVCT_EL0 register using the MRS instruction (*ARM Developer Documentation*, 2023; Cownie, 2021).

Experiment Results

Base Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	500000	0.5
Levene Test	0	1

Table 48 – Statistical Test Results for Base Test vs. Base Test

The base test is used as the control group whereas the other tests are run with Anti-Debugging Techniques acting as the applied treatment. The differences in the measurement of the performance data indicate the impact that the Anti-Debugging technique has on the program's performance. For the Base Test, no treatment is applied, and the results are equivalent. This is confirmed with the statistical results that have a p-value of 1.0 for Levene test. This makes sense as we fail to reject the null hypothesis in each of these cases as the data sets are the same and therefore the median and variance are the same.

ClockGetTime Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	101214	9.95e-264
Levene Test	676	1.53e-128

Table 49 – Statistical Test Results for ClockGetTime Test vs. Base Test

The Mann-Whitney U test results are U=131,213 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the ClockGetTime data set median.

Additionally, the Levene test results are W=676 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and ClockGetTime Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The ClockGetTime data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the ClockGetTime data set model when used with the artifact.

Clock Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	8841	0.0
Levene Test	664	1.05e-126

Table 50 – Statistical Test Results for Clock Test vs. Base Test

The Mann-Whitney U test results are U=8841 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Clock data set median.

Additionally, the Levene test results are W=664 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Clock Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Clock data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Clock data set model when used with the artifact.

Code Checksum Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	14397	0.0
Levene Test	0.15	0.694

Table 51 – Statistical Test Results for Code Checksum vs. Base Test

The Mann-Whitney U test results are U=14,397 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Code Checksum data set median.

Lastly, the Levene test results are W=0.15 and p=0.69. These results are not above the threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and Code Checksum Test data sets have different variances.

Overall, the inability to differentiate the Code Checksum dataset from the Base test data using the Levene test means that it is unlikely to be useful as part of the artifact. The

Mann-Whitney U test demonstrated a difference in medians of the datasets; however, this kind of difference can occur from longer running programs without Anti-Debugging techniques. In this research, we can identify a difference in data sets due to the Anti-Debugging technique but will be difficult to generalize. Since the Levene test did not have a statistically significant difference between the Base Test and the Code Checksum dataset, must limit our expectations for the artifact as the Levene test will likely introduce false positives into the results as the structural change of the data set, based on the distribution variance, is difficult to measure accurately.

Code Obfuscation Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	0.0	0.0
Levene Test	362	2.96e-74

Table 52 – Statistical Test Results for Code Obfuscation Test vs. Base Test

The Mann-Whitney U test results are $U=0.0$ and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Code Obfuscation data set median.

Additionally, the Levene test results are $W=362$ and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Code Obfuscation Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Code Obfuscation data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation

between the presence of an Anti-Debugging technique in the sample and the Code Obfuscation data set model when used with the artifact.

Process Enumeration Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	0.0	0.0
Levene Test	366	5.38e-75

Table 53 – Statistical Test Results for Process Enumeration Test vs Base Test

The Mann-Whitney U test results are $U=0.0$ and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Process Enumeration data set median.

Additionally, the Levene test results are $W=366$ and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Process Enumeration Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Process Enumeration data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Process Enumeration data set model when used with the artifact.

raise(SIGTRAP) (INT3) Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	10961	0.0

Levene Test	68.1	2.74e-16

Table 54 – Statistical Test Results for raise(SIGTRAP) (INT3) Test vs. Base Test

The Mann-Whitney U test results are U=10,961 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the raise(SIGTRAP) (INT3) data set median.

Additionally, the Levene test results are W=68.1 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and raise(SIGTRAP) (INT3) Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The raise(SIGTRAP) (INT3) data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the raise(SIGTRAP) (INT3) data set model when used with the artifact.

BRK (INT3) Scan Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	500496	0.579
Levene Test	0.323	0.570

Table 55 – Statistical Test Results for BRK (INT3) Scan Test vs. Base Test

The Mann-Whitney U test results are U=500496 and p=0.58 which is above the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we fail to reject the null

hypothesis and cannot say that the Base Test data set median is different from the BRK (INT3) Scanning data set median.

Additionally, the Levene test results are W=0.323 and p=0.57. These results are above the p-value threshold to reject the null hypothesis and therefore we fail to reject the null hypothesis in this test. As a result, we cannot say that the Base Test and Process Enumeration Test data sets have different variances.

Overall, the inability to differentiate the BRK (INT3) dataset from the Base Test data using the Mann-Whitney U test and the Levene test means that it is unlikely to successfully detect this Anti-Debugging technique using the artifact. The high p-value for the Mann-Whitney U test identifies a similarity in the median of the performance data distribution compared to the Base test. In addition, the high p-value for the Levene test means the difference in variances between the Base Test data and BRK (INT3) data is minimal. Therefore, the distributions of the Base Test and BRK (INT3) data set are similar in both median and variance and not only difficult to detect in this research but also for the artifact. Since the artifact uses the Levene test to measure variance, this technique, if used as part of the artifact, will have a high false positive rate as it's like the control group data.

IsTracerPidZero Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	0.0	0.0
Levene Test	68.8	2.00e-16

Table 56 – Statistical Test Results for IsTracerPidZero Test

The Mann-Whitney U test results are U=0.0 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the IsTracerPidZero data set median.

Additionally, the Levene test results are W=69 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in

this test. As a result, we can say that the Base Test and IsTracerPidZero Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The IsTracerPidZero data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the IsTracerPidZero data set model when used with the artifact.

Memory Encryption Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	0.0	0.0
Levene Test	864	4.08e-158

Table 57 – Statistical Test Results for Memory Encryption Test vs. Base Test

The Mann-Whitney U test results are $U=0.0$ and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Memory Encryption data set median.

Additionally, the Levene test results are $W=864$ and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Memory Encryption Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Memory Encryption data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging

technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Memory Encryption data set model when used with the artifact.

Parent Process Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	3147	0.0
Levene Test	36.3	2.05e-9

Table 58 – Statistical Test Results for Parent Process Test vs. Base Test

The Mann-Whitney U test results are $U=3147$ and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Parent Process data set median.

Additionally, the Levene test results are $W=36.3$ and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Parent Process Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Parent Process data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Parent Process data set model when used with the artifact.

CNTVCT_EL0 (RDTSC) Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	287197	1.73e-112
Levene Test	348	8.85e-72

Table 59 – Statistical Test Results for CNTVCT_EL0 (RDTSC) Test vs. Base Test

The Mann-Whitney U test results are U=287,197 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the CNTVCT_EL0 (RDTSC) data set median.

Additionally, the Levene test results are W=348 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and CNTVCT_EL0 (RDTSC) Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The CNTVCT_EL0 (RDTSC) data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the CNTVCT_EL0 (RDTSC) data set model when used with the artifact.

Self-Debugging Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	0.0	0.0
Levene Test	33.7	7.60e-9

Table 60 – Statistical Test Results for Self-Debugging Test vs. Base Test

The Mann-Whitney U test results are U=0.0 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set mean is different from the Self-Debugging data set median.

Additionally, the Levene test results are W=33.7 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in this test. As a result, we can say that the Base Test and Self-Debugging Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Self-Debugging data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Self-Debugging data set model when used with the artifact.

Time Test Results

Test Name	Statistic Value	p-value
Mann-Whitney U Test	246104	2.98e-141
Levene Test	869	5.97e-159

Table 61 – Statistical Test Results for Time Test vs. Base Test

The Mann-Whitney U test results are U=246,104 and $p < 0.05$ which is below the conventional threshold for statistical significance. The Mann-Whitney U test measures the differences between the medians of the data sets. For this test, we reject the null hypothesis and can say that the Base Test data set median is different from the Time data set median.

Additionally, the Levene test results are W=869 and $p < 0.05$. These results are above the threshold to reject the null hypothesis and therefore we can reject the null hypothesis in

this test. As a result, we can say that the Base Test and Time Test data sets have different variances.

Overall, both statistical tests returned statistically significant results and can be differentiated from the Base Test data. The Time data set is measurably different from the Base Test using the Mann-Whitney U test based on the p-value being less than 0.05. This means that there is a correlation between the treatment of the Anti-Debugging technique on the Base Test for this research. In addition, the Levene test returned a similarly statistically significant result with a p-value less than 0.05. Due to this result, it is likely, based on the p-value, that the variance can be measured in other datasets and suggest a strong correlation between the presence of an Anti-Debugging technique in the sample and the Time data set model when used with the artifact.

Artifact Experiment

Experimental Set Up

The setup is based on the results from the first three experiments and Anti-Debugging techniques that are likely to be detectable using the Levene test to measure the variance between the data models produced and the sample data. The artifact experiment is run using Python 3.10.7 x64 on a Windows 11 operating system. The sample data was collected using the same implementation of the Anti-Debugging techniques as the previous experiments but with three different base tests: slow_sample, medium_sample, and fast_sample. These samples have different durations of execution time to demonstrate the artifact's ability and limitations when classifying the presence of an Anti-Debugging technique.

Experiment Implementation

The artifact is designed to read in the data models produced during the first three experiments in addition to the sample data and process them both. During the processing, the data is converted to Pandas DataFrame objects for ease of computation and data visualization (The pandas development team, 2020). The artifact then computes the Levene test on the sample data with each of the valid data models for the appropriate architecture and operating system. The Levene test will measure the similarity of variances for the data distributions and

if we fail to reject the null hypothesis that the variances are different, we can say that the variances are similar. This binary determination of the similarity between variances acts as a classifier for the presence of the Anti-Debugging technique using the performance data from the sample and the data models. The sample data was collected using the same implementation of the Anti-Debugging techniques as the previous experiments but with three different base tests: slow_sample, medium_sample, and fast_sample. These samples have different durations of execution time to demonstrate the artifact's ability and limitations when classifying the presence of an Anti-Debugging technique. The fast_sample binary executed a simple loop 1000 times for execution duration. The medium_sample binary executed ten times as long with a loop of 10,000 iterations. The slow_sample executed ten times as long as the medium sample with 100,000 iterations. The success or failure of the artifact will be determined by the Anti-Debugging technique's presence in the output of the artifact as having been successfully identified as the correct Anti-Debugging technique. The result remains true even if other Anti-Debugging techniques were also determined to be similar in variance. If desired, the entire source code implementation for the research, data included, is hosted at the link provided in Appendix A – Source Code.

Experiment Results - Windows

CloseHandle Artifact Results

Sample	Success
fast_sample	True
medium_sample	True
slow_sample	False

Table 62 – CloseHandle Artifact Results (Windows)

The artifact was able to detect the CloseHandle Anti-Debugging technique in the fast and medium execution duration samples. The artifact was unable to detect the CloseHandle technique during the slow sample. This exception-based technique was mildly sensitive to the change in variance across the new samples based on the fact it could detect the technique in the fast sample and the medium sample, which executed ten times as many calculations. This

technique is only more sensitive than the Guard Page technique and less sensitive than the INT 0x2D and UnhandledExceptionFilter tests when it comes to Exception-based techniques.

Code Checksum Artifact Results

Sample	Success
fast_sample	True
medium_sample	True
slow_sample	True

Table 63– Code Checksum Artifact Results (Windows)

The artifact was able to detect the Code Checksum Anti-Debugging technique in all three execution duration samples. This technique is insensitive to changes in execution time based on the ability of the artifact to detect the technique within the fast sample and both the medium and slow samples which had ten and one hundred times more computations than the fast sample. Given that the code checksum was one of the longer running techniques as part of the Scanning-based Anti-Debugging technique group, it is likely that the increased duration of the technique allowed it to be identified over a larger variance of data distributions compared to more sensitive techniques. This result is unique compared to the other Scanning-based Anti-Debugging techniques that were sufficient to be measured by the artifact and even more so compared to the scanning-based techniques that weren't distinguishable from the Base Test.

Debug Registers Artifact Results

Sample	Success
fast_sample	True
medium_sample	False
slow_sample	False

Table 64 – Debug Registers Artifact Results (Windows)

The artifact was only able to detect the Debug Registers Anti-Debugging technique in the fast sample. This technique is very sensitive to the execution duration based on the

experimental results. Despite the ability to detect a difference against the original Base Test, the variance is not significant enough to expand its detectability to longer durations of execution. This technique is the most sensitive API-based technique for Windows within the artifact. Both the NtQueryInformationProcess and NtQueryObject techniques had more successful detections in this experiment.

Process Enumeration Artifact Results

Sample	Success
fast_sample	False
medium_sample	False
slow_sample	True

Table 65– Process Enumeration Artifact Results (Windows)

The artifact was only able to detect the Process Enumeration Anti-Debugging technique in the slow sample. While this is a unique result compared to other tests within the experiment, it demonstrates the very sensitive nature of this technique to being detected with the artifact. The variance was not significant enough to be detected during faster running samples which limits its ability to be used for broader detection using the artifact despite its ability to be identified against the initial Base Test data set. This result is more closely in line with the other Scanning-based Anti-Debugging techniques as extremely, or mildly, sensitive to execution duration.

GetSystemTime Artifact Results

Sample	Success
fast_sample	True
medium_sample	False
slow_sample	False

Table 66 – GetSystemTime Artifact Results (Windows)

The artifact was only able to detect the GetSystemTime Anti-Debugging technique in the fast sample. This technique is very sensitive to the execution duration based on the

experimental results. Despite the ability to detect a difference against the original Base Test, the variance is not significant enough to expand its detectability to longer durations of execution. This result is not entirely surprising based on the NtQueryPerformanceCounter results and the few Timing-based Anti-Debugging techniques that were detectable even from the Base Test. The RDTSC and GetLocalTime tests were not distinguishable from the Base Test and this test's sensitivity demonstrates how quickly and relatively uniformly the Anti-Debugging technique executes.

GetTickCount Artifact Results

Sample	Success
fast_sample	True
medium_sample	True
slow_sample	True

Table 67 – GetTickCount Artifact Results (Windows)

The artifact was able to detect the GetTickCount Anti-Debugging technique in all three execution duration samples. This technique is insensitive to changes in execution time based on the ability of the artifact to detect the technique within the fast sample and both the medium and slow samples which had ten and one hundred times more computations than the fast sample. This result is unique compared to the other Timing-based Anti-Debugging techniques that were indistinguishable from the Base Test and even the Timing-based techniques that were significant enough for the artifact.

Guard Page Artifact Results

Sample	Success
fast_sample	True
medium_sample	True*
slow_sample	True

Table 68 – Guard Page Artifact Results (Windows)

* Unique and accurate detection of the Anti-Debugging Technique

The artifact was able to detect the Guard Page Anti-Debugging technique in all three execution duration samples. This technique is insensitive to changes in execution time based on the ability of the artifact to detect the technique within the fast sample and both the medium and slow samples which had ten and one hundred times more computations than the fast sample. This result is extremely unique compared to the other Exception-based Anti-Debugging techniques and from all other techniques in this research. The Guard Page technique was the only technique that was able to be solely detected as the only, and correct, technique by the artifact during the medium sample test. In addition, this technique is less sensitive than other Exception-based techniques that were significant enough for the artifact experiment.

INT 0x2D Artifact Results

Sample	Success
fast_sample	False
medium_sample	False
slow_sample	False

Table 69 – INT 0x2D Artifact Results (Windows)

The artifact was unable to detect this Exception-based Anti-Debugging technique during any of the samples. Based on the results from the artifact, this technique is unviable for detection by the artifact and is unable to be expanded to detection within other samples. While it was significant enough to be detected compared to the original Base Test, the applicability is limited and is one of the least successful tests from the artifact, not only for the Exception-based techniques, but for all the techniques tested using the artifact.

INT3 Scan Artifact Results

Sample	Success
fast_sample	True
medium_sample	False
slow_sample	True

Table 70 – INT3 Scan Artifact Results (Windows)

The artifact was able to detect this Scanning-based Anti-Debugging technique during two of the three samples. Different from any of the other tests, the Int3 Scan technique was the only technique to be identified during the fast and slow samples but not identified during the medium sample. It is possible this is a false negative or the data was skewed due to the operating system interrupting execution while the test was executing. This test was mildly sensitive to the execution duration and more in line with the Process Enumeration results than the Code Checksum results for Scanning-based techniques.

Memory Encryption Artifact Results

Sample	Success
fast_sample	True
medium_sample	True
slow_sample	True

Table 71 – Memory Encryption Artifact Results (Windows)

The artifact was able to detect the Memory Encryption Anti-Debugging technique in all three execution duration samples. This technique is insensitive to changes in execution time based on the ability of the artifact to detect the technique within the fast sample and both the medium and slow samples which had ten and one hundred times more computations than the fast sample. This result is in line with the other Miscellaneous Anti-Debugging technique, Self-Debugging, which also was able to be detected in all three tests. The Memory Encryption technique was one of the slower techniques across the corpus of techniques in this research and the increase in execution of the technique itself likely provided a more significant variance for the artifact to identify across the samples.

NtQueryInformationProcess Artifact Results

Sample	Success
fast_sample	False
medium_sample	True
slow_sample	True

Table 72 – NtQueryInformationProcess Artifact Results (Windows)

The artifact was able to detect the NtQueryInformationProcess Anti-Debugging technique in two of the three execution duration samples. It was able to detect the technique in the medium and slow samples, but not the fast one. These results demonstrate the mild sensitivity to the execution duration for this sample. This technique is more sensitive than the NtQueryObject and Debug Registers techniques making it the second most sensitive API-based technique used with the artifact for this experiment.

NtQueryObject Artifact Results

Sample	Success
fast_sample	True
medium_sample	True
slow_sample	True

Table 73 – NtQueryObject Artifact Results (Windows)

The artifact was able to detect the NtQueryObject Anti-Debugging technique in all three execution duration samples. This technique is insensitive to changes in execution time based on the ability of the artifact to detect the technique within the fast sample and both the medium and slow samples which had ten and one hundred times more computations than the fast sample. This result is less sensitive than the other API-based techniques, NtQueryInformationProcess and Debug Registers, which was only detected in two of the samples, and one of the samples respectively. The implementation of the NtQueryObject technique required more code to execute which likely leads to a less sensitive technique due to the increase in variance due to the execution duration of the technique itself.

NtQueryPerformanceCounter Artifact Results

Sample	Success
fast_sample	False
medium_sample	True
slow_sample	False

Table 74 – NtQueryPerformanceCounter Artifact Results (Windows)

The artifact was only able to detect the NtQueryPerformanceCounter Anti-Debugging technique in the medium sample. This technique is very sensitive to the execution duration based on the experimental results. Despite the ability to detect a difference against the original Base Test, the variance is not significant enough to expand its detectability to longer durations of execution. This result is not entirely surprising based on the GetSystemTime results from the artifact and the few Timing-based Anti-Debugging techniques that were detectable even from the Base Test. The RDTSC and GetLocalTime tests were not distinguishable from the Base Test and this test's sensitivity demonstrates how quickly and relatively uniformly the Anti-Debugging technique executes.

Parent Process Artifact Results

Sample	Success
fast_sample	False
medium_sample	False
slow_sample	False

Table 75 – Parent Process Artifact Results (Windows)

The artifact was unable to detect this Scanning-based Anti-Debugging technique during any of the samples. Based on the results from the artifact, this technique is unviable for detection by the artifact and is unable to be expanded to detection within other samples. While it was significant enough to be detected compared to the original Base Test, the applicability is limited and is one of the least successful tests from the artifact, not only for the Scanning-based techniques, but for all the techniques tested using the artifact.

Self-Debugging Artifact Results

Sample	Success
fast_sample	True
medium_sample	True
slow_sample	True

Table 76 – Self-Debugging Artifact Results (Windows)

The artifact was able to detect the Self-Debugging Anti-Debugging technique in all three execution duration samples. This technique is insensitive to changes in execution time based on the ability of the artifact to detect the technique within the fast sample and both the medium and slow samples which had ten and one hundred times more computations than the fast sample. This result is in line with the other Miscellaneous Anti-Debugging technique, Memory Encryption, which also was able to be detected in all three tests. The Self-Debugging technique was also one of the slower techniques across the corpus of techniques in this research and the increase in execution of the technique itself likely provided a more significant variance for the artifact to identify across the samples.

UnhandledExceptionFilter Artifact Results

Sample	Success
fast_sample	True
medium_sample	False
slow_sample	False

Table 77 – UnhandledExceptionFilter Artifact Results (Windows)

The artifact was only able to detect the UnhandledExceptionFilter Anti-Debugging technique in one of the three execution duration samples. It was able to identify the technique in the fast sample only; it was not detected in the medium or slow samples. This technique is very sensitive to the execution duration. This Exception-based technique was slightly less sensitive than the INT 0x2D technique but more sensitive than the Guard Page and CloseHandle techniques.

Experiment Results – Linux (Intel)

ClockGetTime Artifact Results

Sample	Success
fast_sample	True
medium_sample	False

slow_sample	False

Table 78 - ClockGetTime Artifact Results (Linux-Intel)

The artifact was only able to detect the ClockGetTime Anti-Debugging technique in one of the three execution duration samples. It was able to identify the technique in the fast sample only; it was not detected in the medium or slow samples. This technique is very sensitive to the execution duration. These results are like the other timing-based Anti-Debugging techniques, Clock and RDTSC. All these tests were only successful for the fast sample and were not identified in the medium and slow samples.

Clock Artifact Results

Sample	Success
fast_sample	True
medium_sample	False
slow_sample	False

Table 79 - Clock Artifact Results (Linux-Intel)

The artifact was only able to detect the Clock Anti-Debugging technique in one of the three execution duration samples. It was able to identify the technique in the fast sample only; it was not detected in the medium or slow samples. This technique is very sensitive to the execution duration. These results are like the other timing-based Anti-Debugging techniques, ClockGetTime and RDTSC. All these tests were only successful for the fast sample and were not identified in the medium and slow samples.

Code Checksum Artifact Results

Sample	Success
fast_sample	False
medium_sample	False
slow_sample	True

Table 80 – Code Checksum Artifact Results (Linux-Intel)

The artifact was only able to detect the Code Checksum Anti-Debugging technique in one of the three execution duration samples. It was able to identify the technique in the slow sample only; it was not detected in the medium or fast samples. This technique is very sensitive to the execution duration. The detection of this technique is comparable to the detection of other scanning techniques on the Linux (Intel) operating system. Each scanning technique was very sensitive to changes in variance causing the artifact to only detect it within one sample.

Process Enumeration Artifact Results

Sample	Success
fast_sample	False
medium_sample	True*
slow_sample	False

Table 81 – Process Enumeration Artifact Results (Linux-Intel)

* Unique and accurate detection of the Anti-Debugging Technique

The artifact was only able to detect the Process Enumeration Anti-Debugging technique in one of the three execution duration samples. It was able to identify the technique in the medium sample only; it was not detected in the slow or fast samples. This technique is very sensitive to the execution duration. Somewhat different from other tests though, but the artifact was able to uniquely identify the technique as part of the medium sample while it was unable to detect it in the fast or slow sample. The strong detection on the medium sample while failing to identify it elsewhere suggests even more strongly how sensitive to changes in variance this technique is. The detection of this technique is comparable to the detection of other scanning techniques on the Linux (Intel) operating system. Each scanning technique was very sensitive to changes in variance causing the artifact to only detect it within one sample.

INT3 Artifact Results

Sample	Success
fast_sample	False
medium_sample	False

slow_sample	True*

Table 82 – INT3 Artifact Results (Linux-Intel)

* Unique and accurate detection of the Anti-Debugging Technique

The artifact was only able to detect the INT3 Anti-Debugging technique in one of the three execution duration samples. It was able to identify the technique in the slow sample only; it was not detected in the medium or fast samples. This technique is very sensitive to the execution duration. Somewhat different from other tests, the artifact was able to uniquely identify the technique as part of the slow sample while it was unable to detect it in the fast or medium sample. The strong detection on the slow sample while failing to identify it elsewhere suggests even more strongly how sensitive to changes in variance this technique is.

INT3 Scan Artifact Results

Sample	Success
fast_sample	False
medium_sample	True
slow_sample	False

Table 83 – INT3 Scan Artifact Results (Linux-Intel)

The artifact was only able to detect the INT3 Scanning Anti-Debugging technique in one of the three execution duration samples. It was able to identify the technique in the medium sample only; it was not detected in the slow or fast samples. This technique is very sensitive to the execution duration. The detection of this technique is comparable to the detection of other scanning techniques on the Linux (Intel) operating system. Each scanning technique was very sensitive to changes in variance causing the artifact to only detect it within one sample.

IsTracerPidZero Artifact Results

Sample	Success
fast_sample	False
medium_sample	False

slow_sample	False

Table 84 - IsTracerPidZero Artifact Results (Linux-Intel)

The artifact was unable to detect the IsTracerPidZero Anti-Debugging technique in any of the samples. This renders the technique unviable from the perspective of the artifact as the data is likely too sensitive to changes in variance and the samples did not represent any similar execution environments like the original execution environment when data was collected. This stands in contrast to the other scanning techniques where the artifact was able to detect the technique in at least one of the samples.

Memory Encryption Artifact Results

Sample	Success
fast_sample	True
medium_sample	True
slow_sample	True

Table 85 – Memory Encryption Artifact Results (Linux-Intel)

Unlike every other Linux (Intel) test using the artifact, the Memory Encryption technique was detected by the artifact for each sample. Since the technique was detected in each sample by the artifact, we can infer this technique is much less sensitive to changes in variance in execution duration. This stands quite opposite from the other miscellaneous technique, Self-Debugging, that was very sensitive to the changes in variance.

RDTSC Artifact Results

Sample	Success
fast_sample	True
medium_sample	False
slow_sample	False

Table 86 - RDTSC Artifact Results (Linux-Intel)

The artifact was only able to detect the RDTSC Anti-Debugging technique in one of the three execution duration samples. It was able to identify the technique in the fast sample

only; it was not detected in the medium or slow samples. This technique is very sensitive to the execution duration. These results are like the other timing-based Anti-Debugging techniques, ClockGetTime and Clock. All these tests were only successful for the fast sample and were not identified in the medium and slow samples.

Self-Debugging Artifact Results

Sample	Success
fast_sample	True*
medium_sample	False
slow_sample	False

Table 87 – Self-Debugging Artifact Results (Linux-Intel)

* Unique and accurate detection of the Anti-Debugging Technique

The artifact was only able to detect the Self-Debugging Anti-Debugging technique in one of the three execution duration samples. It was able to identify the technique in the fast sample only; it was not detected in the medium or slow samples. This technique is very sensitive to the execution duration. Like only a few other instances, the artifact was able to uniquely identify the self-debugging technique within the fast sample while being unable to detect it in the others. This further implies the sensitivity to changes in variance for this technique. This stands quite opposite from the other miscellaneous technique, Memory Encryption, that was insensitive to the changes in variance.

Experiment Results – Linux (ARM)

ClockGetTime Artifact Results

Sample	Success
fast_sample	False
medium_sample	True
slow_sample	True

Table 88 - ClockGetTime Artifact Results (Linux-ARM)

The artifact was able to detect the ClockGetTime Anti-Debugging technique in two samples. The artifact was able to detect the technique in the medium and slow samples; it was not able to detect the technique in the fast sample. This technique is mildly sensitive to changes in variance for execution duration based on the results of this experiment. The ClockGetTime technique was less sensitive than both other timing-based techniques, Clock and CNTVCT_EL0.

Clock Artifact Results

Sample	Success
fast_sample	False
medium_sample	True
slow_sample	False

Table 89 - Clock Artifact Results (Linux-ARM)

The artifact was able to detect the Clock Anti-Debugging technique in one sample. The artifact was able to detect the technique in the medium sample; it was not able to detect the technique in the fast or slow sample. This technique is very sensitive to changes in variance for execution duration based on the results of this experiment. The Clock technique was very sensitive to changes in execution duration like the CNTVCT_EL0 technique. It is also more sensitive than the ClockGetTime technique based on the results of this experiment.

Code Checksum Artifact Results

Sample	Success
fast_sample	False
medium_sample	True
slow_sample	True

Table 90 – Code Checksum Artifact Results (Linux-ARM)

The artifact was able to detect the Code Checksum Anti-Debugging technique in two samples. The artifact was able to detect the technique in the medium and slow samples; it was not able to detect the technique in the fast sample. This technique is mildly sensitive to

changes in variance for execution duration based on the results of this experiment. The artifact is able to detect this technique more than any other scanning-based technique including Process Enumeration, BKPT Scanning, and IsTracerPidZero.

Process Enumeration Artifact Results

Sample	Success
fast_sample	False
medium_sample	False
slow_sample	False

Table 91– Process Enumeration Artifact Results (Linux-ARM)

The artifact was not able to detect the Process Enumeration Anti-Debugging technique in any of the samples. The inability to detect the technique in any of the samples renders the technique for this operating system and architecture unviable for detection by the artifact. For the Linux (ARM) tests, this result was much more common than other operating system and architecture pairs. The artifact had similar detection results for this technique as other scanning-based techniques except for the Code Checksum Anti-Debugging technique.

raise(SIGTRAP) Artifact Results

Sample	Success
fast_sample	False
medium_sample	False
slow_sample	False

Table 92 – INT3 Artifact Results (Linux-ARM)

The artifact was not able to detect the raise(SIGTRAP) Anti-Debugging technique in any of the samples. The inability to detect the technique in any of the samples renders the technique for this operating system and architecture unviable for detection by the artifact. For the Linux (ARM) tests, this result was much more common than other operating system and architecture pairs.

BRK Scan Artifact Results

Sample	Success
fast_sample	False
medium_sample	False
slow_sample	False

Table 93 – BRK Scan Artifact Results (Linux-ARM)

The artifact was not able to detect the BRK Scan Anti-Debugging technique in any of the samples. The inability to detect the technique in any of the samples renders the technique for this operating system and architecture unviable for detection by the artifact. For the Linux (ARM) tests, this result was much more common than other operating system and architecture pairs. The artifact had similar detection results for this technique as other scanning-based techniques except for the Code Checksum Anti-Debugging technique.

IsTracerPidZero Artifact Results

Sample	Success
fast_sample	False
medium_sample	False
slow_sample	False

Table 94 - IsTracerPidZero Artifact Results (Linux-ARM)

The artifact was not able to detect the IsTracerPidZero Anti-Debugging technique in any of the samples. The inability to detect the technique in any of the samples renders the technique for this operating system and architecture unviable for detection by the artifact. For the Linux (ARM) tests, this result was much more common than other operating system and architecture pairs. The artifact had similar detection results for this technique as other scanning-based techniques except for the Code Checksum Anti-Debugging technique.

Memory Encryption Artifact Results

Sample	Success
fast_sample	False

medium_sample	False
slow_sample	False

Table 95 – Memory Encryption Artifact Results (Linux-ARM)

The artifact was not able to detect the Memory Encryption Anti-Debugging technique in any of the samples. The inability to detect the technique in any of the samples renders the technique for this operating system and architecture unviable for detection by the artifact. This result is like the other miscellaneous Anti-Debugging technique, Self-Debugging, which also was unable to be detected by the artifact.

CNTVCT_EL0 (RDTSC) Artifact Results

Sample	Success
fast_sample	False
medium_sample	False
slow_sample	True

Table 96 - RDTSC Artifact Results (Linux-ARM)

The artifact was able to detect the Clock Anti-Debugging technique in one sample. The artifact was able to detect the technique in the medium sample; it was not able to detect the technique in the fast or slow sample. This technique is very sensitive to changes in variance for execution duration based on the results of this experiment. The CNTVCT_EL0 technique was very sensitive to changes in execution duration like the Clock technique. It is also more sensitive than the ClockGetTime technique based on the results of this experiment.

Self-Debugging Artifact Results

Sample	Success
fast_sample	False
medium_sample	False
slow_sample	False

Table 97 – Self-Debugging Artifact Results (Linux-ARM)

The artifact was not able to detect the Self-Debugging Anti-Debugging technique in any of the samples. The inability to detect the technique in any of the samples renders the technique for this operating system and architecture unviable for detection by the artifact. This result is like the other miscellaneous Anti-Debugging technique, Memory Encryption, which also was unable to be detected by the artifact.

CHAPTER 6

CONCLUSIONS

In conclusion, this research was able to achieve its objective of implementing 58 Anti-Debugging techniques across operating systems and architectures to generate a new dataset. In addition, this dataset was used to provide statistics on the performance of each of these techniques. What's more, the performance data was able to be used to generate data models for the Anti-Debugging techniques for their respective operating system and architecture which was combined into an artifact. This artifact was able to successfully identify 27 out of the 58 total techniques with any amount of sensitivity to changes in execution duration and seven of those 27 techniques were insensitive to the variance in execution duration for this experiment. Below we'll give specific details of the comparisons between techniques between operating systems and architectures and the details of the techniques that were successfully detected within the artifact. Overall, many of the Windows operating system's techniques were the least sensitive to variance in execution duration while many were also unviable. The Linux techniques for both architectures tended to fall in the middle being very, or mildly, sensitive to the variance of execution duration.

Comparison of Operating Systems and Architectures

Windows vs. Linux Anti-Debugging Techniques (Intel)

When comparing the results of the different Anti-Debugging techniques across these operating systems, one must keep in mind that the number of techniques were different and many of the techniques did not map one-to-one across operating systems. With that in mind, seven techniques did.

The first technique we'll compare is the Code Checksum. This technique implemented the same code across both operating systems computing the Fletcher-16 checksum on an important function in the program implementation; as a malware author might do. While the

Windows operating system certainly had many very quick runs for 50% of the runs, the upper 50% of the data skewed quite high in execution time. Compared to the Linux (Intel) implementation, the results were highly consistent across the dataset with no more than one microsecond of difference between the minimum value and the 75th percentile. As expected, the maximum was much higher but still orders of magnitude faster than the Windows maximum value.

The comparison between the artifact's results for the Code Checksum were even more striking. The Windows operating system technique was able to be identified in all three samples using the artifact while the Linux (Intel) technique was only identified in one of the three samples. This naturally follows from the differences in variance between the techniques with the Windows technique having significantly more variance in performance data than the Linux (Intel) test that could be measured by the artifact.

The second technique that was available across both operating systems is the process enumeration technique. This technique was implemented using operating system specific API functions which likely caused some differences in execution time but the overall algorithm of listing all the processes and iterating over a list of known bad process names was the same. Like the Code Checksum data, the Windows operating system data had smaller data values for 50% of the data compared to the Linux (Intel) implementation but also quickly rose orders of magnitude in execution time towards the max value. Similarly, the Linux (Intel) implementation had more consistent values through the 75th percentile and the maximum value was not orders of magnitude higher.

The artifact results for this technique were almost identical. The artifact could only identify the Process Enumeration technique in the medium duration sample while the artifact was able to identify the same technique only in the slow sample for Linux (Intel). Given the differences in variance, like the Code Checksum data, the fewer correct identifications of the technique suggest there may be more information that is affecting the variance than was captured in this research. Since we do not determine if the variance was more or less than the others with the artifact, we can only say for certain that the variances were affected differently by the samples than during the model generation.

The third technique is the INT3 technique. Since the technique implements an architecture-based feature, the technique is well-designed to compare differences across

operating systems. The Windows operating system once again showed a large spread in the data compared to the Linux (Intel) implementation. In fact, the entire spread of the Linux (Intel) performance data is below the 75th percentile of the Windows data.

In stark contrast to the previous two techniques, only the Linux (Intel) INT3 technique was able to make it into the artifact for detection. The artifact was able to detect the INT3 technique in the slow sample for the Linux (Intel) operating system. The Windows operating system INT3 technique was not statistically significant enough to be detected using the artifact and was not included in the artifact results. While this is not unexpected given the simplicity and speed of using a single assembly instruction as an Anti-Debugging technique, it is interesting that on the Linux (Intel) side it was detectable while on Windows it could not be statistically discerned from the Base Test at an alpha < 0.05 level.

The fourth technique that can be compared across operating systems is the scanning-based technique, INT3 Scanning. The INT3 Scanning on the Windows operating system followed its normal pattern of very small durations for approximately 50% of the values and quickly grew by orders of magnitude for the longest durations. The Linux (Intel) implementation kept very consistent executions times with the minimum and 75th percentile having equal values. The maximum execution time for the Linux (Intel) technique was only one order of magnitude larger than the others.

The artifact was slightly more effective in detecting this technique than the previous INT3 technique. Once again, the artifact was only able to detect this technique in one of the Linux (Intel) samples, the medium duration sample. However, the artifact was able to detect two samples for the Windows operating system, namely the fast and slow samples. The detection on the Windows operating system was unique in that it detected the technique in the fast and slow samples but not the medium sample in the middle. It's possible this is a false negative result based strictly on this specific research conducted as opposed to a generalized capability of the artifact.

The fifth technique, Memory Encryption, was the most successful technique to be detected in this research. The performance data was in line with operating system norms. The Windows technique had higher performance values for this technique based on the implementation required to do encryption. However, it did maintain the general data structure of most data being consistent with the maximum growing by orders of magnitude. The Linux

(Intel) implementation maintained very consistent results across all the data with the maximum less than three times the minimum but with few very low values as compared to the Windows implementation.

The artifact was able to detect the Memory Encryption technique on both operating systems for all three samples. This technique was the only technique that the artifact was successful in completely identifying the technique using the model data and applying it to new performance data. Based on the natural timing of this technique, it is likely that the increased duration of this technique, with its own natural variance, was more easily detected using the artifact than other samples.

The sixth, and penultimate, technique was a timing-based technique using the RDTSC instruction. This technique was one of the fastest techniques implemented and had some of the lowest values for performance data in this research. The Windows implementation usually operated at a sub-microsecond resolution resulting in many zeros appearing in the data due to the measurement being conducted in microsecond resolution. As expected for the Windows operating system, the maximum value was significantly larger than most of the data. The Linux (Intel) implementation also had very low, consistent values, with the maximum value only being one order of magnitude larger than the minimum value.

The artifact was only able to use the Linux (Intel) data for classification as the Windows operating system data was not statistically significant from the Base Test to be included in the artifact. While the Linux (Intel) technique is included in the artifact, the artifact was still only able to detect the technique in the fast sample, and not the medium or slow samples. These results strongly point to the speed of execution of this technique minimizing the detectability of this technique. Anything that executed too much longer caused the natural variance to overwhelm the technique and result in false negative outcomes.

Lastly, the Self-Debugging technique was used across operating systems but due to the implementation of debuggers on each system, the results varied wildly compared to other techniques that spanned operating systems. The Windows implementation of the Self-Debugging Anti-Debugging technique required creating a new process that had minimal debugging features to debug the parent process. This required operating system specific calls and significantly increased the duration of the technique. The Self-Debugging technique was the longest running technique for the Windows operating system. Although the overall

performance data values were high, the results were much more consistent than other techniques. The Linux (Intel) technique was implemented using ptrace() which is much more light weight than the Windows alternative resulting in significantly smaller values for the performance data. Deviating slightly from the standard data distribution for Linux (Intel) data, the lower 75% were consistent with the maximum ending up higher than other techniques, relative to the technique.

The artifact varied in being able to detect these techniques across operating systems. The artifact was able to detect the Windows technique in all three samples while only being able to detect the Linux (Intel) technique in the fast sample. Once again, the natural variance of the technique likely made detection more reliable for the Windows implementation compared to the Linux (Intel) variation resulting in the observed detections.

Linux (Intel) vs. Linux (ARM) Anti-Debugging Techniques

For this cross-architectural experiment looking at Anti-Debugging techniques and comparing them between Intel and ARM architectures, ten of the Anti-Debugging techniques for Linux were added to the artifact for detection. The artifact varied in its ability to detect each of these techniques but overall, the artifact was more successful in detecting the Intel versions of the techniques than the ARM versions.

The first Linux technique to compare is the ClockGetTime Anti-Debugging technique. The Intel version of this technique had smaller execution duration values up to the 75^{th} percentile compared to the ARM version of the technique. However, the range was also larger for the Intel version than the ARM version. Despite these small differences in execution durations for the models, both techniques were statistically significant enough for the artifact. The artifact was better able to detect the ARM technique than the Intel technique in this instance. The artifact detected the Intel technique in the fast sample only while it was able to detect the ARM technique in the medium and slow samples.

The second technique for comparison across architectures was the Clock Anti-Debugging technique. Like ClockGetTime, the Intel version of the technique had much lower execution durations up to the 75^{th} percentile with a larger maximum value. The ARM execution durations were larger in general but maintained a smaller range than the Intel technique. The artifact was slightly less successful in the detection of these techniques as it

was only able to detect the technique in one sample for each technique. The artifact was able to detect the Clock Anti-Debugging technique in the fast sample for Intel and the medium sample for ARM.

The third technique is the Code Checksum Anti-Debugging technique. Matching the previous techniques performance data, the Intel version had very low execution duration values through the 75th percentile while significantly increasing to the maximum value. The ARM version kept a much smaller range as seen before with the maximum being less than three times the minimum value. Both techniques were incorporated into the artifact but were detected different amounts. The Intel version of the technique was only detected during within the slow sample while the ARM version was detected in the medium and slow sample.

The fourth technique is the scanning-based Process Enumeration technique. This technique's results diverge from the previous results in significant ways in both the performance data and the artifact detectability. The Intel version had higher values almost in its entirety, with the minimum value only approximately 300 microseconds smaller than the maximum from the ARM version of this technique. Otherwise, the Intel performance data maintained consistent values up through the 75th percentile with the maximum being an order of magnitude larger than the rest of the data. The ARM version once again maintained minimal range within the dataset and almost universally lower values than the Intel performance data for this technique. The artifact was less able to detect this technique than others previously including being unable to detect the ARM version of the technique in any samples. The inability to detect the Process Enumeration technique within the ARM sample demonstrates an increased sensitivity on that platform compared to the Intel version. Comparatively, the artifact was only able to detect the Process Enumeration technique in the medium sample, however it detected it uniquely. The artifact classified the Process Enumeration technique as the single possibility for presence of an Anti-Debugging technique in the medium sample which is a rare and interesting result.

The fifth technique is the exception-based technique, INT3, or raise(SIGTRAP), for ARM. Both architectures have an instruction that causes a SIGTRAP signal to be raised to the operating system which is used for debugging. However, the architecture implementations of the INT3 equivalent on ARM does not continue execution which hinders its ability to be used as an Anti-Debugging technique. To account for this, the ARM version raises a SIGTRAP

signal itself rather than relying on the INT3 instruction to do it as seen in the Intel version. The performance data results return to the common data distributions for this technique. The Intel version has lower performance data values up to the 75th percentile with a moderate increase to the maximum value. Again, the ARM values maintain a smaller range and consistent values with the maximum being only three times the value of the minimum. The artifact was not able to detect the ARM technique, just like Process Enumeration, demonstrating the difference in architectures and sensitivities to variance for this technique within the artifact. Also like the Process Enumeration technique, the artifact was only able to detect the technique in one sample, the slow sample, but it was a unique detection. The artifact was able to identify the INT3 technique as the only candidate for the slow sample on for the Intel version of the INT3 Anti-Debugging technique.

The sixth technique is the INT3, or BRK, scanning technique. The BRK instruction is the corresponding instruction on the ARM architecture for the INT3 instruction on Intel. One major difference between these two architectures is the size of the instruction. The INT3 instruction is only a single byte while the BRK instruction is four bytes. While the bytes are different sizes, the performance data maintained a unique and interesting parity across architectures. For both architectures the performance data was entirely consistent up to the 75th percentile. While the specific values differ across architectures, the minimal variance for this technique is interesting. As for the maximum values, the ARM architecture had a smaller maximum value than the Intel version which is consistent with the rest of the technique's performance data. The artifact was less successful detecting the ARM version of the technique in this case too. The artifact was able to detect the Intel version of the technique in the medium sample only. While the artifact is even more sensitive to the ARM version of this technique, with only one detection on the Intel version as well it demonstrates a high sensitivity for this technique across architectures on the Linux operating system.

The seventh technique is another scanning-based technique, IsTracerPidZero. These performance data values are consistent with the other values when comparing techniques across architectures. The Intel version of the technique has lower values up to the 75th percentile with the maximum value being an order of magnitude higher. The ARM version of this technique has a much smaller range than the Intel version; the maximum value is only twice the minimum value. The artifact results for this technique were unexpected based on

both techniques being uniquely identified from the Base Test results to be included in the artifact at all. Despite their detectability against the Base Test, the artifact failed to identify this technique in any of the samples on either architecture. This demonstrates these techniques' extreme sensitivity to changes in variance that is independent of architecture.

The eighth technique is the miscellaneous Anti-Debugging technique, Memory Encryption. The ARM version of this technique had an extremely small range relative to the performance data values and other ARM technique measurements. The values were very consistent and at most varied by at most 10% between minimum and maximum implying a small variance. The Intel version of the technique also had a smaller range compared to other Intel versions of techniques but still a larger variance than the ARM version of this technique. As seen before, the performance data values were smaller up to the 75^{th} percentile and then grew beyond the ARM performance data values up to the maximum. The artifact results were drastically different between architectures. The artifact failed to identify any of the Memory Encryption results for the ARM version. Contrarily, the artifact successfully identified the Intel version of the Memory Encryption technique in all the samples. This complete difference in the ability to identify the technique for the Intel version of the technique compared to the ARM version of the technique displays significant differences between the architectures for this technique.

The ninth, and penultimate, technique is the RDTSC, or CNTVCT_EL0, timing-based Anti-Debugging technique. While acquisition of the timing values between architectures can still be implemented in one instruction on both architectures, the Intel RDTSC instruction operates differently than the CNTVCT_EL0 register. The CNTVCT_EL0 register, which contains the cycle counter, must be read using the MSR instruction while the RDTSC instruction returns the cycle counter abstracting away any of the hardware specificities. The performance data stays consistent with other tests comparing the results between Intel and ARM versions of the technique. The ARM version has more consistent results with a smaller range and therefore smaller variance. On the other hand, the Intel version has smaller values through the 75^{th} percentile and then grows an order of magnitude to the maximum value. The artifact was able to detect both techniques in exactly one of the samples. The artifact was able to detect the Intel version of the technique in the fast sample while it was able to detect the ARM version in the slow sample. Since the artifact was only able to detect each technique

once, we can be infer the techniques themselves are very sensitive to changes in variance, but what's more, the techniques were identified on different samples further implying that the techniques are more sensitive and the model data sets were more similar to the fast and slow models for Intel and ARM, respectively, in order for these techniques to be detected in this manner.

The last technique is the Self-Debugging Anti-Debugging technique. The Intel version of this technique maintained its consistent structure of smaller values, compared to ARM performance data, until the 75th percentile with the maximum value increasing by an order of magnitude. What is different about the ARM performance data for this technique is that it mirrored the Intel structure of data. While the minimum value for the ARM performance data is still larger than the 75th percentile for Intel, the ARM data also was low until the 75th percentile and then increased by an order of magnitude for the maximum value. Despite the likely higher variance for this ARM performance data, the artifact failed to identify the technique in any of the samples. The artifact successfully identified Self-Debugging in the Intel version of this technique including a unique identification. The artifact identified the Self-Debugging technique in the fast sample only, but the unique identification adds to the interest and rarity of this result.

Unviable Anti-Debugging Techniques

Out of the 58 total techniques, there were 31 techniques that were considered unviable for the artifact. The techniques were considered unviable for one of two reasons, either the technique could not be distinguished from the Base Test, or the artifact failed to detect the technique in any of the three samples during the artifact experiment.

There are 22 techniques that were unable to be distinguished from their Base Tests which means that the variance of the execution duration of the performance data was too similar to the Base Test that we failed to reject the null hypothesis after applying the Levene test (Levene, 1960; "Levene Test for Equality of Variances," n.d.). Of the 22 techniques, 15 of the unviable techniques, of the first kind, were Windows techniques. In addition, four techniques were Linux (Intel) and three were Linux (ARM). Eight of the techniques were API-based techniques, six were scanning-based techniques, four of the techniques were timing-based, and four were exception-based techniques. These techniques are:

CheckRemoteDebuggerPresent (Windows), Code Obfuscation (Windows), CsrGetProcessId (Windows), Event Pair Handles (Windows), GetLocalTime (Windows), Heap Flag (Windows), INT3 (Windows), IsDebuggerPresent (Windows), NtGlobalFlag (Windows), NtSetDebugFilterState (Windows), NtSetInformationThread (Windows), OutputDebugString (Windows), RDTSC (Windows), TimeGetTime (Windows), Trap Flag (Windows), Parent Process (Intel), Time (Intel), Trap Flag (Intel), Code Checksum (ARM), BRK Scan (ARM), Code Obfuscation (Intel), and Code Obfuscation (ARM).

Out of the 31 techniques that were not viable for the artifact, there are nine techniques that are identifiable from the Base Test, according to the alpha < 0.05 threshold, but were not able to be detected by the artifact in any of the three samples. Most of these are Linux (ARM) techniques with one Linux (Intel) technique and two Windows techniques as well. Two of the techniques are miscellaneous, five of the techniques are scanning-based, and two are exception-based Anti-Debugging techniques. Given that five of these techniques are scanning-based techniques, and due to their longer execution duration and increased variance, it's a surprising result that so many are unviable for the artifact based on the artifact experiment. The list of techniques is: Self-debugging (ARM), Memory Encryption (ARM), IsTracerPidZero (ARM), BRK Scan (ARM), raise(SIGTRAP) (ARM), Process Enumeration (ARM), IsTracerPidZero (Intel), Parent Process (Windows), and INT 0x2D (Windows).

Very Sensitive Anti-Debugging Techniques

There are 15 techniques across operating systems and architectures that are considered very sensitive to execution duration and variance based on the artifact's ability to only detect the technique once out of the three samples. Looking at the composition of this set of techniques, five are on the Windows operating system, eight are on the Linux (Intel) operating system and architecture, and two are on the Linux (ARM) operating system and architecture. In addition, the groupings of this set of techniques are as follows, one API-based technique, four are scanning-based techniques, seven are timing-based techniques, two are exception-based techniques, and one is miscellaneous. The higher presence of timing-based techniques is not unexpected due to their shorter execution duration and less complex implementation. Many timing-based techniques rely on executing a single instruction or a single operating system API which reduces the execution duration and variance compared to scanning-based

samples that loop, read, and process information to make an Anti-Debugging decision. We can also see pairs of techniques in this sample, for instance the Linux (Intel) and Linux (ARM) single instruction timing-based Anti-Debugging technique (i.e. RDTSC or CNTVCT_EL0) are both in the very sensitive category while the Windows RDTSC technique was not even distinguishable from the base test. This technique is very sensitive to execution duration and variance based on the artifact results. The techniques are Debug Registers (Windows), Process Enumeration (Windows), GetSystemTime (Windows), NtQueryPerformanceCounter (Windows), UnhandledExceptionFilter (Windows), ClockGetTime (Intel), Clock (Intel), Code Checksum (Intel), Process Enumeration (Intel), INT3 (Intel), INT3 Scan (Intel), RDTSC (Intel), Self-Debugging (Intel), Clock (ARM), and CNTVCT_EL0 (ARM).

Mildly Sensitive Anti-Debugging Techniques

There are five techniques that were detected by the artifact in two of the three samples, which is classified for this research as Mildly Sensitive. Since these techniques were not detected in every sample, we know the execution duration and variance played a larger role in the execution of these techniques and therefore their detectability. Three of these techniques are from the Windows operating system and two are from Linux (ARM). There is one exception-based technique, CloseHandle, two scanning-based techniques, INT3 Scan and Code Checksum (ARM), one API-based technique, NtQueryInformationProcess, and one timing-based technique, ClockGetTime (ARM). Overall ARM techniques had less variance with a slight increase in execution duration as compared to their Intel counterparts. This decrease in variance increased the difficulty for detection by the artifact due to its reliance on detecting the technique due to the variance in the performance data.

Insensitive Anti-Debugging Techniques

As a result of this research, seven techniques were identified that demonstrated insensitivity to the execution duration of the samples in this research and were identifiable from the initial base test. Each of these techniques, associated with their operating system and architecture, were detected by the artifact in all three samples. Six of these seven Anti-

Debugging techniques are on the Windows operating system with one on the Linux (Intel) operating system. Overall, one scanning-based technique is included, Code Checksum, one exception-based technique is included, Guard Page, one timing-based technique, GetTickCount, one API-based technique, NtQueryObject, and three miscellaneous techniques including Self-Debugging on Windows and Memory Encryption on both Windows and Linux (Intel). There were no ARM techniques that have the same level of insensitivity for this research. Many of these techniques are longer running techniques that are likely less sensitive to execution duration and variance due to their inherent execution duration and variance during their own execution. However, the GetTickCount result is the most unexpected because of its faster execution duration and the artifact's ability to detect it in all three samples.

Future Work

Not everything can be covered in this research and along with the limitations discussed in the introduction, there are five instances of future work that were not covered here. The five future research topics are anti-debugging techniques for the Windows operating system on ARM, hardware-based optimizations for anti-debugging techniques, connecting the data and Levene test as a classifier to Machine Learning, modifying the detection to work at the function-level as opposed to the program level, and implementing kernel functionality for exclusive execution of measured code.

Windows on Arm has increasing support by Microsoft as the Windows operating system supports native ARM executables in addition to emulating Intel architecture binaries on ARM chips. As recent as 2022, Microsoft is supplying IDEs, .NET Frameworks, and Java virtual machines that will run on ARM. Based on Window's new support for ARM and the hardware requirements for this research, measurement of native ARM executables with anti-debugging techniques was not realistic for this research. However, future research can build off the results here and compare to Windows on Arm results in the future when the ARM development ecosystem is more mature and widely available (mattwojo et al., 2023).

The second future research topic is related to optimizing anti-debugging techniques based on hardware. There exist optimizations that can be implemented that take advantage of the underlying hardware to decrease access and execution times that are not measured in this

research. While this research does focus on the performance aspects of anti-debugging techniques, it does not focus on optimizations for the number of cores, the size of the instruction and/or data caches, nor loop unrolling (except for code obfuscation but, not optimization). The goal of this research is not to find the optimal speed for anti-debugging techniques but to measure the overhead as is. Optimized anti-debugging techniques may be the subject of future research based on the results observed in this research.

Machine learning is an important tool for cybersecurity research and machine learning combined with Anti-Debugging and performance data may prove inciteful. Dasgupta et al published a review of machine learning in cybersecurity in 2022 and it talks about the current state of the intersection of machine learning and cybersecurity research. Based on their over 250 cited works, machine learning algorithms such as the decision tree, k-nearest neighbors, linear regression, neural networks, and more are given enormous amounts of data related to cybersecurity and providing insights from that data faster than humans can process the same information. This data can come from network intrusions, logs, and other attack related information. In a similar way, performance data can be collected and used to profile, measure, and classify different components of cybersecurity using more powerful machine learning algorithms. (Dasgupta et al., 2022). Specifically, using performance data as a feature of a dataset for classification could improve detectability of Anti-Debugging techniques.

Fourth, based on the high sensitivity to changes in variance of some of the techniques in this research, limiting the execution time and allowing the anti-debugging performance metrics to be front and center would provide more accurate detection. This can be performed at the function level of a program which executes less code, minimizes variance that cannot be accounted for, and allows for more precise detection. An implementation may look like developing an artifact to hook functions and capture performance data while then providing the data to a classifier to alert at the possibility of Anti-Debugging techniques being present. This data would be able to capture performance data, but in addition it could identify functions that were likely Anti-Debugging implementations and the function’s frequency of use.

Lastly, since no kernel-mode code was implemented for this research, future research could benefit from the implementation of a kernel-mode driver that reserved exclusive execution on a single CPU for measured code. Many of the performance measurement

capabilities can be implemented in user-mode but this is not one of them. I predict the impact would be minimal as the research here was done using virtual machines with only the samples running, however the performance measurements would be even more accurate with exclusive use of a CPU for an entire measurement of the code under test.

Summary

To summarize, this research was able to achieve its objective of implementing over 50 Anti-Debugging techniques across operating systems and architectures to generate new datasets. These datasets demonstrate the viability of using performance measuring in addition to statistical tests to non-invasively assess the presence of anti-debugging techniques within a binary. Based on these results, the Windows operating system on the Intel architecture had more techniques that were able to be detected than Linux on either architecture. In addition, these datasets were used to provide statistics on the performance of each of these techniques. It is understood that different techniques require different implementations and will have different performance metrics, but now these differences can be quantified. The artifact generated using these datasets and the Levene test, to fail to reject the null hypothesis, are the first technical implementation of this type of non-invasive determination of Anti-Debugging techniques using performance data. Lastly, we've answered the initial research questions, namely the performance impact on execution timing is minimal, but as demonstrated with this research it is measurable using the created artifact that was able to successfully identify 27 out of the 58 total techniques.

REFERENCES

1847. Criminal Copyright Infringement—17 U.S.C. 506(a) And 18 U.S.C. 2319. (2015, February 19). https://www.justice.gov/archives/jm/criminal-resource-manual-1847-criminal-copyright-infringement-17-usc-506a-and-18-usc-2319

Aadp. (n.d.). [Computer software]. Retrieved February 16, 2024, from https://code.google.com/archive/p/aadp/source/default/source

AbdelHameed, M. U., Sobh, M. A., & Bahaa Eldin, A. M. (2009). Portable executable automatic protection using dynamic infection and code redirection. *2009 International Conference on Computer Engineering & Systems*, 501–507. https://doi.org/10.1109/ICCES.2009.5383212

Abrath, B., Coppens, B., Nevolin, I., & De Sutter, B. (2020). Resilient Self-debugging Software Protection. *European Symposium on Security and Privacy Workshops*, 606–615.

Abrath, B., Coppens, B., Volckaert, S., Wijnant, J., & De Sutter, B. (2016). Tightly-coupled self-debugging software protection. *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, 1–10.

Apostolopoulos, T., Katos, V., Choo, K.-K. R., & Patsakis, C. (2020). *Resurrecting Anti-virtualization and Anti-debugging: Unhooking your Hooks*. http://eprints.bournemouth.ac.uk/34823/1/Anti_forensics.pdf

ARM Developer Documentation. (2023). Arm Limited. https://developer.arm.com/

Aycock, J., Manuel Gutierrez Cardenas, J., & Medeiros Nunes de Castro, D. (2009). Code Obfuscation using Pseudo-Random Number Generators. *2009 International Conference on Computational Science and Engineering*.

Bahaa-Eldin, A. M., & Sobh, M. A. A. (2014). A Comprehensive Software Copy Protection and Digital Rights Management Platform. *Ain Shams Engineering Journal*.

Bing, G., Guoyuan, L., & Jiutao, T. (2010). Application of Structured Exception Handling in Software Anti-Debugging. *2010 3rd International Conference on Advanced Computer Theory and Engineering*.

Bonfa, G. (n.d.). EventPairHandle as Anti-Dbg Trick. *EvilCodeCave*. Retrieved January 28, 2024, from https://www.evilfingers.com/publications/research_EN/EventPairsHandle.pdf

Branco, R. R., Barbosa, G. N., & Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat, 1*(2012), 1–27.

Brown, M. B., & Forsythe, A. B. (1974). Robust tests for the equality of variances. *Journal of the American Statistical Association, 69*(346), 364–367.

Cesare, S. (1999, January). *LINUX ANTI-DEBUGGING TECHNIQUES (FOOLING THE DEBUGGER)*. http://www.ouah.org/linux-anti-debugging.txt

Chen, P., Huygens, C., Desmet, L., & Joosen, W. (2016). Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware. *ICT Systems Security and Privacy Protection: 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30-June 1, 2016, Proceedings 31*, 323–336.

Chen, X., Andersen, J., Mao, Z. M., Bailey, M., & Nazario, J. (2008). Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 177–186. https://doi.org/10.1109/DSN.2008.4630086

Collberg, C. S., & Thomborson, C. (2002). Watermarking, Tamper-proofing and Obfuscation Tools for Software Protection. *IEEE Transactions on Software Engineering*, *28*(8), 735–746.

Comparison of operating system kernels. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Comparison_of_operating_system_kernels&oldid=1151237403#Feature_overview

Cownie, J. (2021, March 3). Fun with Timers and cpuid. *CPU Fun*. cpufun.substack.com/p/fun-with-timers-and-cpuid

Dasgupta, D., Akhtar, Z., & Sen, S. (2022). Machine learning in cybersecurity: A comprehensive survey. *The Journal of Defense Modeling and Simulation*, *19*(1), 57–106.

Dinaburg, A., Royal, P., Sharif, M., & Lee, W. (2008). Ether: Malware analysis via hardware virtualization extensions. *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 51–62.

DOMARS. (2023, April 13). *WinDbg Overview—Windows drivers*. https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/windbg-overview

Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons.

Ferrie, P. (2011, May 4). The “Ultimate” Anti-Debugging Reference. *Anti-Reversing*. https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf

Gagnon, M. N., Taylor, S., & Ghosh, A. K. (2007). Software Protection through Anti-Debugging. *IEEE Security & Privacy*, *5*(3), 82–84. https://doi.org/10.1109/MSP.2007.71

Gan, J., Kok, R., Kohli, P., Ding, Y., & Mah, B. (2015). *Using Virtual Machine Protections to Enhance Whitebox Cryptography*. IEEE/ACM 1st International Workshop on Software Protection.

Gao, S., & Lin, Q. (2012). Debugging classification and anti-debugging strategies. *Fourth International Conference on Machine Vision (ICMV 2011): Computer Vision and Image Analysis; Pattern Recognition and Basic Technologies*, *8350*, 729–734.

Giffin, J. T., Christodorescu, M., & Kruger, L. (2005). Strengthening software self-checksumming via self-modifying code. *21st Annual Computer Security Applications Conference (ACSAC’05)*, 10-pp.

Godbolt, M. (2023). *Compiler Explorer*. https://godbolt.org/

Gozzini, S. (2022). *PINvader: A dynamic analysis tool for evasive techniques detection and bypass in 64-bit windows binaries*.

Guoyuan, L., Jiutao, T., & Bing, G. (2010). A Survey on Shareware Protection Schemes. *2010 2nd International Conference on Signal Processing Systems*.

Guttman, D. (2017, December 27). *Common Anti-Debugging Techniques in the Malware Landscape*. https://www.deepinstinct.com/blog/common-anti-debugging-techniques-in-the-malware-landscape

Hex-Rays. (2023). *IDA Pro*. https://hex-rays.com/ida-pro/

Intel® 64 and IA-32 Architectures Software Developer Manuals. (2023, September 18). Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

IsDebuggerPresent function. (2021, June 28). *IsDebuggerPresent Function*. https://learn.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent

Kim, J.-W., Bang, J., & Choi, M.-J. (2020). *Defeating Anti-Debugging Techniques for Malware Analysis Using a Debugger*. https://pdfs.semanticscholar.org/135b/23507b0033b76722633ea1d4cdc9d2a9f106.pdf

Kim, J.-W., Bang, J., Moon, Y.-S., & Choi, M.-J. (2019). *Disabling Anti-Debugging Techniques for Unpacking System in User-level Debugger*.

Kim, M.-J., Lee, J.-Y., Chang, H.-Y., Cho, S., Park, Y., Park, M., & Wilsey, P. (2010). Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering. *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. https://doi.org/DOI 10.1109

kirschju. (2018). *Kirschju/debugmenot* [Computer software]. https://github.com/kirschju/debugmenot

Kulchytskyy, O. (2019, May 23). Anti Debugging Protection Techniques with Examples. *Apriorit*. https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software

Lee, J., Kang, B., & Gyu Im, E. (2013). *Rule-based Anti-anti-debugging System*.

Levene, H. (1960). Robuts tests for equality of variances. *Contributions to Probability and Statistics*, 278–292.

Levene Test for Equality of Variances. (n.d.). *NIST Engineering Statistics Handbook*. Retrieved January 28, 2024, from https://www.itl.nist.gov/div898/handbook/eda/section3/eda35a.htm

Li, A., Zhang, J., & Zhu, G. (2015). A Token Strengthened Encryption Packer to Prevent Reverse Engineering PE Files. *2015 International Conference on Estimation, Detection and Information Fusion*.

Mann, H. B., & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 50–60.

mattwojo, jamshedd, guijan, & alexbuckgit. (2023, March 28). Windows on Arm. *Windows on Arm*. https://learn.microsoft.com/en-us/windows/arm/overview

Miller, H. (2005). Beginners Guide to Basic Linux Anti-Anti-Debugging Techniques. *CodeBreakers Journal*.

Nahm, F. S. (2016). Nonparametric statistical tests for the continuous data: The basic concept and practical use. *Korean J Anesthesiol*, *69*(1), 8–14.

Onofri, S. S.-D. (2022, December 19). *Malware Analysis: GuLoader Dissection Reveals New Anti-Analysis*. Crowdstrike.Com. https://www.crowdstrike.com/blog/guloader-dissection-reveals-new-anti-analysis-techniques-and-code-injection-redundancy/

OpenSSL. (2024). [Computer software]. https://www.openssl.org/docs/

Paoloni, G. (2010). *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures* (p. 37). Intel Corporation.

https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

Park, J., Jang, Y.-H., Hong, S., & Park, Y. (2019). Automatic detection and bypassing of anti-debugging techniques for microsoft windows environments. *Advances in Electrical and Computer Engineering*, *19*(2), 23–28.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45–77.

Pék, G., Bencsáth, B., & Buttyán, L. (2011). nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. *Proceedings of the Fourth European Workshop on System Security*, 1–6.

Qi, Z., Li, B., Lin, Q., Yu, M., Xia, M., & Guan, H. (2012). SPAD: Software Protection Through Anti-Debugging Using Hardware-Assisted Virtualization. *Journal of Information Science and Engineering*, *28*(5), 813–827.

ReactOS Team. (2023). *ReactOS*. ReactOS. https://reactos.org

Roccia, T., & Lesueur, J.-P. (2023). The Unprotect Project. *The Unprotect Project*. https://unprotect.it/

Rutkowska, J. (2016). *Subverting VistaTM kernel for fun and profit*. Blackhat. http://www.orkspace.net/secdocs/Conferences/BlackHat/Asia/2006/Subverting%20Vista%20Kernel%20For%20Fun%20And%20Profit.pdf

Saad, M., & Taseer, M. (2022). Study of the Anti-Debugging Techniques and their Mitigations. *LGU International Journal for Electronic Crime Investigation*, *6*(3). https://ijeci.lgu.edu.pk/index.php/ijeci/article/download/139/102

Schallner, M. (2006). Beginners Guide to Basic Linux Anti Anti Debugging Techniques. *CodeBreakers Magazine*, *1*(2).

scottt. (2021). *Debugbreak* [Computer software]. https://github.com/scottt/debugbreak

ScyllaHide. (2023). [Computer software]. https://github.com/x64dbg/ScyllaHide

Shi, H., & Mirkovic, J. (2017). *Hiding Debuggers from Malware with Apate*.

Shi, H., Mirkovic, J., & Alwabel, A. (2017). Handling Anti-Virtual Machine Techniques in Malicious Software. *ACM Transactions on Privacy and Security*, *21*(1). https://doi.org/10.1145/3139292

Shields, T. (2010). *Anti-Debugging—A Developers View*.

Sikorski, M., & Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press.

Spisak, M. (2016, August 8). Hardware-Assisted Rootkits: Abusing Performance Counters on the ARM and x86 Architectures. *10th Usenix Workshop on Offensive Technologies*. WOOT. https://www.usenix.org/system/files/conference/woot16/woot16-paper-spisak.pdf

Stone, J., Greenwald, M., Partridge, C., & Hughes, J. (1998). Performance of checksums and CRCs over real data. *IEEE/ACM Transactions on Networking*, *6*(5), 529–543.

the GDB developers. (2023). *GDB: The GNU Project Debugger*. https://sourceware.org/gdb/

The pandas development team. (2020). *pandas-dev/pandas: Pandas* (latest) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.3509134

Title 17—Copyrights. (2021). https://www.govinfo.gov/content/pkg/USCODE-2021-title17/pdf/USCODE-2021-title17-chap5-sec506.pdf

Title 18—Crimes and Criminal Procedure. (2021). https://www.govinfo.gov/content/pkg/USCODE-2021-title18/pdf/USCODE-2021-title18-partI-chap113-sec2319.pdf

Update: Destructive Malware Targeting Organizations in Ukraine | CISA. (2022, April 28). https://www.cisa.gov/news-events/cybersecurity-advisories/aa22-057a

Vasudevan, A., & Terraballi, R. (2005). *Stealth Breakpoints*. 21st Annual Computer Security Applications Conference.

Vasudevan, A., & Yerraballi, R. (2006). Cobra: Fine-grained malware analysis using stealth localized-executions. *2006 IEEE Symposium on Security and Privacy (S&P'06)*, 15-pp.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Voisin, J. (2016). *Jvoisin/pangu* [Computer software]. https://github.com/jvoisin/pangu

Vrba, Ž., Halvorsen, P., & Griwodz, C. (2010). Program Obfuscation by Strong Cryptography. *2010 International Conference on Availability, Reliability and Security*, 242–247. https://doi.org/10.1109/ARES.2010.47

Wan, J., Zulkernine, M., & Liem, C. (2018). A Dynamic App Anti-Debugging Approach on Android ART Runtime. *2018 IEEE 16th International Conference on Dependable, Autonomic and Secure Comp.*

Xu, J., Zhang, L., Yang, L., Mao, Y., & Shi, X. (2016). An Effective Android Software Reinforcement Scheme Based on Online Key. *2016 IEEE 18th International Conference on High Performance Computing and Communications*. https://doi.org/DOI 10.1109/HPCC-SmartCity-DSS.2016.62

Yi, T., Zong, A., Yu, M., Gao, S., Lin, Q., Yu, P., Ren, Z., & Qi, Z. (n.d.). Anti-debugging Framework Based on Hardware Virtualization Technology. *2009 International Conference on Research Challenges in Computer Science*.

Yoshizaki, K., & Yamauchi, T. (2014). Malware Detection Method Focusing on Anti-Debugging Functions. *2014 Second International Symposium on Computing and Networking*.

APPENDIX A – SOURCE CODE

The full source code and data are viewable on Github at https://github.com/quantumite/dissertation.

APPENDIX B – PERFORMANCE DATA (GRAPHS)

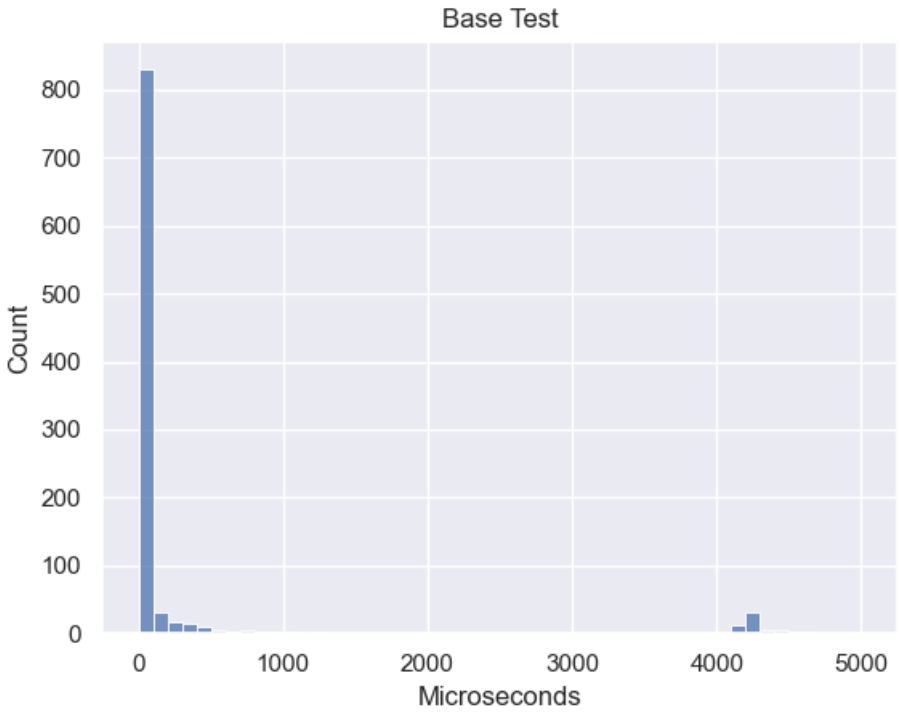

Figure 2 – Base Test Distribution of Data (Windows)

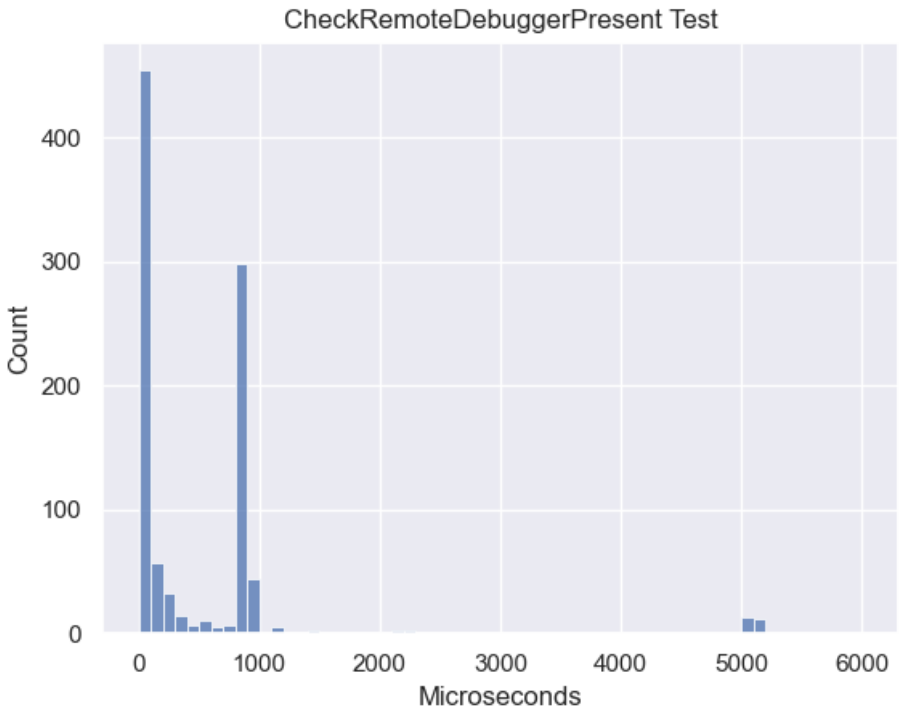

Figure 3 – CheckRemoteDebuggerPresent Test Distribution of Data (Windows)

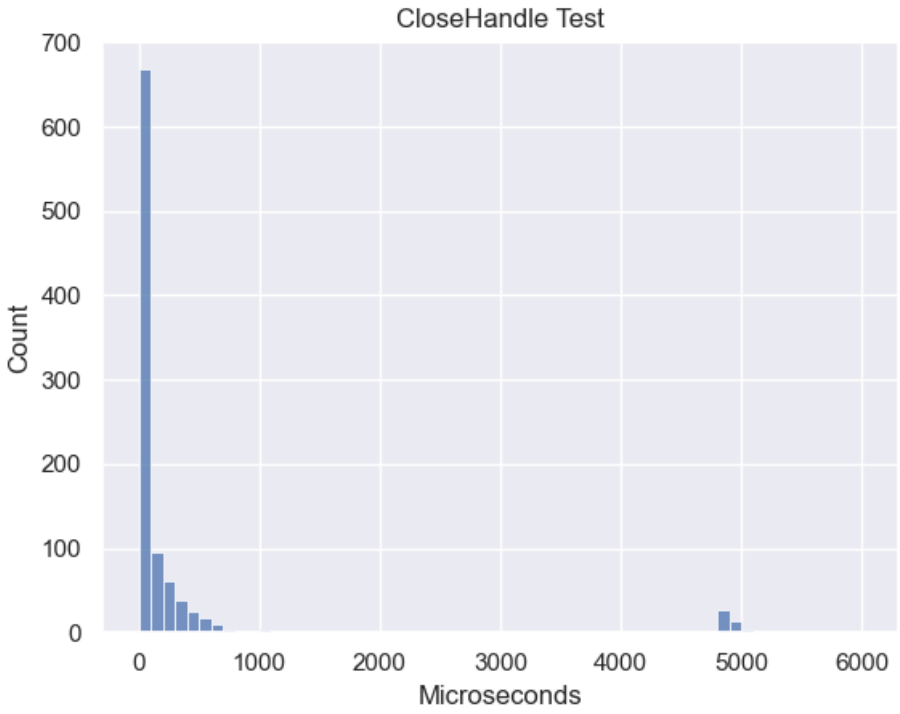

Figure 4 – CloseHandle Test Distribution of Data (Windows)

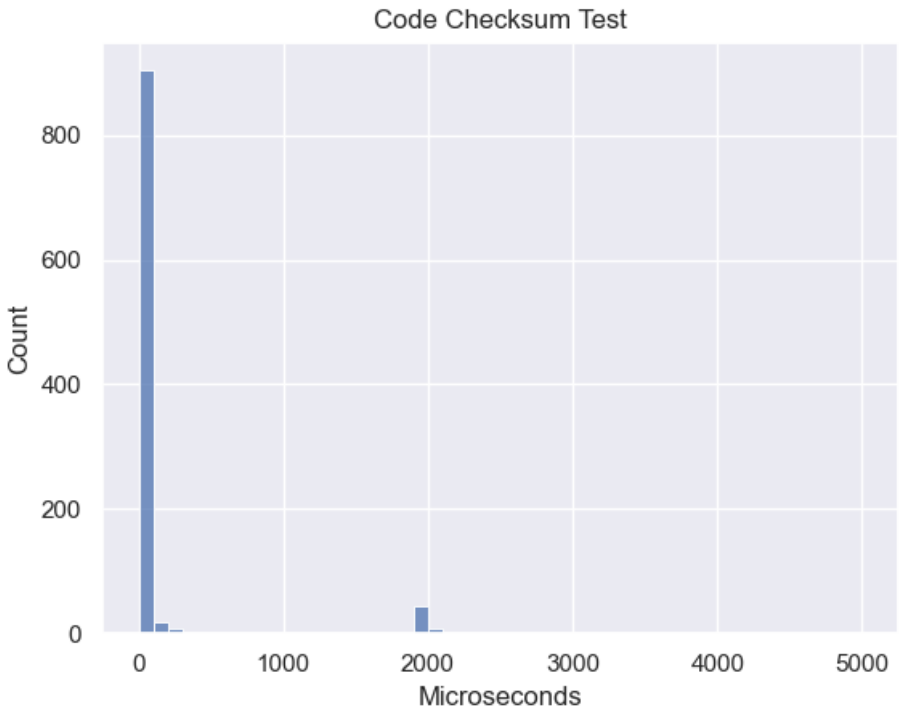

Figure 5 – Code Checksum Test Distribution of Data (Windows)

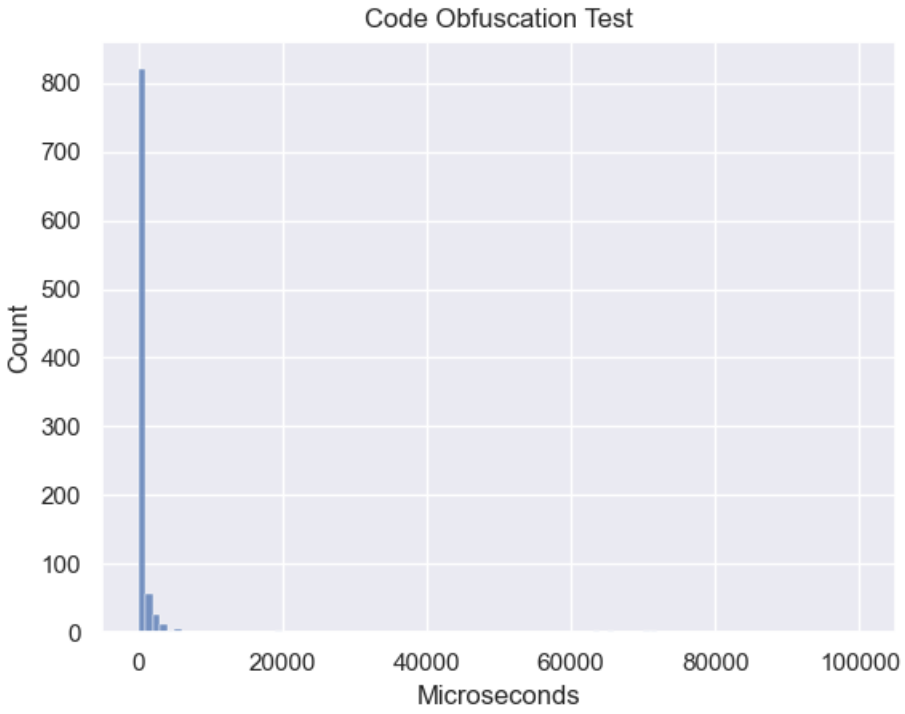

Figure 6 – Code Obfuscation Test Distribution of Data (Windows)

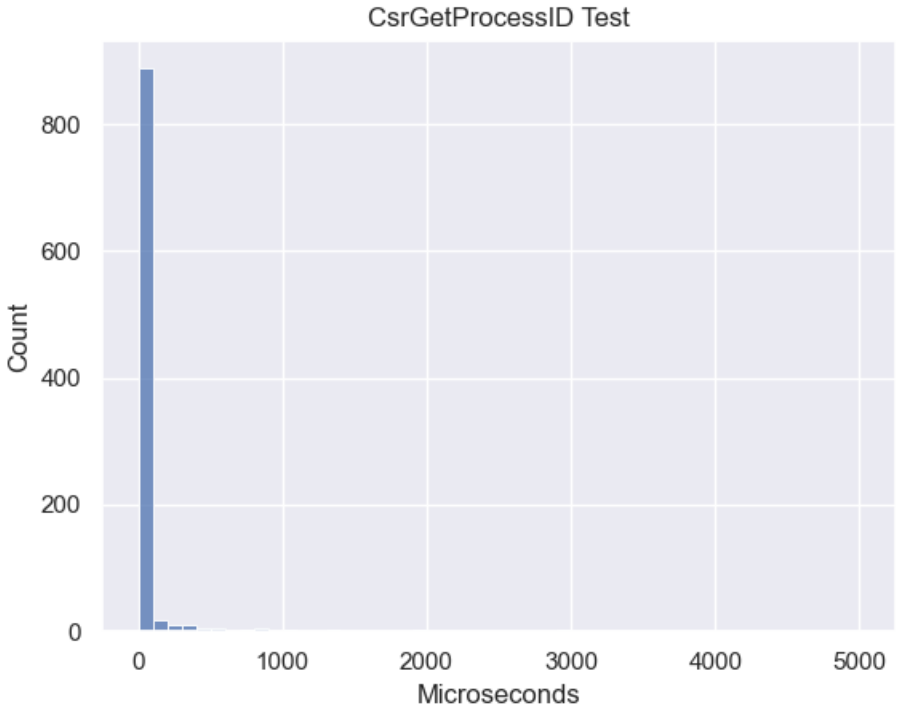

Figure 7 – CsrGetProcessID Test Distribution of Data (Windows)

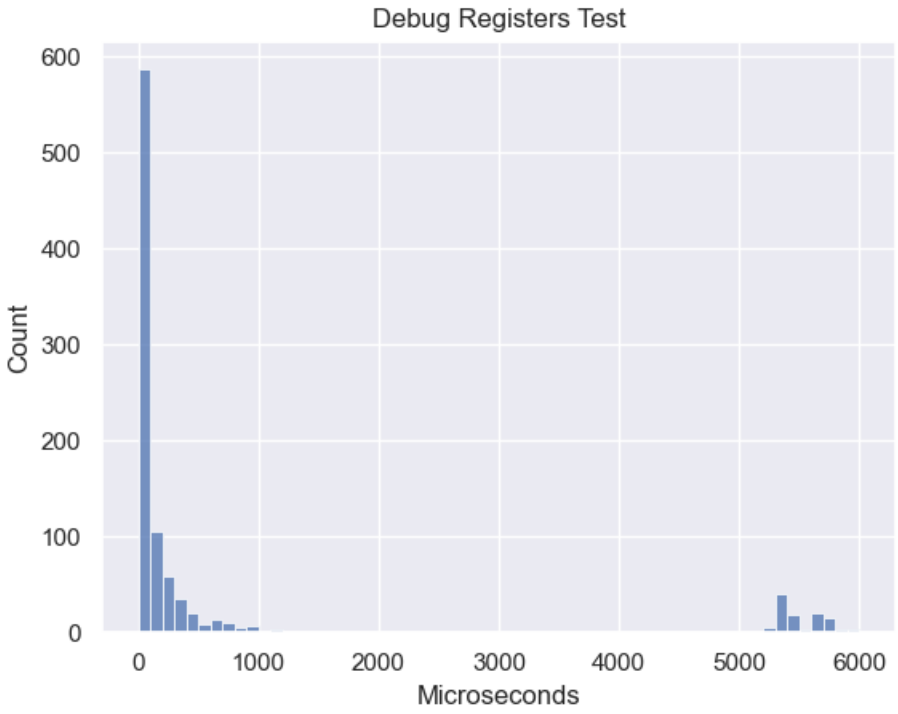

Figure 8 – Debug Registers Test Distribution of Data (Windows)

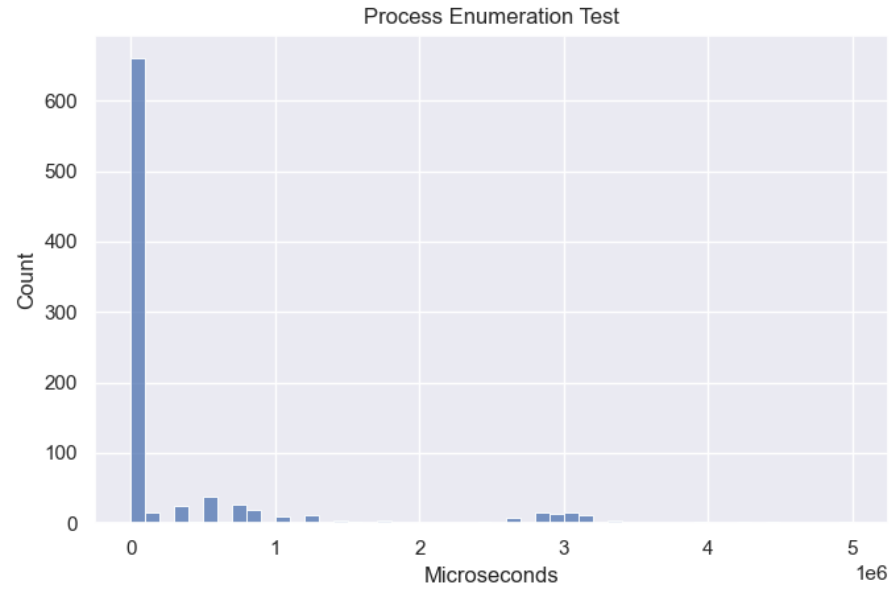

Figure 9 – Process Enumeration Test Distribution of Data (Windows)

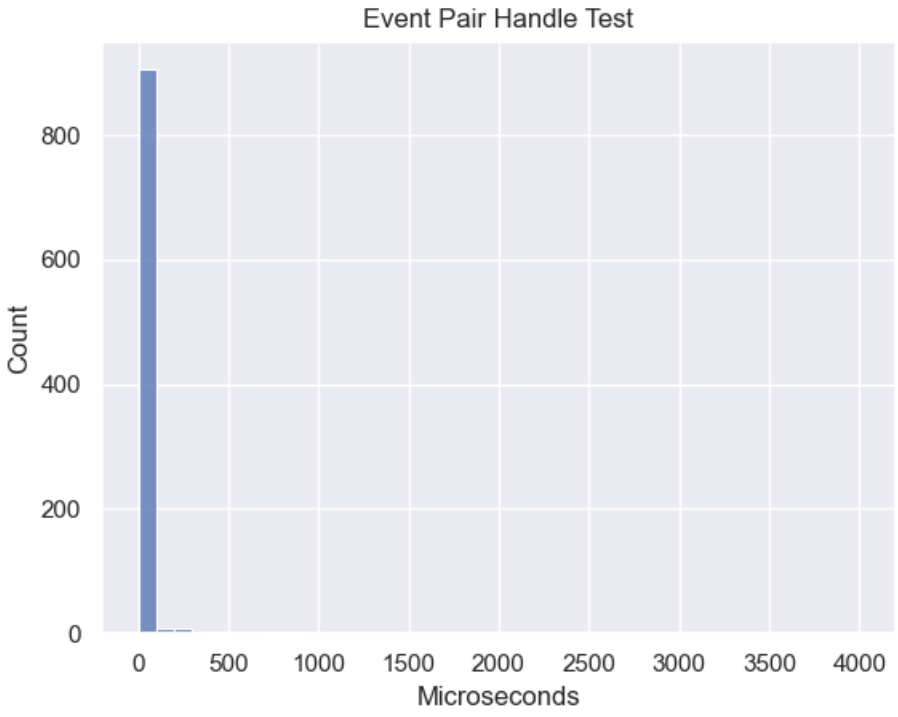

Figure 10 – Event Pair Handle Test Distribution of Data (Windows)

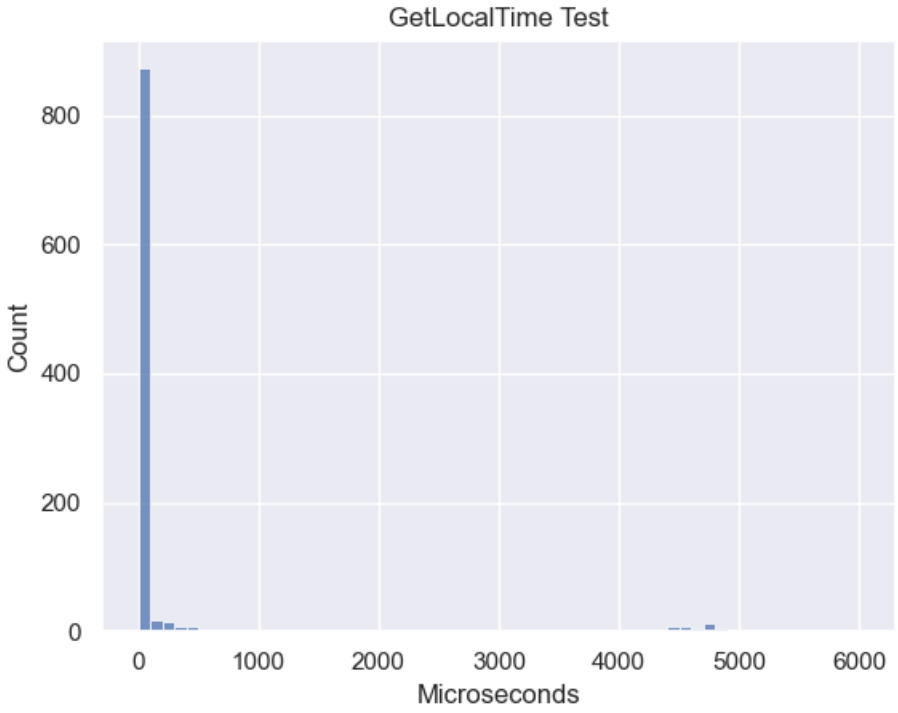

Figure 11 – GetLocalTime Test Distribution of Data (Windows)

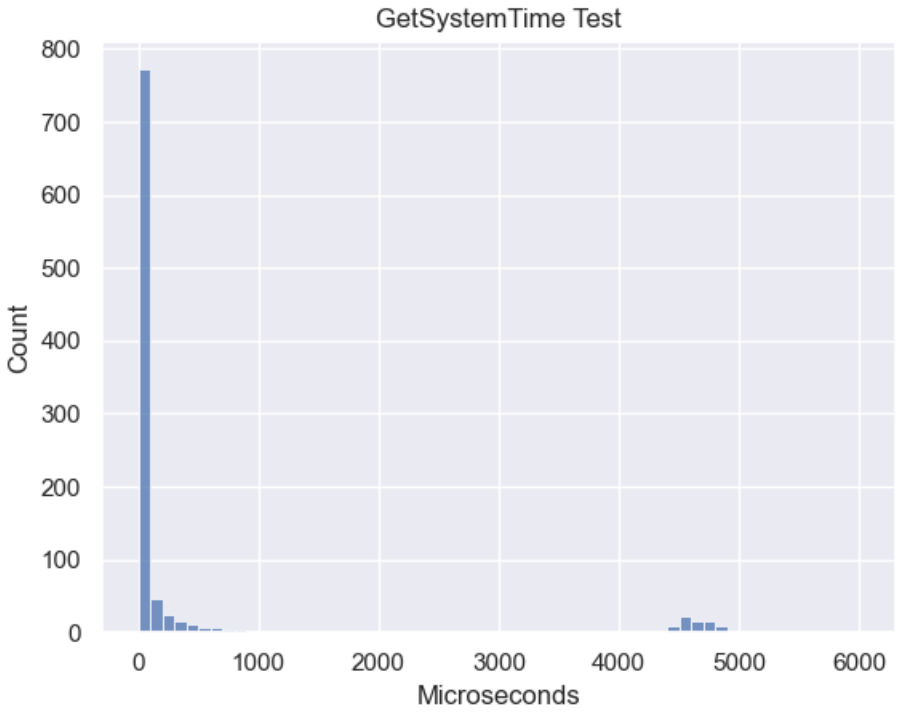

Figure 12 – GetSystemTime Test Distribution of Data (Windows)

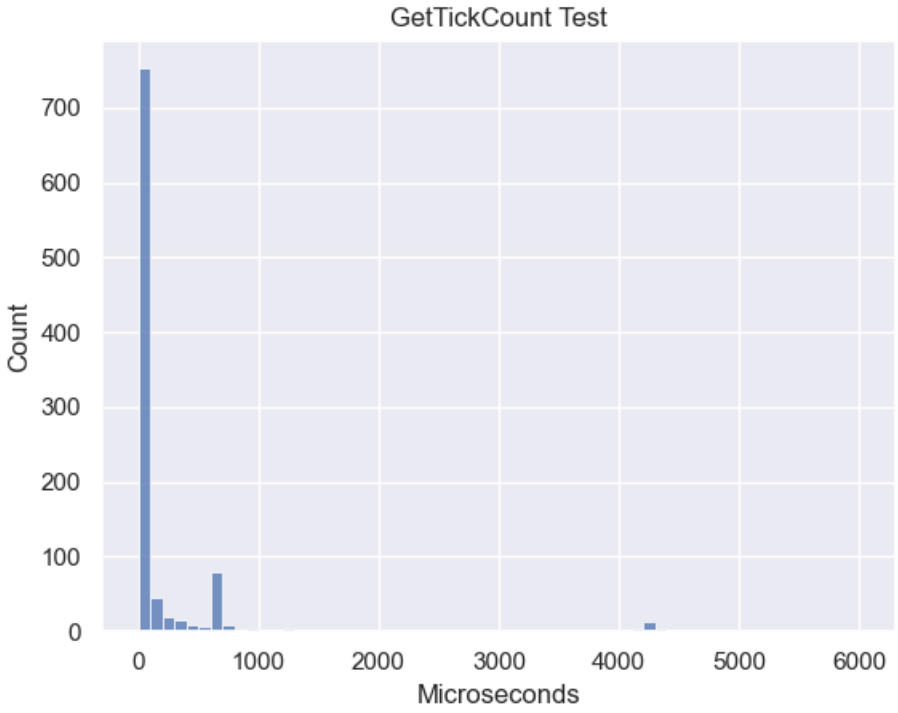

Figure 13 – GetTickCount Test Distribution of Data (Windows)

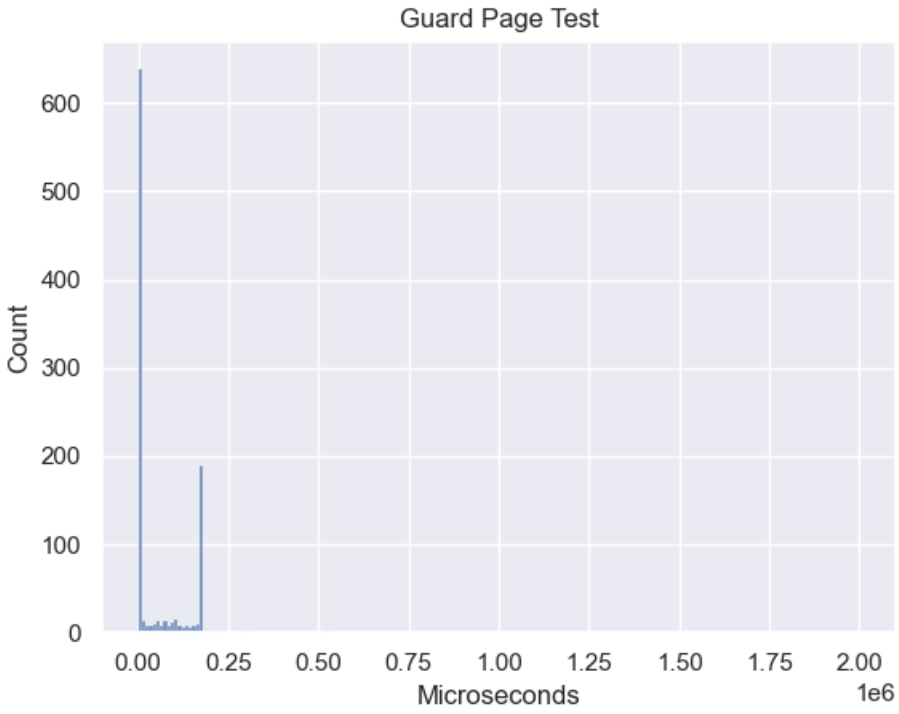

Figure 14 – Guard Page Test Distribution of Data (Windows)

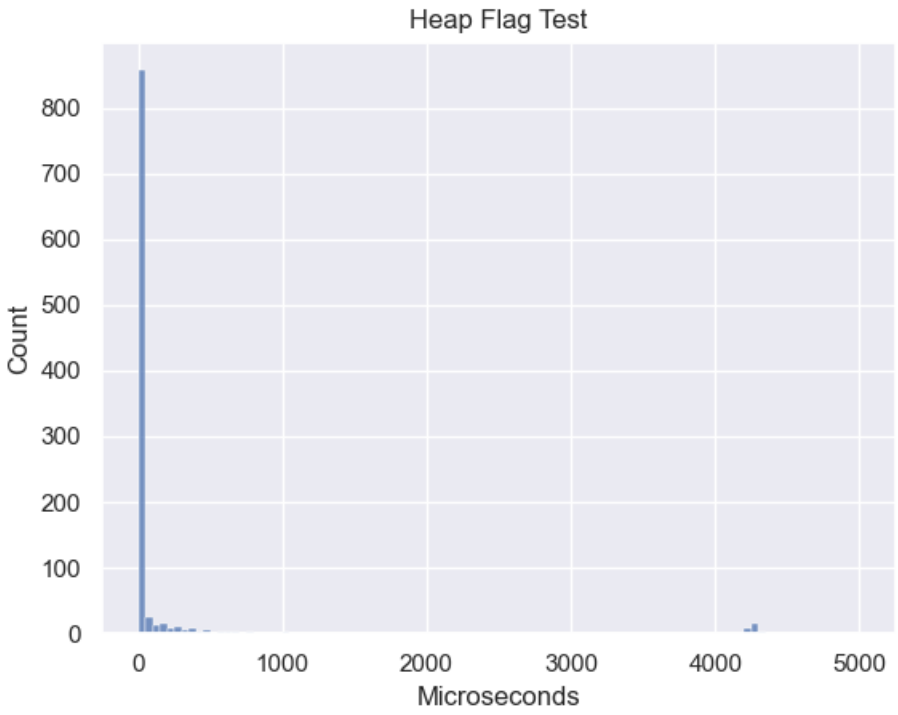

Figure 15 – Heap Flag Test Distribution of Data (Windows)

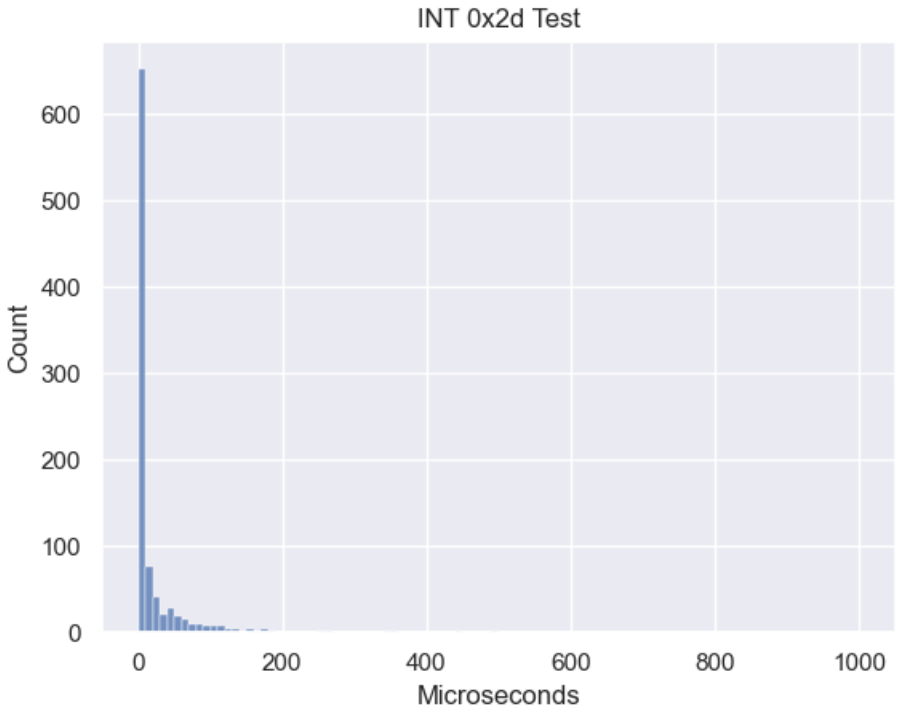

Figure 16 – INT 0x2d Test Distribution of Data (Windows)

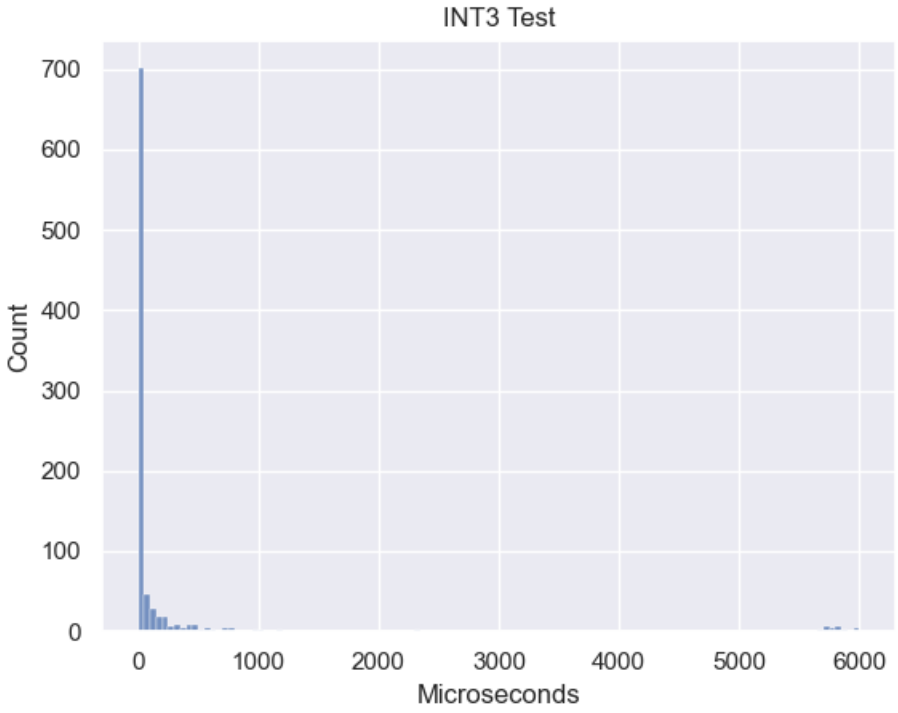

Figure 17 – INT3 Test Distribution of Data (Windows)

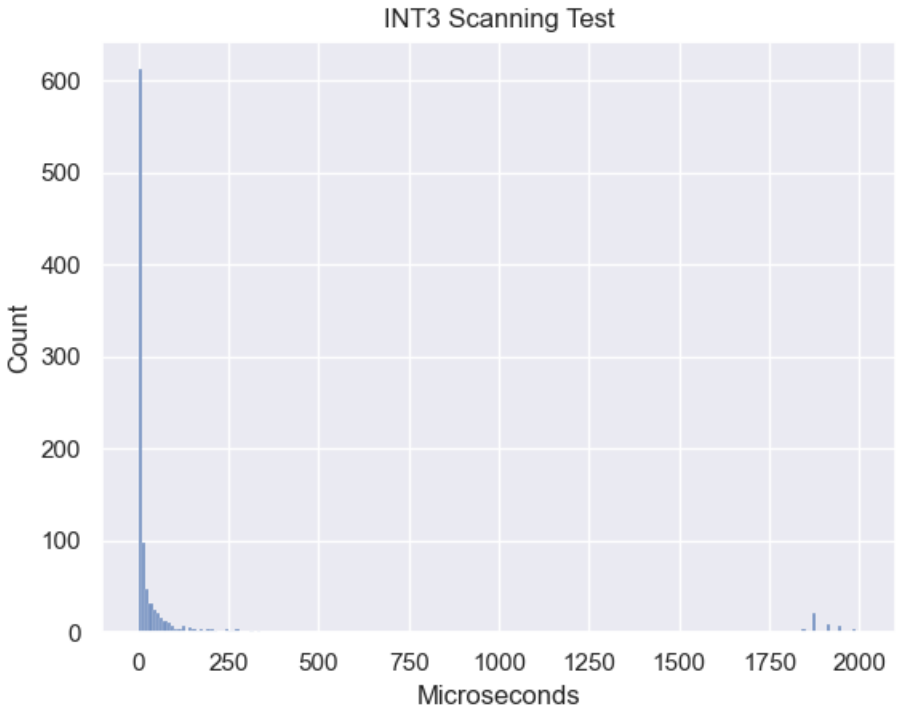

Figure 18 – INT3 Scan Test Distribution of Data (Windows)

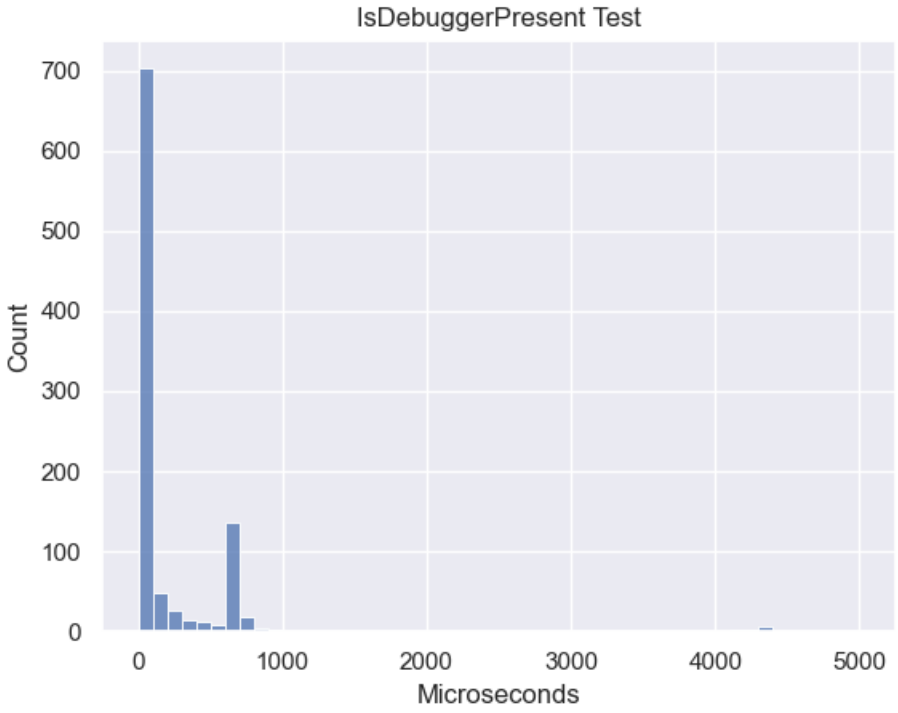

Figure 19 – IsDebuggerPresent Test Distribution of Data (Windows)

Figure 20 – Memory Encryption Test Distribution of Data (Windows)

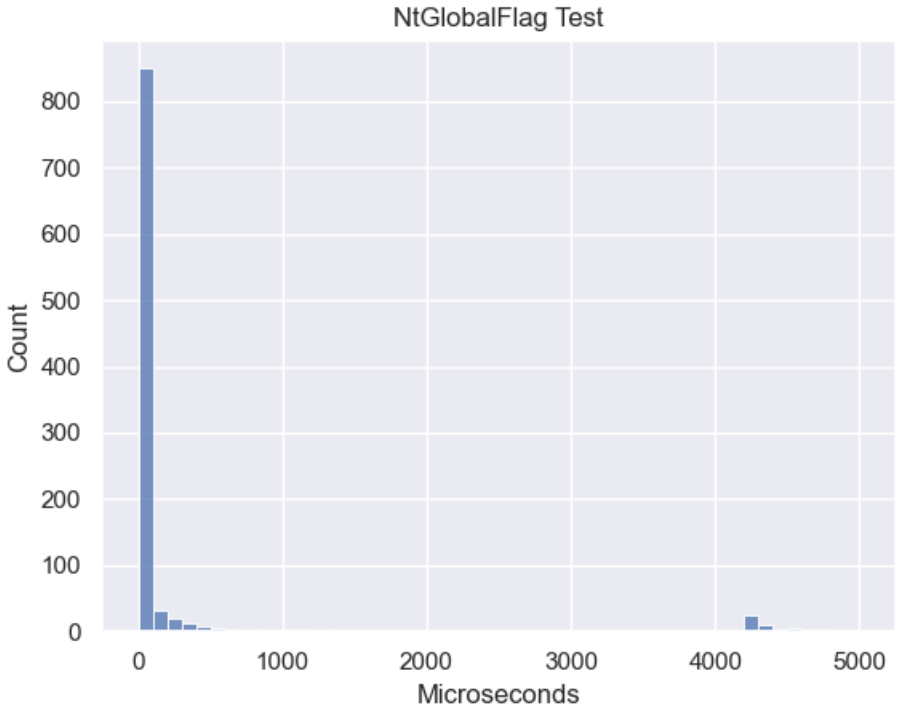

Figure 21 – NtGlobalFlag Test Distribution of Data (Windows)

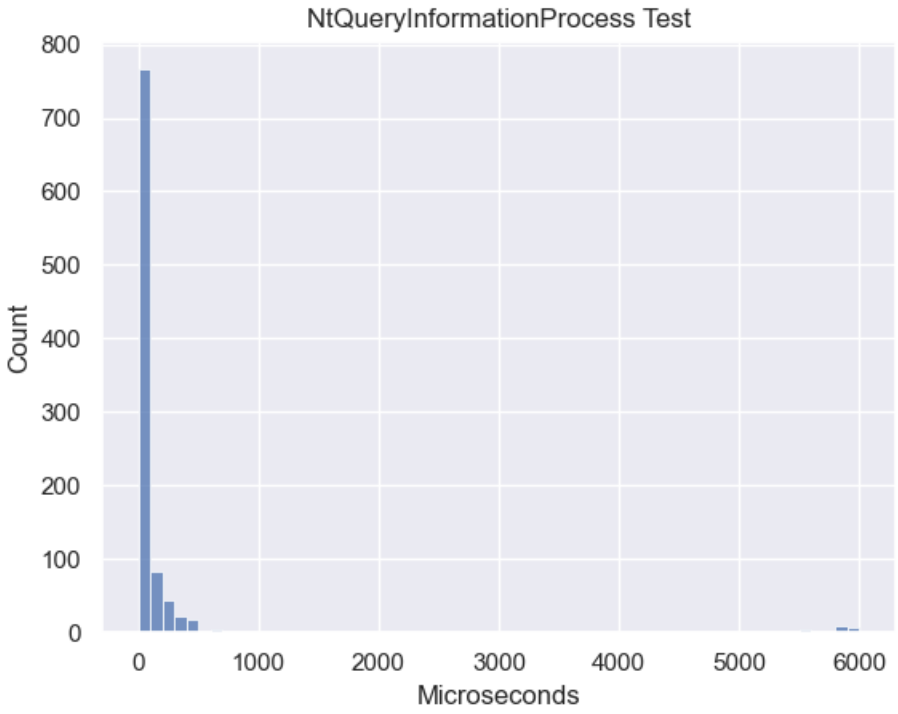

Figure 22 – NtQueryInformationProcess Test Distribution of Data (Windows)

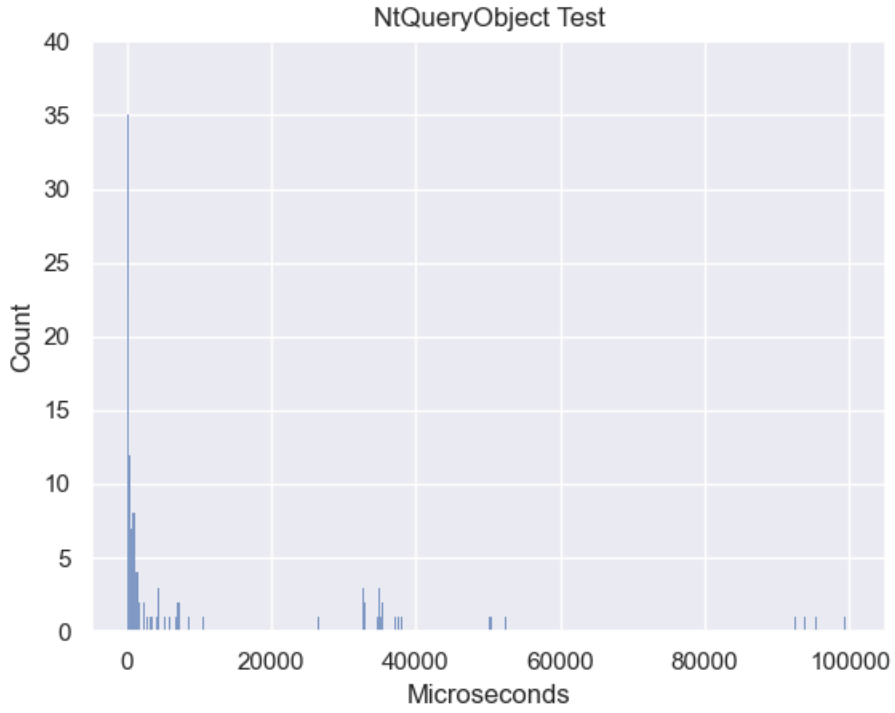

Figure 23 – NtQueryObject Test Distribution of Data (Windows)

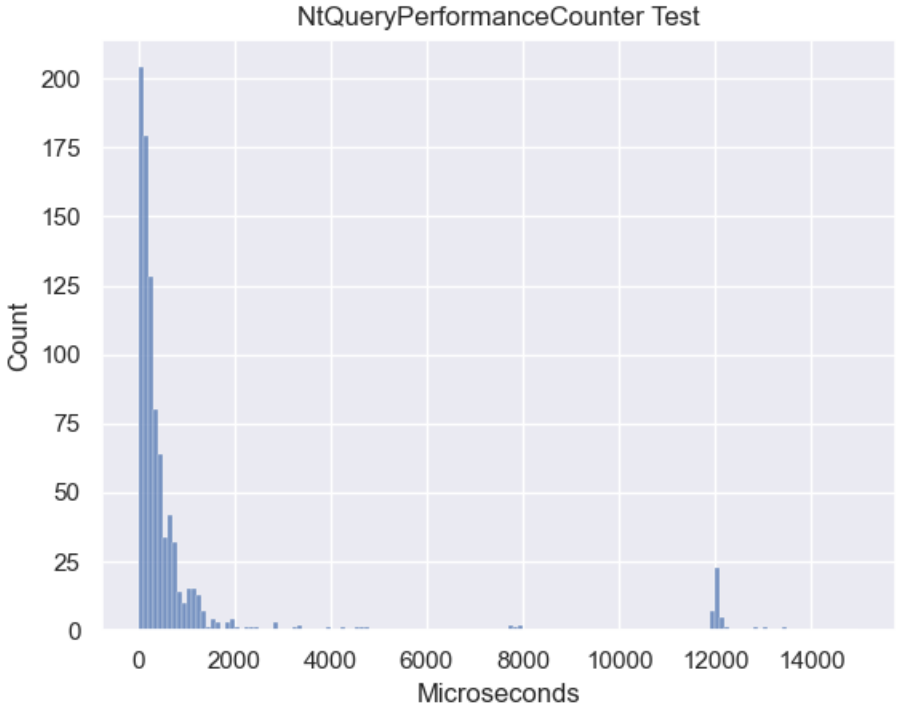

Figure 24 – NtQueryPerformanceCounter Test Distribution of Data (Windows)

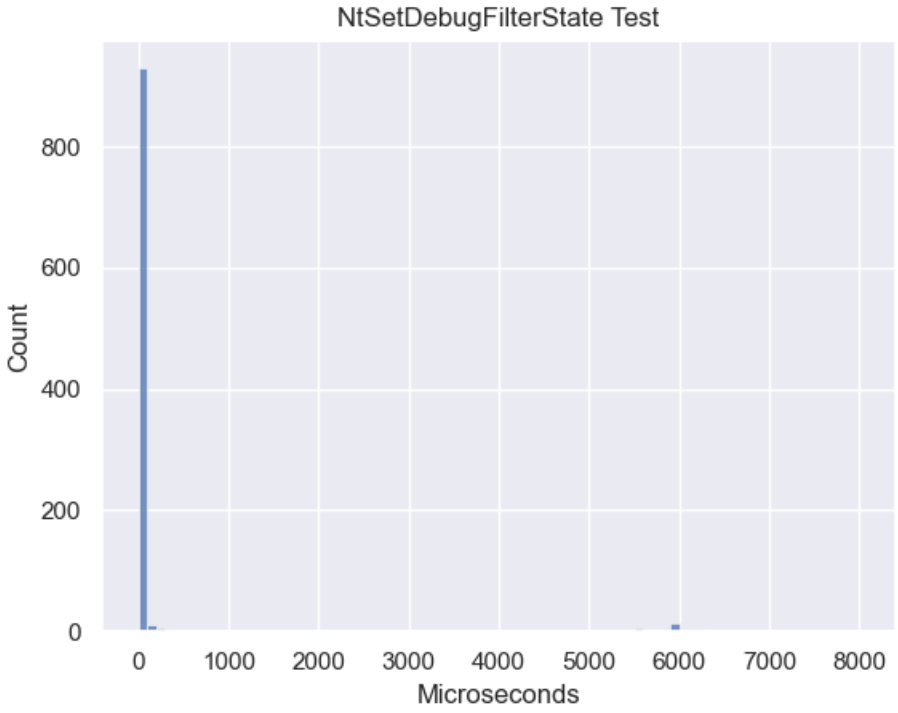

Figure 25 – NtSetDebugFilterState Test Distribution of Data (Windows)

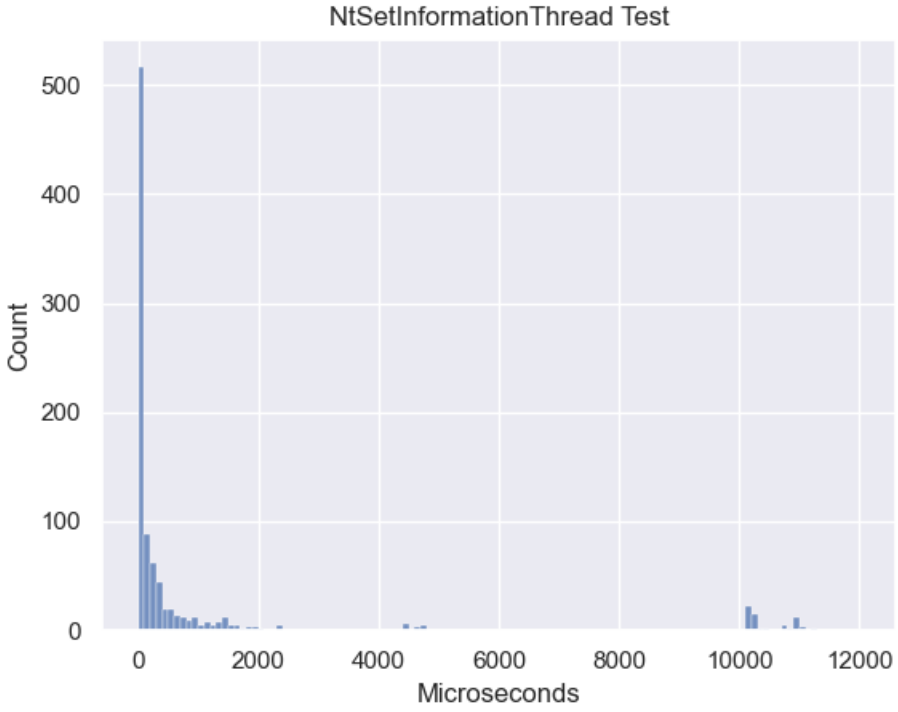

Figure 26 – NtSetInformationThread Test Distribution of Data (Windows)

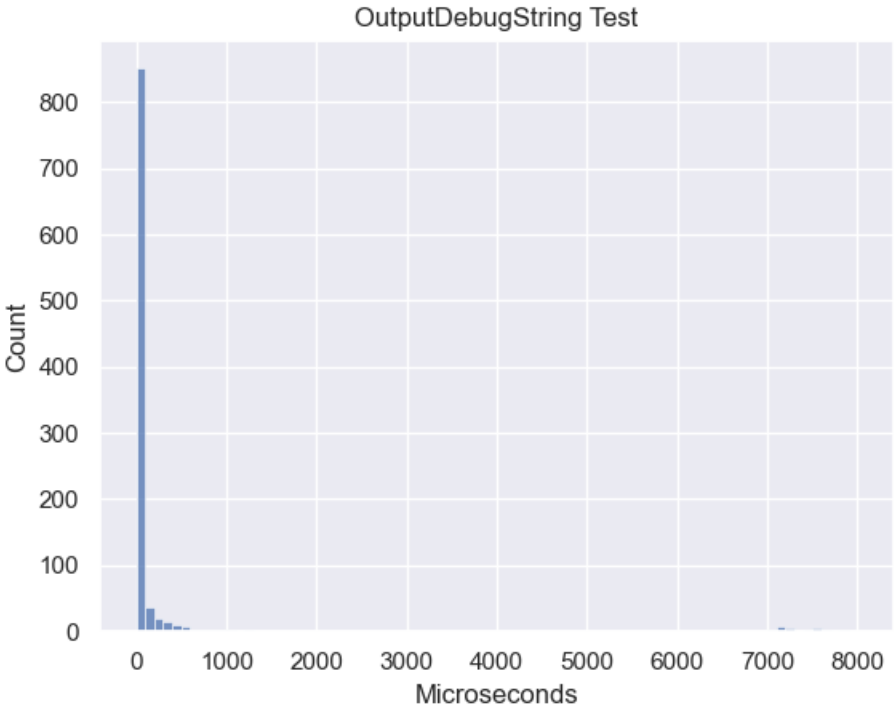

Figure 27 – OutputDebugString Test Distribution of Data (Windows)

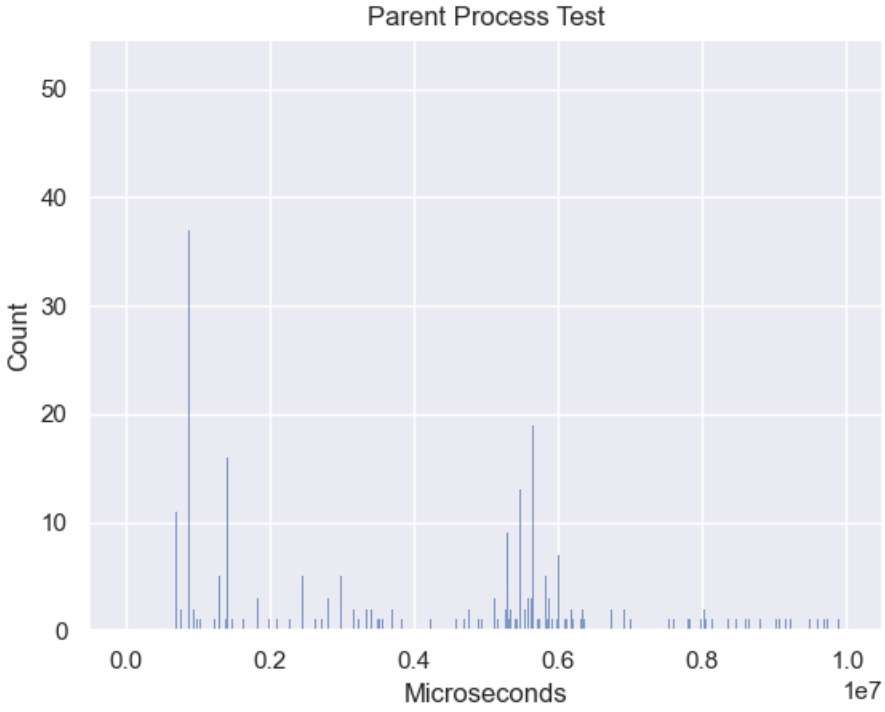

Figure 28 – Parent Process Test Distribution of Data (Windows)

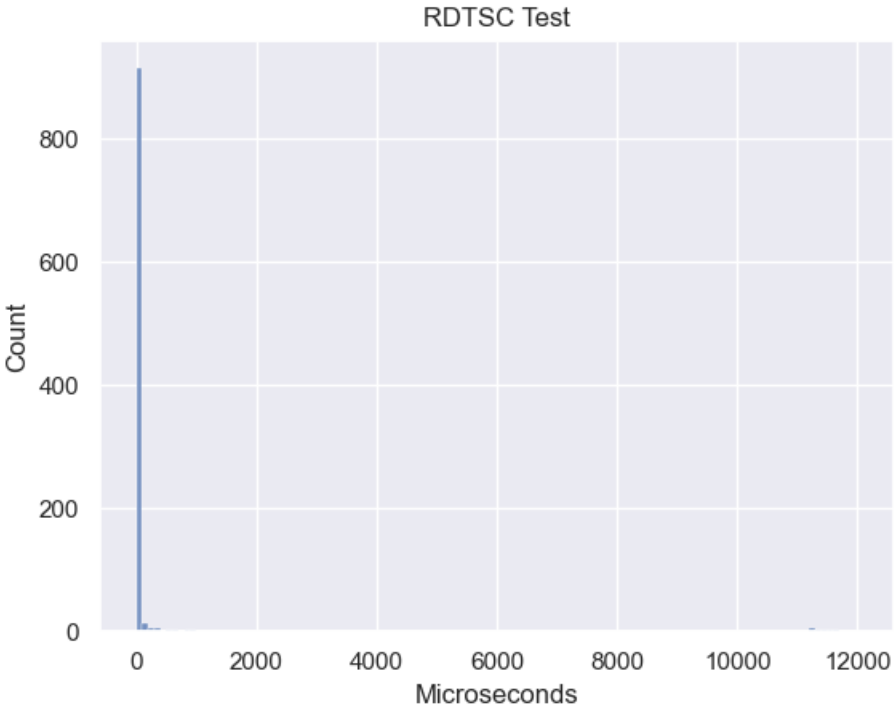

Figure 29 – RDTSC Test Distribution of Data (Windows)

Figure 30 – Self-Debugging Test Distribution of Data (Windows)

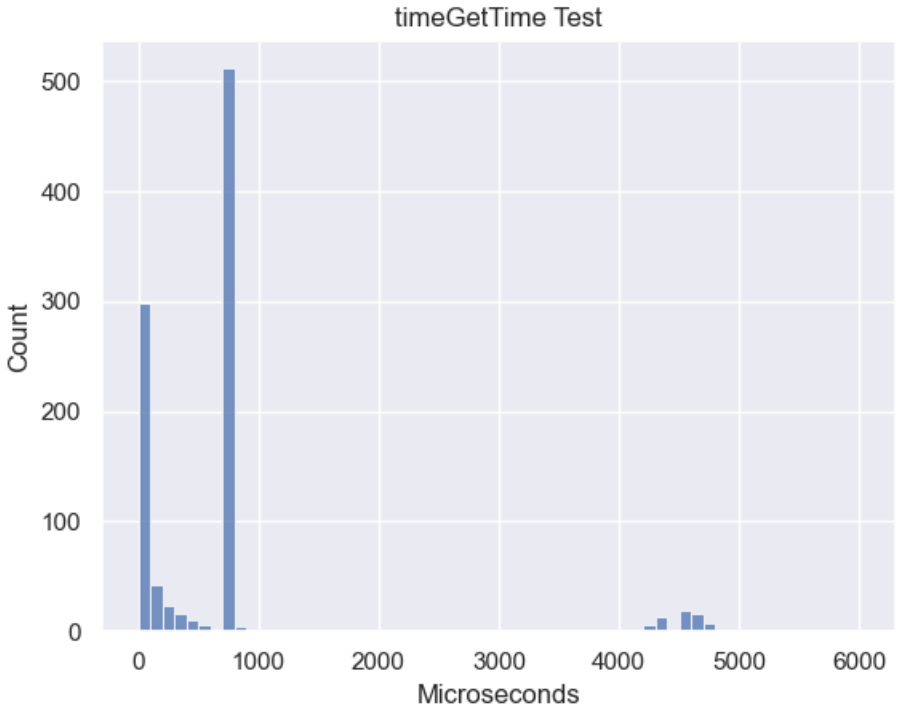

Figure 31 – timeGetTime Test Distribution of Data (Windows)

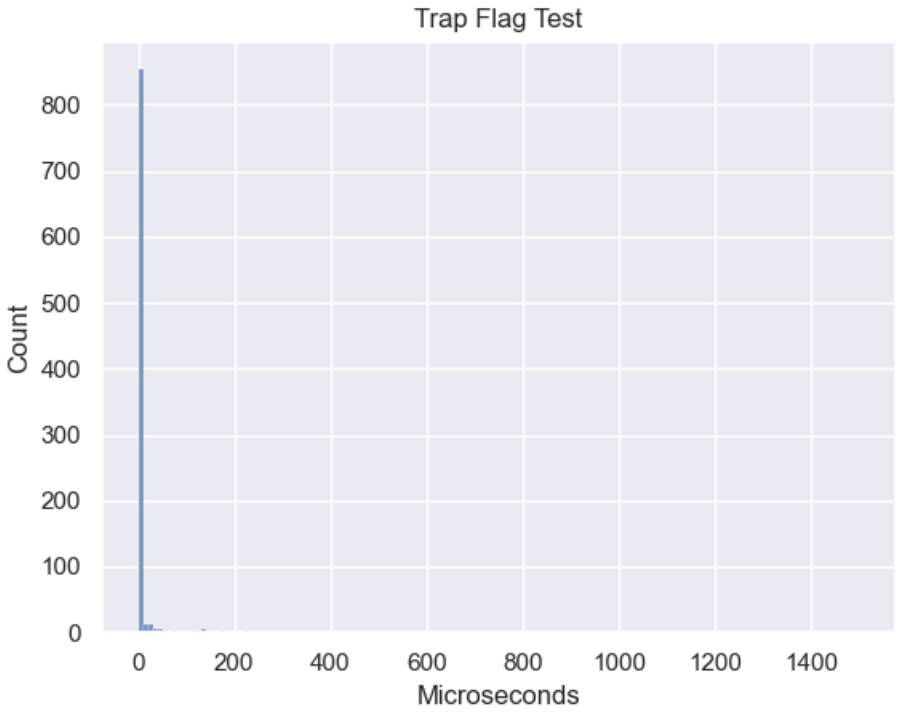

Figure 32 – Trap Flag Test Distribution of Data (Windows)

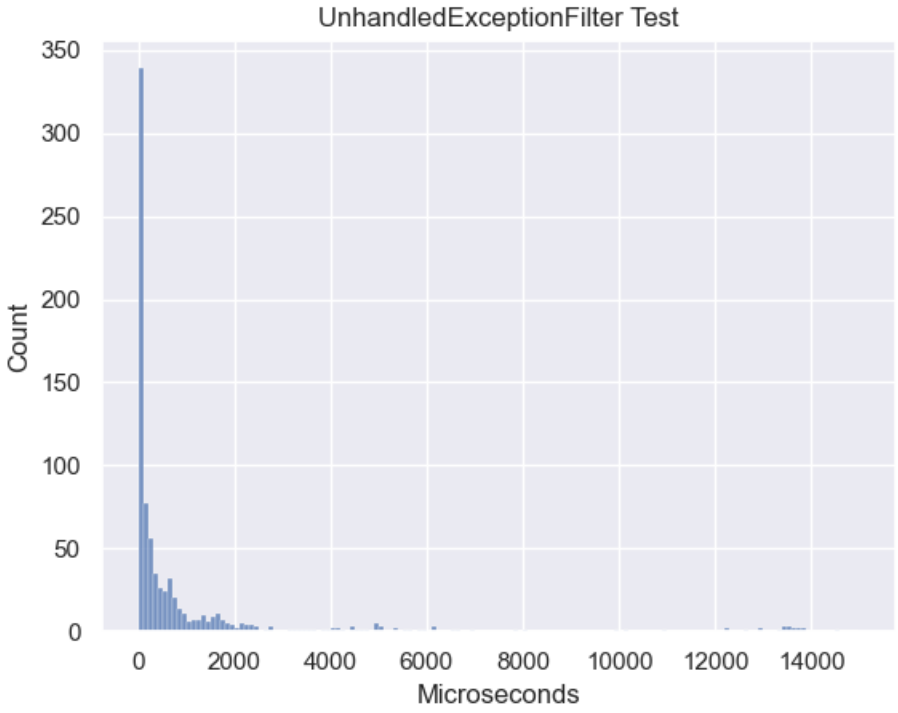

Figure 33 – UnhandledExceptionFilter Test Distribution of Data (Windows)

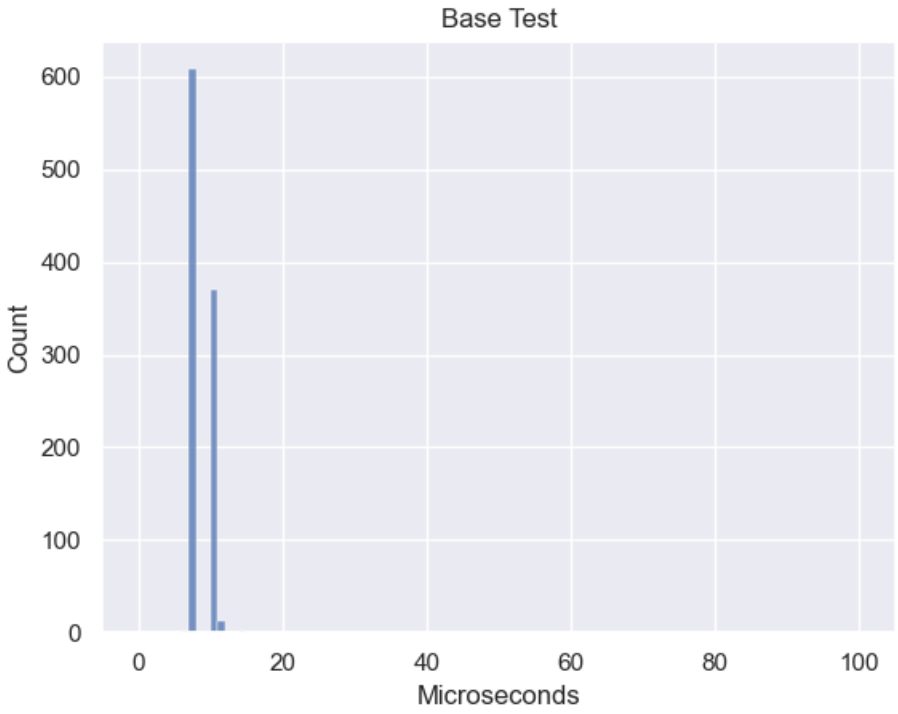

Figure 34 – Base Test Distribution of Data (Linux, Intel)

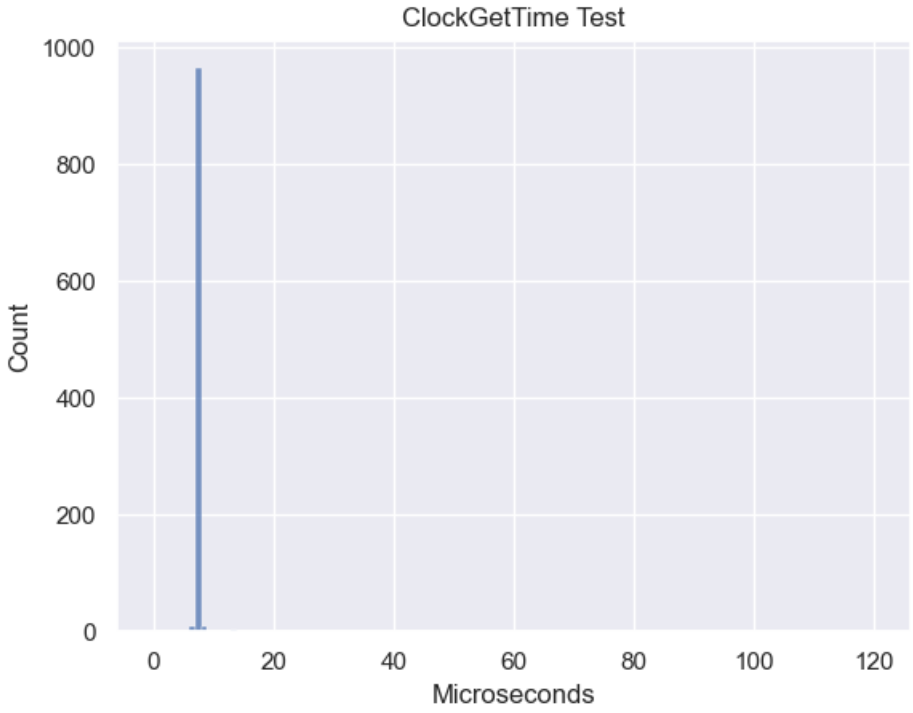

Figure 35 – ClockGetTime Test Distribution of Data (Linux, Intel)

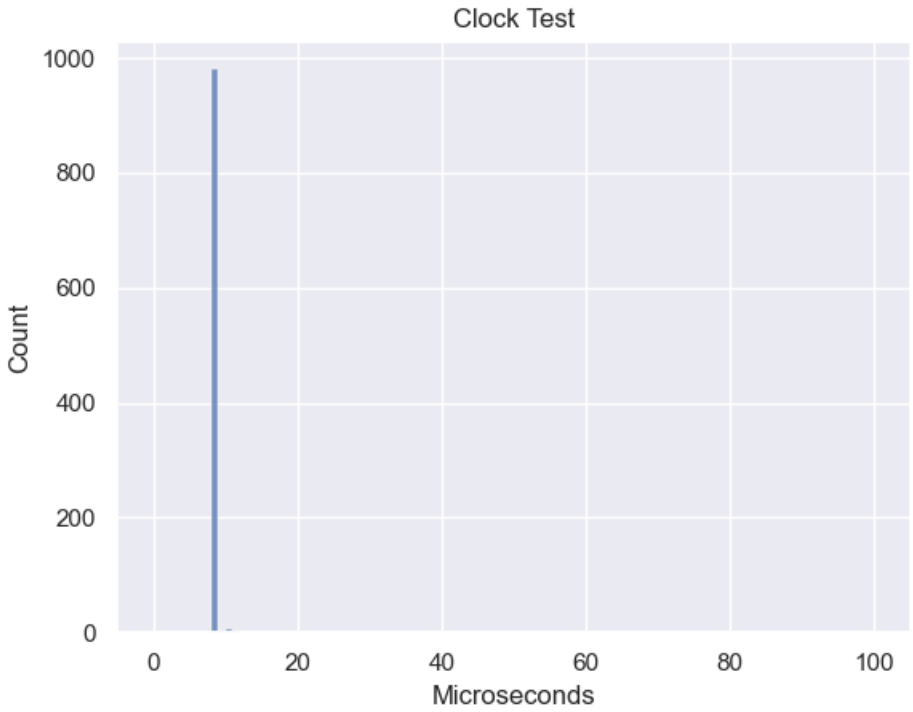

Figure 36 – Clock Test Distribution of Data (Linux, Intel)

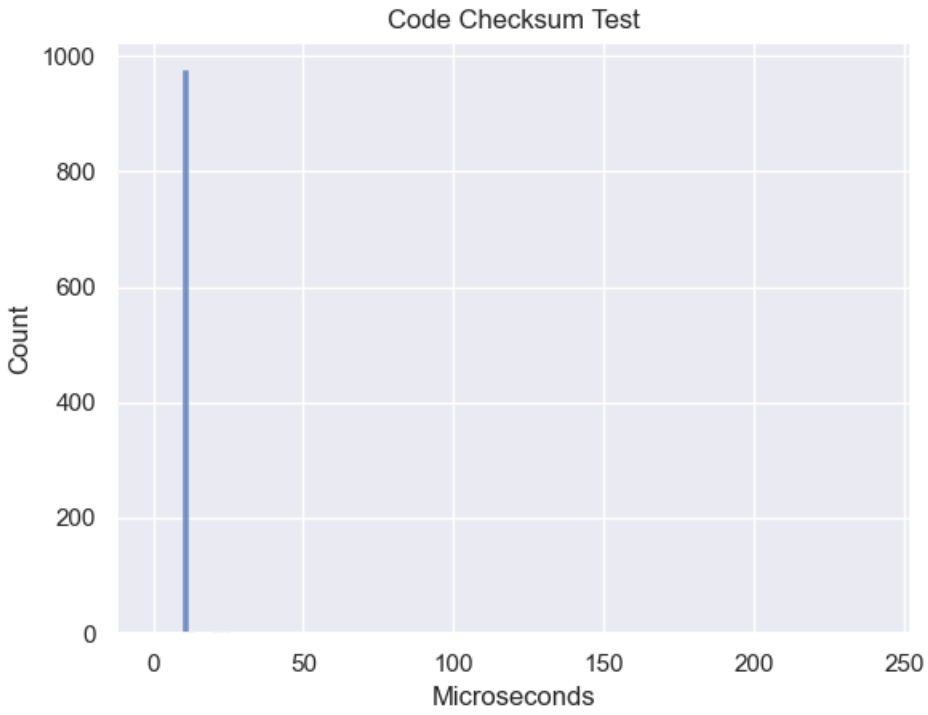

Figure 37 – Code Checksum Test Distribution of Data (Linux, Intel)

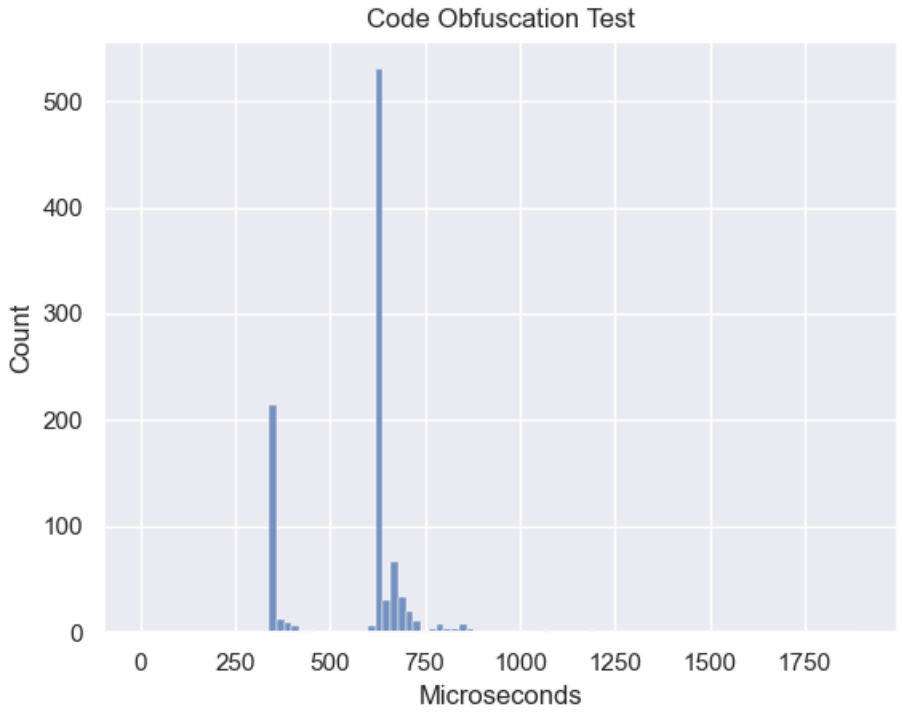

Figure 38 – Code Obfuscation Test Distribution of Data (Linux, Intel)

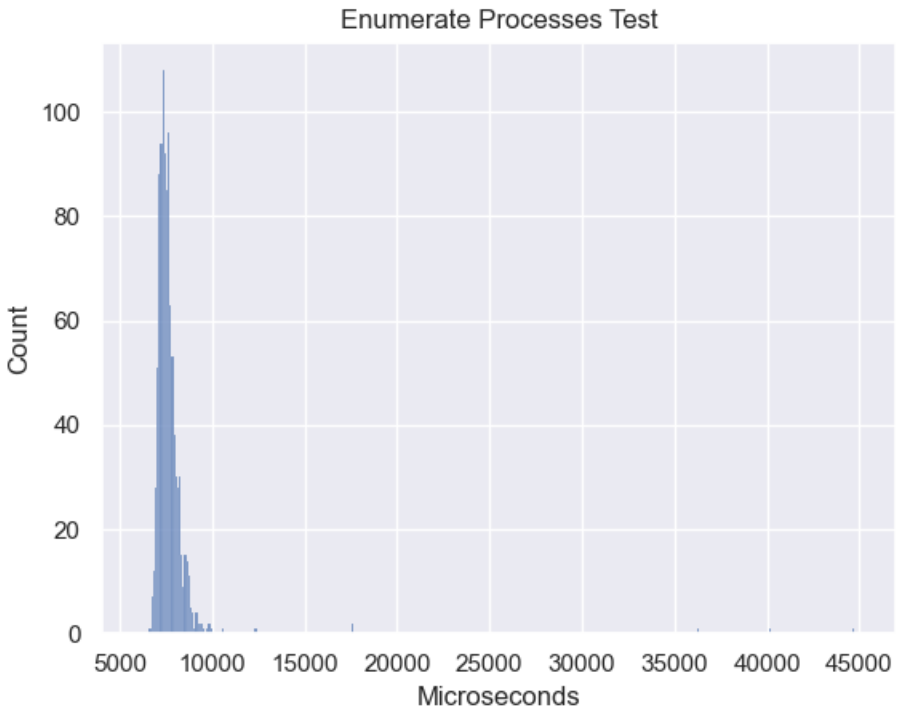

Figure 39 – Process Enumeration Test Distribution of Data (Linux, Intel)

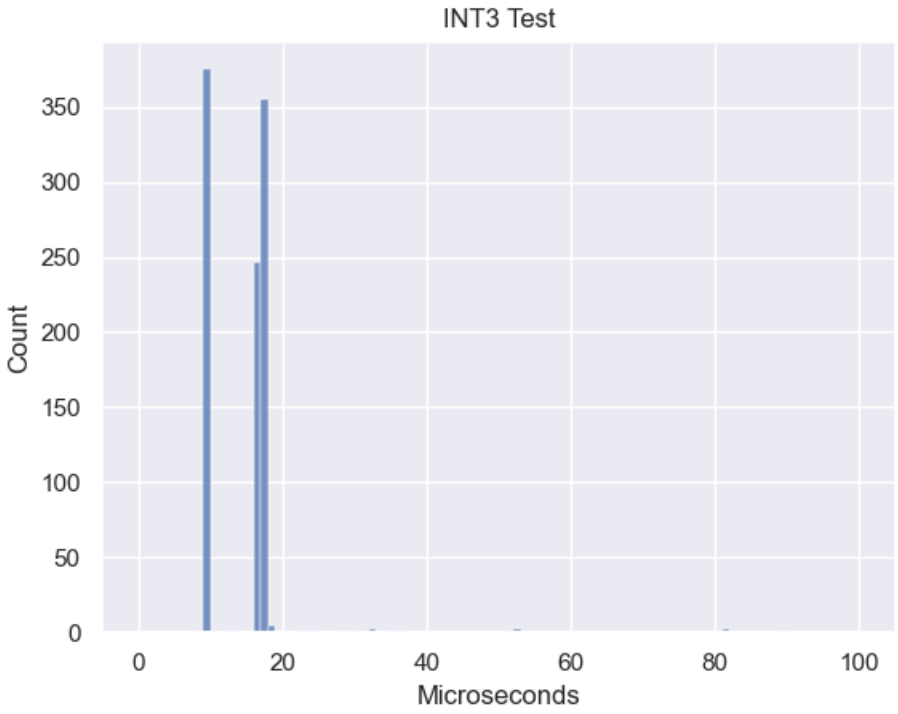

Figure 40 – INT3 Test Distribution of Data (Linux, Intel)

Figure 41 – INT3 Scanning Test Distribution of Data (Linux, Intel)

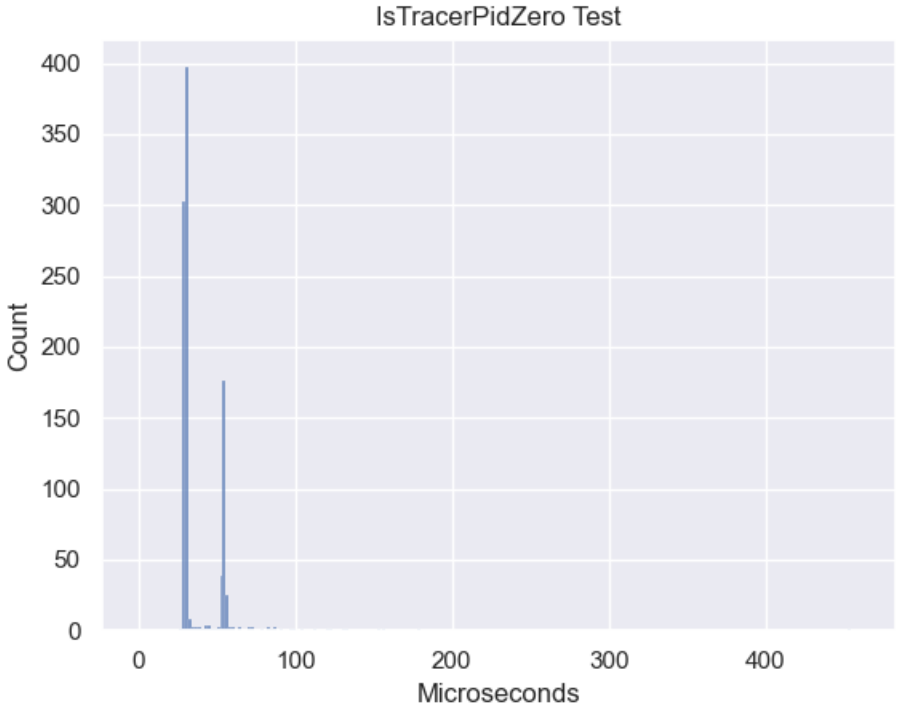

Figure 42 – IsTracerPidZero Test Distribution of Data (Linux, Intel)

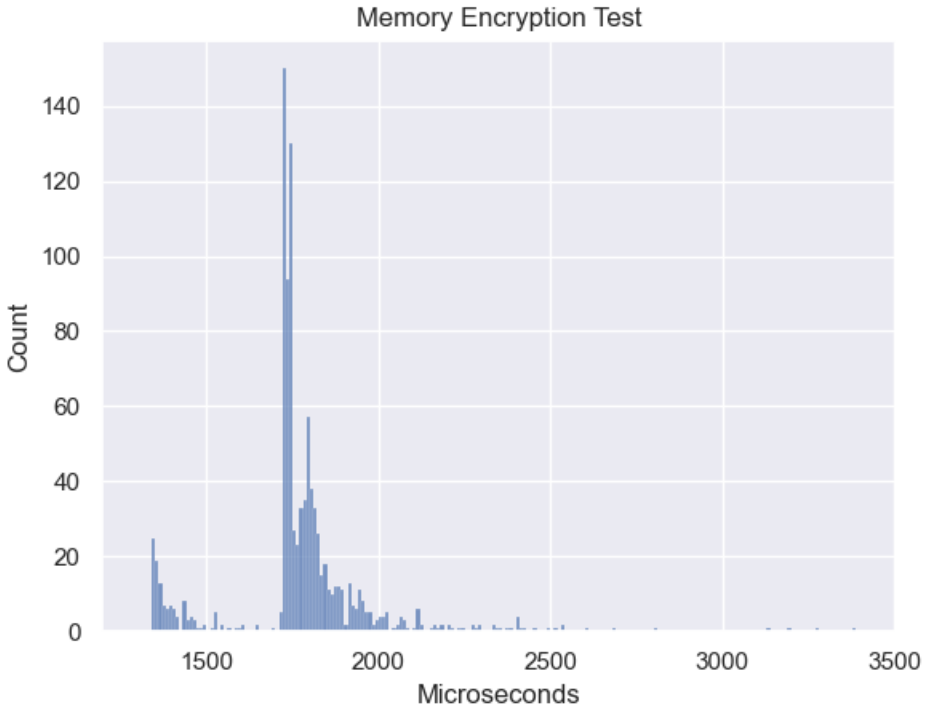

Figure 43 – Memory Encryption Test Distribution of Data (Linux, Intel)

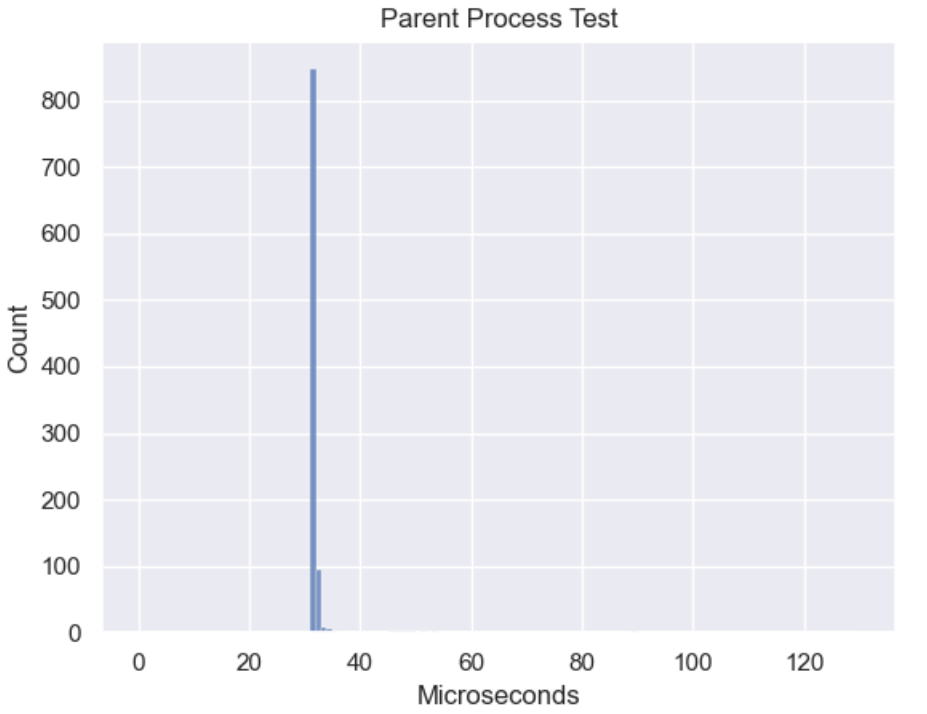

Figure 44 – Parent Process Test Distribution of Data (Linux, Intel)

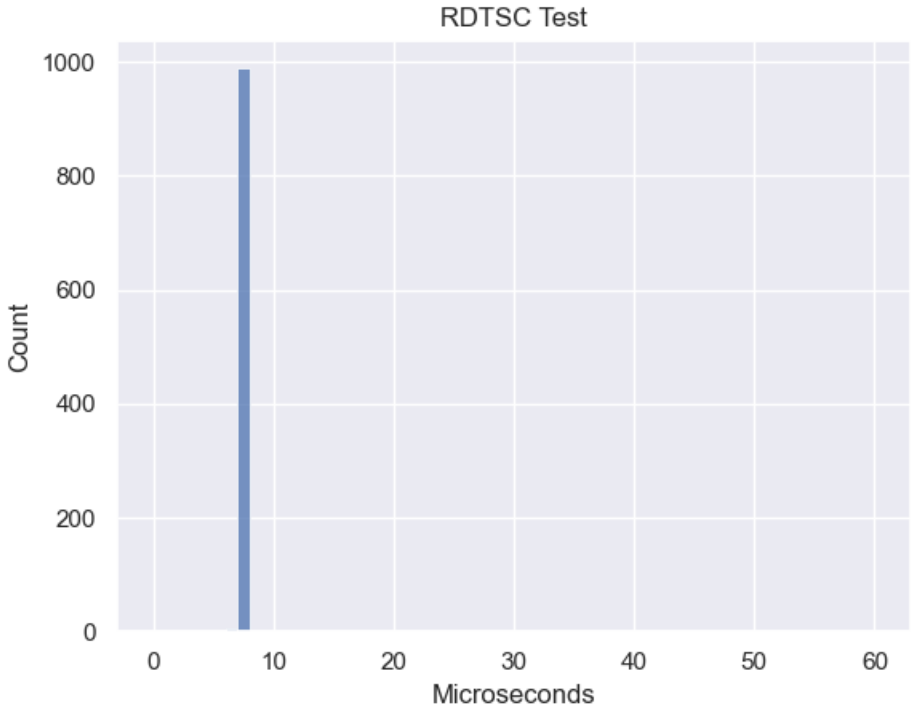

Figure 45 – RDTSC Test Distribution of Data (Linux, Intel)

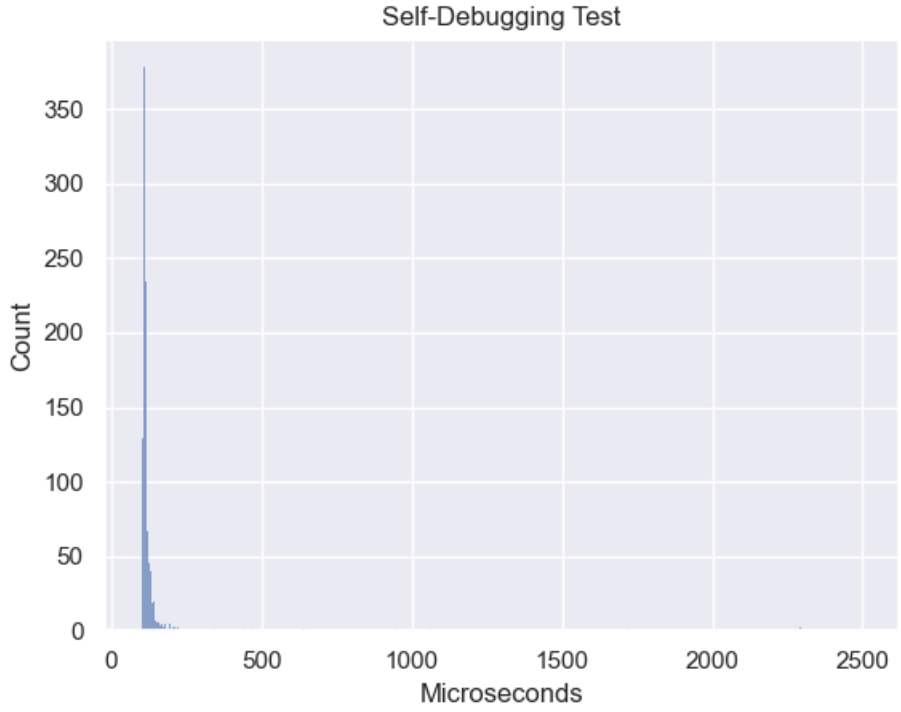

Figure 46 – Self-Debugging Test Distribution of Data (Linux, Intel)

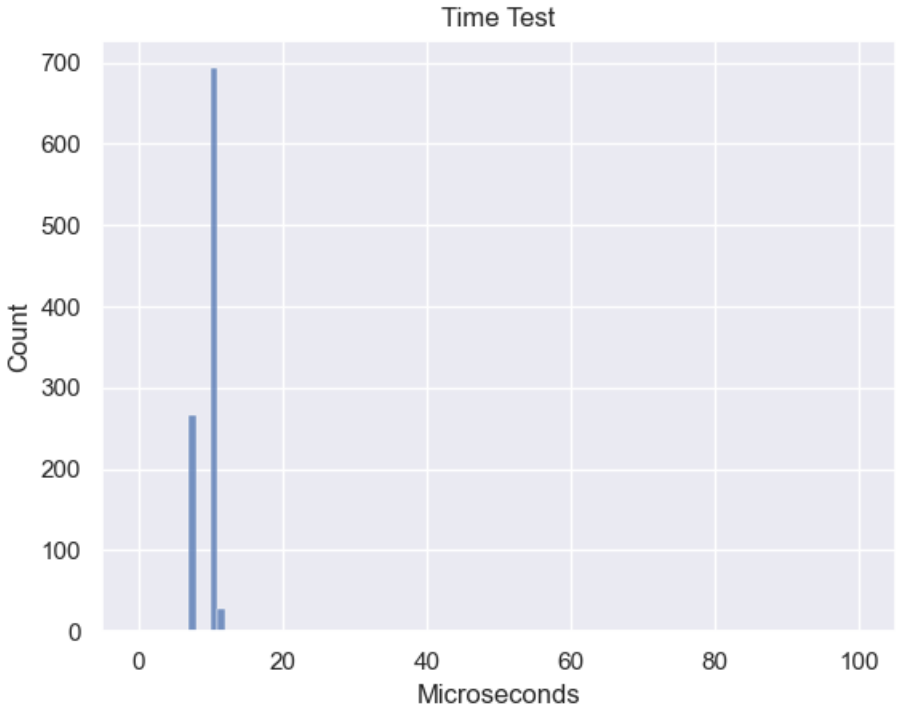

Figure 47 – Time Test Distribution of Data (Linux, Intel)

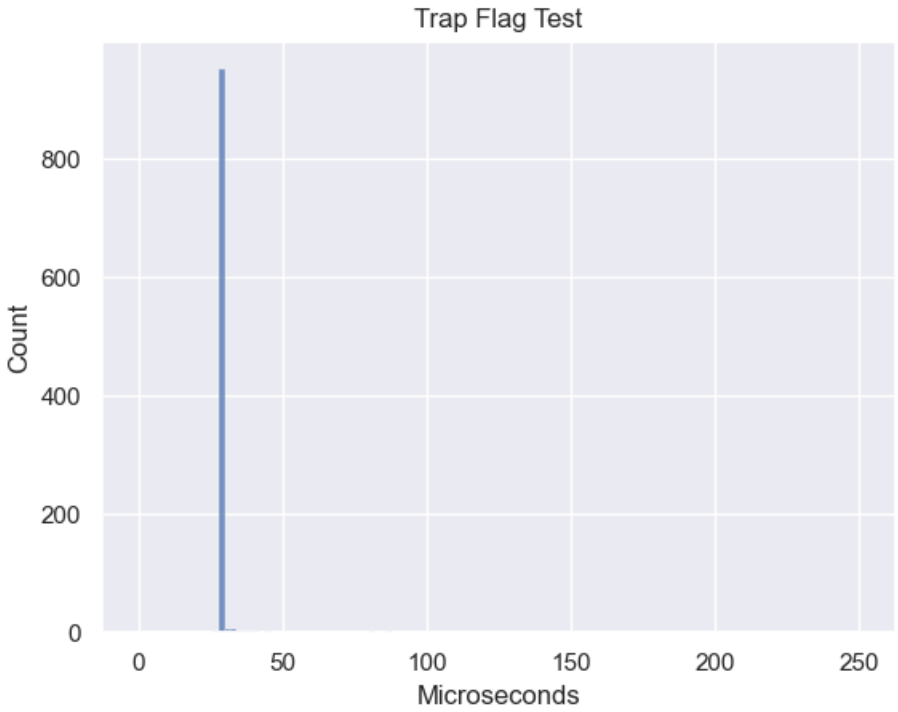

Figure 48 – Trap Flag Test Distribution of Data (Linux, Intel)

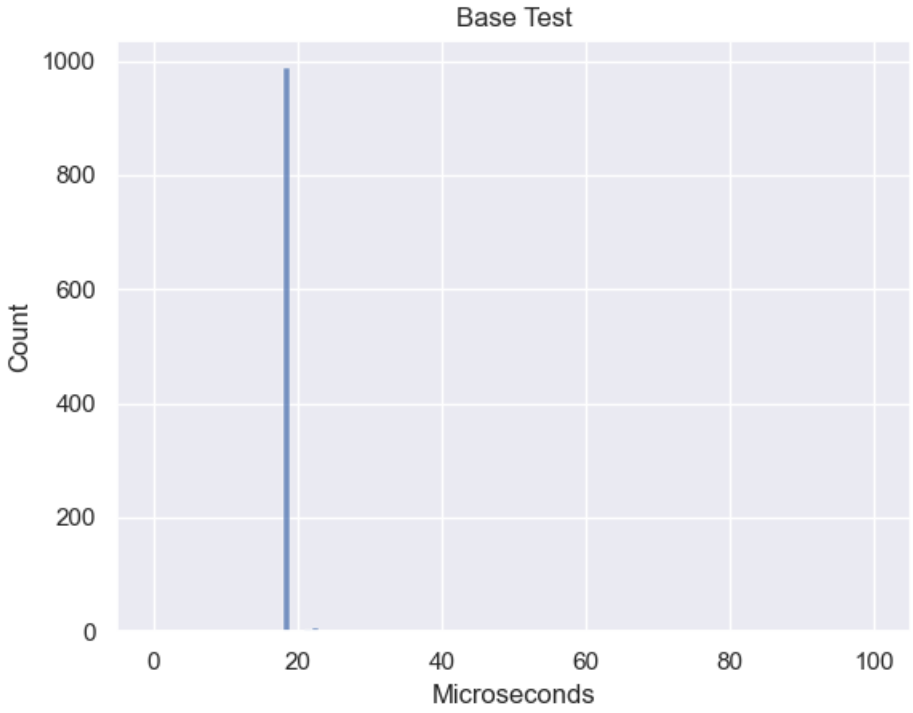

Figure 49 – Base Test Distribution of Data (Linux, ARM)

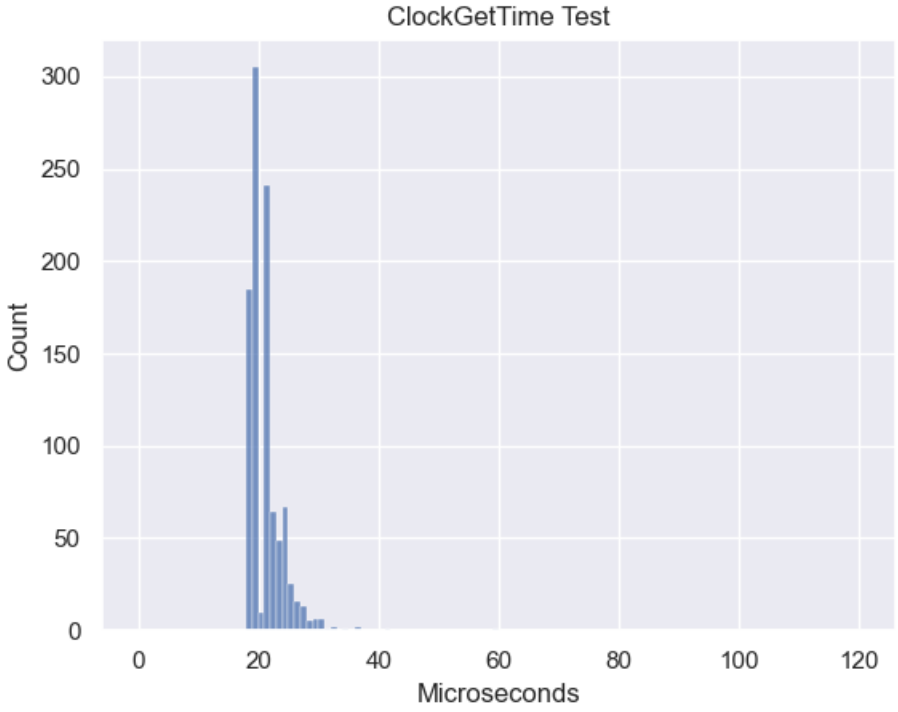

Figure 50 – ClockGetTime Test Distribution of Data (Linux, ARM)

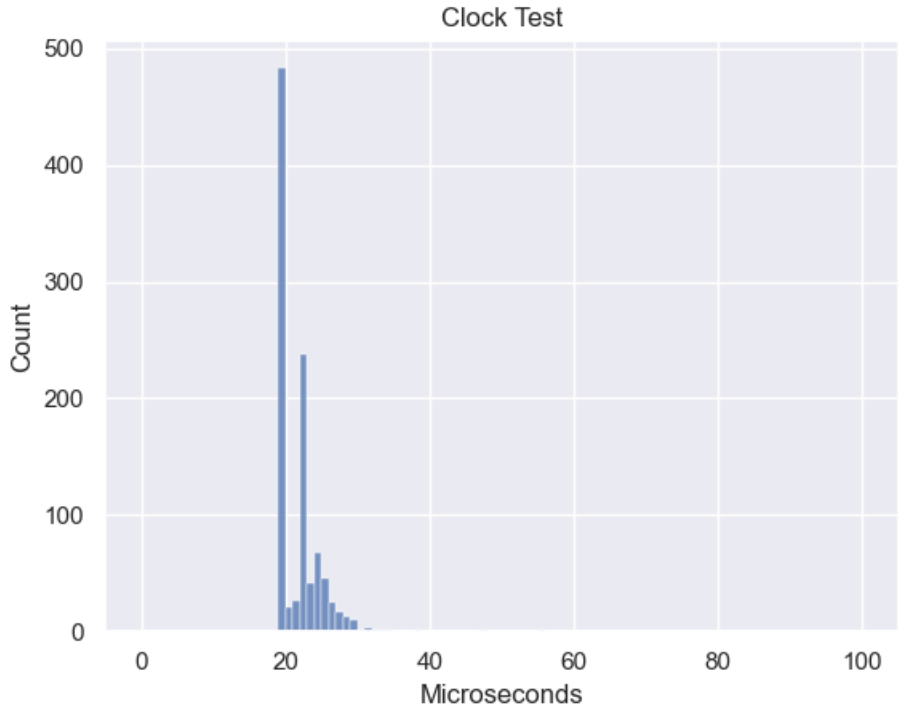

Figure 51 – Clock Test Distribution of Data (Linux, ARM)

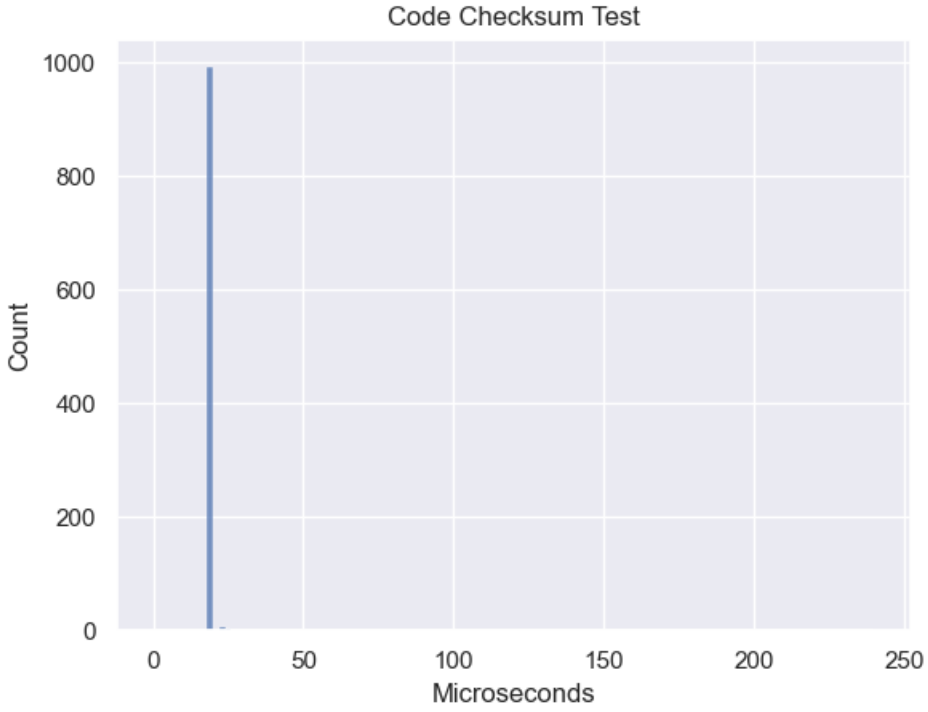

Figure 52 – Code Checksum Test Distribution of Data (Linux, ARM)

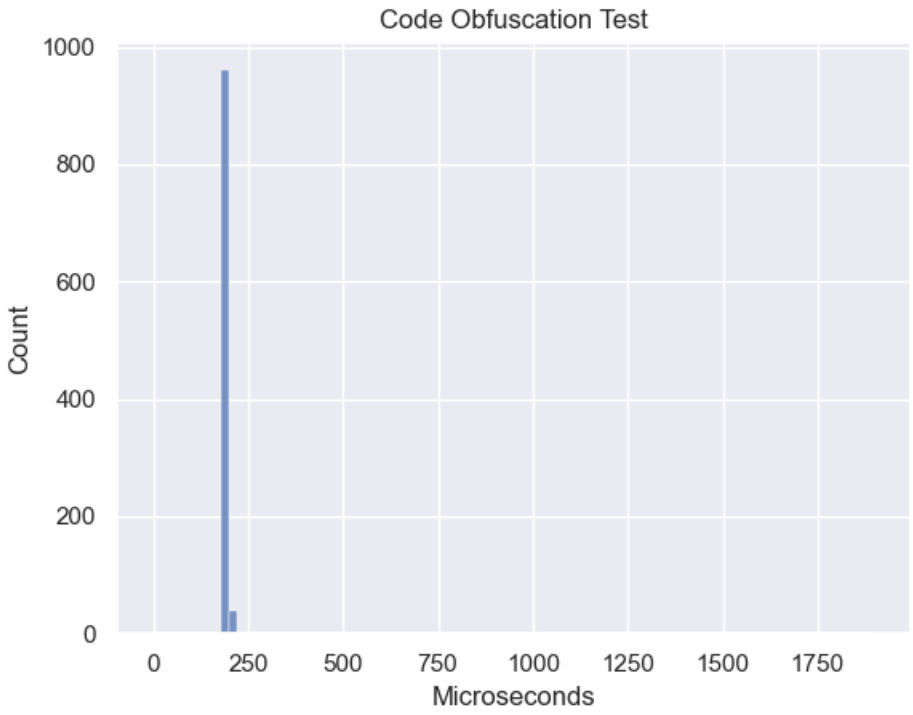

Figure 53 – Code Obfuscation Test Distribution of Data (Linux, ARM)

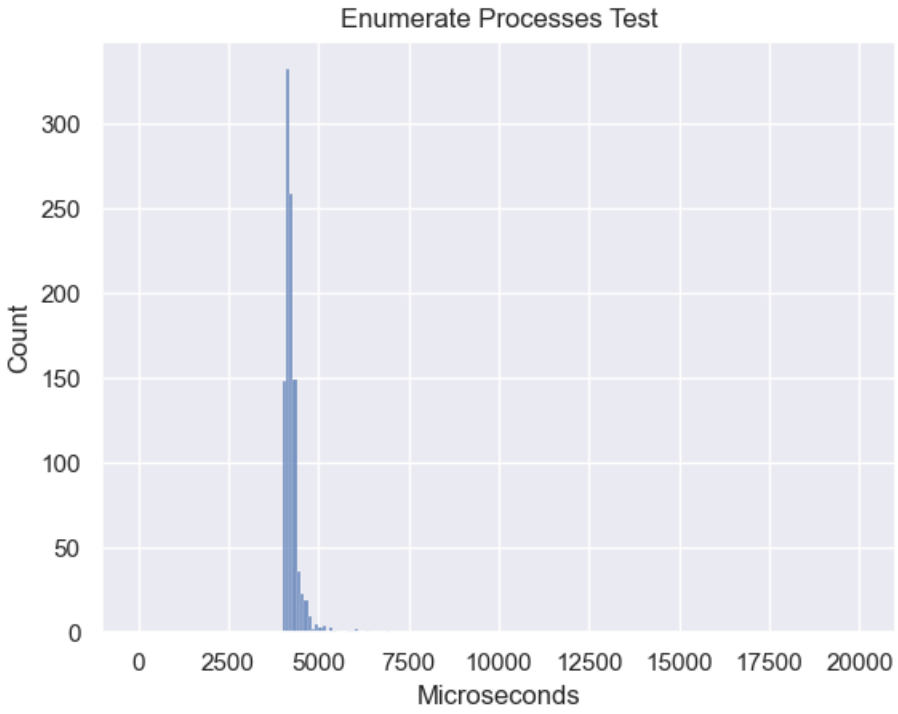

Figure 54 – Process Enumeration Test Distribution of Data (Linux, ARM)

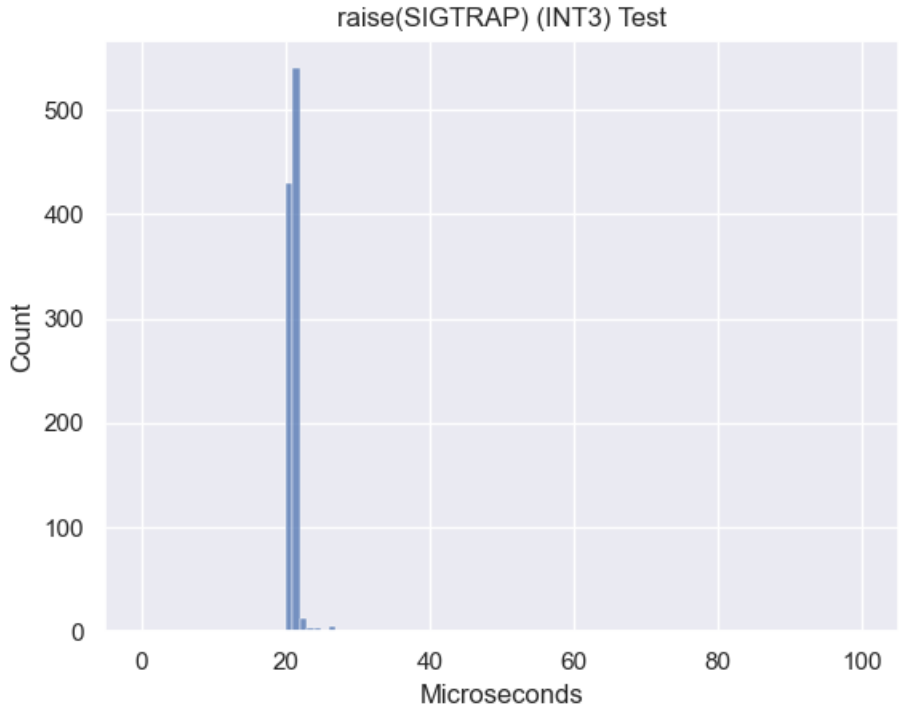

Figure 55 – raise(SIGTRAP) (INT3) Test Distribution of Data (Linux, ARM)

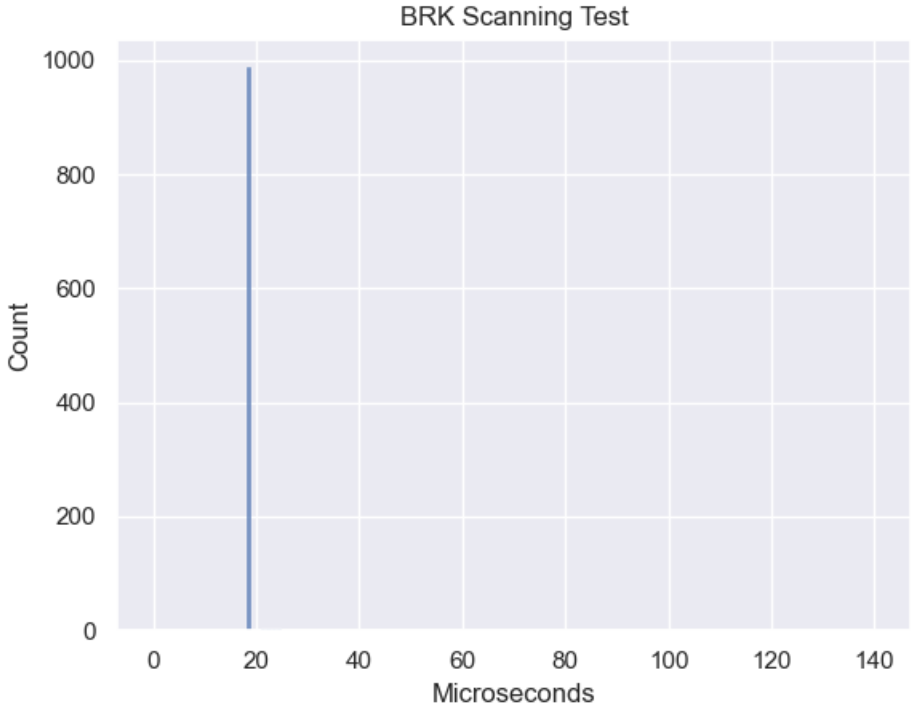

Figure 56 – BRK (INT3) Scanning Test Distribution of Data (Linux, ARM)

Figure 57 – IsTracerPidZero Test Distribution of Data (Linux, ARM)

Figure 58 – Memory Encryption Test Distribution of Data (Linux, ARM)

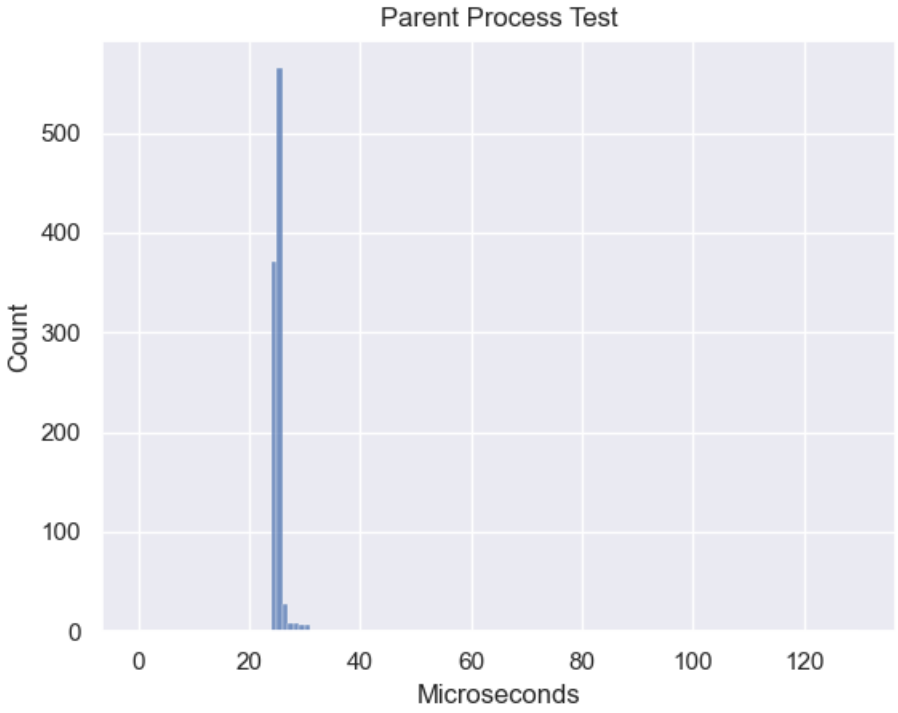

Figure 59 – Parent Process Test Distribution of Data (Linux, ARM)

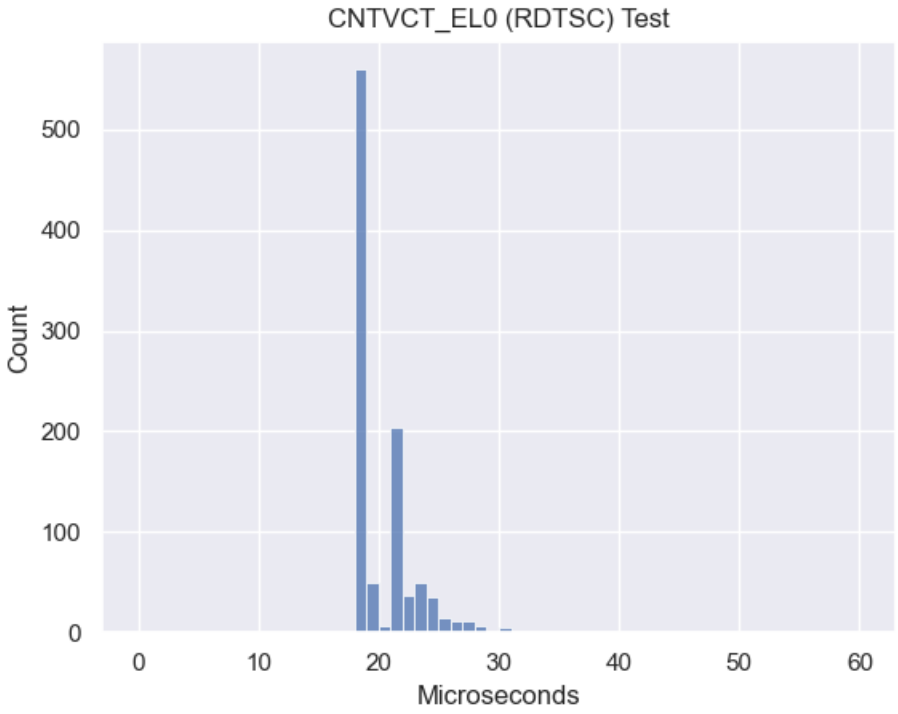

Figure 60 – CNTVCT_EL0 (RDTSC) Test Distribution of Data (Linux, ARM)

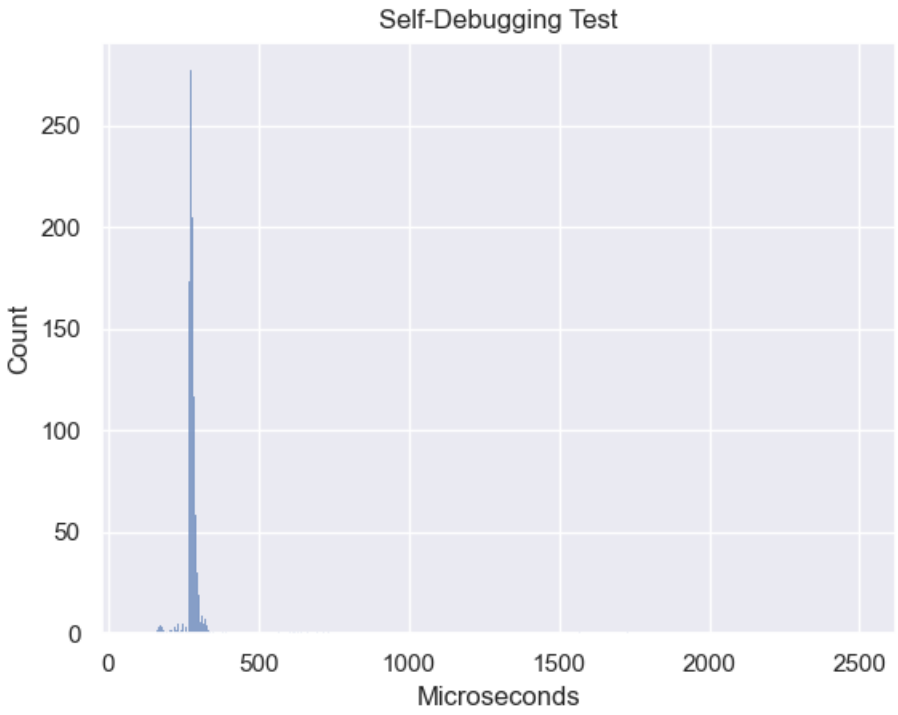

Figure 61 – Self-Debugging Test Distribution of Data (Linux, ARM)

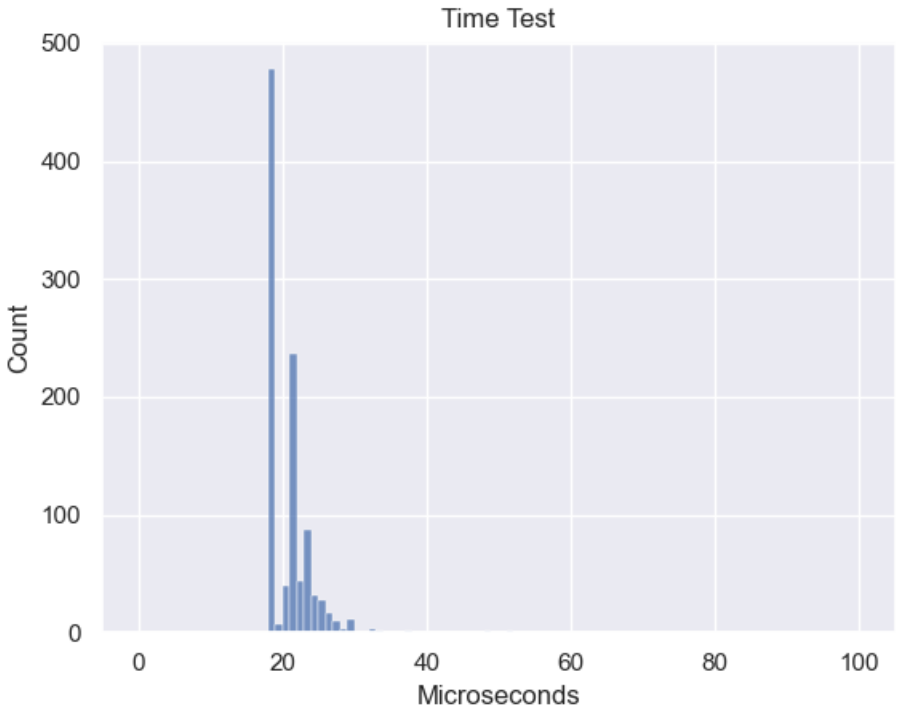

Figure 62 – Time Test Distribution of Data (Linux, ARM)

APPENDIX C – PERFORMANCE DATA (QUARTILE)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	0
75^{th}	16
Maximum (100^{th})	92336

Table 98 – Percentile Values for Base Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	9
Median (50^{th})	181
75^{th}	885
Maximum (100^{th})	14808700

Table 99 – Percentile Values for CheckRemoteDebugger Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	2
75^{th}	176.25
Maximum (100^{th})	432762

Table 100 – Percentile Values for CloseHandle Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	3

75th	21
Maximum (100th)	199612

Table 101 – Percentile Values for Code Checksum Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	0
25th	0
Median (50th)	31.5
75th	490
Maximum (100th)	30862681

Table 102 – Percentile Values for Code Obfuscation Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	0
25th	0
Median (50th)	0
75th	0
Maximum (100th)	217110

Table 103 – Percentile Values for CsrGetProcessID Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	0
25th	12
Median (50th)	60.5
75th	289.75
Maximum (100th)	216172

Table 104 – Percentile Values for Debug Registers Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	194
25th	196

Median (50th)	196
75th	704961
Maximum (100th)	237732950

Table 105 – Percentile Values for Process Enumeration Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	2
25th	2
Median (50th)	2
75th	2
Maximum (100th)	32065428

Table 106 – Percentile Values for Event Pair Handle Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	0
25th	0
Median (50th)	0
75th	0
Maximum (100th)	223427

Table 107 – Percentile Values for GetLocalTime Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	0
25th	0
Median (50th)	0
75th	69
Maximum (100th)	208014

Table 108 – Percentile Values for GetSystemTime Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	0

25th	0
Median (50th)	0
75th	94.75
Maximum (100th)	537794

Table 109 – Percentile Values for GetTickCount Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	1
25th	1
Median (50th)	889
75th	118569.5
Maximum (100th)	14172526

Table 110 – Percentile Values for Guard Page Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	0
25th	0
Median (50th)	0
75th	0
Maximum (100th)	176223

Table 111 – Percentile Values for Heap Flag Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	0
25th	0
Median (50th)	2
75th	24
Maximum (100th)	176249

Table 112 – Percentile Values for INT 0x2d Test (Windows)

Percentile	Value (Microseconds)

Minimum (0^{th})	0
25th	0
Median (50^{th})	2
75^{th}	103
Maximum (100^{th})	5605482

Table 113 – Percentile Values for INT3 Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	4
75^{th}	28
Maximum (100^{th})	176111

Table 114 – Percentile Values for INT3 Scan Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	4
75^{th}	28
Maximum (100^{th})	176111

Table 115 – Percentile Values for IsDebuggerPresent Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	220.75
Median (50^{th})	1008.5
75^{th}	3167.5
Maximum (100^{th})	48536409

Table 116 – Percentile Values for Memory Encryption Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	0
75^{th}	5
Maximum (100^{th})	176244

Table 117 – Percentile Values for NtGlobalFlag Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	3
75^{th}	88.5
Maximum (100^{th})	764263

Table 118 – Percentile Values for NtQueryInformationProcess Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	1
75^{th}	582.75
Maximum (100^{th})	4729432

Table 119 – Percentile Values for NtQueryObject Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	1
25th	122.75
Median (50^{th})	285
75^{th}	740.25
Maximum (100^{th})	535585

Table 120 – Percentile Values for NtQueryPerformanceCounter Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	0
75^{th}	0
Maximum (100^{th})	176112

Table 121 – Percentile Values for NtSetDebugFilterState Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	18
Median (50^{th})	90
75^{th}	604.25
Maximum (100^{th})	14185486

Table 122 – Percentile Values for NtSetInformationThread Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	0
75^{th}	2
Maximum (100^{th})	102290308

Table 123 – Percentile Values for OutputDebugString Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	704997
25th	1938639.75
Median (50^{th})	6214420.5
75^{th}	28250372.5
Maximum (100^{th})	338658057

Table 124 – Percentile Values for Parent Process Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	0
75^{th}	0
Maximum (100^{th})	217904

Table 125 – Percentile Values for RDTSC Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	22
25th	1062962.5
Median (50^{th})	1235103
75^{th}	1564856
Maximum (100^{th})	9993505

Table 126 – Percentile Values for Self-Debugging Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	43.5
Median (50^{th})	728
75^{th}	731
Maximum (100^{th})	172215

Table 127 – Percentile Values for timeGetTime Test (Windows)

Percentile	Value (Microseconds)
Minimum (0^{th})	0
25th	0
Median (50^{th})	0
75^{th}	0

Maximum (100th)	32772514

Table 128 – Percentile Values for Trap Flag Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	0
25th	33.75
Median (50th)	374.5
75th	4957.25
Maximum (100th)	3615535

Table 129 – Percentile Values for UnhandledExceptionFilter Test (Windows)

Percentile	Value (Microseconds)
Minimum (0th)	6
25th	7
Median (50th)	7
75th	10
Maximum (100th)	128

Table 130 – Percentile Values for Base Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	6
25th	7
Median (50th)	7
75th	7
Maximum (100th)	119

Table 131 – Percentile Values for ClockGetTime Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	8
25th	8
Median (50th)	8

75th	8
Maximum (100th)	97

Table 132 – Percentile Values for Clock Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	10
25th	11
Median (50th)	11
75th	11
Maximum (100th)	238

Table 133 – Percentile Values for Code Checksum Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	345
25th	591.75
Median (50th)	624
75th	633
Maximum (100th)	1835

Table 134 – Percentile Values for Code Obfuscation Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	6647
25th	7272.25
Median (50th)	7527.5
75th	7850.25
Maximum (100th)	44666

Table 135 – Percentile Values for Process Enumeration Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	8
25th	9

Median (50th)	16
75th	17
Maximum (100th)	89

Table 136 – Percentile Values for INT3 Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	10
25th	10
Median (50th)	10
75th	10
Maximum (100th)	138

Table 137 – Percentile Values for INT3 Scanning Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	27
25th	29
Median (50th)	30
75th	53
Maximum (100th)	452

Table 138 – Percentile Values for IsTracerPidZero Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	1341
25th	1728
Median (50th)	1749
75th	1819.25
Maximum (100th)	3382

Table 139 – Percentile Values for Memory Encryption Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	31

25th	31
Median (50th)	31
75th	31
Maximum (100th)	127

Table 140 – Percentile Values for Parent Process Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	6
25th	7
Median (50th)	7
75th	7
Maximum (100th)	56

Table 141 – Percentile Values for RDTSC Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	104
25th	111
Median (50th)	114
75th	120
Maximum (100th)	4393

Table 142 – Percentile Values for Self-Debugging Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0th)	7
25th	7
Median (50th)	10
75th	10
Maximum (100th)	97

Table 143 – Percentile Values for Time Test (Linux, Intel)

Percentile	Value (Microseconds)

Minimum (0^{th})	27
25th	28
Median (50^{th})	28
75^{th}	28
Maximum (100^{th})	228

Table 144 – Percentile Values for Trap Flag Test (Linux, Intel)

Percentile	Value (Microseconds)
Minimum (0^{th})	18
25th	18
Median (50^{th})	18
75^{th}	18
Maximum (100^{th})	40

Table 145 – Percentile Values for Base Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	18
25th	19
Median (50^{th})	20.5
75^{th}	22
Maximum (100^{th})	59

Table 146 – Percentile Values for ClockGetTime Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	19
25th	19
Median (50^{th})	20
75^{th}	22
Maximum (100^{th})	55

Table 147 – Percentile Values for Clock Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	18
25th	19
Median (50^{th})	19
75^{th}	19
Maximum (100^{th})	46

Table 148 – Percentile Values for Code Checksum Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	192
25th	194
Median (50^{th})	196
75^{th}	196
Maximum (100^{th})	220

Table 149 – Percentile Values for Code Obfuscation Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	4008
25th	4135
Median (50^{th})	4204
75^{th}	4303.25
Maximum (100^{th})	6973

Table 150 – Percentile Values for Process Enumeration Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	20
25th	20
Median (50^{th})	21
75^{th}	21
Maximum (100^{th})	60

Table 151 – Percentile Values for raise(SIGTRAP) (INT3) Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	18
25th	18
Median (50^{th})	18
75^{th}	18
Maximum (100^{th})	66

Table 152 – Percentile Values for BRK (INT3) Scan Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	51
25th	51
Median (50^{th})	52
75^{th}	52
Maximum (100^{th})	102

Table 153 – Percentile Values for IsTracerPidZero Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	2168
25th	2177
Median (50^{th})	2183
75^{th}	2189
Maximum (100^{th})	2274

Table 154 – Percentile Values for Memory Encryption Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	24
25th	24
Median (50^{th})	25
75^{th}	25
Maximum (100^{th})	95

Table 155 – Percentile Values for Parent Process Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	18
25th	18
Median (50^{th})	18
75^{th}	21
Maximum (100^{th})	41

Table 156 – Percentile Values for CNTVCT_EL0 (RDTSC) Test

Percentile	Value (Microseconds)
Minimum (0^{th})	161
25th	275
Median (50^{th})	279
75^{th}	286
Maximum (100^{th})	2548

Table 157 – Percentile Values for Self-Debugging Test (Linux, ARM)

Percentile	Value (Microseconds)
Minimum (0^{th})	18
25th	18
Median (50^{th})	20
75^{th}	21
Maximum (100^{th})	51

Table 158 – Percentile Values for Time Test (Linux, ARM)