ADVANCED CODE-REUSE ATTACKS: A NOVEL FRAMEWORK FOR JOP

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

Doctor of Philosophy

in

Cyber Operations

March 2019

By
Bramwell J. Brizendine

Dissertation Committee:

Dr. Joshua Stroschien
Dr. Jared DeMott
Dr. Yong Wang
Dr. Mark Hawkes

DISSERTATION APPROVAL FORM

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: Bramwell Brizendine

Dissertation Title: Advanced Code-Reuse Attacks: A Novel Framework for JOP

Dissertation Chair/Co-Chair: _____ Date: 3/26/19

Committee member: _____ Date: 4/19/19

Committee member: Y Wang Date: 3/26/19

Committee member: _____ Date: 3-26-19

ACKNOWLEDGMENT

I dedicate this dissertation to every student and every professor I have encountered. I would like to extend my gratitude to everyone who has helped me throughout the dissertation process. I dare not name names, lest I forget someone.

ABSTRACT

Return-oriented programming is the predominant code-reuse attack, where short gadgets or borrowed chunks of code ending in a RET instruction can be discovered in binaries. A chain of ROP gadgets placed on the stack can permit control flow to be subverted, allowing for arbitrary computation. Jump-oriented programming is a class of code-reuse attack where instead of using RET instructions, indirect jumps and indirect calls are utilized to subvert the control flow. JOP is important because can allow for important mitigations and protections against ROP to be bypassed, and some protections against JOP are imperfect. This dissertation presents a design science study that proposes and creates the Jump-oriented Programming Reversing Open Cyber Knowledge Expert Tool, the JOP ROCKET. This is a novel framework for jump-oriented programming (JOP) that can help facilitate binary analysis for exploit development and code-reuse attacks.

The process for manually developing exploits for JOP is a time-consuming and tedious process, often fraught with complications, and an exhaustive review of the literature shows there is a need for a mature, sophisticated tool to automate this process, to allow users to easily enumerate JOP gadgets for Windows x86 binaries. The JOP ROCKET fulfills this unmet need for a fully-featured tool to facilitate JOP gadget discovery. The JOP ROCKET discovers dispatcher gadgets as well as functional gadgets, and it performs classification on gadgets, according to registers used, registers affected, and operations performed. This allows researchers to utilize this tool to be very granular and specific about what gadgets they discover. Additionally, there are a variety of options available to modify how the gadgets are discovered, and this will expand or narrow the quantity of gadgets discovered. This design science research presents original

significant contributions in the form of an instantiation and five new or highly reworked and enhanced methods. Some of these methods pertain directly to JOP, while others could be adapted and utilized in other reverse engineering projects. The JOP ROCKET allows researchers to enumerate JOP gadgets for software easily, allowing for a JOP exploit to be more efficiently constructed, whereas before the task would have been a time-consuming process requiring expert knowledge and the use of multiple tools.

Keywords: binary analysis, code-reuse attacks, Jump-oriented programming, return-oriented programming, JOP, ROP, reverse engineering, software exploitation

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,

Bramwell J. Brizendine

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Buffer overflows, were not new in 1995, when Peter Zatko, or "Mudge," first made it publicly known, and then in 1996, when Aleph One made it much more widely known (Zatko, 1995; One, 1996 Nelißen, 2002). As the 1990s continued on, this memory corruption bug as well as others led to the development of numerous exploits and malware, that took advantage of software vulnerabilities such as the buffer overflow. A software vulnerability may be the result of a program behaving in a fashion unintended or not anticipated by the programmer, and some of these can be weaponized and used to perform arbitrary computation (Krusl, 1998). Arbitrary computation could be benign in nature, such as executing a trivial Windows application, like Calculator, or it could be much more malicious, allowing an attacker to perform theft, modification, or destruction to the contents of one computer or even that of an entire network (Stroschien, 2017; Engebretson, 2013). While such vulnerabilities that manifested themselves in malware would be caught by antivirus industry and added to quarantine lists when caught discovered, that did nothing to curtail the onslaught of zero-day exploits (Bilge & Dumitras, 2012). A zero-day exploit is one that makes use of a hitherto unknown vulnerability, and for which defenses, such as earlier forms of antivirus, would lack protection, since having not been known, signatures would not exist. In the hands of a well-funded and capable adversary, a zero-day exploit could be used to such extremes as causing millions of dollars in loss to businesses and organizations, to crippling national critical infrastructure (Zetter, 2014). Indeed, lives could be lost,

or wars could be decided, based on the efficacy of an undetected zero-day vulnerability that had been fully weaponized. Typically, a memory corruption bug that has been weaponized will not lead to such dramatic results; however, the possibility exists of serious ramifications from a zero-day vulnerability, underscoring the importance of strong cyber security.

The most direct way to achieve enhanced cyber security is through protections at a binary or system level, to prevent a malicious action from taking place in the first place. Inevitably, the continued development of numerous attacks utilizing memory corruption bugs led to the development of various protections and mitigations over the years (Bania, 2010). Broadly, these efforts were designed to curtail or stop such attacks. The situation evolved to what could be described as an escalating arms race, with both defenders and attackers, continuously innovating and improving techniques, alternatively to overcome protections or to thwart defenses. These mitigations include stack cookies, Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), Enhanced Mitigation Experience Toolkit (EMET), and Control Flow Guard (CFG), to name a few of the most well-known ones that are most prevalent today. Each will be explained in detail in their appropriate section.

As part of the arms race, one emerging attack methodology has been code-reuse attacks. This first took the form of return-into-libc buffer overflows. This involves using the stack overflow to call a function directly with the needed parameters (Nelißen, 2002). That technique would later be expanded to return-oriented programming (ROP) (Checkoway, et al., 2010; Roemer, et al., 2010). The central idea with ROP is that to execute borrowed chunks of executable code that exist in the virtual memory of the process image. Each borrowed chunk will terminate in a RET instruction, and this is called a ROP gadget. This returns the instruction pointer to the next instruction or address that is on the stack. Thus, a user can easily chain ROP gadgets together,

simply by having the addresses fall one after the other in the stack. Because of the very nature of how the RET instruction works, it will always return and execute whatever is next on the stack. This result is the user can manipulate control flow, directing it from one gadget to the next, and in the process being able to achieve arbitrary execution. Instead of simply being able to call certain functions with parameters, now any arbitrary computation would be feasible (Shacham, 2007). This included being able to disable or circumvent mitigations that otherwise would have protected the binary from attack.

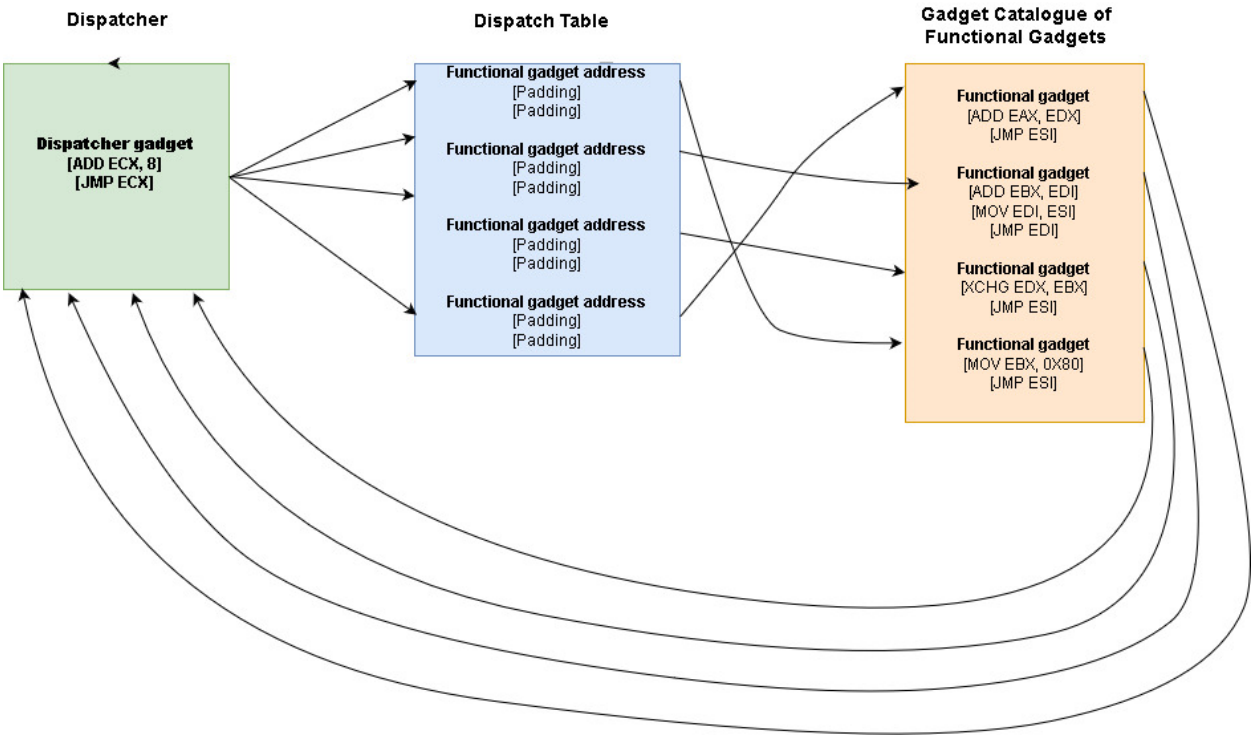

Figure 1. Diagram of JOP utilizing a JOP dispatcher gadget and dispatch table to reach functional gadgets

Code-reuse attacks have exploded in popularity since the formal introduction of return-oriented programming, though the tradition dates back much further with return-to-libc (Shacham, 2007). We have seen since then other code-reuse attacks develop, such as jump-oriented programming (JOP). JOP is a more advanced code-reuse attack that subverts control flow, allowing for arbitrary computation as with ROP, but in a very specific fashion. JOP can take on various

forms, both in a Windows environment as well as on other architectures. In a Windows environment, the most useful variation of JOP takes the form of the dispatcher gadget paradigm. JOP will be discussed exhaustively in chapter 2, but we can introduce it briefly here.

The mechanics of how JOP works is a little more labyrinthine than ROP. With JOP, instead of using the stack to subvert control flow, we instead craft a dispatch table, as shown in Figure 1. This dispatch table serves to replace the stack, as it provides the order in which the gadgets will occur. The dispatch table does this by providing functional gadgets that perform arbitrary computation, much in the same fashion as ROP gadgets. The difference here is that instead of using a RET instruction to return to the stack, the JOP functional gadget will instead terminate in an indirect jump or indirect call. This gadget will jump back to the address of the dispatcher gadget. The dispatcher gadget is a specialized gadget that is used to advance the instruction pointer forwards or backwards in the dispatch table (Bletsch, 2011). It does this by adding, subtracting, or otherwise modifying in a predictable fashion, the address of the dispatcher table. This address is to be contained in a register. For instance, if EBX contained the address of the dispatch table, the dispatcher gadget would modify EBX in a predictable fashion, such as by the following instruction: *ADD EBX, 8 / JMP EBX*. Thus, each invocation of the dispatcher gadget would modify where the dispatcher gadget begins execution; this provides the analyst with a means of subverting control flow. JOP can be useful if exploitation limited to the heap that may be achieved through corruption of the heap or UAF.

JOP is a code-reuse attack that has been written about in the academic literature for the last decade, but it has only rarely been used in the wild for exploits, to the extent that some researchers have claimed erroneously that it has never been used. JOP is the code-reuse attack that this research will focus on.

As a point of information, this dissertation at times will refer to users of code-reuse attacks using a variety of terms, most frequently as analyst or user. This work takes a neutral stance, as many people who utilize code-reuse attacks may do so for purely benign purposes, such as security researchers, or others who may have appropriate authorization. Very occasionally, the user of code-reuse attacks may be referred to as an attacker, but this too is used in a neutral sense, referring simply to an individual carrying out an attack, which may be for benign purposes. At the opposite end of the spectrum, a user of code-reuse attacks may use them with malintent; this work is not directed at such persons.

In response to code-reuse attacks, we have seen the rise and fall of various mitigations to try limit or mitigate their efficacy. Inevitably we have witnessed bypass after bypass emerge against these and other defenses, making these mitigations less and less useful, necessitating endless updates. Over the years, we have had additional mitigations evolve in response to advances in exploitation techniques, as with the release of the isolated heap mitigation being quickly followed up by the release of the deferred free mitigation, both addressing the Use-After-Free (UAF) bug (Bo Qu & Lu, 2014). A UAF a memory corruption bug that can be used to launch into a code-reuse attack

Ultimately, the question is begged, if an attack is can overcome many of these mitigations, even if it raises the level of difficulty, how useful are these protections really? It may deter amateurs, but it will not prevent more dedicated individuals. In recognizing the fallibility of these types of defenses, defensive efforts in the last few years turned more in the direction of control flow integrity (Carlini, et al., 2015). Control flow integrity in concept is a perfect solution and can provide total defense against ROP and JOP, as it enforces the control flow graph. Put simply, control flow integrity (CFI) is a theoretical defensive mechanism for try to prevent or reduce the

subverting of execution to unintended instructions. Strong CFI would make impossible all feasible control flow attacks, such as ROP or JOP, but in actual practice, a resilient CFI that is impervious to attacks and has no performance lags has been very difficult to achieve (Göktas, et al., 2014), as will be discussed in the literature. A control flow graph represents all the valid paths a program can embark upon during the path of execution (Abadi, et al., 2009).

Practical considerations have necessitated that efforts focus more on coarse-grained implementations of CFI that are software-based, such as Microsoft's Control Flow Guard (CFG) (Tang, et al., 2015). While CFG does offer some protection to certain classes of use-cases, there have been bypasses (Wojtczuk & DeMott, 2015). Software-based CFI by necessity must make sacrifices as there can be considerable performance overheads if one tries to do too much, and the end result is not complete defense, but just enhancing the difficulty. A hardware-based CFI has the potential to afford much greater protection by providing fine-grained CFI (de Clercq & Verbauwhede, 2017). Leading efforts for hardware-based CFI currently in development include Capability Hardware Enhanced RISC Instructions (CHERI), Dover Riscv, and CET. CET likely will become the more predominant form of CFI, as it is a joint effort by Intel and Microsoft and will eventually be deployed on new machines on the market.

While CET will be able to offer fine-grained defense against code-reuse attacks such as ROP and JOP, that does not mean it will be invulnerable to code-reuse attacks. Advanced code-reuse attacks such as Counterfeit Object-oriented Programming (COOP) and DOP adhere to a control flow graph perfectly and yet still effectuate an attack successfully (Schuster, et al., 2015; Hu, et al., 2016). These data-based attacks have data side effects that can help facilitate an attack that likely can overcome CET as well as other CFI solutions. CFI is not equipped to deal with these types of attacks; these attacks are outside the scope, since they follow valid execution paths. Yet

while these attacks show tremendous potential for future attacks, their use thus far has been confined to academic journals; they simply are not much done in the wild, at least not that we have seen yet.

As this work will introduce a framework for advancing one particular type of code-reuse attack, we should then return to the issue of the merits of introducing a tool that may potentially help overcome defenses. We are to be reminded though that strong offensive security results in stronger defensive security, as shortcomings, opportunities for improvement, and vulnerabilities can be found, and then they can be remediated. Performing advanced code-reuse attacks, including JOP, COOP, DOP, or others, is not a trivial matter, it can involve a significant preparation. This extra difficulty, owning to the need to manually discover gadgets or to craft one's own tools, is among some of the reasons why easier and simpler attacks such as ROP continue to be more prevalent. The tools available for ROP help provide the ease and reduces the learning curve, such that any motivated attacker could use ROP. However, there will come a time when attackers may be forced to use more advanced code-given attacks. Being able to simplify the process for achieving these advanced code-reuse attacks is certainly a great need.

This chapter will introduce the reader to the research problem and how this research will address the topic. The chapter describes the creation of a framework that helps to reduce the level of complexity and manual labor needed to undertake certain advanced code-reuse attacks. This proposed framework will serve to enhance security research by providing the necessary features and search architecture to improve the state of studies in advanced code-reuse attacks. With the realization of the proposed JOP framework, analysts can have at their disposal a means to more readily produce necessary data for their exploits, and security practitioners then can make progress towards enhancing security. Chapter 2, consisting of the literature review, will introduce more of

the background for this subject matter

Statement of the Problem

As mitigations become increasingly more advanced and more sophisticated, popular modes of code-reuse attacks such as ROP run the risk of being made irrelevant. Mitigations such as ASLR, DEP, and particularly CFG can greatly complicate matters for attackers. CFG is present on Windows 8.1 on up, and that represents over 500 million instances of it in use. Yet many users still need to use Windows 7 or lower for legacy reasons, and CFG remains absent on such systems. Windows 7 with ASLR and EMET, as well as third-party solutions, such as VTint (Zhang, 2015) would provide an effective defense against ROP, but would still come short of defending against JOP, and that is to say nothing of the myriad other third-party ROP mitigations out there, as described in the literature review. Thus, one might assert that under certain circumstances, in a well-defended, hardened Windows 7 environment, JOP could be feasible means of attack, capable of overcoming all mitigations. In actual practice, JOP is rarely used in the wild, in part owing to the difficulty of setting it up. In an ideal world, all systems would utilize the far more hardened, secure Windows 10, where CFG would make JOP impractical. However, the fact remains that as of the end of 2018, Windows 7 is still on 36.9% of all personal computers and 42.8% of all Windows PCs (Keizer, 2019), and as of only December 2018, Windows 7 was the leading operating system (Keizer, 2018). Thus, the relevance of JOP is still high because of the prevalence of Windows 7 systems.

CFG, additionally, is not impervious on Windows 10 with respect to JOP. As an approximation of control flow integrity, it has some shortcomings, due to practical considerations. In February 2018, researchers from University of Padua presented a technical paper that CFG's

defense against indirect calls or jumps can be bypassed with a Back to the Epilogue (BATE) attack. They wrote that CFG works only if a target is aligned to 16 bytes. Failing that, the "unaligned" targets can be used in a bypass, owing to the 16-byte imprecision. Their research concluded that even on Windows 10, with CFG, this allowed for there to be numerous gadgets on Windows libraries (Biondo, et al., 2018). With BATE, CFG could be bypassed, and then more common code-reuse attacks could be used. This will soon be remediated by Microsoft, but it demonstrates that future attacks that may arise may allow opportunity for JOP in a Windows 10 environment, although such possible attacks likely are to be remediated within a reasonable period of time.

We inevitably return to the prospect of JOP, a significantly more advanced form of code-reuse attacks. It is also far more limited than ROP, under the most ideal of circumstances. Although it is more challenging to use, JOP remains a valid and viable attack paradigm, one which can be powerful. Some tools, such as ROPgadget and Mona provide extremely limited functionality, but to such a limited extent as to be of only very marginal value (Salwan, n.d.; Van Eckhaute, n.d.). That is to be expected, as these are ROP tools, and they are highly effective in their problem domain; both provide an abundance of functionality and tools to help make the construction of ROP gadget chains uncomplicated and painless. That ease and functionality provided by those tools, is lacking within the province of JOP. As a far more complicated form of code-reuse attacks, the argument could be made that it more strongly needs some of that automation, to better assist security researchers.

The problem is there is a lack of tools to automate and facilitate building JOP exploits, as the current workflow makes it a manual, tedious, time-consuming process (Bletsch, 2011; Checkoway & Shacham, 2010; Quiao, 2015; Davi, 2015; Erdődi, 2013; Min, 2012). A versatile, powerful framework that can account for the complexity of JOP could be a solution to the research

problem. In endeavoring to address the research problem, this work will endeavor to answer secondary research questions that will be useful in crafting such a solution:

How can software most effectively implement in an automated fashion the discovery of JOP gadgets?

How can software most effectively implement in an automated fashion a means of discovering dispatcher gadgets?

How can software most effectively classify these gadgets into relevant categories and allow these to be displayed on demand, as a means to provide utmost utility to the security researcher?

How can potentially unusable, highly impractical gadgets most effectively be expunged from results, in an automated fashion?

How can software most effectively ensure that no potential JOP gadgets of value are missed?

All the above are questions that will help guide the DSR process of creating a framework with a high degree of usability, versatility, portability, and customization. These are subproblems that need to be responded to, in order devise a solution to the problem statement.

We can try to expand upon some of the above. First, existing tool sets lack functionality to directly discover JOP Gadgets. Both Mona and ROPgadget provide very limited functionality, but only such that it is a footnote in passing; neither is equipped for any serious undertaking with JOP. A team of researchers did develop a tool to find ROP and JOP gadgets for ELF binaries that were implemented with the NSA's proposed CFI solution (Brandon, 2016). Their work was not suited for PE files, and they lack some of robust features necessary that this framework will endeavor to address. Finally, many existing reverse engineering tools could be used to manually search for the

opcodes for the desired indirect jump or call, or to search for the disassembled instructions. Either effort would be thought of as a manual search, and neither would adequately address the matter of unintended instructions.

Manually searching for gadgets for code-reuse attacks can be a time-consuming matter without proper tools, regardless of whether it is for ROP or JOP. For instance, Roemer was attempting manual discovery of Turing-complete features. He spent three weeks doing manual analysis of Assembly in a Solaris libc file, to come up with 19 gadgets in his Turing-complete features gadget catalog for SPARC (Roemer, 2009). That is a considerable amount of time, and it addresses just one file. Roemer later developed a tool to facilitate this process, allowing for Turing features to be discovered more quickly on binaries. A manual process to exhaustively search for JOP Turing-complete features would require even more time and effort.

Second, while some general-purpose reverse engineering tools provide the facility to discover gadgets, such as by manually searching for desired opcodes or disassembly, there is not a way to accomplish this in automated fashion. One specific gadget that is essential is the dispatcher gadget; this helps allow the instruction pointer to go forwards or backwards in the dispatch table, as a means of providing order to the control flow. There is not presently a publicly accessible dispatcher gadget finder.

Next, there is a need to provide classification for JOP gadgets. ROP gadgets do not require classification, other than simply distinguishing between useful gadgets and those that are less likely to be so. ROP gadgets, unlike JOP gadgets, all end in RET, and they abound in plentiful numbers, so presenting a collection of useful ROP gadgets is often adequate for a good starting point. Some tools will group like gadgets with like gadgets, providing some informal sense of classification. With JOP, the necessities of control flow dictate that more careful classification

must be employed, as the gadgets will end in an direct jump or call to a specific register. Thus, having them ordered as such would be a significant improvement over simply providing them all, with no classification. Still, even that classification falls short; the ability to be very granular and specific, with respect to the operations performed in the gadgets, would allow an analyst to much more readily locate the needed functionality, rather than having to wade through endless possibilities or to have to perform an ad hoc search on the results. Thus, there is a need for classification based additionally on specific operation performed.

Forth, the volume of data presented to users from some gadgets is overwhelming, depending on the binary. Moreover, many of these gadgets would have no practical use, and may have just been unintended instructions. For instance, if a user were attempting to add a value into EAX and then in the next line EAX is clobbered, that gadget would be useless for that purpose. Alternatively, if a user were seeking gadgets that performed the SUB operation, they would not wish to encounter an example such as the following: SUB BYTE PTR [EAX+0x53532],0x4 / JMP EBX. That gadget is certainly valid disassembly, but if its only purpose for being presented to the user was on account of the presence of SUB, then it should be excluded, as the chances of it being of practical use for exploitation would be remote. Thus, many gadgets may border on being unusable. Sometimes these are intended instructions; other times they stem from unique combinations of opcodes, creating unusual instructions that may have marginal value to security researchers. Because some of these potentially impractical gadgets stem from certain opcode combinations, there sometimes can be a deluge of such gadgets, thereby making it more challenging to find useful gadgets.

Finally, there is a need to ensure no potential gadgets are missed. Some researchers have stated it likely is impossible with many binaries to do JOP without considering all unintended

instructions (Bletsch, et al., 2011). Depending on the technique employed or the tools in question, it could be easy to overlook some less than obvious gadgets. There are two ways to find gadgets: by looking at the opcodes and by looking at the disassembly that stems from the opcodes. Doing both of this is not adequate, as some gadgets could be missed. Thus, there is a strong need for a JOP tool that considers all possibilities of disassembly, not just those intended by the compiler, and this can only be done by iterating through all valid possibilities. With JOP gadgets being relatively scarce with many binaries, this is essential.

Purpose of the Study

The purpose of this study is to utilize design science research (DSR) to create a powerful, versatile framework, to facilitate the creation of exploits that utilize JOP. This framework will be known as the JOP ROCKET, or Jump-oriented Programming Reversing Open Cyber Knowledge Expert Tool. The motivation for the JOP ROCKET is threefold. First, there is a lack of tools that address JOP. Constructing JOP through a manual process without tools is a time-consuming, tedious, and often difficult effort. As it currently stands, if a security researcher wanted to craft a JOP exploit, they would need to create their own tool set, or make use of tools not well-suited for that purpose. The process of finding suitable JOP gadgets would be made monumentally more difficult than need be. Second, the JOP ROCKET could help discover gadgets that would be missed by utilizing general-purpose reverse engineering tools. JOP will not generate nearly as many fruitful gadgets as ROP, so there is also an urgent need to make sure no gadget is missed. In many cases, there simply are not enough gadgets for a JOP exploit otherwise, and while other work addresses how to find these, it does so imperfectly. Finally, JOP can allow us to bypass mitigations against ROP that do not defend against JOP. This is important because once successful JOP

exploits are created, they could be used to bypass many traditional mitigations that would have been unavailable due to anti-ROP defenses. Many tools, as described in the literature review, have heuristics to detect ROP, but would fail at detecting JOP.

This research is focused on ensuring the JOP ROCKET is made to be a meaningful, fully-featured artifact, to be used by a security practitioner. At present, the security researcher interested in working with JOP would have significant challenges, due to absence of tools. This work will help provide a robust solution to this problem. Additionally, although it is not designed for this purpose, the framework should also provide a response to some supplemental knowledge research questions:

> How common or uncommon on certain categories of gadgets, according to different classifications that may be applied?
>
> On average what is the breakdown of indirect jumps or calls to specific registers?

These questions will not drive the creation of the design science artifact. However, the artifact should be able to answer these open knowledge questions as a by-product, with only minimal additional work being done. These questions are of value because they can allow the security researcher to have a better understanding of some of the intrinsic properties of JOP. This knowledge could help guide some of their decisions as they contemplate attack strategies.

One of the hallmarks of design science is that it is intended to produce posterior knowledge, that which is known only after the conclusion of the research (Wieringa, 2014). This can help serve as a contribution to the discipline. It is felt that the above supplemental questions could be used to help draw out some of this posterior knowledge with respect to some of the intrinsic qualities of JOP. To be clear though, the work will not attempt a comprehensive, quantitative investigation of

JOP, and thus it will not make rigorous use of quantitative methodology. The results produced will be indications of what a more exhaustive study might confirm.

Significance of the Study

The JOP ROCKET is not likely to impact exploit development significantly. ROP is the dominant code-reuse attack for good reason, even in the face of numerous mitigations. Even with a framework such as the proposed JOP ROCKET, the fact remains JOP still will be complicated to use and set up. While JOP can be used in some situations where third-party tools make ROP impractical, many times it will be far simpler and easier to use ROP. In a Windows 10 environment, as mentioned, CFG will provide defense against JOP. However, even in a Windows 7 environment, where JOP could safely execute, the absence of appropriate tools would make the prospect of using JOP highly improbable. This would be from the standpoint of the researcher not wanting to invest significant time into a tedious manual process. Additionally, even if an analyst were interested in investing the time, they would not likely discover the unintended instructions, necessary to successfully execute JOP, unless they invested significant labor into the process. Many would-be JOP users might lack the expertise to find JOP gadgets, even with available general-purpose tools, through a manual process, and even if they were motivated to create their own tool set, they might lack the skills or the time.

Thus, this framework could make JOP more accessible to security analysts and allow for more independent research to be done, thereby giving back to the community. The JOP ROCKET may allow researchers to discover hitherto unknown vulnerabilities and thereby patch against said vulnerabilities. This would serve to further harden systems, making them safer from malicious actors. Most researchers would be able to make little tangible progress in the domain of JOP-

facilitated exploitation without significant efforts, including as mentioned, having to create a toolset, so the JOP ROCKET serves as contribution that helps eradicate such barriers.

With respect to the significance of this study, the scope of this work primarily deals with Windows 7, but it can also encompass Windows 8. CFG was released with Windows 8.1. Update KB3000850 (Kennedy & Satran, "Control Flow," 2018). Windows 8 is a distinct operating system from Windows 8.1, as the latter was a free release to those who had purchased Windows 8 at retail, as a sort of *mea culpa* for some of the public's perceived shortcomings. They marked it as an update, rather than an upgrade, to distinguish it from a service pack. In actual practice there is little difference between it and a service pack, other than some restrictions on how it is made available. It is important to note, however, Windows 8 remains unsupported, as users must install Windows 8.1 or Windows 10 to continue to receive updates. Thus, at the current time of writing, it seems unlikely that home users would continue to use an unpopular operating system such as Windows 8, when superior alternatives are available to them at no cost. Of note though and applicability to this research is that users of Windows 8 Enterprise from volume licensing must upgrade to 8.1 through an upgrade process or clean install, for which a special Windows 8.1 product key is needed. That is not the case for OEM or retail customers who can upgrade easily as if it were a service pack. Due to this requirement, many small business administrators were upset about and did not wish to undergo the tedious, time-consuming process of upgrading to Windows 8.1 (Keizer, 2013). While Windows 8 is not widely used, it is feasible there are some small businesses or organizations who obtained Windows 8 through volume licensing, but due to poor or inadequate IT staffing, may not have gone through the more complicated upgrade procedure. Such organizations that did not upgrade to Windows 8.1 and who are still using Windows 8 would then be susceptible to JOP attacks.

Although Windows 7 represents a significant share of Windows users out there, with it comprising 42.8% of all Windows OSs as of January 2019, support for Windows 7 will end January, 2020. However, upgrades will continue to be available at a cost to business users of Windows 7 Professional for an additional three years (Bacchus, 2019). This could significantly expand the viability of JOP in an enterprise environment for an additional three and a half years from the time of this writing.

As Windows 7 and versions of Windows 8 that lack CFG, they provide a significant number of machines vulnerable to JOP. Once Windows 7 is phased out and these numbers diminish, we may see the importance of JOP come into play in other operating systems. Other work could produce similar results on other operating systems. It serves to underscore the need for stronger security, not only on Windows OS, but elsewhere as well, as very strong anti-ROP defenses potentially could be overcome.

The broader implications of this study are noteworthy. In short, JOP provides a "side door," an alternate way of gaining entry, through the use of code-reuse attacks. The JOP ROCKET makes an entire class of code-reuse attacks plausible, whereas before that simply was not the case due to JOP being a difficult and tedious, manual process. The JOP ROCKET would enable attacks potentially to bypass systems that lack CFG, such as Windows 7 or Windows 8. With nearly 43% of Windows computers lacking CFG, this a significant number of machines that could otherwise be employing strong security, including EMET or other anti-ROP defenses, but that now potentially could be vulnerable to compromise. The susceptibility of the Windows 7 and Windows 8 operating systems to JOP underscores the need for strong control flow integrity (CFI).

CFG can provide control flow integrity, but various attacks and bypasses have existed allowing opportunity for exploitation, before they could be remediated. In some cases, Microsoft

even ignored attacks on CFG (Wojtczuk & DeMott, 2015). CFG is an imperfect, software-based CFI solution, and as such it will always be vulnerable to emerging bypasses, just as other mitigations have given rise to myriad bypasses. The only complete solution for CFI would be a hardware-based shadow stack, designed to integrate with a control flow integrity solution, such as with the forthcoming CET from Intel and Microsoft. CET is planned to provide support to CFG to give it additional resiliency, making it truly impervious to any bypass for JOP. It still may be years out before this happens, and that does nothing for all the existing computers that would lack CET. This research underscores the need for stronger cyber security in the form of CFI. At present, Microsoft's CFG is strong, but is it adequate? A strong argument could be made against that. Additionally, this research underscores the need for organizations and information security personnel to be cognizant of the threats posed by JOP, even if they utilize anti-ROP defenses.

This research is significant also because it will answer important design science questions. Some of these questions do not concern themselves specifically with code-reuse attacks, but could be applied to other reverse engineering tools or techniques. How this work responds to these questions will materialize in the form of a new artifact, as an instantiation of a framework, and five methods that support the framework, as will be described in chapter 3. Some of these methods could be used with other reverse engineering tools.

Nature of the Study

This research is guided by the research problem, and the response will be a DSR endeavor. This work follows in the DSR traditions of Hevner et al. and Wieringa, to allow for the iterative development of a tool that would present an original contribution to design science and to the province of JOP research and development. It will closely follow the iterative processes described

by Hevner and Wieringa, as a means to best understand the research problems and allow for those to organically result in a strong artifact. Design theories explain how an artifact is able to interact with its intended problem context (Wieringa, 2014). Due to its very nature, a design science theory for any framework can be thought of as those a way to satisfy all goals, address all stakeholders, and satisfy all requirements (2014). Thus, the proposed JOP ROCKET artifact should be able to function and meet all these needs, not only under isolated conditions, but given any conditions. This design science theory will help guide this work, and it will be elaborated upon further in chapter 4.

Objectives and Approach

The objective is to develop an approach that fully responds to the problem definition and research questions that have motivated this research, as will be covered in more detail in chapter 3. In so doing, DSR techniques will be employed that will culminate in the construction of an instantiation of an artifact, the JOP ROCKET, and its supporting methods. Briefly, we can assert that the artifact creates a practical solution to the problem of not having adequate tools to facilitate JOP. The JOP ROCKET is intended to produce output that is highly useful and relevant, and whose interface will be intuitive and require only nominal effort from the user.

Because this research meets the requirements of a Ph.D. in Cyber Operations, it was felt that it would be too simple to create a tool that merely integrates with an existing dynamic tool, such as WinDbg or Immunity Debugger. To do so would be to detract from the necessary rigor such a doctorate must demand. Thus, a static analysis approach is employed, even though a dynamic approach with one of the aforementioned tools would reduce the workload tremendously. By embarking on a static approach, many simple, reverse engineering problems that we would not

need to be concerned with, had we taken the dynamic approach, then must have appropriate solutions devised. The approach here is to endeavor to organically devise solutions to some of these various problems, rather than relying on past work done by others. Some technical matters are handled with libraries such as Pefile or Capstone, simply for convenience sake.

Assumptions

Research can aspire to be well done but often imperfections may arise, despite the best efforts to have an approach that is flawless. As part of the process of research, we must make some assumptions to proceed forward. For example, it is assumed that the binaries that are utilized for testing purposes are representative of typical 32-bit PE files and there are not anomalies present. They could, however highly improbable, be aberrations and negatively influence the design of this program, leading to an artifact that is best equipped to deal with them, but not more representative examples. It is assumed that once accuracy for disassembly, accuracy for addresses, and accuracy for offsets is assured, and after testing provides verification in support this, that this will be true for all normal PE files that are analyzed. There could be unexplained software problems that somehow prevent this, but extensive efforts at verification of results would likely detect such. It is assumed that the disassembly produced by the Capstone disassembly engine will be fully accurate for the opcodes supplied. All preliminary testing done as part of this research has not turned up any errors that could not be accounted for, and Capstone is very widely used by several hundred leading reverse engineering tools, however, so its output likely can be trusted, and testing has revealed no discrepancies (“Showcase”, n.d.). It is assumed that the Python library Pefile will be accurate and work as intended, and Pefile is a frequently used, open source library, used as part of countless reverse engineering tools (Carrera, n.d.). However, if the library were to somehow work

improperly, it could result in false disassembly, wrong opcodes, wrong addresses, wrong offsets, or other anomalies. Throughout preliminary investigations into Pefile, no problems have been discovered.

Scope and Limitations

From the initial dissertation proposal, the scope has been narrowed down to focus exclusively on JOP, one of the more widely known forms of advanced code-reuse attacks. All others required complicated, difficult setup, and they fraught with their own unique set of problems. Each would be worthy of exploration within a dissertation, but to group them together would unnecessarily inflate the scope to an untenable level. Moreover, so doing would detract from the clarity and focus needed to refine the JOP artifact iteratively, allowing it to become the best tool that it can be.

Scope has also been focused specifically on 32-bit Windows PE files. Other architectures are dissimilar, and it is impractical to generalize about platforms and architectures that differ so widely. Because the scope is limited to 32-bit Windows, one limitation is scrutiny is not given to the vast 64-bit landscape. There are countless 64-bit applications used on a daily basis, which could lend themselves to exploitation with JOP. We also lose the ability to study how the new, expanded 64-bit registers and the Microsoft x64 calling convention may affect JOP, in terms of the number of gadgets produced relative to each area of classification. How the new registers and the x64 calling convention affect the overall usefulness of JOP gadgets produced would be worthy of study as well.

Dissertation Organization

This dissertation is organized into chapters, adhering to conventions for DSR dissertations. This first chapter has provided the reader with an introduction to the research problem, while providing the reader introductory material on code-reuse attacks and other relevant topics pertinent to this study. Chapter 1 has also introduced secondary research questions and discussed the significance of this study; it introduces the theoretical framework, while delineating scope, limitations, and assumptions. In chapter 2, the literature review will provide a more in-depth background to all relevant topics concerning code-reuse attacks, including JOP, ROP, COOP, DOP, as well as the prevalent mitigations in place, including ASLR, DEP, and various forms of CFI. Chapter 3 will provide a discussion on the research methodology, while exploring some of the design choices that will be made throughout the development of the tool, as it iterates through the DSR cycle. Chapter 4 then will discuss the results of the framework, while highlighting many of the contributions that are to be provided by this novel architecture. Chapter 4 additionally will provide validation and evaluation of the tool. In the final chapter, conclusions and recommendations are made for this research, while also proposing future work.

CHAPTER 2

LITERATURE REVIEW

Chapter 2 will survey the literature surrounding this study, while providing the necessary background on code-reuse attacks as well as other relevant topics. First, chapter 2 will give insights into memory corruption bugs. It will provide a historical overview and then introduces many of the most prevalent memory vulnerabilities, giving information on stack buffer overflow, heap corruption, heap spraying, Use-After-Free, and Double-Free. While this is not an exhaustive list of possible memory corruption bugs, knowledge of these topics is necessary, these vulnerabilities and others are necessary to provide an attacker with that first opportunity to perform a code-reuse attack. The literature review will provide insights into relevant topics for the most prevalent code-reuse attacks; these include return-to-libc, return-oriented programming, Turing-complete features, and jump-oriented programming.

Chapter 2 will provide discussion the numerous mitigations that have developed to respond to memory corruption bugs as well as code-reuse attacks, such as Data Execution Prevention, Address Space Layout Randomization, and the Enhanced Mitigation Experience Toolkit. Code-reuse attacks are often used to bypass some of these protections, so knowledge of them is important. Other countermeasures specific to ROP will be discussed, such as Kbouncer, ROPecker, G-Free, and ROPGuard.

Chapter 2 will then introduce the concept of control flow integrity. Control flow integrity is important as a stronger, better solution against code-reuse attacks. Insights will be given into the NSA's control flow integrity solution, Control Flow Guard, Control-flow Enforcement

Technology, Cryptographically Enhanced Control Flow, and Lockdown.

Code-reuse attacks and the many countermeasures that have evolved do not make for light reading. To be truly fluent in code-reuse attacks, a deep understanding of these topics is necessary. This literature review will introduce the important contributions from scholars who have worked in this problem domain. The literature review can only provide a general survey of these topics; interested readers are encouraged to investigate these topics further.

Memory Corruption

When performing code-reuse attacks, one is not able to simply decide to start doing ROP or JOP at will. There must be a way to gain entry, to get one's payload somehow be executed by the CPU. This initial foothold is often accomplished via memory corruption. Having done that, an attacker could then launch different varieties of attacks, e.g. ROP, return-to-libc, JOP, shellcode, etc. This research does not examine extensively at how to accomplish these tasks, although it does provide a gentle introduction. It is a given that an attack will necessitate having some means of supplying input, often alongside some vulnerability.

Historical Perspective and Introduction

While credit is often given to this disclosure by Levy under his *nom de plume*, buffer overflows were known as early as 1972, per a presentation of the Computer Security Technology Planning Study. However, knowledge of buffer overflow remained unknown to the public, until 1995. It was at this time rediscovered and published on Bugtraq and then finally popularized by Levy, writing as Aleph One in the hacking journal *Phrack.* Levy describes a buffer overflow as follows:

On many C implementations it is possible to corrupt the execution stack by writing past

the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind (1996).

In the years that have followed, the details of the buffer overflow have become well known. At the heart of the buffer overflow, the stack is compromised, such that control flow can be overcome to facilitate the execution of unintended, often malicious code. That is, when a process is mapped to virtual memory, the stack is naturally used by programs and the instruction pointer, so that stack frames can be pushed and popped onto the stack. The stack frames may include local variables, function parameters, and more. The amount written to the buffer can exceed its intended target, and that can result in an overflow, overwriting other values on the stack (1996). It had been an effective way of placing arbitrary shellcode onto the stack, so that the instruction pointer could flow there, thereby executing the payload. Indeed, for a period of time, the stack buffer overflow as that simple, absent other mitigations.

Following Levy's disclosure, the buffer overflow gained more widespread traction, and it was widely used. Microsoft was slow to respond, but remediation efforts began in an earnest in 2004, with the first release of Data Execution Prevention as an opt-in choice for users in Windows XP SP 2. There have been many variations of buffer overflows being exploited for exploitation. Related techniques have involved heap-based buffer overflows, integer overflows, and vulnerabilities in format string. The central theme is having the instruction pointer being overtaken to allow flow of control to be placed under the direction of the attacker.

Memory corruption can affect a wide number of operating systems and applications. The two primary purposes for doing so are to modify data or to hijack control flow in order to execute code. Data that has been modified may indirectly influence control flow, leading to a desired

outcome that otherwise would not have been reached. If control flow can be overcome, it is possible then to direct execution to shellcode, although whether or not it can be successfully executed will depend upon different mitigations. Buffer overflows have also been used to leverage information leakage, where the leaked data is the end game.

Stack Buffer Overflow

At the forefront of memory corruption has been the stack buffer overflow, popularized by Levy, in the seminal *Phrack* article. While these are often thought of as the beginnings of the buffer overflow, it was first shared more than two decades earlier, although this was known only by few until its 1990's rediscovery. In a 1972 Air Force report, the buffer overflow was disclosed, as follows:

By supplying addresses outside the space allocated to the users programs, it is often possible to get the monitor to obtain unauthorized data for that user, or at the very least, generate a set of conditions in the monitor that causes a system crash (Meer, 2010, 9).

The report subsequently describes how this technique can be used to “inject” code and then “seize control” of the computer. In 1988, Cornell graduate student Robert Tappan Morris created the Morris Worm, launching it at MIT. Morris had hoped the worm would be attributed to MIT, with which he was unaffiliated, although he later become a tenured professor there. The Morris worm created a denial of service attack, and it was the first widespread use of the buffer overflow, causing economic disruption and leading to Morris’ conviction.

Later, in late 1995 Peter “Mudge” Zatko wrote an informal document on writing buffer overflows and shellcode (Zatko, 1995; Meer, 2010, 11). After the publication of Levy's work on smashing the stack the next year, activity and writings on buffer overflows proliferated in great

abundance.

With the buffer overflow, a buffer is filled with data that exceeds the buffer's boundaries, causing the program to overwrite adjacent memory (Zatko, 1995; Levy, 1996). The buffer overflow is perhaps the most well-known type of security vulnerability. Stack overflows were most relevant for lower level languages closer to Assembly, such as C or C++, where programmers deal with memory allocations and deallocations, and where there wis raw access to memory with pointers.

A buffer overflow could have many uses. For instance, the attack could place arbitrary shellcode on the stack and then execute it, barring any mitigations, if a return address can be overwritten. Often a stack pivot is performed, allowing for control flow to be subverted, so that it returns to another location on the stack. This could then contain shellcode or ROP gadgets. These ROP gadgets could help bypass a mitigation, and then direct execution to shellcode. All of this could lead to arbitrary computation, benign or malicious. Other possibilities with buffer overflow exist, such as overwriting local variables, a function pointer, or an exception handler.

Initially, defenses against buffer overflows were weak, but mitigations have been developed. These include stack cookies, such as Microsoft's Stack Guard. Buffer overflows can also be mitigated by checking the size of the data prior to writing. A programmer may do this by using more secure programming techniques and choosing functions that are not vulnerable, or if they use a vulnerable function, to do it in a safe and secure fashion. For instance, strcpy is a vulnerable function, and while strncpy is a safer alternative, a programmer could write additional error checking to be able to use strcpy in a secure fashion (Ye, et al., 2016).

Heap corruption

The heap is fundamentally different than the stack, and it is controlled by memory

management algorithms from the operating system. The heap also contains data regarding heap attributes, not limited to its relationship to other memory blocks, linked list pointers, data about its state, and other metadata such as vtables.

Heap overflow is one type of corruption that can occur, and it can function as the result of a buffer overflow. In environments with lower level programming languages, such as C or C++, the programmer must explicitly manage memory through API function calls to allocate and deallocate memory. Malloc is used to allocate memory in C, returning a pointer to an uninitialized region of memory for the allocation. Heap overflows could overwrite critical areas, such as vtables, to achieve arbitrary computation. Heap overflows that are effective and succeed are much more complicated and may involve heap coalescing and other advanced heap techniques.

Heap spraying

While not a form of memory corruption or an exploit, heap spraying has been one way by which to more easily exploit a vulnerability. In the past it was possible to use heap spraying with great effect. Heap spraying involves writing values to the heap at locations that can be found or that may be predictable. Mitigations have arisen that have made this a lot more challenging in some environments than was previously the case. Heap spraying has been used in browser exploits, involving VBScript, Flash, and Javascript. In the past, heap spraying has been used as a way to reach a specific target address, by placing NOP or similar instructions on the heap, thereby creating in effect, a NOP sled. The NOP or similar instructions would have no effect, until it reached a target address or shellcode.

Use-After-Free

A use-after-free (UAF) vulnerability happens once an area of memory has been freed and

then is used again after having been freed (Caballero, et al., 2012). Once the memory has been freed, it is possible for an attacker to allocate to that previously freed memory location, filling it with arbitrary data, such as shellcode, ROP gadgets, or changing functionality by modifying data. Use-after-free can lead to several possibilities, such as achieving arbitrary computation or modifying data. Recent mitigations such as deferred free and isolated heap in IE have eliminated short-lived UAF bugs, although they do not affect long-lived UAF bugs (Bo Qu & Lu, 2014).

Double-Free

The double free is a memory corruption in which an area of memory is freed twice at the same allocated memory, a condition that has the potential to result in a buffer overflow attack (Caballero, et al., 2012). Once memory is freed twice, the memory's data structures can become corrupt. This could lead to the program crashing, or it could lead to a condition in which the next two calls to malloc lead to the same pointer ("Doubly freeing," 2018). This means an attacker could try to create a situation in which he is given access to this pointer to memory that has been double-freed. If it occurs, it is possible the attacker could subvert control flow or

Code-Reuse Attacks

Code-reuse attacks are often used by necessity in exploits, as a means of overcoming countermeasures, which serve to harden binaries as well as operating systems, making what was once trivial, now a more labor-intensive process. Code-reuse attacks in modern Windows often address, at minimum, DEP or ASLR. This chapter will discuss in detail the relevant code-reuse attacks that have evolved as well as the countermeasures in place to combat them. These attacks have evolved in simplicity from return-to-libc, to JOP, and this chapter will provide a necessary background to understand these topics.

Return-to-libc

ROP derived from an earlier attack, return-to-libc, as a vessel through which to skirt around various OS safeguards. Return-to-libc (ret2libc) is a way of make simulated function calls with attacker data from the stack being used as function parameters. Through ret2libc and return-chaining multiple functions and parameters could be placed on the stack and then used, with the "stack unwind[ing] upward" (Dai Zovi, 2010, 18). This could be achieved by performing a buffer overflow and supplying an address for a function and appropriate parameters. Return-to-libc allowed for some arbitrary execution, but was limited in its capabilities, relative to code-reuse attacks that would arise later.

Return-Oriented Programming

Ret2libc provided the attacker with flexibility to subvert control flow, but Schacham had found ret2libc to be based upon "false assumptions" and marred by various "shortcomings," and he expanded upon it extensively, resulting in what he called return-oriented programming (Schacham, 2007; Kornau, 2010, 10). Schacham points out an important distinction between ret2libc and ROP. The former deals with entire functions calls that "perform substantial tasks," and they may involve many lines of code, whereas ROP gadgets may be just one to a few lines of code (Schacham, 2007, 5). ROP gadgets are not limited just to existing functions, but Assembly instructions such as ADD, SUB, XOR, etc., can be used to modify values in registers or memory. Each ROP gadget may perform only a small number of trivial actions that independently may mean very little, but a chain of ROP gadgets from discontinuous locations in the virtual memory collectively can result in arbitrary execution that enables a substantial task to be performed (Dai

Zovi, 2010, 28). The RET instruction that each gadget ends in will ensure that execution will return to the next location on the stack, thereby allowing for gadgets to be chained together.

When creating a ROP chain, an attacker will need to create a plan for what values to load where, what values to modify, etc., and the necessary gadgets will be found in the virtual memory. If any attacker wishes to call an arbitrary function, he will need to discover the address of such and create a ROP chain that will result in that function being executed with the correct parameters. The attacker would need to be careful to ensure that registers being used to hold a value are not inadvertently clobbered by other instructions in a ROP gadget. This could happen easily, as some gadgets may consist of multiple lines of instructions, and some registers may be used repeatedly in different instructions. Sometimes flexibility is necessary with ROP chains, and an attacker sometimes may be able to find another ROP gadget to compensate from the effects of less than desirable actions performed by a previous gadget.

Roemer writes that ROP works on the principle that there is a flawed idea that “preventing the introduction of malicious code is sufficient to prevent the introduction of malicious computation" by taking and using borrowed chunks of code from the process image or modules (Roemer, 2010, 2). A module or dynamic-link library (DLL) is a file extension in the PE file format, that allows executable code to be imported and used by executables. Whether compromised of code from module or the executable itself, ROP gadgets independently are neither benign nor malicious. However, they can be used to thwart mitigations such as DEP, ASLR, etc., which may lead to malicious computation.

The algorithm to discover ROP gadgets works by searching the process image and modules for the opcode that designates a RET, which is C3, although others are possible, such as C2, C6, CA (Kornau, 2010; Schacham, 2007). C3 is a near return, C2 imm16 is a near return with a stack

unwind, CB is a far return, and CA imm16 is a far return with a stack unwind. Once a RET is found, the algorithm can disassemble backwards, trying to discover useful instructions. A useful ROP gadget is one that does not cause flow of control to escape by not reaching the RET (Schacham, 2007). A C3 may be found in the middle of machine code for instructions that had not been intended to contain a RET. That C3 would be recognized by the ROP discovery algorithm as a RET, even though that was not what had been intended by the compiler. Disassembling backwards from that C3 could produce a useful, valid ROP gadget. This would likely include new Assembly instructions. Though unintended by the compiler, these instructions would be executed if control flow were subverted there. Schacham refers to this as the "geometry" of a language, where moving a byte can produce instructions that are valid instructions that are unintended (Schacham, 2007). This technique of finding unintended instructions, known as opcode-splitting, may vastly increasing the attack surface by producing many more ROP gadgets than would be available otherwise. Opcode-splitting frequently leads to viable results in x86, where not just words but whole sequences of words can be discovered (Schacham, 2007).

Opcode-splitting works due to the nature of Intel x86 architecture. Because the Intel x86 architecture descends from the 8-bit 8088 processor that was used in the original IBM PC, it must support many legacy features. One such legacy feature involves having memory access be unaligned, allowing for the 8-bit and 16-bit portions of a 32-bit register to be referenced. The length of instructions are also variable. This differs from an architecture like SPARC, where instructions are not variable-length, but instead are fixed-width with enforced alignment.

With x86 ISA, we also have an inherent flexibility not present in some other architectures, where we are free to do as we like with regard to calling conventions, using the stack and calling conventions without restrictions. whereas with SPARC there is much less flexibility, as various

registers are typically used for calling conventions and function arguments. With x86 we also have less registers than other architectures, which some might at first think of as a hindrance, but it is more beneficial, because it makes it easier to coordinate data flow from various registers throughout various instructions, all of which may come from different ROP gadgets (Roemer, et al., 2012).

There are several considerations to be aware when constructing ROP exploits. The size of the binary and its loaded modules will limit the number of possible gadgets, as there may not be enough useful instructions in sufficient quantity to perform desired actions. Alternatively, a specific, desired instruction may be present, but there could be actions that occur in successive lines that induce side effects to other registers, prior to the RET being reached. Those side effects could render some potential gadgets unusable. Other problems could exist, such as losing control flow, such as a conditional jump, which would make a gadget unusable.

The success of ROP is reliant directly upon the amount of code size available. Certain instructions may appear with less frequency, but after one reaches a certain threshold of code size, the probability of having a usable number of gadgets available increases greatly. In some cases, with sufficiently large amounts of instructions it may be possible to have Turing-complete sets of gadgets, though some may be partial (Roemer, et al., 2012).

ROP: Turing-complete Features

ROP has been proven to be Turing-complete in all architectures that have been encountered, as will be discussed (Checkoway, et al., 2010). Thus, theoretically ROP is capable of producing arbitrary computation in those environments purely by using only ROP gadgets, although in practice that may not always be the case, if there is a limited number of gadgets available. New defenses against code-reuse attacks could be evaluated from the standpoint of how

they may adversely impact Turing-complete features. If a defense can completely block one category of Turing-complete features, then from an evaluative standpoint we might say that the defense is at least partially effective. If a binary can produce Turing-complete features, then the argument has been that it is feasible to construct exploits allowing arbitrary computation, given a binary with sufficient gadgets. This section will explore in detail what constitutes Turing-complete features.

Turing-complete attributes can be broken down into the following categories: (1.) memory load and store operations; (2.) arithmetic gadgets, which includes addition, subtraction, negation, multiplication, and division; (3.) logical gadgets, which includes exclusive or, and, as well as or; (4.) branching, which includes both conditional and unconditional jumps as well as loops; and (5.) system calls.

Demonstrating ROP to be Turing-complete has been achieved on every architecture, thus far, absent any mitigations. Many coarse-grained CFI solutions, such as ROPecker, kBouncer, CFI for COTS binaries, EMET, and ROPGuard, etc. have claimed that they can stop ROP attacks and that Turing-completeness had been eliminated, owing to the reduced codebase. Yet in spite of these claims, Davi & Sadeghi were able to achieve a Turing-complete gadget set on all (Davi, 2014; Davi, 2015). Many of these solutions, while they do indeed raise the bar for difficulty, also allow for many more execution paths than are necessary, which a dedicated researcher can take advantage of (Davi & Sadeghi, 2014, 402). Others have done advanced work with ROP; Roemer, et al., have created a simple proof of concept compiler on the SPARC architecture. Using ROP, this compiler can implement a "dedicated exploit programming language" (Roemer, et al., 2012, 28).

Memory Load and Store Operations

Loading a constant is fairly straight forward, often accomplished via a POP, e.g. POP EDX / RET. It could also be accomplished through loading from memory, e.g. MOV EAX, EBX / RET. The lods and stos instructions are other possibilities as well. Storing to registers is straightforward as well, with instructions like MOV EAX, EDX. Writing values to a place in memory or the stack is also possible. Other less obvious ways of memory load and store operations can be implemented via ADD or SUB where an intermediate value is 0 (Homescu, et al., 2012).

Arithmetic Operations

Arithmetic operations are simple to perform. One needs to first load a value from memory and then perform the desired arithmetic operation, whether it be addition, subtraction, negation, multiplication, or division. The result must be stored to memory, if it is not already at the currently desired location. Addition can be accomplished via instructions such as ADD EDX, EAX. Subtraction can be accomplished via the SUB instruction, e.g. SUB EAX, EBX, or one could also emulate it via addition. It is important to be aware of less desirable alternatives for addition or subtraction, as ADD and SUB have same meaning as ADC and SBB, once the carry flag is cleared (Homescu, et al., 2012). This could be useful with smaller binaries and limited options.

With negation, one could use NEG EAX, to give the opposite. It is also possible to use two's complement and add one.

Multiplication can be accomplished via a similar process to addition or subtraction, although multiplication can be somewhat rare at times. In fact, Roemer et al. found no way to do multiplication in libc, though it was possible to emulate it with addition (Roemer, et al., 2012). Looping was used to facilitate multiplication or division operations when they may not been

otherwise available. Indeed, in the ARM architecture Davi used the ADDS or SUBS instructions in a loop to achieve MUL and DIV functionality (Davi, et al., 2010).

Logical Operations

Logical gadgets are necessary to achieve Turing-complete status, but not all may be as useful for ROP, with the exception of NOT. Exclusive Or is one that can be very useful, and it is included among logical operations, represented by the XOR instruction, e.g. XOR EDX, EAX. Others include AND, OR, as well as NOT. The use of NOT and AND can help achieve branching, depending on one's modus operandi. Finally, shift and rotate instructions are possible as well. Though it is interesting to note shift gadgets are not always included in Turing-complete gadget sets (Davi, 2015). Clearly though shifting can also be used to help achieve multiplication or division, although one is limited. On the whole, Davi found that logical gadgets were not quite as commonplace as were arithmetic gadgets (Davi, 2015).

Branching Operations

As a whole, branching operations are not the simplest of operations. Unconditional branching can be instructions such as XCHG REG, REG. Alternatively, a simple POP ESP; RET may sufficient to get desired branching (Roemer, et al., 2012). Davi also suggested LEAVE for unconditional branching, which will load ESP with a new address that had been previously loaded into. Davi also suggested the possibility of simply adding an offset to ESP, such as ADD ESP, 0Ch (Davi, 2015).Alternatively, one could load an address onto the stack or memory; whatever register is being used for control flow could then be moved to that location.

While establishing branching was not difficult, doing the same with conditional branching proved to be the most challenging of all the Turing-complete features to achieve using ROP (Roemer, et

al., 2012). The conditional jump instructions are not useful for ROP, and many require certain flags to be set. To achieve conditional branching, Davi required four instruction sequences: negation of a register value, subtraction with carry of that value and itself, *anding* the value, and LEAVE:

```
NEG EAX;
SBB EAX,
EAX;
AND EAX,[EBP-4];
LEAVE
```

The result is EAX would be the same as EBP-4 if EAX was zero, and if it was not, then EAX would be zero.

There are other ways to achieve the conditional jump. Conditional jumps could also be done with ADC. There would need to be two values in registers, then the ADC, add with carry, could be used to give the sum of the operand as well as the carry flag. If the two operands were zero, then the result would be either 0 or 1, which can be used to perturb a register, such as ESP (Roemer, et al., 2012). Conditional branching could be achieved by using addition or logical operations to set or clear a flag; PUSHFD could then be used to put EFLAGS onto the stack (Chen, 2011). One can extract the flag of interest from the stack through an arithmetic or logical operation. Then, one can then use neg on the register containing the extracted value. It will be 0 or -1 (Chen, 2010). One then could set the offset to perturb as either 0 or whatever the original value was, based on flag information (Chen, 2011). Various other ways can be used to achieve conditional branching. Davi found more than a dozen possible ways of doing unconditional branching in Kernel32 (Davi, 2015).

System and Function Calls

System calls and function calls are important for exploits, as they allow for different functions of the operating system to be performed, such as memory allocation, executing a file, or changing memory protections, etc. In Linux, the system call is typically done via a software interrupt, such as INT 0x80 or Syscall, for 32-bit and 64-bit Intel respectively, and the EAX register to specify the system call (Davi, 2015). In Google Android, rather than using EAX for system calls, the system call number would be in R7 (Davi, et al., 2010). On Windows, one typically uses wrapper functions, such as the Windows API and the necessary parameters, to gain access to system calls. For the attacker, system calls can be more time consuming to perform in Windows, than in Linux. It is more straightforward to use syscall, to call on any of the numerous system calls available in Linux[1]. For instance, in Linux the mprotectto function could make a page writable and executable, and this would use syscall with 125 in EAX, as well as appropriate parameters for EBX, ECX, AND EDX. Additionally, in Linux sys_execve can be used to implement a function call from libc.

Other Useful Operations

Additional operations can help facilitate the smooth functioning of Turing-complete features on the ISA-86, even though they outside their purview. For instance, in dealing with function calls, NULL is sometimes required, and so having a NULL-byte gadget would be advantageous. One illustrative function is strcpy, which stops coping data after a NULL, yet the attacker is at an impasse, as a NULL would typically break a payload. One solution is the AND instruction, which can be used to create a NULL, after strcpy had been used (Davi, 2015).

1 See Linux System Call Table at http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html for 190 system calls and required register values.

The Long-NOP gadget can be particularly useful in allowing a ROP gadget to stay viable. Some behavioral heuristics can be set off with seven or more short gadgets, and having a Long-NOP gadget is a way to do this. A long-NOP gadget could be one that is 20 or more instructions that perform actions of no consequence to the attacker. Ideally, the Long-NOP is one that makes use of few registers, helping ensure that the registers used for the attack are preserved. The Long-NOP should be selected to ensure no side effects are caused for registers used by the ROP (Davi & Sadeghi, 2014)

Beyond ROP

While return-oriented programming was cutting edge a decade ago, there are presently myriad defenses against with others in development. This has necessitated that attackers to consider other possibilities. We must think beyond ROP, into the more general concept of code-reuse attacks. Me may define these as attacks that subvert control flow by manipulating existing code to be used in an unintended fashion (Davi, 2015, 9). We may view ROP as a subset of code-reuse attacks. Code-reuse attacks need not confine itself simply to ROP or ret2libc; the possibilities are limited only by the imagination of its creators. And there have been a number of variations, some of which this research will discuss.

As mitigations increased the difficulty of performing ROP, more advanced code-reuse attacks were developed, as introduced by Checkoway & Shacham (2010, 2-3). The idea was the same: use gadgets to modify the control flow and allow for execution of gadgets to achieve arbitrary computation, chaining them together with some other element of control flow.

One alternative subset of code-reuse attacks that emerged was Jump-Oriented Programming (JOP). With JOP, instead of using RET instructions to chain together gadgets, an indirect call or jump could be used to subvert control flow. Significant, additional complexity is

also required for JOP.

Different paradigms evolved for JOP, but the most useful of which was introduce by Bletsch. This paradigm uses a dispatcher gadget in conjunction with a program counter to move along control flow (Bletsch, et al., 2011). In a sense, a dispatcher gadget happens automatically on the stack with ROP, due to the inherent nature of IA86. With JOP though, since RET is not being used, it does not naturally unwind to the next location, and thus a special dispatcher gadget and program counter are necessary to emulate the functionality of what happens with the stack. Most current ROP mitigations do not work with JOP attacks, so that is a strong reason in favor of using JOP. In actual practice, JOP is far more difficult to construct, due to absence of appropriate tools, much higher complexity in implementing, and much fewer viable gadgets, relative to ROP.

Jump-Oriented Programming

JOP may make use of many varieties of vulnerabilities, including heap over flow, buffer overflow, use-after-free are vulnerabilities (Goktas, et al., 2014). ROP has been *en vogue* for some time since its academic “discovery” in 2007, but JOP has not caught on nearly to the extent. Even as of 2015, there have been claims there have been no real world JOP attacks (Qiao, et al., 2015). That is not accurate, but they are very rare; the first real world JOP attacks were known to have started in 2010, targeting Adobe PDF. In the literature, Erdődi demonstrated in a JOP attack which added a user to Windows (Erdődi, "Attacking x86," 2013).

With JOP we can identify three distinct varieties of way of performing JOP. The first is the "Bring your own pop jump (BYOPJ)" paradigm, as introduced by Checkoway and Shacham. The second involves a method of setting up a dispatcher table, as introduced by Bletsch in 2011. Third, there is a method that involves control gadgets and “combinational gadgets,” which we consider a distinct category in its own right, although it could also be seen as a variation on Bletsch's

dispatcher gadget paradigm (Chen, et al., 2011).

There is a good reason JOP has not been used much. In the past, JOP simply was not necessary, as DEP was not rigidly enforced, and ROP defenses were in their infancy (Davi, 2015). ROP is far simpler and easier, and often inadequate defenses are in place to prevent ROP from being performed. Why would someone want to spend hours in a manual process to discover JOP gadgets, when there is a possibility there may not be sufficient JOP gadgets to mount an attack. Clearly, ROP would be the better choice. However, as chapter 1 discussed, there can be times when there are strong ROP defenses in place, but inadequate security for JOP. If countermeasures mean there is a forced absence of ROP, which is a goal of many CFI implementations, then JOP would be viable.

Intel's proposed CET is intended to make JOP impossible, although it may be years off, and it would require hardware-based support. It would involve indirect branch tracking and a shadow stack. CET would result in forced alignment to some extent, which would force the absence of opcode-splitting ("Control-flow Enforcement," 2017). This by itself would severely reduce the attack surface, such that attacks likely would not be probable, purely on that reason. Indirect branching would prevent attackers from entering in the middle of functions as well, which would require longer gadgets. This would likely induce side effects, which would clobber values that need to be preserved in some registers. These two features would have the effect of making nearly all JOP gadgets unusable.

Barring those restrictions from potential CFI solutions, JOP could present itself as a useful code-reuse attack paradigm. As example of JOP's potential value, Chen et al. created a sophisticated, kernel-level rootkit constructed out of JOP, using linux-2.6.15 to perform arbitrary computation. This enabled them to evade most methods to check kernel integrity as well as the

latest ROP detection schemes then present (Chen, et al., 2011).

Jump-Oriented Programming: Bring Your Own Pop Jump Paradigm

Checkoway and Shacham introduced a new type of code reuse attack, the first variety of JOP attacks, bring your own pop jump (BYOPJ) or ROP without using return instructions. This paradigm of JOP required a special sequence to initialize, though inconveniently it is not often found naturally in code (Davi, 2015). Checkoway & Schacham used a POP X; JMP *X sequence, with X being any register. This sequence shares attributes with RET, in that it both retrieves a value at the top of the stack and sets the EIP to said value. In addition, an attacker could use POP X; JMP *C(X), where C is a constant. Instead of placing a value of Y into X, the attack would simply then do Y-C, to take into account the offset The POP X; JMP *X is not common; in fact, the POP X; JMP *X sequence is so rare it is not found even once in libc. Thus, this paradigm is not highly realistic for practical use in the wild. However, Checkoway & Shacham speculate it could be in large enough binaries, and they point out the sequence of POP X; JMP *X is needed only once. The rest can be JMP *Y with Y serving as a pointer to the POP X; JMP *X sequence. The register holding the POP X; JMP *X address need not be static; that is to say, an attacker could be creative and shuffle it from register to another, as long as it ends up going to the register containing the address of the sequence. Other means can be used to load an address into *X, so only JMP X would be necessary (Checkoway & Shacham, 2010).

Jump-Oriented Programming: Dispatcher Gadget Paradigm

With JOP, each gadget ends with an indirect jump. However, unlike ROP, JOP gadgets are "uni-directional," and they keep going forward without an easy way to bring control back. The solution to this impasse was the dispatcher gadgets to both "dispatch and execute" the gadgets,

thereby serving to "govern control flow among various jump-oriented gadgets" (Bletsch, et al., 2011). Bletsch, et al., had found that the BYOPJ paradigm difficult and cumbersome, and thus they developed dispatcher gadgets. The logic is much the same as with the BYOPJ paradigm, but instead of a rare sequence of POP X / JMP X, the dispatcher gadget is utilized instead. The dispatcher gadget a will subvert control flow to "an internal dispatch table that explicitly specifies the control flow of the functional gadgets" (Bletsch, et al., 2011).

The program counter is used in conjunction with the dispatcher gadget; it may be used to point to the dispatcher table with be any register. With every invocation, the dispatch table advances to the next functional JOP gadget and launches it. Given that the stack is not necessary for JOP, the dispatcher can use potentially any memory, even that which is not contiguous, to begin the dispatch table, absent any consideration for possible mitigations in place (Bletsch, et al., 2011). With the JOP dispatcher, one register can hold the address to the dispatch table. Instead of the stack, we have something that is predictable, such as adding or subtracting. The hacker have appropriate gadgets in memory. If a constant such as 4 is used to advance the program counter, then the table can be organized as an array, while if a memory deference occurs, it can be done more in the manner of a linked list (Bletsch, et al., 2011).

The functional gadgets in JOP are designed to go to code pointers that perform the desired instructions, or what would be considered the just simply gadgets under ROP, only now they end with an indirect jump (Bletsch, et al., 2011). The functional gadgets could be operations such as Turing-complete features, loading system calls, etc., or any arbitrary computation. The requirements of the functional gadgets are twofold. First, they must jump back to the dispatcher table after executing the function. Second, they cannot have side effects that are destructive to values within the dispatcher gadget (Erdődi, 2013, "Finding Dispatcher Gadgets"). It may be

possible to have a functional gadget that could survive side effects, if needed registers were moved to onother registers or places in memory and could then be restored. Such measures may be necessary at times, given the paucity of functional gadgets available.

Careful thought is required for selection of dispatcher gadgets. The dispatcher gadgets should be as short as possible, because each register that is modified in other lines may not be able to be used much, as that register would be clobbered endlessly. The more registers that are clobbered, the less viable the dispatcher gadget candidate is. The only exception would be an intermediate gadget to somehow restore the gadgets. A less commonly used register would be more well suited as a dispatcher gadget, to state the obvious. Erdődi points out there are many possible ways to change the pointer of dispatcher gadget, such as incrementing or decrementing a register, adding or subtracting a value from a register, moving a constant value and a register to a register, or moving a dereferenced register to a register (Erdődi, 2013). It is considered suboptimal to have a conditional jump within the dispatcher gadget, even though it is feasible the EFLAGS value would evaluate to false. Whilst it is certainly unfavorable, it is still potentially viable, though it adds greater complexity both to the gadget searching and the JOP construction.

Jump-Oriented Programming: BLX Attack on ARM

JOP need not confine itself purely to one architecture. JOP is more limited on ARM. In his dissertation and other work, Davi establishes one way to perform JOP on ARM using a BLX attack. Davi's work with ARM included developing a Turing-complete gadget set for a BLX attack. Davi's gadget catalogue included the following categories: memory operations (e.g. load and store), data processing (e.g. arithmetic and logical operations), control flow (e.g. branching), and system and functional calls (Davi, 2015).

The BLX jump-oriented attack centered around the indirect call instruction BLX, or

Branch-Link-Exchange. Only the BLX will allow indirect function calls, accessing addresses of a branch that may be held in a register, hence this instruction being pivotal to the attack (Davi, et al., 2010). Davi faced hurdles because ARM has memory alignment, thereby significantly reducing potential gadgets due to lack of opcode splicing (Davi, et al., 2010). Developing a BLX-based JOP attack on ARM may be somewhat similar to developing an attack for the NSA CFI scheme, due to the rigid memory alignment, thereby prohibiting opcode splicing and thus creating a paucity of instructions. Davi was able to use a heap-overflow vulnerability in conjunction with JOP to attack an Android app and thereby send SMS message (Davi, 2015).

Solutions and Countermeasures

As buffer overflow attacks and later ROP became more prevalent, it was necessary to employ different countermeasures. Two of the earliest and most well-known were DEP and ASLR, although bypasses soon developed against both. Other detection and mitigation strategies employ different techniques, tools, and heuristics in order to harden binaries against ROP. Some ROP detection programs can identify and block ROP. With heuristics, it is possible to create a signature of different attributes, such as what is referred to as length, or the number of memory address that appear on the stack, one after the other. ROP attacks often are comprised of long sequences of short gadgets. Thus, length can help find ROP as that attribute would not occur otherwise (Carlini, et al., 2014).T he behavior of the binary can also be useful with heuristics in detecting ROP. Other strategies such as control flow integrity can provide some degree of protection against code-reuse attacks, although this varies greatly among implementations.

DEP

Microsoft attempted to provide mitigation against shellcode attacks with Data Execution

Prevention (DEP). This was in response to vulnerabilities that had arisen out of buffer overflows. This dealt with memory pages, and it prevented code that was readable and writeable from also being executable ("Data Execution Prevention," 2009). With DEP in place, attempting to execute memory protected by DEP will result in an access violation and program termination. With the /NX used on a compiler, DEP could be utilized on binaries compiled with Visual Studio. As a mitigation, DEP's utility is limited, as various ways have been devised to bypass it, most commonly involving ROP and function calls to VirtualAlloc or VirtualProtect. Once DEP has been bypassed, then it is possible to have a region of memory that is both readable and writable as well as executable, meaning an attacker could write and execute shellcode at the same location.

ASLR

Address Space Layout Randomization (ASLR) was designed to provide a challenge against ROP. ASLR randomizes the regions of memory, including the heap, parts of the stack, as well as module base addresses. ASLR makes it unknowable where certain items will be in memory. Prior to ASLR, these memory addresses would typically be predictable, based on different factors, such as operating system, software version number, etc. ASLR was first deployed on Windows Vista on, but it has evolved and subsequently been further hardened. There have been a couple problems associated with ASLR. Memory disclosures can allow for it to be overcome; once a memory disclosure is found, offsets can then be calculated to figure out addresses of desired functions or gadgets, thereby negating the effect of ASLR. The second problem was low entropy in terms of randomization for ASLR. This meant memory addresses were not predictable, but they could be brute-forced. However, this has since been remediated by introducing high entropy ASLR with x64 Windows 8 and up.

EMET

Microsoft's EMET is a free security toolkit from Microsoft. For several years it was a cutting-edge security tool, providing features to attempt to mitigate or stop different exploitation strategies, including ROP. It contains several mitigations to provide protections, such as Caller check, SimExecFlow check, LoadLibrary check, EAF check, MemProt check, and Stack Pivot check. While these and others checks provide strong protection against ROP, they do not stop JOP. EMET could best be viewed as a hodgepodge of various tricks and tools to try to limit the efficacy of code-reuse attacks and other exploitation techniques. EMET provided strong defenses for several years, but it soon became subject to a number of bypasses, most notably by DeMott (2015). DeMott found several shortcomings in EMET that allowed a total bypass. As others created bypasses, Microsoft continued to update the tool and mitigate against those bypasses. However, it became clear that EMET would continuously be an arms race with attackers, with the different protections that were to evolve serving as merely the next hacker challenge.

EMET is not a tool that enforces CFI. Control flow integrity was viewed by some as the ultimate solution to ROP. After all, a perfect control flow graph would not allow the unintended ROP gadgets to be reached in the first place. In actual practice, achieving this is fraught with difficulty, some of which is detailed later in the literature review. Given EMET's inherent limitations, efforts were made to replace EMET with a more resilient tool that would be impervious to bypasses. Microsoft worked to develop their own instantiation of control flow integrity, resulting in Control Flow Guard (CFG).

Additionally, Windows 10 also provides similar functionality built into the operating system, in the form of Process Mitigation Management Tool and Windows Defender Exploit Guard (Bright, 2017; "ProcessMitigations," 2018; "Windows Defender," 2018).

While EMET has since been deprecated and is no longer supported as of July 2018, it still may be used by those utilizing older operating systems that lack CFG and other defenses.

Control Flow Integrity Background

Control flow integrity (CFI) could be used as a final solution to code-reuse attacks, given the perfect yet at present unobtainable implementation. CFI refers to how an operating system may implement the natural control flow graph for a program for which it was intended.

The principle behind CFI is to determine an application's control flow graph before program execution, allowing the control flow graph to be used to require that control flow adheres to paths defined in the control flow graph. CFI can be either looser or stricter, i.e. fine-grained or coarse-grained. Failure to follow said paths as set forth in the control flow graph generally results in application termination. What makes up those paths is defined by different CFI solutions, with varying levels of granularity (Davi & Sadeghi, 2014). Precision is important in a control flow graph. In a fully-precise static control flow graph, an indirect control flow is permitted only when there is a legitimate trace that follows the edge, avoids malicious attempts at control hijacking, and does not limit functionality. Often real-world implementations rely upon static analysis to create the control flow graph, and the result often is an overly loose, coarse-grained control flow graph (Carlini, et al., 2015). A fine-grained defense is a closer attempt to fully-precise control flow graph, but it can be overly restrictive, meaning it may block paths that are legitimate.

According to Davi & Sadeghi, CFI can be broken into three categories of policies. The first deals with rules for indirect branching. Second, there may be heuristics that attempt to capture attributes of ROP behavior that may try to undermine CFI. The final category is when the inspection occurs (Davi & Sadeghi, 2014, 404). The strength of CFI is in how precise the control flow graph is, i.e. the hard limit for precision, as well as how precise run-time checks may be. If it

is inadequately precise, then unintended control flow may occur (Payer, et al., 2015).

In the control flow graph, there are basic blocks (BBLs) represented as nodes. A BBL is a sequence of Assembly instructions and edges, which function to connect two nodes. Control flow may occur directly or indirectly, as with return, jump, and call. To facilitate this, CFI often inserts labels at the start of BBLs. CFI validates control flow transfers that occur at runtime, to ensure indirect branches go to BBLs that have the appropriate label (Davi & Sadeghi, 2014). CFI that is looser, in that it allows for fewer labels, may improve performance, but it results in a control flow graph that is less precise. On the other hand, as far as practical implementations are concerned, CFI that is based on a strict control flow graph could limit legitimate returns, so there is the dichotomy of being too strict and causing potential exceptions with greater overhead, and then being less strict with lower overhead but leaving more opportunity for exploitation (Goktas, et al., 2014).

When discussing CFI, it is important to be aware of the different protections that can be offered to different types of attacks. Forward-edge CFI protects against indirect JMP or CALL sites, while backward-edge CFI protects against RET instructions (Spisak, 2017).

CFI needs to have a shadow stack in order to have a strong defense (Carlini, et al., 2015). The shadow stack is merely one way to help implement some of the policies of CFI, though it is one of the strongest ways of doing so. There can be a sharp cost though, as some CFI implementations with shadow stacks can have performance overheads, with an average as high as 21% (Davi & Sadeghi, 2014). This clearly means that CFI without dedicated hardware is not quite ready for widespread deployment. Dedicated hardware is not available, although Intel is working on the forthcoming CET.

One of the main efforts at restricting returns is performed via a shadow stack. The idea is

that only legitimate return addresses, i.e. those placed on a shadow stack, can be returned to without triggering an application termination. This helps prevent the arbitrary arrangement of code pointers ending in ret from achieving a ROP payload. CFI with a shadow stack will typically limit an attacker to just system calls that are available within the application (Carlini, et al., 2015). There have been several drawbacks to previous efforts at shadow stacks. One is incompleteness, in that they protect against code from compilers, but not so from hand-written assembly, potentially leaving some third party libraries unprotected. There are also incompatibility issues, where complex program may have returns that do not match calls, leading to false positives. These are serious issues that need to be addressed, before any implementation should have widespread deployment (Qiao, et al., 2015).

Control Flow Guard and Return Flow Guard

Control Flow Guard (CFG) is Microsoft's attempt to prevent control flow to unintended locations. It accomplishes this chiefly by storing valid addresses in a bitmap and performing a check before every indirect call, in an effort to ensure the target address is valid. CFG offers defense against ROP, though it is very coarse-grained. CFG does provide forward-edge CFI, so it offers protection against indirect CALL or JMP sites. Microsoft’s Return Flow Guard (RFG) provides complimentary support for backward-edge CFI, by providing a software-based shadow stack. With RFG, protection is not assured for return addresses, as it is possible to call valid functions out of context. It is also possible to corrupt return addresses on the stack. CFG willl integrate with the proposed CET hardware enhancement from Intel, further hardening the CFI protection offered. Many individuals have bypassed CFG and RFG (Wojtczuk & DeMott, 2015; Spisak, 2017), and it is anticipated bypasses will continue to emerge.

NSA's CFI Implementation

The NSA developed a proposal for CFI that would do very well in mitigating both ROP and JOP attacks. Their proposed implementation consists of a shadow stack, which is implemented at the hardware level (NSA, 2015). The hardware for the shadow stack has not been prototyped, and they left some questions as to how it would work up to the industry. For it to be adopted would require considerable industry buy-in of the idea. It seems that Intel has taken notice, and they have worked to develop a solution that offers similar functionality, CET. The second key part of the NSA proposal was the addition of three landing point instructions to Assembly. These are for the RET instruction and movement to CALL or JMP sites. That is, the compiler would produce these landing point instructions at the start of the function. If a code-reuse attack attempted to enter a function at a location other than the entry point, then there would be an exception, causing the processor to abort the program. This would prevent opcode-splitting from being used additionally. Currently, the opcodes for these landing point instructions would function as NOPs. The NSA's proposed CFI scheme could function with just the landing point instructions, or it could function with the landing point instructions and the hardware shadow stack. The work of several researchers indicates it is feasible though very unlikely that the landing point instructions could be defeated in any meaningful way, in the absence of the shadow stack, though some Turing-complete features can be performed. The chances of doing an actual attack with landing point instructions present is highly unlikely, and if it were possible to do so, it would take considerable effort and likely would be very limited. With the presence of the proposed shadow stick, the chances of this CFI scheme being compromised in any meaningful way are highly remote.

CET

Intel and Microsoft have proposed a new technology to deal with CFI. Control-flow

Enforcement Technology (CET) is a joint effort to rigidly enforce CFI (Intel, 2017). CET is comprised of two elements, a hardware-based shadow stack and Indirect Branch Tracking. The CET shadow stack is used only for control flow operations, such as control flow addresses and shadow_stack_load and shadow_stack_store; no other values are kept there (Intel, 2017). It is held in system RAM and protected by the CPU memory management unit (Williams, 2016). With Indirect Branch Tracking, we have a new instruction, Endbranch. This is used to mark and keep track of control flow targets that are valid. As with the NSA proposal, the opcode is otherwise a NOP for legacy systems. Indirect Branch Tracking uses a sate machine, to keep track of calls and jumps. If there is a JMP or a CALL, the state will go from IDLE to WAIT_FOR_ENDBRANCH. If the endbranch is not forthcoming, there will be a fault. Thus, the endbranch instruction is able to help find control flow violations. For CET to function, the processor needs to offer support for it. CET is currently in preview. However, GCC and Microsoft Visual Studio both have begun compiler support for CET. It is unknown when Intel will roll out CET.

CET is similar in some respects to the efforts from the NSA Information Assurance Directorate. Both involve a shadow stack for to implement backward-edge CFI. NSA's CFI scheme is more fine-grained, however, with different landing point instructions. CET was influenced by the NSA proposal, but it has taken a different approach. Both require hardware support, and as such there are not implementations available as yet to researchers, though the NSA made compiled ELF binaries publicly available to researchers on their GitHub.

Other ROP Solutions

While EMET, CFG, DEP, and ASLR all provide strong yet imperfect defense against ROP, there are various other solutions that have been developed. A sampling of some other solutions to

ROP are provided here. Some of these mitigations, while providing lower overhead and good security, can be compromised (Carlini & Wagner, 20014).

Kbouncer

Kbouncer can utilize hardware to examine a variety of indirect branches, to try to determine if ROP is being attempted. If a system call is made, kBouncer can examine the history to verify the call was valid and not part of a ROP gadget. Whenever a system call is made kBouncer, it will utilize Last Branch Record to examine a history of 16 of the last indirect jumps, making sure that they all return to only an address that was call-preceded. If kBouncer can detect that 8 of the last memory addresses utilized are gadget-like, then it will kill the process. One behavioral heuristic is to specifically check that each RET is preceded by a CALL instruction (Carlini, et al., 2014). Another common heuristic is examining the histories of gadgets. If enough of what appears to be ROP gadgets are used in a set period of intervals, then it could be flagged as ROP. While this is a valid signature pattern, an attacker could still overcome it. For instances, the histories kBouncer maintains only go so far before they are overwritten, and an attacker could exploit this. A history flushing attack could be performed by making a 16 safe indirect jumps sufficient to wipe out kBouncer's history (Carlini, et al., 2014).

ROPecker

ROPecker is an extension available to kBouncer. ROPecker can look at different places for an attack throughout the execution of a program (Carlini, et al., 2014). ROPecker performs its inspections more frequently and in a more meticulous fashion. ROPecker additionally allows for only a small number of pages to be executable. Thus, if there is an attempt to execute a page outside this range, a page fault will occur. ROPecker will then commence searching for an attack. Like

kBouncer, ROPecker also performs efforts at detection whenever a system call is made. If a page fault occurs, not only will ROPecker look at the past history via the Last Branch Record, but it will also look forward at what it is about to happen. With ROPecker, if 11 or more of what could be considered gadgets that are less than six instructions in length occur back to back, then it will terminate the process (Carlini, et al, 2014).

As with most ROP defenses, ROPecker is not impenetrable. One way is to use an evasion attack and make use of a termination gadget after 10 or less gadgets; this would ensure that the number of gadgets needed to trigger the process being terminated (11) is not reached. An attacker would need to be careful choosing a termination gadget, to ensure that any registers that need to be preserved are not clobbered.

G-Free

G-Free is another tool that can help combat ROP (Onarlioglu, et al., 2010).
G-Free achieves its defensive capabilities through recompilation. How it accomplishes this is by taking away unintended ret instructions and by also encrypting return addresses. It enhances this security by going further and encrypting the return address with a random nonce at the function entry point and then ensuring it is there at the function exit (Goktas, et al., 2014; Qiao, et al., 2015). If recompilation is occurring, then the result will be that ROP gadgets are then made to be challenging to use. Another of the ways in which G-Free can provide defense is through enforcing alignment, which is critical in reducing the attack surface (Goktas, et al., 2014). One of the key ways in which ROP is able to increase the attack surface is through opcode splicing, as IA-86 does not enforce alignment; G-Free would mitigate this.

ROPGuard

ROPGuard is a tool to detect and prevent ROP, and it won second prize for Microsoft's BlueHat Prize contest. As part of that, some of its functionality was later baked into EMET. ROPGuard works to ensure essential Windows API functions must have a call-preceded instruction occur first. ROPGuard also checks that the word that precedes the return address is the true start of an essential function. This will protect essential functions, but it does not offer the same protection for non-critical functions. ROPGuard also provides additional security by examining the stack to look for what could be other possible ROP gadgets, and if these are found, the program then terminates. ROPGuard makes use of heuristics in order to check and see if a stack pointer does not go beyond what is defined as the limits for the stack, thereby mitigating against attacks on the heap (Davi & Sadeghi, 2014).

ROPGuard offers no protection with respect to indirect jumps or calls, so while it may provide defense against ROP, it does not provide defense against JOP.

Other CFI Solutions

Microsoft's Control Flow Guard is the preeminent implementation of CFI, but there have been many different CFI solutions that have been developed. Section examines some prominent ones that have appeared in the academic literature.

Cryptographically Enhanced Control Flow

Cryptography can be used to enhance CFI. The proposal for Cryptographically Enhanced Control Flow still requires a fair amount of work to be effective, but it holds great promise. The approach is to utilize message authentication codes (MAC) to provide assurance for vtable pointers, return addresses, and function pointers, thereby greatly hardening the control flow

(Mashtizadeh, et al., 2014). This provides a number of advantages. First, CCFI requires that in order for an adversary to launch a ROP attacker, they must somehow observe a specific MAC that it is utilized. Second, CCFI requires that a MAC can only be utilized with the pointer that it is associated with. These two facts together can help avoid the need for an open, coarse-grained CFI, as that assurance will be provided instead by the MACs.

Whenever any objects pertaining to control flow are encountered, the return address is placed on the stack alongside MACs generated using the return address and the frame pointer. Subsequently, the MAC is verified in the epilogue before it is returned to the caller. This is key in helping to avoid a hypothetical attack where the adversary would pop the upper stack frame, in order to then move to different return addresses.

CCFI implements its cryptography by storing a random key in a dedicated register. Utilizing a register can help prevent a memory disclosure. In addition, the MAC also includes additional metadata to help mitigate against the threat of swapping pointers that could have been leaked. CCFI also makes effort to mitigate against other emerging attacks. Thus, RELRO is utilized to ensure that the GOT is made read-only, thus getting rid of an older attack that is seeing increased use (Mashtizadeh, et al., 2014).

CCFI is subject to criticism. Some programs required manual computations for MAC to be inserted, which would be impractical for a real-world implementation. Other issues can arise, as when void* is used for both arguments of memcpy. Most concerning of all is the high performance overhead, which can range from a lower 3-18% to an unacceptably high 38%. It is also possible the hashing may be too expensive an operation to perform for any real-world CFI implementation. CCFI, while imperfect, is mentioned because of the innovation and the tremendous that hashing may add.

Lockdown

Lockdown is another prominent tool that provides a more fine-grained, dynamic approach to CFI (Payer, et al., 2015). With Lockdown, import and export definitions are used to approximate the jump targets, and target approximation can also occur at time of execution through dynamic binary analysis, thereby dynamically changing the control flow graph for the process being executed. Furthermore, the current object can only proceed to where it is dictated by the current object imports and exports, thereby allowing for only valid functions to be reached (Payer, et al., 2015). One of the more robust ways in which Lockdown enforces control flow integrity is through the shadow stack. Thus, valid addresses from function calls to specific addresses are placed onto a shadow stack. If there is an attempt made to reach an address that has not already been placed on the shadow stack, then the program will fault. In addition, the shadow stack within Lockdown also requires that control can only be given back to the caller or sometimes the previous caller, thereby preventing many addresses for ROP gadgets (Payer, et al., 2015). Lockdown is fine-grained in its approach to CFI, and it becomes increasingly more fine-grained when there are more modules or libraries associated with a process. Thus, the more libraries there are, the more fine-grained the CFI (Payer, et al., 2015). Lockdown also augments its defensive capabilities against ROP with strong restrictions on intra-module indirect calls. Beyond these powerful capabilities, Lockdown can analyze and make sense of symtab and dynsym information from ELF binaries to better restrict boundaries for jumps (Payer, et al, 148-150).

Summary

Chapter 2 has explored the evolution of code reuse attacks from return-to-libc, to ROP, and to JOP. Prominent forms of memory corruption have gently been introduced to the reader. There

has been discussion on the different mitigations and countermeasures that have arisen, such as DEP, ASLR, EMET, as well as various implementations of CFI, e.g. Microsoft's CFG. Other solutions to ROP and CFI have been examined. Chapter 2 will help provide a broad overview of the necessary concepts one must grasp to be fluent in code-reuse attacks.

CHAPTER 3

RESEARCH METHODS

Chapter 2 provided a broad overview of the literature that comprises the background needed for this research. We were able to ascertain the current state of research with respect to code-reuse attacks and the various mitigations in place that attempt to curtail their usage. In the sections that are to follow, Chapter 3 will present the research methods that will be employed in this study. In addition, this chapter will not only discuss how this framework works, but it will \ demonstrate why this framework is the inevitable result of a DSR inquiry into this research question. This chapter will discuss how this research satisfies the design science guidelines of both Wieringa (2014) and Hevner, et al. (2014). Finally, chapter 3 will define in detail the requirements, development, as well as implementation of the framework.

We are reminded that purpose of this research broadly is to facilitate advanced code-reuse attacks. More specifically, this research aims to develop a novel framework can greatly simplify the process of constructing JOP exploits, making what was once a manual and potentially labor-intensive process, into a far simpler and more fruitful activity.

Hypothesis

A hypothesis takes a clearly worded stance that predicts the behavior of different variables within a system (Creswell, 2018). The hypothesis for this study is that a software tool can be created to greatly reduce the effort required to construct a JOP exploit. This hypothesis will guide the work that will be done in this study. Without the presence of this framework and its methods,

the effort needed to construct JOP exploits would be significantly higher. This is a causal explanation, describing how this framework will facilitate a possible occurrence.

The null hypothesis will be that a software tool cannot be created to greatly reduce the effort required to construct a JOP exploit.

Research Approach

This research will make use of design science methodology, and the resulting artifact, the JOP ROCKET, will answer the research question. While Hevner, et al., are much more well-known with design science, this research will pursue a DSR inquiry by looking at Wieringa, due to the highly technical nature of this work. Wieringa's approach to design science is better suited to this study, as it is more apposite to software development, reverse engineering, and exploit development. The JOP ROCKET is not intended for information systems or business use, but because of its widespread use in design science, this chapter will endeavor wherever possible to show how it not only follows Wieringa, but also meets the guidelines set forth by Hevner, et al.

The problem, having been identified in chapter 1, is the need for a novel framework that helps facilitate the discovery of JOP gadgets and simplify the construction of JOP exploits. The motivation is the lack of available tools to provide this needed functionality and the difficulty and potentially time-consuming nature of doing so without such a tool. As mentioned previously, Roemer spent weeks finding just a couple dozen gadgets that met certain classification criteria for Turing-complete features. This was work done by manually looking at disassembly from a Solaris libc file, and it was using the far simpler ROP. Without proper tools, a manual undertaking seeking JOP would be time-consuming effort.

This will employ the design science research methodology, and it will result in the creation

of an artifact, the JOP ROCKET. According to Wieringa, design science entails performing both the investigation and design of artifacts in a specific context. The artifacts in turn are designed to address the problem context with an aim of improving it (Wieringa, 2014). DSR is intended to develop a practical approach or method to allow the problem to be better understood, allowing for the cycle of re-evaluation and improvement of design, to lead to a superior solution that satisfies the research problem (Watts, 2009). Validation and evaluation are hallmarks of the DSR cycle, as different iterations inevitably lead to a continuously improving artifact and bringing into focus different DSR choices.

This research will adhere Hevner's DSR guidelines. They assert design science is very much concerned with the process of discovery through artifacts that have been created in order to address certain problems (Hevner, et al., 2004). Design science, thus, can be fruitful and allow for new and useful knowledge to emerge from the creation of an artifact, and that knowledge can pose a significant contribution to the collective knowledge of the cyber security discipline.

Design science problems also demand a careful analysis of stakeholder goals (Wieringa, 2014). The stakeholders here can be identified as a security researcher engaged in exploit development. The security researcher is one who wishes to have the ability to more easily discover relevant JOP gadgets, classify them, and exclude those which have minimal value. To do this through an automated tool would save a monumental effort that would need to be expended doing it manually. Another stakeholder could be defenders; these could consist of individuals who might devise or utilize defenses against JOP. The JOP ROCKET could perhaps lead to superior defenses. The stakeholder goals will have been carefully considered when devising the requirements for the artifact.

Hevner's Desigin Science Guidelines

Hevner provides seven guidelines in their seminal piece on design science. These guidelines are listed in Table 1, alongside both a description of the guideline and details as to how the JOP ROCKET satisfies these guidelines (Hevner, et al., 2004).

Table 1. Hevner Design Science Guidelines

Guideline	**Definition**	**Artifact**
Design Science as an Artifact	Design science should result in creation of an artifact in the form of model, construct, instantiation, or method	JOP ROCKET will adheres t these principles. It provides an instantiation of a framework and five supporting methods.
Problem Relevance	Design science research is focused on developing solutions to important business and information systems problems	JOP ROCKET will fully address the problem of not having an automated process to help construct JOP exploits. This will eliminate what could be very time-consuming, tedious work.
Design Evaluation	The design artifact must be able to withstand scrutiny from rigorous evaluation to ensure sufficient utility, quality, quality, and effectiveness.	The accuracy and validity of its results will be tested by making use of several existing reverse engineering tools to ensure accuracy. The artifact will be developed in a protype environment and made use of single-case mechanism experiment (laboratory

		simulation), to demonstrate its efficacy and utility.
Research Contributions	The design science research must provide verifiable contributions with respect to design foundations, design artifact, or design methodologies.	JOP ROCKET provides contributions to design science research, by providing a sorely needed tool that is highly useful for creating JOP exploits or exploring the possibility of doing so. Several of the novel methods could be adapted and used in other reverse engineering tools unrelated to JOP.
Research Rigor	Rigorous methods must be used both during the artifact's construction and evaluation.	JOP ROCKET will be subjected to a cycle of testing throughout the entire development cycle. A single-case mechanism experiment will be used to provide validation and evaluation of the artifact.
Design as a Search Process	Available means must be used during the search for an effective artifact, while also not violating laws within the problem environment	JOP ROCKET was developed through a highly iterative cycle to help ensure the best possible artifact apposite to the task was created. Additionally, other code-reuse attack tools relating to ROP and more general-purpose reverse engineering tools were

		examined closely, to determine what was effective and what could be improved.
Communication of Research	Design science should be able to be communicated effectively not only to a technology-oriented audience, but also to one that is managerial.	JOP ROCKET provides rich, detailed information to the intended technical audience. It generates files and statistics on its findings, to be of use to security researchers and technical users. A managerial audience can understand that it produces desired results.

The DSR process inevitably is part of a cycle, with different iterations allowing for the design of the artifact build to improve, while the problem can be closely examined and research methods can be enhanced as needed.

Design as an Artifact

In the first guideline, the design research must yield a viable artifact that results in a construct, a model, a method, or an instantiation (Hevner, 2004). Wieringa writes that an instantiation of a software solution should be referred to simply as an artifact; both will be used interchangeably in this chapter (2014). This research will produce an instantiation of a framework that is designed to help facilitate exploit development. The JOP ROCKET will also encompass several novel methods.

Together the artifact and its methods will solve a problem that has hitherto remained unsolved. This is achieved by creating a framework that allows for the discovery of JOP gadgets in an innovative fashion. In addition, it expands upon and refines the algorithm for JOP gadget discovery. Refinements on the algorithm will allow for considerably more gadgets to be found, helping to ensure that potential gadgets are not missed.

Problem Relevance

Problem relevance asserts that DSR must yield technology-based solutions to important and relevant business problems (Hevner, 2004). The JOP ROCKET will achieve such, by meeting the needs of those in the business of exploit development, as well as by providing additional motivation for enhanced cyber security to those play a defender role within organizations.

The primary problem relevance is the lack of supporting tools, as addressed in chapter 1. Existing tools that make use of gadget discovery for PE Files only provide very minimal support of JOP gadgets, such that they are of minimal utility. The Mona python script (Van Eeckhoutte, n.d.) and ROPGadget (Salwan, n.d.). provide scant to barely existent coverage; this is certainly understandable as these tools were designed to discover ROP gadgets. The only way to obtain the needed information to construct a JOP exploit would require using multiple reverse engineering tools, and it would be a time-intensive, manual process. Moreover, it would require a mastery of those tools that many would-be users of JOP simply may not possess. Roemer, as described in chapter 1, spent three weeks just manually searching for ROP gadgets in a Solaris libc file to find gadgets for his Turing-complete features gadget catalog for SPARC, as there were no tools available. JOP is a much more complicated process. This research will also provide a classification of gadgets, as part of one of the methods, and that includes many Turing-complete features. This classification will greatly simplify and enhance the process of constructing the JOP chains.

By having a way to find JOP gadgets in an automated fashion, then JOP becomes viable as an attack paradigm, which previously was not the case. This can allow JOP to then be used on systems that may have strong defenses against ROP, but whose defenses would be inadequate for JOP, as discussed in chapter 1. Exposing many systems to the threat of JOP underscores the need for better CFI and enhanced cyber security. It motivates defenders to find solutions to JOP, or to upgrade to operating systems that can provide defense.

Design Evaluation

The third guideline, that of design evaluation, asserts that the triad of efficacy, quality, and utility must arise from the design artifact; furthermore, these attributes should be demonstrated via well-executed evaluation methods (Hevner, 2004).

The framework will make abundant usage of validation techniques in the form of a single-case mechanism experiment. According to Wieringa, single-case mechanism experiments are used for implementation and evaluation of a design science artifact. This allows researchers to investigate, in a lab or natural setting, the cause and effect of the object of the study within an environment containing the intended context. The mechanism describes the interaction among the various elements at play, showing the effect the artifact has on the natural phenomena in the real world. The response can then be understood through the unique mechanisms intrinsic to the model (Wieringa, 2014) . Single-case mechanism experiments can be done in two ways, either by investigating an implementation in a natural, real world setting, or by making use of an artifact protype in a laboratory environment (Wieringa, 2014). This research will pursue the latter.

Single-case mechanism experiments are highly effective in the validation of new technologies, such as the JOP ROCKET, by performing evaluation of the utility and effectiveness of the design. One of the most labor-intensive parts of the design cycle of the artifact will be in

demonstrating these attributes. These experiments may respond to questions such as, can the framework be used in a way that is useful to the practitioner? We will find this is likely to be the case, as it will provide classification of JOP gadgets, using some Turning-complete features as well as others that are useful for JOP. This ensures the gadgets were organized in a way that was meaningful and useful to the practitioner. We will find that with this framework, the practitioner will be able to be very granular and specific about the types of gadgets they are searching for, and to get results that are specific to those needs. Additionally, gadgets that likely would be of little use will be excluded, thereby saving the security researcher from the need to manually consider each gadget for potential use.

If the artifacts are able to respond fully the research problem and the primary research questions, there could be an argument that the JOP ROCKET has met evaluation criteria, if testing with a single-case mechanism experiment is able to validate the technology. Each of the artifacts have specific aims that will be described in chapter 4, and from a DSR standpoint, if this research creates a tool that can satisfy all those goals, while producing the intended results without errors, then the evaluation will be successful. As a very complex tool, the fact that it simply works and does what it sets out to accomplish some very non-trivial tasks demonstrate it has met evaluation criteria

Tied closely to evaluation is the accuracy of the gadgets. While we can view this from a more high level perspective, in that we need the artifact to meet various goals, if the accuracy is off, then it cannot satisfy many of these goals. Widely used reverse engineering tools, such as IDA Pro and WinDbg, will ensure accuracy of the gadgets. First, it will ensure the disassembly produced is accurate. While this framework makes use of the Capstone disassembly engine, this is not altogether straightforward. The JOP ROCKET is a static analysis tool, and is not looking at

a process, but is working with code from the text section of the PE file, reading from disk, rather than from process memory. If the disassembly begins at the wrong byte, then the disassembly that it produces is not accurate. The resulting disassembly will be checked in IDA Pro or WinDbg as appropriate, both at the Assembly mnemonic level and also looking at the opcodes.

Second, the algorithm to find JOP gadgets will search for the desired indirect jump or call, and then it will disassemble backwards to discover opcodes that could then become useful gadgets. If the algorithm goes back 8 or 10 bytes to begin disassembly, this will cause opcode-splitting, meaning the resulting disassembly could be different, depending on which byte the disassembly starts at. Ensuring these results are accurate will be critical, if the JOP ROCKET is to have any value.

Third, while having accurate disassembly is necessary, having the correct virtual memory addresses as well as the correct offsets is equally important. After all, an inaccurate address will not allow us to reach the intended gadgets, and thus the gadgets would be of minimal value. Any of the aforementioned reverse engineering tools can allow us to perform inspections to see if these are correct.

Research Contribution

Guideline four states that an artifact needs to contribute to the province of design artifact, potentially encompassing artifact design, design methodologies or design foundations (Hevner, 2004). This framework will contribute materially to the discipline, satisfying hitherto, unmet needs. Because of the capabilities provided by this framework, exploit developers and security researchers will all stand to benefit from the JOP ROCKET, as it helps simplify the necessary tasks that otherwise would have had to been performed manually or semi-manually. This could be a monumentally difficult and highly time-consuming task. It would require specialized knowledge

and use of multiple tools, barring entry to many researchers to JOP. The value and importance of having this tool for security researchers cannot be overstated. JOP ROCKET will enrich the area of design artifact by providing a well-tested artifact that that provides a solution that has some unique, novel methods. Some of the methods used, as will be described more fully in chapter 4, independently could constitute worthy research contributions. Some could be taken and adapted for more general-purpose reverse engineering tools. In sum, this artifact will utilize existing knowledge in ways that are both innovative and novel, allowing for unanswered research problems to be solved.

Research Rigor

The fifth guideline concerns research rigor, and it asserts that DSR should employ rigorous methods not only during the construction of the design artifact, but during the evaluation (Hevner, 2004). This rigor has been achieved by iterating through the different stages of design science as well as working to ensure that the results it produces are accurate and valid. The work accepts as valid the results produced independently by different existing reverse engineering tools; producing results that line up with those help to provide rigor. In one sense, the question of research rigor can be addressed in an objective fashion by whether or not it achieves its stated aims. If it can do this and those results can be verified, then it will have demonstrated research rigor. If the artifact is able to provide classification of gadgets to enable security researchers to take a very fine-grained approach to gadget discovery, that too will have made a contribution towards research rigor.

Design as a Search Process

Guideline six, which deals with design as a search process, states that the search for an effective artifact requires making use of all possible means to reach an effective artifact, while

satisfying the laws inherent in the problem environment (Hevner, 2004). This framework meets these guidelines. This tool will be developed in a prototype environment, after multiple years of studying code-reuse attacks, close examination of the literature, examination of existing reverse engineering and exploitation tools, and through iterating through the design science cycle to help achieve the most effective tool apposite to the research problem. The design science cycle by its very nature is iterative, allowing for multiple cycles of development, validation, and improvement, to achieve a superior tool.

Communication of Research

The final Hevner guideline, that of communicating research, asserts that DSR must be presented not only to a technology-oriented audience, but also one that is managerial (2004). This work is communicated effectively to a technical audience through this dissertation, which provides abundant technical detail and discussion. Hevner's DSR guidelines deals with information systems, businesses, and the management that oversees those businesses, thereby necessitating that part of the research be understandable by them. This is one of the guidelines where this study follows Wieringa more. The communication of the results that will be created by this framework will be readily understandable to the intended users, consisting of security researchers and exploit developers. The JOP ROCKET is not intended to break down the results in layman's terms for management, as it is assumed that largely this work would be beyond their understanding. This largely is one of the shortcomings of Hevner, since not all design research will be on a level that is accessible to a management audience. However, the fact that the framework will produce results in well documented files and provide statistics on the number of gadgets found is a form of communication that a managerial audience may appreciate. They may recognize that it is doing

what it sets out to do, generating JOP gadgets that can be used to construct a JOP exploit; any more in-depth understanding would be unnecesary.

As has been alluded to, the intent is not to follow Hevner's design principles, although we find that, according to the above, these guidelines largely would have been satisfied. Because of the highly technical nature of this framework, it instead follows DSR guidelines set forth by Wieringa. His work, while much less widely known, seems more appropriate to creating software.

Wiering'a DSR model attempts to answer two types of research problems, design problems as well as knowledge problems (2014). The first attempts to improve upon something, where utility is a goal and knowledge is a side effect. The second attempts to discover knowledge, explaining, predicting, and describing, with truth as the ultimate goal and utility as an inevitable side effect. The algorithm for finding JOP has already been established, so this research does not endeavor to rediscover the wheel, but it instead improves upon and expands previous research.

Wieringa's Design Science Guidelines

Wieringa describes a scientific theory as a conceptual framework that can help frame a research problem, allowing for phenomena to be both described and subjected to analysis, such that generalizations can be formed, allowing for the structure of the artifact and its context to be laid out (2014). These generalizations may help explain some the causes or reasons why the phenomena are such. Having formed useful generalizations, these may be in turn used to help provide the necessary justification for designs of artifacts that may arise. The framework can be viewed, not just as a way of specifying an artifact, but a way of defining methods, asking questions, and interpreting the specifications; it can be a way of discussing the architecture, as well as its different capabilities and components (Wieringa, 2014).

Wieringa writes that with an architectural framework, a system's various components are responsible for its behavior. Thus, we could view the proposed instantiation of the JOP ROCKET as an architecture whose various components are responsible for its "system-level phenomena" (Wieringa, 2014). As designers, we then can explore the various available options by attempting different possibilities and observing the effect they have on system behavior (Wieringa, 2014). With respect to the JOP ROCKET, we can abstract different parts of it, different methods, different features, different algorithms, and explain the effect those have on the system phenomena. These phenomena would include JOP gadgets, the opcodes, the disassembly, and the different classifications. As we embark upon this research, the iterative process of design science will enable us to observe the effects different methods may have on the results. By closely looking at these effects and by making changes to enhance results, we can create the best artifact possible. Some DSR activity through the iterative process may lead us to rethink or refine different algorithms, variables, or methods, and this too will enhance the artifact.

Goals are used to help provide a definition to the research problem; these goals have been enumerated in chapter 1. Design research can then address a design problem, which necessitates an iterative process of designing an artifact in order to satisfy said goals. These design problems stem from both the stakeholder's goal as well as the problem's context (Wieringa, 2014). The artifact is the vehicle through which the stakeholder goals can be satisfied. This can be broken down to the form of a template: "Improve <a problem context> by <(re)designing an artifact> that satisfies <some requirements> in order to <help stakeholders achieve some goals> " (Wieringa, 2014). This template could be applied to any design science problem. In the case of this research, we will express the aim of this research as such: This research will improve the need for automated tools for JOP discovery by designing a framework that satisfies both its stated technical and major

functional requirements, in order to facilitate the development of advanced code-reuse attacks and allow users to save significant time and effort.

According to Wieringa, the three primary phases of a design science engineering cycle consist of problem investigation, treatment design, and treatment validation (Wieringa, 2014). The problem has been thoroughly investigated in chapter 1. Treatment design consists of those requirements that contribute to the goals of the implementation. These have been expressed separately within this chapter, as technical requirements and major functional requirements. Wieringa's work on treatment validation will be addressed in chapter 4.

Objectives

The next task is in defining the objectives for the framework. These objectives are an integral part of the design for this artifact:

It should identify all gadgets for each indirect call or indirect jump to a particular register. These could form the foundation for the functional gadgets.

It should then use criteria to establish and discover potential dispatcher gadgets. These will be few and far in between for ones that are viable, and good ones will be even more scarce.

It will use criteria to eliminate gadgets that are by likely to be of no practical use. These can be identified through analysis and if found may be discarded before the analyst even needs to look at them.

It will classify gadgets into different categories that roughly correspond to features of the Turing catalogue as well as additional classifications that are practical in constructing a JOP exploit. Not all Turing-complete features will be utilized, as they are beyond the scope of what this tool is intended to do. Some, for practical purposes, would be very difficult to do with JOP in real-life,

non-toy binaries. These will make extensive use of analysis to reject gadgets that most likely are to be of marginal or no practical value, saving the analyst time that otherwise would be wasted. This feature will be of tremendous utility to the security researcher, as instead of having to potential comb through vast numbers of gadgets, they can look at the ones germane to the task at hand. Because JOP in essence has its own stack-like structure, the dispatch table, which serves to order control flow, many gadgets by their very nature may need to be discarded outright, so being able to be very granular and specific will be immensely useful to the security researcher.

It will be a primarily static analysis tool, although some DLLs will be briefly loaded into memory but not executed, so a handle can be obtained for the module, allowing for its file location to be found. This is needed for its text section may be extracted for analysis. The reason for this is because one of the overarching goals is to not have this tied down to an existing tool, such as a script for WinDbg, Immunity, or IDA Pro, and for it to be its own standalone tool.

It will ensure that by working at both the opcode and assembly mnemonic level, that all potential gadgets are found. With opcode-splitting, we increase the attack surface drastically by changing where we begin disassembly, or eventually where the execution would begin in an exploit. This is important because there is a paucity of available gadgets in comparison to ROP, so we must make sure we enumerate all potential gadgets. Doing this can sometimes result in gadgets that would be of very marginal value, simply because of what they do and registers used. Thus, having strong criteria to exclude will enhance the utility of this framework. The JOP ROCKET will make use of a technique to do this, and in addition, it will provide the user the ability to be granular and specific with this, to increase or decrease the number of results.

The tool will allow the user to decide how many opcodes to go back from the desired indirect jump or indirect call. It then will then iterate through and enumerate all the possibilities.

Thus, if a user specified to go back 14 bytes, it would go back 14 bytes and then test it with all possibilities at an opcode level, until at reached the minimum number of bytes, effectively ensuring the whole of the attack surface is covered. This enhanced flexibility, rather than just hardcoded defaults, can allow a user to narrow or widen results as needs dictate.

It will let the user be specific about the number of lines of other instructions that should be able to exist between the desired operation (e.g.MOV or ADD to a desired register) and the terminating indirect jump or indirect call to the target register. Having more lines in between can certainly make some of these gadgets more difficult to use, as they might clobber needed registers, but it may be necessary if there are limited gadgets otherwise. Likewise, a user may just want to exclude all results that exceed a certain distance. As a practical tool, this framework will provide the analyst with the ability to make those judgment calls if need be, but it will provide reasonable defaults so that the novice user need not to be concerned with such details.

Artifacts of Design Science

DSR has an end product of artifacts that address the research problem. The artifacts created should have utility and be effective at achieving an artifact's purported aims. This design science research inquiry is part of an effort to answer the research question, and it will result in the creation of different artifacts. These artifacts can be immensely useful to the security researcher that wishes to make use of JOP.

This research will culminate in the creation of two types of artifacts, a collection of methods and an instantiation of a framework, the JOP ROCKET. Accurate requirements will be provided to help provide assurances that the artifacts created fully address the all aspects of the research question. Throughout review of the literature, an instantiation such as outlined below was not seen,

nor were the methods as described here, although as mentioned before, some methods are significant improvements upon existing algorithms.

Within the province of design science, we can regard a method as those operations, steps, or algorithms involved in performing a task (March & Smith, 1995). The following methods will be created during this research:

A method to discover JOP functional gadgets;

A method for discovering JOP dispatcher gadgets while applying exclusion criteria;

A method for printing the disassembly of gadgets found;

A method for classifying JOP gadgets into, while applying exclusion criteria;

A method for statically enumerating and obtaining modules in the import table and obtaining their JOP gadgets, while applying exclusion criteria.

A method that utilizes an object-oriented approach for storing executable content, JOP gadgets, and other bookkeeping information for the PE image and its modules.

The most significant contribution, the primary effort, is in creating an instantiation, which embodies all the methods and serves as a practical tool that the user can employ to obtain the desired information. According to March & Smith, we can regard an instantiation as the final culmination of the artifact (1995). Throughout the research, the methods will be developed with the notion of how they can be implemented effectively within the instantiation. The methods provide the most benefit embodied in an instantiation, where collectively they can provide the desired functionality needed to address the research question and provide utility to the security researcher. Taken independently, some could be adopted and used for other, unrelated projects.

Requirements for Instantiation of the Artifact

The requirements for the instantiation are provided below in Table 2.

Table 2. Requirements for Instantiation of the Artifact

Number	Requirement
1.	Must be portable across platforms, such that it is able to be run on all modern Windows operating systems.
2.	Must be portable across platforms, such that it is able to be run on Linux and MacOS with limited functionality.
3.	Must be able to run with minimal hardware specifications.
4.	Must make use of only publicly available libraries as dependencies.
5.	Must provide a user interface that is intuitive and that has an easily accessible help sub-menu.
6.	Must provide output to user in a manner that is logical and convenient.
7.	Must provide the user with a high degree of freedom to customize the types of gadgets obtained and how they are generated, by allowing them to change the mechanics of how the gadgets are produced.

Assumptions and Limitations

This research has intentionally limited the scope of the framework instantiation to 32-bit Microsoft Windows PE files. The reason for this is simple: developing a tool for another architecture or for 64-bit would be non-trivial and would involve rewriting all the labyrinthine rules for the numerous subroutines that use strict criteria to exclude gadgets. Testing each of these subroutines to ensure accuracy can be a labor-intensive endeavor. Such effort is best left to future work.

While 64-bit has become commonplace, a significant number of programs are still 32-bit and simply run under SysWoW64, which provides compatibility on a 64-bt system for 32-bit applications to be run natively (Kennedy & Satran, “File System,” 2018). 32-bit applications are more prevalent and likely will remain so for a while, and so targeting these is more important. One limitation of no examining the 64-bit binaries is that we do not gain an understanding of the nature of JOP in that environment that could prove useful. Because of the nature of JOP, having more available registers in theory could be highly beneficial. However, what that means in actual practice will be another matter. Statistically, as this research will demonstrate, indirect jumps and indirect calls to certain registers in 32-bit applications are much less plentiful, making some of them even potentially difficult to use. Thus, it stands to reason that while there may be more registers available to use, in actual practice some may not on average produce many practical JOP gadgets. Nor would we gain an appreciation for how the x64 calling convention impacts the efficacy of JOP in general and with specific registers. A more in depth study of JOP in 64-bit programs could be fruitful. While one could produce anecdotal responses or conjectures to some of the above, the JOP ROCKET produced highly detailed statistical information on the numbers and types of gadgets produced for the binaries that it analyzes.

This framework does consider the matter of ASLR, but it does not provide the ability exclude modules that utilize ASLR. The artifact provides information on different security protections in place for each application and associated modules, such as ASLR, DEP, CFG, and GS (Stack Guard). Providing the functionality to exclude them could be done without significant effort, but they are included as the user may be aware of a memory disclosure to overcome ASLR or other bypasses. The framework also conveniently provides offsets for all modules, in the users has the ability to obtain a memory disclosure. It is assumed that the security researcher who uses

this tool has a high level of sophistication and familiarity with more basic code-reuse attacks, e.g. ROP, and they understand the ramifications of all protections.

This framework will be limited in the number of modules that it is able to enumerate and thus obtain gadgets for. This is perhaps the most significant limitation. This is due to the limitation of this being a static analysis tool. This is due to the fact that modules are loaded into a process in a variety of ways. First, they may be imported. Some may be done dynamically during runtime, as with a Windows API such as LoadLibrary and GetProcAddress, or through a malware technique such as traversing the PEB to find desired modules and then desired Windows API functions. Some modules will also call upon other DLLs as a dependency. Other dynamic methods of loading libraries can include loading libraries for forwarded as well as delayed API calls. Thus, to be able to enumerate all modules, one must make use of a dynamic approach, allowing the process to be examined in memory, rather than from disk. As this is a static analysis tool, it does not provide that functionality, and as such it will only enumerate those that are statically bound to the import table, which is a limitation, since some modules may be missed. When one makes use of general-purpose reverse engineering tools such as WinDbg, we may take it for granted that some libraries are loaded through other means early on. The end result is that some libraries that are loaded relatively early on are excluded. There does not seem to be an easy or simple way to statically enumerate these via examination of the PE; dynamically loading the PE file as a process would need to be a necessity to discover some of these libraries that are not loaded via the import table. This research will identify other unique ways to obtain some likely modules that will be loaded and that exist outside the IAT, but while this will be pragmatic, it will not be all inclusive.

Another limitation is that the user needs to establish the proper environment with necessary dependencies, namely Python as well as the following Python libraries: Capstone, PEFile, and

Pywin32. They will need an environment in which to make use of Python; the recommendation is Cygwin, for convenience sake, although any environment would do. Setting up these dependencies might take a moderate amount of effort, but it likely would be a trivial undertaking for the intended users, who may already have some or all of these set up. By utilizing different existing libraries and being a standalone tool, the JOP ROCKET avoids the need to have to integrate with an existing reverse engineering tool, such as WinDbg or Immunity, making users reliant that tool. By having a standalone framework, the user can quickly and easily make use of the artifact from a command line interface to obtain the necessary data.

There is a limitation with respect to malformed PE files, as those may not be read properly. It is possible have code in places other than the standard text section, and in such a case the code would not be extracted, and thus it would be excluded from analysis. Indeed, any executable code that may lie elsewhere in the PE file would be excluded. It is also possible the code could be self-modifying in memory, behavior that is typically associated with malware. Were this the case, then the gadgets generated may not be valid.

Another limitation is that this framework does not support long term storage of data that is collected on the target PE file and its DLLs. This would be beneficial, but it is a feature left to future work. It is anticipated that security researchers will make use of the tool to generate gadgets, which will then be written to disk in appropriately named files, so that users can refer to the data contained in the files as needed.

This framework will be created using the Python language to provide optimal portability, allowing it potentially to run on Windows, Linux or MacOS. To enable this to occur, the framework will make use of different existing Python libraries, namely PEFile, Capstone, and Pywin32. There will be one limitation, however. While it is possible in the appropriate Linux or

MacOS environment to read and extract gadgets from an executable, on those systems it will only be able to extract gadgets from the executable image, not the modules. This is for two reasons. First, Pywin32 is not supported on those systems. That library provides access to the Windows API in a Python environment. Pywin32 is used on Windows to very briefly load libraries so that a handle can be returned to modules, and this handle is used to help ascertain the file locations of all DLLs in the import table. Thus, the Windows API is used to help facilitate the discovery of file locations for DLLs in the import table, so that their text sections can be extracted and the subjected to analysis. Outside of Windows, this will not be possible as the Windows API will not be present, and even if it were, the necessary supporting DLLs would be absent in a Linux or MacOS environment. Of course, individual modules could be loaded one by one, such as Windows DLLs, but that would be very tedious to acquire all of them and individually analyze them.

Data Collection

The data that is of consequence here are the gadgets themselves. Data that is obtained by the framework will be stored in hundreds of parallel lists that maintain the necessary data, that enables the search, the classification of gadgets, the exclusion of impractical gadgets, and finally the printing of gadgets. The data stored in the data structures will not be the gadgets themselves, but bookkeeping information that can enable the gadgets to be generated immediately on the fly. However, for purposes of data collection, we can think of the stored data as the gadgets.

The JOP ROCKET will utilize methods to obtain the gadgets and perform classification; these methods serve as artifacts themselves, and the full implantation details will be discussed in chapter 4. Thus, part of the data collection involves classification. Classification occurs automatically as the algorithms are run. As a carved out, small chunk of code from the text section

is analyzed, different features of that code might result in several different classifications. One could be for ADD or one for MOV, etc., all ending in the same indirect jump or call. The gadgets stored would be based on the classification, as the search algorithm will only produce the most minimal form of the gadget. Thus, if ADD EAX, EBX appeared 3 lines back from the indirect jump, and MOV EBX, EAX appeared 4 lines back, then if the classification was for addition, then the gadget would begin at ADD EAX, EBX. Thus, we see the same data can manifest itself in different gadgets, serving different purposes.

Prior to data collection, validity and reliability of data must be assured. The sample of binaries to be analyzed will be processed through the framework, and some results will be compared with various reverse engineering tools to ensure accuracy and reliability. These tasks are discussed more fully under Validity and Reliability. Once those steps have been performed successfully, it will be assumed that the data produced is valid and accurate. As such, data collection will then concern itself with the collection of gadgets that are generated by the framework. These are stored in data structures and are available to be immediately printed. Immediately implies that no additional CPU time is required, other than the minimal amount needed to print to terminal or save to disk. The latter will be the chief means of delivering the data; the former is just a convenience to the analyst.

The development of the aforementioned algorithms and workflows will be iterated until all errors have been corrected and the algorithms have been optimized. Once the design science process has completed and the tool has achieved a high level of efficacy and utility, there will be a single-case mechanism experiment that will be performed. This will be done to perform validation, to ensure that the design science cycle has resulted in a tool rich with utility and efficacy.

Data collection adhering to quantitative methodology will also begin once the DSR cycle has been completed. A single-case mechanism experiment will be done on 32 executables. These will include a variety of both open source and commercial executables, ranging in size from small to large. These executables will be run in the framework, and the data will be subjected to analysis and classificaiton. The purpose here will be to demonstrate the utility and efficacy of the tool. It is anticipated that a large number of gadgets will be able to be classified into appropriate categories. Furthermore, a large number of potentially impractical gadgets will be able to be excluded. There will be no data collected on the number of gadgets excluded, as they are silently discarded, as this is a tool that is intended to be useful to analysts, not to perform an exhaustive study on the nature of JOP in x86. The classification and exclusion of impractical gadgets will be a significant contribution in the area of utility and efficacy. This will be discussed in the data analysis section and then finally in the validation section.

Validity and Reliability

To provide assurances as to the validity and reliability of the gadgets produced, it will be necessary to verify the results against those produced by general-purpose reverse engineering tools. For instance, for gadgets produced by the framework, then the disassembly produced must be verified as accurate, and both the virtual memory address and the offset must also be verified. IDA Pro potentially can be used to validate some of the gadgets. However, if opcode splitting occurs, then validating through IDA Pro may not be straightforward, because the disassembly will be different. In that case, it is necessary to look at the opcodes and to see if those opcodes would produce the disassembly offered by the framework. This is not a straightforward process, and it would involve having to use other tools. The framework does not produce opcodes to the user.

However, during the design of the tool, opcodes can directly viewed for debugging purposes. Alternatively, one could use a simple utility or common web site to convert the disassembly produced by the framework back into opcodes, thereby allowing for a comparison (Defuse, 2018). In the case of opcode-splitting, the usage of IDA Pro can verify only that the address or offsets are correct, not the disassembly. To validate the accuracy of the disassembly in view of opcode-splitting, the most straightforward technique is simply to make use of WinDbg and the command *u [address],* such as *u 0x00435352* and then view the disassembly that results from un-assembling at that location. It should be identical to that produced by the framework. Another option could be to take the opcodes and use a utility, such as the online assembler and disassembler at Defuse, to see if those opcodes produce the correct disassembly that the framework reports. That is a more laborious process, but it can provide additional confirmation beyond WinDbg. These techniques will be employed to ensure the accuracy of all gadgets produced.

Reliability also concerns the gadgets produced, beyond simply whether or not the disassembly and addresses are correct. There should be no instructions on a gadget prior to the target operation. Thus, if MOV were the target operation and it was at a distance of two lines from the indirect jump, then the gadget should begin on the target operation, as anything before that would be extraneous and irrelevant. Reliability can come into play with classification as well. If gadgets are to end with an indirect jump to a specific register, then only those that do so must be produced. Similarly, if gadgets are to modify only a certain register, then only those must be produced. Similarly, if the user is searching for a specific operation, such as ADD or MOV, or may be searching for a dispatcher gadget, then the results should include only those that fall under those classifications. The framework will provide the capability for the user to be highly specific, and the results must match what has been requested by the user.

Reliability will ensure that the same results occur when comparisons are made from one tool to another. Results will be checked with a minimum of two reverse engineering tools in parallel. This provides assurance as to a higher degree of a reliability, as the results can be reproducible from one tool, to the next.

Overview of Framework

At a high level, this framework will parse the PE file, extract the executable portion, and then obtain the necessary JOP gadgets. First, the framework will extract the text section for the target PE file in addition to all DLLs contained in the import table. An algorithm will then be run to find the opcodes for all instructions for indirect jumps or indirect calls. Once these are found, disassembly will be obtained for all possible combinations of opcodes that terminate in an indirect jump or indirect call. The resulting Assembly will then be searched with classification being performed according to strict criteria, to exclude gadgets that would otherwise be impractical. Finally, the end result will be hundreds of parallel arrays populated with necessary data to create the appropriate gadgets that the user may be seeking, based on whether it is an indirect call or indirect jump, the registers in question, and the operation performed. In addition, the JOP ROCKET will feature additional customization the user can perform, to be more granular and specific as to what gadgets they wish to search for or print. This could increase or decrease the number of gadgets, while at the same time having an effect of lowering or increasing the potential quality of said gadgets. As to the matter of quality, we could regard a high quality gadget as one where the target operation is close to the indirect jump or call, whereas a low quality gadget is once where that distance is high. Thus, the ability for the user to identify specific criteria for the functional or dispatcher gadgets they seek will add significant utility to the framework.

The framework will be run on the command line. This will allow a user to more quickly and easily make use of the artifact, if they will not need to waste time with a mouse or cursor. The program is interactive, where the user supplies relatively brief keyboard commands to perform operations. For ease of use, these commands are limited generally to one to three characters, and a help sub-menu provides a brief explanation on how to use the different commands. The artifact allows for a great deal of customization to be possible, so taking command line arguments would be impractical, although there is a default command line option that will attempt everything with reasonable settings. The user interface has reasonable default settings, so every setting need not be set, but they can be adjusted with great flexibility as need be.

Specification of Major Functional Requirements

The functional requirements should provide explicit, detailed descriptions to aid in the design of the application. Wieringa asserts, “The desirability of a requirement must be motivated in terms of stakeholders goals by a so-called contribution argument” (2014). This research has had its requirements motivated by these stakeholder goals. Wieringa defines as a requirement a property that will be sought by a stakeholder, to be realized in the final implementation of the artifact (2014). A functional requirement is one specific type of requirement, which Wieringa defines as a requirement for some functions that will be sought from the artifact. All the primary functional areas will be discussed with a detailed description of the proposed functionality.

Command Line Interface

For ease of use, this tool will be built with a command line interface. The purpose of this will be to provide a versatile user interface for the user, one which does not require the use of a

mouse. This will be the primary means through which the user enters commands and can interact with and utilize the various algorithms and methods, to allow for effective use of the framework.

The command line interface needs to be run in an environment with Python support and with the previously described dependencies, listed under Technical Specifications, and these dependencies must be configured properly. On Windows, the ideal environment is Cygwin, and the tool will be optimized for that. However, any environment with the dependencies is adequate.

User commands will be entered in one of three primary areas. The first will be the primary screen, where input can be entered. The second will be a printing sub-menu. This will allow for results to be printed to screen and saved to appropriately named files. There is a tremendous amount of customization on the printing menu, with over 240 different combinations possible. At this stage, all the results will have been stored in one of hundreds of data structures. It will be simply a matter of providing the input needed to obtain the desired results. The final area will be a sub-menu for searching for dispatcher gadgets. As with the printing sub-menu, this will allow for a great deal of customization. Some additional screens will be able to be accessed, and these will take some input but will be relatively straightforward. These will include a help sub-menu, settings for extracting DLLs, and others.

User commands will be minimal, typically one to three characters, just enough for a quick keystroke. The intent will be to minimize the typing the analyst must do. The commands all will in some way have some mnemonic that can be connected to the operation that is being performed. This will allow an experienced user to quickly enter these commands without the need to consult the help sub-menu. To that end, reasonable default settings for some of the more esoteric settings will be set, so that a user will be able to simply select from among various preset options, if they do not wish to specific each individually. Because the intend will be for the tool to be very flexible

with a high degree of customization available, there will not be a one-size-fits-all setting that will be the standard default. Though the user will be able to simply select everything, it is assumed that they will wish to customize their output by using some more specific preset options or even specifying some options individually.

The commands are introduced alongside a description in Table 3 and Table 4. These will describe the commands the user is able to enter in the main screen as well as the print screen. There will be a simple interface to a very robust, powerful set of algorithms and methods. There is a stakeholder goal in having a degree of usability, so this is felt to be preferable to something with arguments supplied on the command line. With the level of customization that will be present in this framework, the command line would be too limiting. Moreover, extracting and performing the search does take a period of time, so the user may explore several options for printing after those steps have been completed, and it would be highly inefficient for them to have to continuously repeat the extraction and search. Once the extraction and search have been completed, it will only take seconds to print the results, regardless of the level of customization applied.

Table 3. The main screen user interface commands.

Command	**Description**
F	This allows for the target PE file to be changed.
R	This specifies the target 32-bit registers, delimited by commas, e.g. EAX, EBX.
T	This sets the target control flow, indirect target, e.g. JMP, CALL, or both.
P	This allows for the user to configure the settings to print output according both to target registers and target operations.

Command	Description
D	This grants access to a sub-menu to provide settings for getting dispatcher gadget.
M	This enumerates all modules.
N	This changes the number of opcodes to disassemble. This is done through an iterative process, so the more that you have, the more gadgets you generate, but the higher you go, the longer the resulting gadgets become.
L	This changes the lines to go back when searching for a specific operation, e.g. 3 or 4 lines.
S	This sets the scope to perform the search and subsequent printing of results. The default is set to just the PE file itself, but the scope can be expanded to include all modules in the import table.
G	This gets gadgets. Once the settings have been set, the framework will search through the extracted text section in order to discover the target gadgets, perform classification, and perform exclusion.
C	This clears all the data structures.

Table 4. The print screen user interface commands.

Command	Description
R	Set registers to print. The functional gadget classified by operation will print only those that pertain to registers set here. For instance, if EAX was set and ADD was selected, then only gadgets that somehow add to EAX/AX would be included in results. All registers are possible as well as ALL.

Command	**Description**
J	Print all JMP [REG]. This is done according to registers set.
Ja	Print all JMP EAX. This is done according to registers set.
Jb	Print all JMP EBX. This is done according to registers set.
Jc	Print all JMP ECX. This is done according to registers set.
Jd	Print all JMP EDX. This is done according to registers set.
Jdi	Print all JMP EDI. This is done according to registers set.
Jsi	Print all JMP ESI. This is done according to registers set.
Jbp	Print all JMP EBP. This is done according to registers set.
C	Print all CALL [REG]. This is done according to registers set.
Ca	Print all CALL EAX. This is done according to registers set.
Cb	Print all CALL EBX. This is done according to registers set.
Cc	Print all CALL ECX. This is done according to registers set.
Cd	Print all CALL EDX. This is done according to registers set.
Cdi	Print all CALL EDI. This is done according to registers set.
Csi	Print all CALL ESI. This is done according to registers set.
Cbp	Print all CALL EBP. This is done according to registers set.
Ma	Print all arithmetic (ADD, SUB, MUL, DIV). This is done according to registers set.
A	Print all ADD. This is done according to registers set. This is done according to registers set.
S	Print all SUB. This is done according to registers set.
M	Print all MUL. This is done according to registers set.

Command	**Description**
D	Print all DIV. This is done according to registers set.
Move	Print all movement operations. This include all MOV, XCHG, and LEA. This is done according to registers set.
Mov	Print all MOV. This is done according to registers set.
Movv	Print all MOV value, where a value is moved into a register. This is done according to registers set.
Movs	Print all MOV shuffle, where one register is shuffled to another. This is done according to registers set.
L	Print all LEA. This is done according to registers set.
Xc	Print all XCHG. This is done according to registers set.
St	Print all stack operations (POP, PUSH). This is done according to registers set.
Po	Print all POP. This is done according to registers set.
Pu	Print all PUSH. This is done according to registers set.
Id	Print all INC and DEC. This is done according to registers set.
Inc	Print all INC. This is done according to registers set.
Dec	Print all DEC. This is done according to registers set.
Bit	Print all bitwise operations. This is done according to registers set.
Sl	Print All Shift Left. No customization is available due to paucity of results making it impractical.
Sr	Print All Shift Left. No customization is available due to paucity of results making it impractical.

Command	Description
Rr	Print All Shift Left. No customization is available due to paucity of results making it impractical.
Rl	Print All Shift Left. No customization is available due to paucity of results making it impractical.
All	Print all options.
Rec	Print only all Turing catalogue operations, the recommended setting for ease of use and speed.

Parsing of Input

The command line will provide tremendous flexibility in terms of the commands that can be input into the framework. Many of these commands will be settings that will change the results that are printed. For instance, on the main screen there will be an option for which registers to include in the search, or "get," as in the language of the JOP ROCKET. This will specify the registers that indirect jumps or calls should go to, as only some may be of interest to the security researcher. To provide input for the registers to "get," the user simply will need to enter the command r and then provide the desired registers, delimited by commas. The input will then be validated and then stored in a list. Once all other settings will have been finalized, the user then will be able to enter the command g for get, thus starting the process of searching for JOP gadgets. Similarly, in the printing sub-menu, the user will have an option to enter registers of interest for printing. This will work utilizing data that has already been obtained. If the user had sought only gadgets that end in JMP EAX or CALL EDX, then the registers entered on the printing sub-menu would pertain to the operations performed on just those registers. Thus, if EBX was selected as a

register to print, then gadgets such as MOV EBX, 5 / JMP EAX or MOV EBX, EDX / CALL EDX would be printed. All of this allows the security researcher to be very granular and specific as to exactly what they may be seeking.

Capture of Text Section

The framework will provide functionality to capture the text section. The text section is traditionally where the executable section is contained. Traditionally, executable code exists in the text section. The text section consists of the opcodes that provide the Assembly instructions to be executed by the CPU, beginning at the entry point specified by the PE file. It is possible to have the executable code be placed outside the text section or in multiple locations. This framework will not consider these, as it will only consider traditional, well-formed PE files. In the case of malicious programs or those with heavy obfuscation to protect intellectual property, there can be a concerted effort to complicate analysis for researchers; these will not be considered.

The framework will utilize the Pefile library to help extract the executable image's text section. This is the best option for the language used, and this work does not endeavor to recreate the wheel here, as these steps are well established. Once it has been extracted, other text sections from DLLs will be extracted for analysis. As a static analysis tool, by default it will extract just the executable image text section, but the user can select to add all DLLs. This may be useful if ASLR is not on a specific DLL or if the researcher has a memory disclosure to allow for an ASLR bypass. Then gadgets on those modules would be relevant. If the user decides to search the DLLs, then the JOP ROCKET will iterate through and capture the text section for each module. It will then perform a search for the specified target registers, storing the results in the appropriate data structures for later usage.

Search for opcodes

This framework will utilize and improve upon existing ROP and JOP algorithms that seek to find the opcode for a specific instruction, such as RET. Once found, it will disassemble backwards, finding all possible gadgets. Thus, the artifact will search for and enumerate all opcodes that produce the target indirect jump or call to a register. In x86 Assembly, there are three combinations that produce this. For instance, for CALL EAX, it will search for the opcode for CALL EAX, \xff\xd0, the opcode for CALL PTR EAX, \xff\x10, as well as the opcode for CALL FAR EAX, \xff\x18. The latter two are much less common, but this framework will endeavor to be comprehensive. With jump EAX, we have only one possibility to pursue, JMP EAX, or \xff\xe0. There are no indirect jump conditionals that lead to a register; they do not exist. Once the opcodes have been discovered, their location will be recorded in a data structure along with other necessary data. These will be used subsequently by other algorithms as needed.

Disassemble and Search

Once opcodes have been found, they can then be disassembled and then allow for searching. This has involved usage of the geometry of innocent flesh on the bone, as opcode-splitting was referred to as in the seminal journal article introducing ROP (Shacham, 2007). Again, this describes the unintended instructions that can be discovered due to the lack of forced alignment on x86. This concept can be applied to JOP, where we might find a gadget terminating in CALL EAX or JMP EAX. The JOP ROCKET must then discover all possible combinations.

To do this, it will address this problem in a novel fashion. It will start by generating a chunk of opcodes as specified by the Number of Opcodes. It will disassemble this chunk and then subject it to analysis. Once disassembly has been created for the chunk of opcodes, it will then be subjected to analysis. This will take the form of hundreds of lines of complicated regular expressions to

exclude gadgets that would be deemed impractical. Because opcode-splitting can produce a lot of unintended gadgets, some end up being highly impractical. Without exclusion criteria, there could be a significant number of useless gadgets. Having strict, carefully considered exclusion criteria will help to ensure these are not present.

After that analysis has been completed, the number of opcodes will be decremented, creating new chunk of opcodes ending in the terminating functional gadget, e.g. JMP EAX. This will be disassembled and added to a data structure for further analysis. This decrementing will continue until there are only 5 bytes of opcodes left. This process will be described more in chapter 4.

Search for Dispatcher Gadget

A dispatcher gadget is absolutely essential, as it is a required first step for JOP. The dispatcher gadget is a way to somehow advance or go backwards in the dispatch table in a predictable pattern. With ROP, the security researcher does not need to worry about such matters, as the nature of RET and the stack already provide want is tantamount to this functionality. When the researcher must create their own stack-like structure, the dispatch table, it needs a special gadget to direct control flow.

For this algorithm, the user will specify target registers to look for. For instance, the user might try to find a dispatcher gadget that advances a dispatch table whose address is contained in EAX. The search criteria would then try to find a gadget that ends in an indirect jump or call to EAX. It must also advance it in a predictable fashion; it could be adding or subtracting. The amount to be advanced is not as important, as padding can be used, within reason; the predictability is what is key. This advancing operation should ideally occur very close to the indirect jump or call. Each line of instruction between the ADD/SUB and the JMP [REG] runs the risk that other registers

could be clobbered. These registers then might be made permenantly unusable for functional gadgets or other purposes. Thus, the lines between should be kept minimal.

Practical dispatcher gadgets can be scarce. Therefore, the security researcher has the ability to increase or decrease the number of lines between the advancing operation and the indirect jump or call. This could be useful if an analyst needs to consider less desirable options if no other options exist. Once dispatcher gadgets have been found, then additional exclusion criteria will be applied, and those results will be stored in relevant data structures.

Data Structures

There will be hundreds of different data structures in this framework, based on specific classification criteria. This is necessary because when the search for gadgets operations is being performed, it will automatically search for everything, all gadgets that meet all classification criteria, storing the results in data structures, so that they are then ready to be of use immediately. There are unique data structures for every operation for which there is classification. For every operation there are data structures that hold all gadgets that perform that operation, and there are also data structures for operations that modify each register. Thus, for one specific operation, such as ADD, there will be different data structures for ADD operations that modify each register, as well as one that is a catch-all for all ADD operations. Additionally, each operation that modifies a particular register has four unique data structures associated with it, as described in the next section, Save to Data Structures. Thus, for just the ADD operation, there will be a total of 32 different data structures.

At no time will any disassembly be saved to any data structures. Disassembly will be generated as needed, whether for analysis or printing. The data structures only contain three numerical values as well as the name of the module.

Each data structure will belong to an object, and each object will be a part of a list of objects. The object will correspond to an executable image or one of its modules, and for each object, there will be distinct data structures maintained. The list of objects will correspond to the PE file itself, including the executable and all its modules.

Save to Data Structures

This framework will make extensive use of data structures. Because this will be a static analysis tool that does not build off of an existing dynamic analysis tool, such as WinDbg or Immunity, all that it has to go off of is a captured text section. Once the relevant opcodes have been found, the disassembly has been performed, and the classification and exclusion have been performed, then the JOP ROCKET will be read to save the necessary information to data structures.

For this program, some will be stored in lists belonging to an object, while others will be stored in lists that do not belong to an object, but primarily the former. Because of the nuances of Python, given that it is not a compiled language, this rather than a more complex data structure works best. The vast majority of data structures will belong to an object, and each object will correspond to the executable image or one of its modules. For each gadget that is found, there will be four parallel lists: listOP_Base, listOP_Base_CNT, listOP_Base_NumOps, and listOP_Base_Module. ListOP_Base is the location in the text section of the target JMP [REG] or CALL [REG]. This is the location where the chunk to be generated will end. ListOp_Base_CNT indicates the number of lines of instructions to go back. This is important because gadgets will come in many sizes. Having a gadget with a RET four lines before the JMP EAX would be impractical; starting after the RET would make it a potentially usable gadget. The JOP ROCKET whenever possible will reflect these nuances, so that the gadgets given to the user are clean.

ListOP_Base_NumOps will specify the number of ops that the binary chunk to be generated should be. This goes back to the idea of opcode-splitting, as different combinations of opcodes will produce completely different Assembly instructions. Finally, the listOP_Base_Module contains information on what module it is, whether it is the image executable itself or one of its DLLs, if applicable. The data structures will not store or contain any disassembly, only bookkeeping information so that the correct disassembly can be produced very quickly if requested by the user. There will be helper functions that save all needed data to the appropriate data structures.

Provide Disassembly for Printing of Gadgets

This framework will provide the disassembly in a novel fashion. It will employ helper subroutines, disHereJmp and disHereCall, to generate a binary chunk according to the bookkeeping information provided by the data structures. The gadgets then will be generated on the fly using this bookkeeping data to carve out and disassemble small chunks from the text section of whatever executable or module is needed. Disassembly as before will be performed by the Capstone library, which has become an industry standard and is used in over 400 tools. While there will be hundreds of data structures created, allowing for the security analyst to be granular and specific about what gadgets they may wish to seek, many may be of no interest. They will all have been populated, but the user may only be interested in a handful. They can provide input given their needs. Once input has been provided, then only those requested gadgets meeting the requested specifications will be printed. Given the number of registers and options, there are a couple hundred unique combinations available.

The disassembly will be printed to the screen as well as saved in files. The files will be classified according to operation and register. If an existing file of the same name exists, the framework will maintain the existing file and created another one, incrementing the version

number of the file. While printing to the screen is convenient, sometimes the sheer number of gadgets will make this impractical, making files far more convenient.

Summary

The JOP ROCKET will be constructed as part of a DSR inquiry into the research problem. The iterative cycle and the results obtained throughout the DSR process will help to continuously improve the artifact throughout development. This chapter has described the design of the framework and delineated its various components and its architecture. It has described how together they can allow for a novel, innovative solution to be found. The chapter has explained how this design science research conforms to the guidelines set forth by Hevner, et al., as well as Wieringa. The chapter has described how one goal of this research will be to employ classification and exclusion criteria to ensure the JOP gadgets are more usable and pragmatic. It has shown how the JOP ROCKET will allow for this to occur in an automated manner, via software, rather than through an arduous, time-consuming manual process. This chapter has also introduced the use of the single-case mechanism experiment as a means of demonstrating the utility and efficacy of this framework. Finally, the chapter has addressed how validity and reliability will be assured, and how validation will be performed.

CHAPTER 4

RESULTS AND ANALYSIS

The primary goal of this dissertation is to provide a full-featured artifact to assist reverse engineers and exploit developers in their ability to construct JOP gadgets, meeting a need for a sophisticated toolset. A close examination of the literature had revealed that there were no publicly available tools that provided this needed functionality, and this study effectively provides a validated solution to the research problem. Because of the greater complexity of JOP relative to ROP, creating a tool that can fully satisfy the needs of the exploit developer is much more challenging than it would have been to have created one of the similar, already existing tools for ROP. After all, it would be trivial to develop a tool that could search through disassembly to find all gadgets that terminate in an indirect jump or indirect call. However, such a tool would be severely lacking and be of only marginal use to the researcher, who would need much more help to relieve some of the tedium inherent in their efforts. A fully developed JOP artifact, one which has significant exclusion criteria, one that allows classification of gadgets, one that includes robust searching for dispatcher gadgets, is a more labor-intensive project, one that is fraught with myriad difficulties that need to be handled delicately. A high level of expertise is needed to carry such a project to fruition, and chapter 4 will showcase the results of those efforts.

The artifact developed in this research fully satisfies those requirements, as well as those set forth in the requirements specification. This chapter will discuss the results obtained by from the research that was discussed in the previous chapter. The research contributions are discussed in chapter 5.

Evaluating the Instantiation

As an instantiation, the JOP ROCKET is an implementation in the Python programming language of all the methods, brought together under a cohesive shell. It allows for different methods that individually would only contribute to part of the puzzle, to come together and collectively bring life to the instantiation. The JOP ROCKET follows what was set forth in the requirements specification provided in chapter 4.

Performance, reliability, functionality, supportability, and usability are necessary evaluation criteria (Wieringa, 2013). These attributes are used broadly as a way of determining how robust an artifact is. Reliability in software indicates that it should conform to the guidelines set forth in the requirement specification. In short, it describes whether the program works as the author intends and as the user would expect. Functionality describes all the different features and capabilities that a program can provide to its user. Functionality provides for an abundance of features to meet the needs of the software. Chapter 3 laid out in detail numerous functions, which were achieved in developing the instantiation. Additional functionality was also developed during the iterative development of this artifact. Supportability describes how easily the program lends itself to users or technical staff being able to troubleshoot issues that arise. The criterion of usability describes how easily a human can interact with the system. The ROCKET has a clean, intuitive interface with straightforward, simple language describing the features. Finally, performance relates well the program works under different loads; i.e. its speed and memory usage. The ROCKET does well with performance, although some large binaries with numerous modules would require a 64-bit installation of Capstone and Python due to memory constraints in the 32-bit version. As will be explored in greater detail below, this work does effectively embody all these characteristics.

When evaluating an instantiation, to try to identify the presence or absence of these features, we can look at code inspection, testing, and verification. Code inspection is intended to discover and allow for remediation of different defects that may exist in the source code. It can involve looking at different parts of the code to try to determine if it meets the set specifications. It also can address how robust the software is, looking at performance, usability, functionality, supportability and reliability (Wieringa, 2014). This work did not follow a formal process of code inspection with multiple individuals involved, but it utilized a more informal approach as adopted by the author, as he endeavored to address many of the elements of code inspection. Code testing is an important part of the process to ensure that the program works as intended and to ensure reduction of software defects. This can encompass many areas, such as how it handles widely differing input, whether it achieves stakeholder goals, whether it meets the requirements set forth, whether it is usable, and if it can perform its functions in an adequate amount of time (Wieringa, 2014). Finally, code verification is an important feature as it ensures that the results produced are accurate. Verification has been dealt with extensively elsewhere in this dissertation.

In looking at the five criteria for evaluating an instantiation, i.e. performance, usability, reliability, supportability, and functionality, we will endeavor to discuss briefly how this artifact measures up. We can assert that this artifact is relatively fast and efficient, in terms of its performance. The typical user will be analyzing one binary at a time, not a vast multitude, and the time frame for this is acceptable. When using Mona, a python script that can integrate with WinDbg or Immunity, on some occasions in can take in excess of 40 minutes to process a binary; others may take much less, and it depends upon system resources as well. Mona has many features, so this also depends upon the commands being used. The JOP ROCKET performs different tasks, but the time in which it does it is comparable, often taking 10 to 25 minutes to process a binary,

depending on its size. This includes the image executable as well as all modules. Formal metrics have not been used to measure the time, as it is felt that it is clearly well within an acceptable range, and no additional benefit would be conferred for it to be much faster. Additional work to improve speed would likely not amount to significant performance enhancements and would be of negligible value.

Performance can be attributed to a few different areas. First, the tool does not need to integrate with other software. Secondly, when it searches for different categories of gadgets, it is all done concurrently. That is, there is just one pass made for each running of the search algorithm for a specific indirect jump or call. The resulting bookkeeping information for all the gadgets are stored in appropriate data structures. A portion of the time could be reduced if printing to the screen was removed, as speed of terminal increases the time it takes, sometimes by as much as a few minutes, if it is a very large binary. That time could be reduced to seconds with just printing to files. Again, these numbers are all felt to be reasonable for the task at hand, so more rigorous measuring of time is not pursued.

The JOP ROCKET provides a high degree of usability. The user interface (UI) is accessible directly from the command line, thereby not requiring the use of a mouse, and the tool features a minimalist design. Where possible, the shortest possible keystroke combinations were used, provided they correspond to some sort of mnemonic. Thus, a frequent user effortlessly could memorize various keystrokes, enabling them to perform the desired operations. Much attention was placed on how to simplify how a user can interact with the program, accessing the needed functionality, while presenting this in a UI that is clean, simple, and intuitive. Such a UI would more likely correlate with a pleasant user experience (UE). A poor or broken UI can create a deeply unpleasant user experience, and while JOP is a complicated topic, this tool provides an effective

UI, allowing an interested user to successfully use the tool, even if they do not have a deep understanding of all the various options available. If a tool lacks usability, even if it has a high degree of functionality, then it may have little value, so efforts were made to ensure it was highly usable. Once the user got the feel for the UI for the JOP ROCKET, then likely they would be able to anticipate how certain features would be accessible. Whenever there was a doubt regarding how to proceed, the user simply need type h to see possible commands. Finally, for users who want to get all possible results with minimal UI interaction, the software provides them an opportunity to obtain reasonable results for everything with just a single command.

This tool meets the criterion for supportability. Repeatedly throughout the artifact, there are dozens and dozens of try catches, so that graceful degradation is handled; that is, the program can recover safely from exceptions that otherwise would make the program crash, allowing for the user to still get results and not have an interrupted experience. In a few cases, if less than desirable behavior occurs, descriptive error messages are provided to the user. For instance, in very rare cases when it cannot load a dll from its file location, it will provide output informing the user of this error. Additionally, the UI provides documentation in the form of a help interface that is implemented with straightforward, concise descriptions of different functionality.

Functionality is perhaps the most important criteria by which we judge a DSR artifact, and this does well to embody everything that is laid out in the software requirements specification. The need for some additional functionality also evolved over the course of the design cycle, and these would be implemented as their need was ascertained. This was typically happened with various helper functions and internal workings of the artifact. For instance, it was noticed that although imports from a PE were loaded, there were additional imports that would load once the PE file began execution. This is because the Pefile library only loads modules that are statically bound

through the Import Address Table (IAT). Thus, creative ways to find some of these missing modules were sought out, and this involved some slightly complicated logic that called for new helper functions and internal structures to be implemented. Functionality does not just encompass features, but it addresses how well these features work together and to what purpose. A feature may be ineptly implemented or prone to serious errors; this is the opposite of one that works flawlessly. Functionality was a chief consideration in designing the JOP ROCKET, and considerable testing was performed to help remove all known defects and to ensure various algorithms functioned smoothly without problems. If there was some, rare unusual case, the artifact is designed to gracefully handle the error, simply passing an exception. Functionality can also be measured by the quality and depth through which the JOP ROCKET meets its functional requirements, and it does fully satisfy them.

Reliability of software is critical, as software must be able to run given optimistic conditions, as well as those that are less so. To ensure reliability, a large number of test cases were done with the JOP ROCKET, as a frequent part of the iterative design cycle. This allowed for anomalies to be found, their causes discovered and then remediated. When it was not possible to completely rule these out, they were gracefully handled with try catches. The test cases used fell within the normal range, inclusive of both small and large PE files, with and without many DLLs. Because the artifact is an interpreted Python script, we need not worry about software defects, such as memory corruption bugs, that could be exploited, so from a reliability standpoint, such issues need not be addressed. Additionally, occasional regression tests would be carried out to see if any minor bug fixes or refinements may have introduced new defects, and when these had been identified, they have been remediated. Input validation has also been employed to ensure that

inaccurate user input doesn't break the software. In some parts of the software, this has taken the form of whitelisting acceptable input, with all other input being discarded.

Evaluating the Methods

This artifact is the realization not only of an instantiation, but also of several methods. As such distinct criteria comes into play concerning the evaluation of methods. With methods, we can look to consistency, completeness, and appropriateness as evaluation criteria. Completeness looks at how complete the results produced are, and it looks to the completeness of the software specification in terms of describing functionality. The specification must fully and completely address all aspects regarding the algorithms, with nothing left unstated. Consistency is in reference to the specifications not having any "internal contradictions" (Zowghi, etal., 2002). Correctness is viewed as the presence of both consistency and completeness; satisfying the stakeholder goals can also be described as correctness (Zowghi, et al., 2002). Those are the criteria, and they can be demonstrated in a number of ways, such as laboratory research, case studies, surveys, etc. This research will look to Hevner's evaluation criteria for a design science artifact, i.e. efficacy, quality, and utility. This will in part be demonstrated via validation techniques from Wieringa's single-case mechanism experiment.

Evaluation of the artifact's instantiation and its methods will be performed jointly, as in this artifact they are inseparable. To try to separate them for purposes of evaluation and validation would be overly artificial, and it would introduce unnecessary redundancy.

Artifact 1: A Method to Discover JOP Functional Gadgets

As the literature indicates, previous methods to discover JOP gadgets already exist. As chapter 2 discusses, there have been different paradigms for finding JOP, and the JOP ROCKET

exclusively uses the JOP dispatcher gadget paradigm. This dissertation presents an original approach to an existing algorithm. It is an enhanced variation on the algorithm for finding JOP. With this approach, it searches for a combination of opcodes for a specific indirect jump or indirect call. For instance, for CALL EAX, it would search for the opcodes for CALL EAX (FFD0), CALL PTR EAX(FF10, and CALL FAR EAX (FF18); these are shown in Figure 3. It would then search through the whole of the text section. Each time it found one of these combinations of opcodes, it would carve out a chunk of a bytes to be analyzed. The default is 20 bytes, although this could be changed to any arbitrary amount. For the opcodes carved out, it would search the resulting disassembly, perform classification, and save necessary bookkeeping information. The tool then would decrement the number of opcodes being searched, and the it would carve out those bytes, create disassembly, and then perform classification. This method would continue as long as there was at least six bytes of opcodes. After a point, there could sometimes be insufficient opcodes to generate meaningful disassembly, and this was chosen as a safe value.

This method is an original reworking of a simpler JOP algorithm (Bletsch, et al., 2011). The method presented in this artifact is more robust, and the inner workings of the method differ. It is assured that discovery will occur for each possible permutation that could be formed with chunks of binary that end in a desired indirect call or indirect jump. Because the attack surface with JOP is very small relative to ROP, it is critical every conceivable gadget be located and catalogued. The specific exclusion criteria and the usage of regular expressions is original as well.

Although this algorithm could include some redundancy, it ensured that no viable gadgets were missed. While there is value in ensuring all practical gadgets are located, sometimes the output generated would be highly impractical and of no value to an analyst, and accordingly exclusion criteria prevented such gadgets from being saved. For instance, *SBB BYTE PTR [EAX-*

0X5E5B10C4], *AL* would be of no use to an exploit developer. To deal with these, the tool makes use of a series of extensive regular expressions to exclude unwanted results that were felt to be nonsensical or far removed from the possibility of being useful. There is not functionality for the user to change this in the UI, although they could modify the source code as needed, to add to or detract from the exclusion criteria. Once desired gadgets were found, they were saved to appropriate data structures that indicated the location in the array of text where the indirect jump or indirect call was, the number of lines to go back, the number of opcodes to disassemble, and the name of the module, as shown in Figure 4.

```
7106 def get_OP_JMP_EAX( NumOpsDis):
7107     global o
7108 
7109     numOps = NumOpsDis
7110     while numOps > 6:   # Num of Ops to go back
7111         t=0;
7112         for v in objs[o].data2:
7113             test = ord(OP_JMP_EAX[0])
7114             test2 = ord(OP_JMP_EAX[1])
7115             if (ord(objs[o].data2[t]) == test):
7116                 if (ord(objs[o].data2[t+1]) == test2):
7117                         disHereJmp(t, numOps, "ALL")
7118             t=t+1
7119         numOps = numOps -1
7120     objs[o].listOP_JMP_EAX2 = copy.copy(listOP_Base)
7121     objs[o].listOP_JMP_EAX_CNT = copy.copy(listOP_Base_CNT)
7122     objs[o].listOP_JMP_EAX_NumOps = copy.copy(listOP_Base_NumOps)
7123     objs[o].listOP_JMP_EAX_Module = copy.copy(listOP_Base_Module)
7124 
7125     listOP_Base_CNT[:] = []
7126     listOP_Base[:] = []
7127     listOP_Base_NumOps[:] = []
7128     listOP_Base_Module[:] = []
```

Figure 2. Function get_Op_JMP_EAX

```
11 OP_JMP_EAX = b"\xff\xe0"
12 OP_JMP_EBX = b"\xff\xe3"
13 OP_JMP_ECX = b"\xff\xe1"
14 OP_JMP_EDX = b"\xff\xe2"
15 OP_JMP_ESI = b"\xff\xe6"
16 OP_JMP_EDI = b"\xff\xe7"
17 OP_JMP_ESP = b"\xff\xe4"
18 OP_JMP_EBP = b"\xff\xe5"
```

Figure 3. Hexadecimal opcodes for various JMPs.

Another value that is important in this algorithm is the LinesGoBackFindOp. This is a global variable that is utilized as part of the searching mechanism, and the UI allows the user to easily change this as needed. LinesGoBackFindOp allows the analyst to specify the depth to search

for gadgets. The depth refers to how many lines exist from the indirect call or jump to the target operation. The further back a line is from the indirect jump or call, the greater the chance that other instructions might be disruptive to registers that contain values that need to be preserved. Thus, by either enlarging or reducing the LinesGoBackFindOp variable, the analyst can cast a broad net or do the opposite.

Figure 4. A diagram of method addListBaseAdd

```
1175 def addListBaseAdd(address, valCount, numOps, modName):
1176     objs[o].listOP_BaseAdd.append(address)
1177     objs[o].listOP_BaseAdd_CNT.append(valCount)
1178     objs[o].listOP_BaseAdd_NumOps.append(numOps)
1179     objs[o].listOP_BaseAdd_Module.append(modName)
```

Figure 5. The source code for method addListBaseAdd

Once the desired gadget is found, bookkeeping information is stored in specific data structures, as shown in Figure 4 and Figure 5. There are three primary data structures as well as a less important forth once; each is populated with information once a functional JOP gadget has been discovered. The first three contain numerical values. The first is the location of the indirect jump or call. The second is the number of lines to go back; this is in reference to the number of lines between the target indirect jump or call and the target operation. The third is the numOps, which is the number of opcodes needed to produce the desired disassembly. NumOps is important because in x86, line enforced alignment does not exist, and therefore execution may begin in the middle what would have been an intended instruction. There is a global value for numOps, so the user can easily change this through the UI and affect the numOps in all the functions, while

allowing a user to also modify the source code at the function level if need be. However, the algorithm will discover all possible combinations of opcodes from the set amount of numOps, down to 6. The printed results also list the number of opcodes associated with a gadget. Finally, the forth parameter is the module name; this indicates the module that the gadget stemmed from.

```
3397    CODED2 = b""
3398
3399    x = NumOpsDis
3400    for i in range (x, 0, -1):
3401        CODED2 += objs[o].data2[address-i]
3402    CODED2 += objs[o].data2[address]
3403    CODED2 += objs[o].data2[address+1]
3404    CODED2 += b"\x00"
3405
3406    # I create the individual lines of code that will appear>
3407    val =""
3408    val2 = []
3409    val3 = []
3410    address2 = address + objs[o].startLoc + 1000
3411
3412    for i in cs.disasm(CODED2, address-x):
3413        add = hex(int(i.address))
3414        add2 = str(add)
3415        add3 = hex (int(i.address + objs[o].startLoc    + objs[o].VirtualAdd))
3416        add4 = str(add3)
3417        val =  i.mnemonic + " " + i.op_str + "\t\t\t\t"  + add4 + " (offset " + add2 + ")\n"
3418        val2.append(val)
3419        val3.append(add2)
```

Figure 6. The "carve out" portion of function disHereJmp

Now that we have a better understanding of some of what goes on in this method, we can provide a broad overview of how the method is implemented. The algorithm consists of two functions, a primary function and a helper function. The primary function, as show in Figure 2, first searches through the entirety of the extracted text section, for the opcodes that correspond to a target indirect jump or call. Once it finds an instance of one, it sends it to a helper function that carves out a chunk of disassembly based on the number of opcodes, as shown in Figure 6. This will then search for all the specific operations, of which there are dozens, and if found, it will record the proper bookkeeping information. It will then decrement the number of opcodes, without falling below 6, and iterate through a loop, generating all possible combinations of opcodes below the NumOpsDis, and allowing the resulting disassembly to be searched for specific operations.

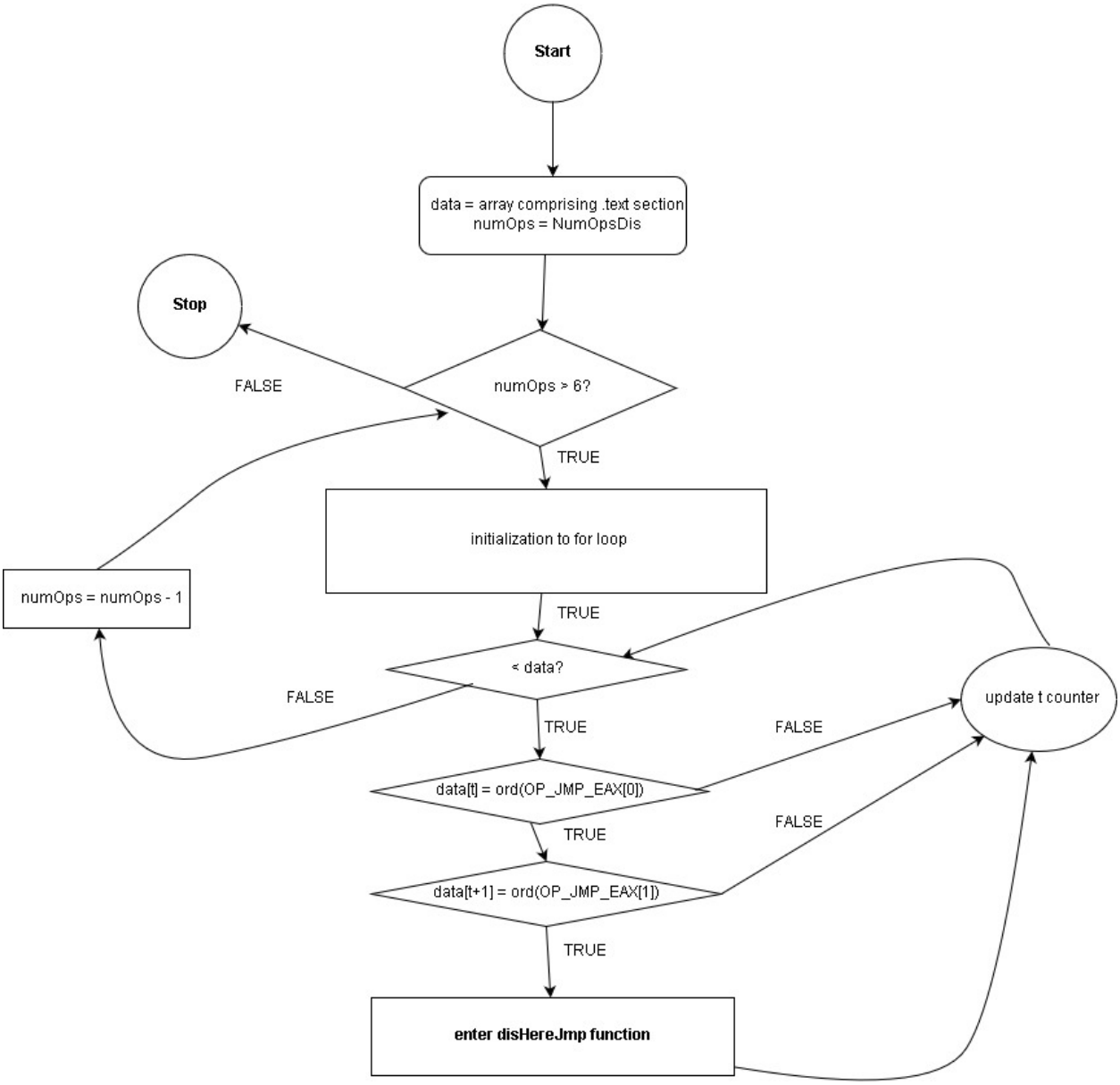

Figure 7. A diagram depicts the function get_OP_JMP_EAX

The method improved iteratively as it was subjected to a series of validation efforts. Frequently nuanced reworkings would emerge as desirable, and over a period of time these resulted in a profoundly improved version of the method.

Artifact 2: A Method to Discover JOP Dispatcher Gadgets

This artifact is also a reworking of a previously extant algorithm to make it more robust and useful. A dispatcher gadget is necessary as a critical first step in JOP. With ROP, we would use the stack typically which integrates with RET instruction, which will return from the function call and then execute the next operation or go to the next address on the stack. However, with JOP, the luxury of using RET is not available; instead, a functional gadget would jump to the location of the dispatcher gadget. The dispatcher gadget would then advance forward or backwards by a predictable amount a location in the target dispatch table, which would be where the dispatcher gadget would jump to, allowing us to reach the address for the next functional gadget. One very convenient dispatcher gadget would be ADD EAX, 8 / JMP EAX, where EAX would contain the location of the dispatch table. The dispatch table would then be populated with virtual memory addresses for functional gadgets, as well as filler, as there often may be wide gaps in between the addresses. Filler could consist of NOPs or functional NOPs. A functional NOP is one that effectively does nothing of importance, preserving registers as they were and not affecting control flow; an example would be MOV EDI, EDI, which Microsoft uses for runtime hot patching.

```
9508 def get_Dispatcher_G(NumOpsDis, howDeep, Reg):
9509     global o
9510     for obj in objs:
9511         t=0
9512         if Reg == "EAX":
9513             for v in objs[o].listOP_JMP_EAX2:
9514                 findDG_EAX(objs[o].listOP_JMP_EAX2[t], objs[o].listOP_JMP_EAX_CNT[t], objs[o].
                        listOP_JMP_EAX_NumOps[t], objs[o].listOP_JMP_EAX_Module[t],linesGoBack, howDeep)
9515                 t=t+1
9516             t=0
9517             for v in objs[o].listOP_CALL_EAX:
9518                 findDG_EAX(objs[o].listOP_CALL_EAX[t], objs[o].listOP_CALL_EAX_CNT[t], objs[o].
                        listOP_CALL_EAX_NumOps[t], objs[o].listOP_CALL_EAX_Module[t],linesGoBack, howDeep)
9519                 t=t+1
9520             t=0
```

Figure 8. Excerpt from function get_Dispatcher_G

```
5165 def findDG_EAX(address, valCount, NumOpsDis, modName, linesGoBack, HowDeep):
5166     global w
5167     CODED2 = b""
5168     numOps = NumOpsDis # was x
5169     for i in range (numOps, 0, -1):  #was x
5170         CODED2 += objs[o].data2[address-i]
5171     CODED2 += objs[o].data2[address]
5172     CODED2 += objs[o].data2[address+1]
5173     CODED2 += b"\x00"
```

Figure 9. Excerpt from findDG_EAX function depicting the "carve out" portion

The JOP ROCKET's method for finding dispatcher gadgets contains original contributions, and it improves upon the existing algorithm. First, it utilizes an the method of grabbing a chunk of binary to search and then iterating through all possible permutations, as partly shown in Figure 8 and Figure 9; this is the same technique as used in the artifact to discover JOP function gadgets. The method to discover a dispatcher gadget differs from the rest of the tool, in that the search for these is not done concurrently with all other searches. Instead, this algorithm works off of previously obtained results. This helps provide the analyst with much more flexibility, and the dispatcher gadget finding method takes additional parameters to allow for this.

Rather than being overly permissive about possible dispatcher gadget candidates, the algorithm has strict criteria for what is acceptable. For instance, with ADD or ADC, there is a requirement that it must add only a register or a numerical value, as seen in Figure 10. While others could possibly result in valid dispatcher gadget candidates, this helps reduce unacceptable results, which could make results too unwieldy to wade through.

The algorithm provides classification of dispatcher gadgets, which is an original contribution. First, it identifies all common, viable dispatcher gadgets. From this, it then determines which of these would be best. An example of a best dispatcher gadget would be one that adds only a small numerical value to the register that it will subsequently jump to. The best dispatcher gadgets are scarce. Finally, it includes an Other Dispatcher Gadget category, which

identifies some other gadgets that would be feasible but would require more careful, special planning and set up and maintain.

```
5198        matchObj = re.match( r'\badd\b|\badc\b|\bsub\b|\bsbb\b', e, re.M|re.I)  #do inc or dec separate
5199        if matchObj:
5200            matchObj = re.match( r'^[add|adc|sub|sbb]+ [e]*a[x|l|h]+|^[add|adc|sub|sbb]+ [byte|dword]+ ptr \[eax
                \]', e, re.M|re.I)
5201            if matchObj:
5202                matchObj2 = re.match( r'^[add|adc|sub|sbb]+ [e]*a[x|l|h]+, [e]*a[x|l|h]+|^[add|adc|sub|sbb]+ [
                    byte|dword]+ ptr \[eax\], [e]*a[x|h|l]+|^[add|adc|sub|sbb]+ [e]*a[x|l|h]+, [byte|dword]+ ptr
                    \[[e]*a[x|l|h]+|^[add|adc|sub|sbb]+ [e]*a[x|l|h]+, 0\t', e, re.M|re.I)
5203                if not matchObj2:
5204                    save2 = address
5205                    LinesSave = i+linesGoBack
5206                    addListBaseDG_EAX(save2, LinesSave, numOps, modName)
5207                    counter()
5208                    matchObj3 = re.match( r'^[add|adc|sub|sbb]+ [e]*ax, 0x[0-9a-f]+[0-9a-f]*|^[add|adc|sub|sbb]+
                        [byte|dword]+ ptr \[eax\], 0x[0-9a-f]+[0-9a-f]*', e, re.M|re.I)
5209                    if matchObj3:
5210                        matchObj3 = re.match( r'^[add|adc|sub|sbb]+ [e]*ax, 0x[0-9a-f]+[0-9a-f]+[0-9a-f]+[0-9a-f]
                            +|^[add|adc|sub|sbb]+ [byte|dword]+ ptr \[eax\], 0x[0-9a-f]+[0-9a-f]+[0-9a-f]+[0-9a-f
                            ]+', e, re.M|re.I)
5211                        if not matchObj3:
5212                            addListBaseDG_EAX_Best(save2, LinesSave, numOps, modName)
5213        if tk == HowDeep:   #This determines how far deep it goes searching for the Dispatcher gadget. That is,
            how many lines can be between the jmp/call [reg] and the add/sub
5214            break
```

Figure 10. Excerpt from findDG_EAX function depicting regular expressions to find the primary category of dispatcher gadgets

The Other category includes criteria for finding dispatcher gadgets, and this constitutes a more significant, original contribution. Because the number of dispatcher gadgets produced can at times be very slim or even non-existent, it is important to consider other less practical possibilities. We can look to Appendix A to see the few dispatcher gadgets found, which average to 0, for many of the registers, demonstrating the urgency to look at alternative sources for dispatcher gadgets. Firstly, the method considers multiplication in the form of MUL and IMUL. It the multiplication was a very small number, in some cases this would lend itself to a plausible dispatcher gadget candidate. For instance, if an analyst were to start writing to the heap, and then multiplied it by 2 or even 3, potentially a few to several addresses then could be hit, given the right set up. At some point, the numbers would increase too much, and there would be no way to go forward. However, if one were to find another functional gadget that subtracted a large, appropriate value from the register containing the location of the dispatch table, then the dispatcher gadget could then return

again to an area of the heap under the attacker's control. Of course, doing all this would require a special set up and very careful planning, but it is feasible. Unfortunately, gadgets that do multiplication are rare, so MUL and IMUL will be are rare, but they are considered in the code because no stone should be left unturned. It is anticipated that being able to use them in this fashion as a dispatcher gadget for a real-world application would be rare.

```
5218    for i, e in reversed(list(enumerate(val2))):
5219        tt = val2.__len__()
5220        if tk < 1:
5221            save = valAdd[i]   # This allows me to save the initial  address of the target jmp [reg] address.
5222        tk += 1
5223        matchObj = re.match( r'\bshl\b|\bshr\b|\bsar\b|\bsal\b|\bshlb\b|\bshrb\b|\bsarb\b|\bsalb\b|\bshlw\b|\b
            shrw\b|\bsarw\b|\bsalw\b|\b', e, re.M|re.I)  #do inc or dec separate
5224        if matchObj:
5225            matchObj = re.match( r'^s[h|a]+[l|r]+[dwl]* [e]*a[x|l|h]+, [1|2]+', e, re.M|re.I) #only 1 or 2
            because anything else is too inconcievable, and shift left / shift right can provide can provide
            other options.
5226            if matchObj:
5227                save2 = address
5228                LinesSave = i+linesGoBack
5229                addListBaseDG_EAX_Other(save2, LinesSave, numOps, modName)
5230        if tk == HowDeep:   #This determines how far deep it goes searching for the Dispatcher gadget. That is,
            how many lines can be between the jmp/call [reg] and the add/sub
5231            break
```

Figure 11. Excerpt from findDG_EAX function depicting regular expressions to find the "other" category of dispatcher gadgets

What is more practical and could be found in the wild, are gadgets that do multiplication or division through bitwise operations, namely shift left or shift right. Thus, the algorithm searches for shift left and shift right by values or 1 or 2, as it is felt that anything higher is simply too impractical, as can be illustrated in Figure 11. With shifting bitwise operations, the same logic can apply. Traditionally, multiplication and division have not been considered as useful for dispatcher gadgets, for obvious reasons. However, as previously explained, it is feasible that the attacker could advance forward or backward a very limited number of times and then get a functional gadget to subtract or add a large value from register containing the address of the dispatch table. This also means shifting operations could be feasible if the attacker can gain control over a very large area of the heap, because within the heap the dispatch table could be laid out with significant distances between the addresses for functional gadgets. Because dispatcher gadgets are necessary

for JOP to work, and due to the scarcity of these gadgets, being able to expand what is acceptable for dispatcher gadgets candidates could potentially make some binaries now potentially viable. These are binaries that otherwise may have been impractical due to paucity or absence of dispatcher gadgets. With JOP the urgency of consider all possible gadgets is no more true than with dispatcher gadgets.

The dispatcher gadget algorithm allows for needed flexibility for the analyst. Specifically, the howDeep variable indicates the distances between the indirect call or jump and the desired dispatcher opertion (e.g. add, sub, shift, etc.). Ideally, the howDeep should be within one or two lines, because the greater the distance, the higher the chances that other registers could be made un-usable or only usable on a limited basis. However, if no viable candidates can be found with howDeep within 1 or 2 lines, that amount can be increased easily in the UI. After all, it is possible gadgets may be found that would work in spite of a larger howDeep value. The default for this parameter allows for only two lines of instructions, so as to not inundate the analyst with results that mostly would be impractical.

In summary, this artifact is an original contribution due to its unique method of searching to discover dispatcher gadgets. It is an original contribution also because of its expansion to include other dispatcher operations, other than what had previously been used. It is an original contribution because of its use of regular expressions to exclude unsuitable gadgets as well as to classify dispatcher gadgets into distinct categories. Finally, it is an original contribution because of how granular the analyst can be when searching for dispatcher gadgets, to either expand or narrow the scope of what is acceptable.

Artifact 3: A Method for Printing Disassembly for JOP Gadgets

This dissertation presents an entirely new method, not based on previous work, for the printing of disassembly for JOP gadgets. Such a method could be extended to artifacts that make use of other code-reuse attacks, such as ROP. The JOP discovery algorithms described above result in pertinent bookkeeping information being saved to appropriately named data structures. There are approximately 600 data structures that save bookkeeping information for each object that exists in the list of objects. Each object holds all the information for one PE file, whether it is an executable or one of its modules. The extracted text section for the executable image and each module is also stored within each object. As described in previous sections, the contents of each of these data structures consists primarily of just a few values used to carve out correctly sized chunks of binary that are then disassembled. At no point are any opcodes or disassembly saved for any gadget. This is useful for many reasons, as it allows enhanced ability to search, manipulate data, and it does not waste memory storing many lines of gadgets, disassembly, or other printed information. No other software that prints out disassembly for code-reuse gadgets using similar methods is known to exist..

```python
def printlistOP_CALL_EDI(NumOpsDis):
    global o
    idval = 1
    while os.path.exists("%s-CALL EDI ALL_%s.txt" % (peName, idval)):
        idval += 1
    filename = peName +"-"+"CALL EDI ALL_" + str(idval) + ".txt"
    with open(filename, 'a') as f:
        total = 0
        for obj in objs:
            i=0
            cnt = objs[o].listOP_CALL_EDI.__len__()
            for i in range (cnt):
                print "\n*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^\n"
                print >> f, "*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^"
                counter()
                counterShow()
                addy =0
                cnt = 0
                num = 0
                addy = objs[o].listOP_CALL_EDI[i]
                cnt = objs[o].listOP_CALL_EDI_CNT[i]
                num = objs[o].listOP_CALL_EDI_NumOps[i]
                mod = objs[o].listOP_CALL_EDI_Module[i]
                out = "Ops: " + str(num) + "\tMod: " + str(mod)
                print out
                print str(addy) + "\t" +str(cnt)+ "\t" +str(num)+ "\t" + str(mod)
                sp()
                print >> f, Ct () + "\t" + out
                print >> f, disHereClean(addy, cnt, num)
            val =objs[o].listOP_CALL_EDI.__len__()
            total = val + total
            out = "# CALL EDI total: " + str(val)
            print out
```

Figure 12. Excerpt from function printlistOP_CALL_EDI

Each specific operation has its own printing function, as can be illustrated in Figure 12. Each operation has its own printing function because of the hundreds of different data structures used for the many possibilities. This is easier and simpler to read and maintain than a few with much more convoluted logic.

Each printing operation displays the disassembly both to screen as well as to disk. To utilize a function, such as printing SUB, the register would be passed as a parameter. Thus, it could retrieve all functional gadgets that perform SUB a specific register, such as EAX or EBX, or alternatively it could simply retrieve all SUB operations. This parameter will allow the appropriate data structures to be accessed to then print the desired output. Once the bookkeeping values were obtained from the appropriate data structures, they are then provided to a helper function, as shown in Figure 13, which then uses those values to obtain the necessary disassembly and then return it as a string, to be printed to terminal and disk.

```
4946    returnVal = ""
4947    trueVal2Cnt = val2.__len__()
4948    if trueVal2Cnt == valCount:
4949        for i in range (valCount):
4950            print val2[i]
4951            returnVal += str(val2[i])
4952    else:
4953        while trueVal2Cnt > valCount:
4954            del val2[0]
4955            trueVal2Cnt -= 1
4956            if trueVal2Cnt == valCount:
4957                for i in range (valCount):
4958                    print val2[i]
4959                    returnVal += str(val2[i])
4960    return returnVal
```

Figure 13. Excerpt from function disHereClean

Each operation is printed to an appropriately named file, with the output provided in a more aesthetically pleasing manner; sample output can be found in Appendix B**.** Segregating the specific operations to small text files named for the the operation in question is intended to make it easier for an analyst to find the gadgets they need, as they would not be required to wade through one large or even massive file. Representative, truncated output is included in Appendix B, for printed results from dispatcher gadgets and functional gadgets. Each line of output has both the virtual memory address location as well as the offset. In many cases, the offset will be more relevant, due to ASLR and other factors that will allow for the virtual address memory to be unpredictable. Each line in the output has both values, so the user can quickly obtain the needed offset, and not need to spend 10 seconds calculating it. For experienced users, calculating an offset is a nominal effort, but for more novice users, it could be another unnecessary source of confusion.

Other printing functions are simpler, if less data structures are involved, but all very similar in that they obtain the necessary values from data structures, provide these values to an original helper function, and then get strings of the disassembly returned.

The speed of printing output is fast. Depending on the size of the binary and its modules, it could range from 5 to 20 seconds for a binary with a smaller number of results, to a few minutes, for those with significant output. At the upper end, much of that time is in printing to terminal.

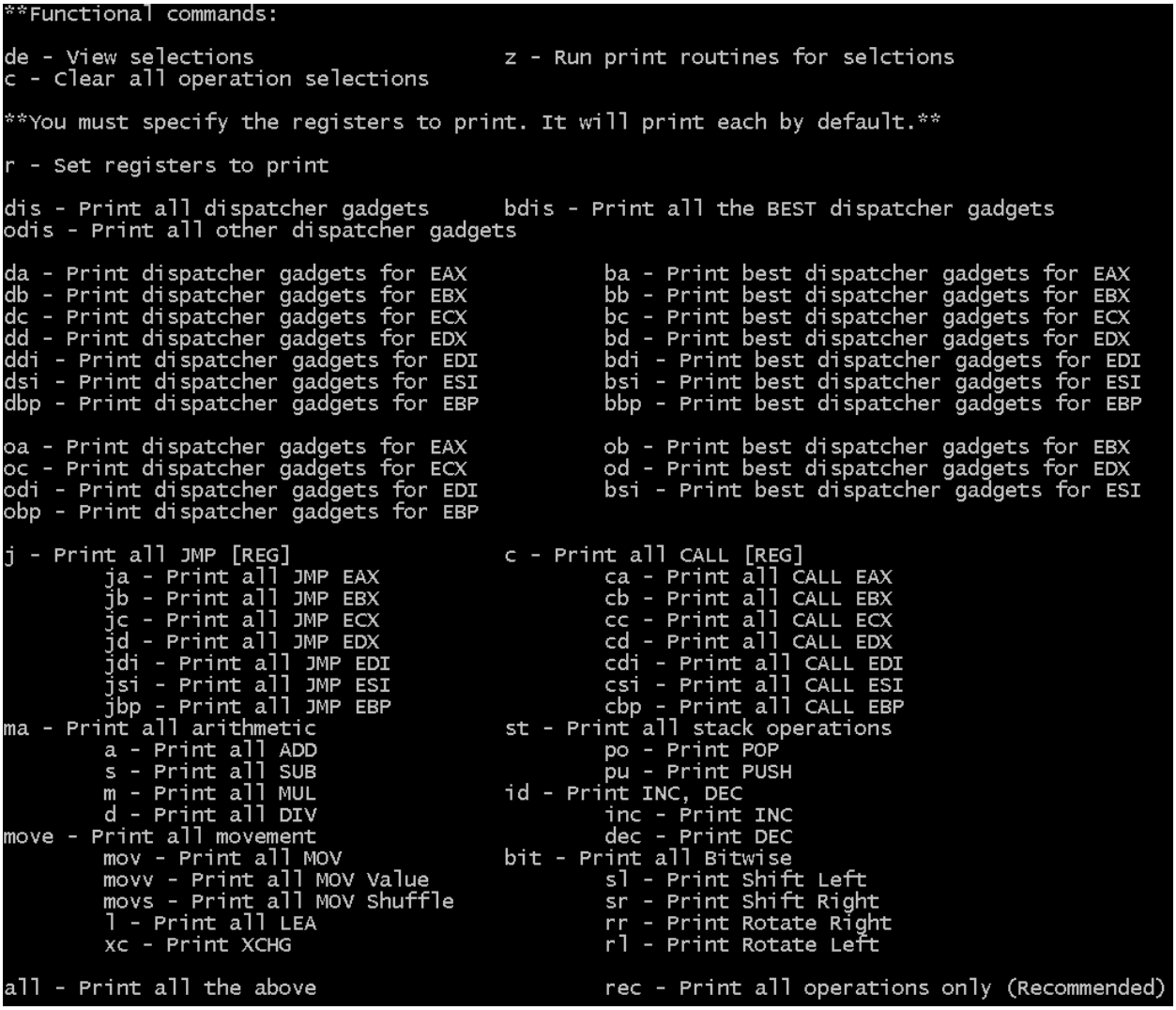

Figure 14. Print sub-menu options

The user is able to go to a printing menu, as seen in Figure 14, to select exactly what they wish to print, as there are numerous options with many variations. The user can easily enter input in an intuitive UI, utilizing brief keystrokes. With a few more keystrokes, they can obtain the highly customized results that they want. Again, emphasis here is on allowing the user to be very granular and specific about what they need, rather than providing a one-size-fits-all option for output. Thus, although all data has been obtained for all possibilities, they may wish to only print output for certain operations or that jump or call a certain register. With output, sometimes less is more, as time is not wasted on irrelevant data. Finally, the user can print a .csv file that shows the

number gadgets generated for each of the various operations. This can allow an analyst at a glance the opportunity to decide if a binary is likely to be useful for JOP and worthy of further investigation.

Artifact 4: A Method for Classifying JOP Gadgets into Categories Based on Turing Catalogue Features,

Many studies on code-reuse attacks have demonstrated Turing-complete catalogue features for ROP on various architectures. In fact, ROP has been demonstrated as a feasible in some form on all architectures, even voting machines (Shacham, 2007). This tendency to demonstrate Turning-complete features has lent itself to JOP as well. This research does not attempt to exhaustively enumerate the presence of all Turing-complete features. There is no interest for this research to enumerate gadgets that facilitate branching, nor are there any attempts within the artifact made to enumerate these, as they are not relevant for the types of attacks being envisioned. They certainly would be feasible, but they introduce a level of complexity that is simply unnecessary for this type of work. Their use on real-world applications, given the relative scarcity of JOP gadgets, would be highly minimal.

Finding or not finding Turning-complete features does not achieve any goals for this research; there is no vested interest to demonstrate these features exist. However, for the analyst, being able to rapidly access some of these as well as other classifications, in an organized, easy to use fashion, will simplify their work. For example, if an analyst is interested in adding or subtracting, they can quickly obtain the functionality that is relevant to them. They might be interested in adding to a certain register for an indirect jump or call to a certain register.

While many other tools have used methods to enumerate Turing features, it is believed this is the first that is done so with respect to JOP while using Python and regular expressions for classification and exclusion.

As previously described, the artifact searches for a specific indirect jump or call. To do this, it first searches for the appropriate opcodes. Once it finds these opcodes, it then sends the appropriate bookkeeping data to a helper function. This will carve out a binary chunk of opcodes and then produce the relevant disassembly. For instance, 20 bytes of opcodes might produce 4 to 8 lines of disassembly. This disassembly would end in the target indirect jump or call. Once this disassembly has been obtained in the helper function, it exists in the form of a list. The helper function then iteratively combs through each line in the list. For both the disHereCall and disHereJmp functions, there are approximately 800 lines of code, much of it regular expressions, to facilitate searching the disassembly. Figure 15 depicts a search in the disHereJmp function for adding to any register and adding to EAX.

```
3505                    addListBase(save, val2.__len__(), NumOpsDis, modName) # fist parameter: address of
                        target jmp [reg]; second parameter: number of lines to go back. third parameter: number
                        of ops to go back.
3506                    while lGoBack > 1:
3507                        try:
3508                            matchObj = re.match( r'\badd\b|\badc\b', val2[i-lGoBack], re.M|re.I)
3509                            if matchObj:
3510                                matchObj = re.match( r'^[add|adc]+ [byte|dword]+ ptr+ \[e[abcds]+[px]+ [+|-]+
                                     0x|^add [byte|dword]+ ptr+ \[e[abcds][px] \+ 0x|^add e[abcds][px], [
                                    dword|byte]+ ptr \[e[abcds][xp] \+ 0x|^add [byte|dword]+ ptr \[eax\], [
                                    al|eax]|^add [byte|dword]+ ptr \[ebx\], [bl|bx]|^add [byte|dword]+ ptr \[
                                    ecx\], [cl|ecx]|^add [byte|dword]+ ptr \[edx\], [dl|edx]|^[add|adc]+
                                    eax, [dword|byte]+ ptr \[[e|a]+[a|l]+|^[add|adc]+ ebx, [dword|byte]+ ptr
                                    \[[e|b]+[b|l]+|^[add|adc]+ ecx, [dword|byte]+ ptr \[[e|c]+[c|l]+|^[
                                    add|adc]+ edx, [dword|byte]+ ptr \[[e|d]+[d|l]+|^[add|adc]+ edi, [
                                    dword|byte]+ ptr \[[e|d]+[d|i]+|^[add|adc]+ esi, [dword|byte]+ ptr \[[e|s
                                    ]+[s|i]+|^[add|adc]+ ebp, [dword|byte]+ ptr \[[e|b]+[b|p]+|^[add|adc]+
                                    esp, [dword|byte]+ ptr \[[e|s]+[s|p]+|^[add|adc]+ a[l|h]+, a[l|h]+|^[
                                    add|adc]+ b[l|h]+, b[l|h]+|^[add|adc]+ c[l|h]+, c[l|h]+|^[add|adc]+ d[l|h
                                    ]+, d[l|h]+|^[add|adc]+ di, di|^[add|adc]+ si, si|^[add|adc]+ sp, sp|^[
                                    add|adc]+ bp, bp', val2[i-lGoBack], re.M|re.I)
3511                                # I am using regular expressions to eliminate what would be garbage gadgets,
                                    of which there would be countless, off the wall, unintended instructions
                                    that would do nothing of any practical value.
3512                                if not matchObj:
3513                                    addListBaseAdd(save, lGoBack, NumOpsDis, modName)    # Saving all add [
                                    reg]
3514                                    #eax - saving add to specific registers -- far more useful.
3515                                    matchObj = re.match( r'^[add|adc]+ [dword|byte]* [ptr]* [\[]*[e]*a[x|l|h]
                                        +|[add|adc]+ [e]*a[x|l|h]+', val2[i-lGoBack], re.M|re.I)
3516                                    if matchObj:
3517                                        addListBaseAddEAX(save, lGoBack, NumOpsDis, modName)
3518
```

Figure 15. Excerpt from function disHereJmp pertaining to the operation of adding to EAX

For each operation, there are many regular expression filters to select target operations, and additionally regular expressions are used heavily to discard results that would be regarded as impractical. Because the tool engages in opcode-splitting, it can produce some unintended gadgets that are simply irrelevant, such as *SBB BYTE PTR [EAX-0X5E5B10C4],AL*. Seeing similarly outlandish results would have little value to the analyst, only wasting their time to wade through such irrelevant results. With regular expressions, patterns for impractical gadgets can be determined and excluded. These specific criteria vary widely from gadget to gadget. Each run through this helper function iteratively subjects the list of lines of disassembly through all the regular expressions. Once an appropriate instruction is discovered that corresponds to a desired operation, all the relevant bookkeeping information is then saved to appropriate data structures, as explained elsewhere. In all, there are approximately 600 data structures to save bookkeeping data for all classifications.

In summary, this research includes an original method for classification of JOP gadgets. Some of the classification categories are based on Turing-complete features, while others are simply operations likely to be of interest for constructing a JOP exploit.

Faceted Classification

This research engages in faceted classification for software reuse by organizing knowledge into very specific categories (Prieto-Diaz, 1990). Not only does faceted classification classify the knowledge, in the form of gadgets, but it makes them available for near instantaneous retrieval after the classification has been completed. This research does this in part by making limited use of Turing-complete features, but it extends it further by adding additional, more granular classifications. Some of these sub-categories, such as for JOP dispatcher gadgets, have a grouping

that presents only the very best gadgets. These additional categories, outside the purview of Turing-complete, make classifications based on registers used and other relevant functionality.

Faceted classification helps to make sense of the number of gadgets that may be generated. One of the hallmarks of systems with faceted classification is that a user is able to very easily navigate and search through vast amounts of data that match up with the different orderings as provided by the facets. With adequate familiarity, users can intuitively circumnavigate the established categories and subcategories to retrieve the desired data.

With faceted classification, under the facets, we then have what are described as items (Zendler, et al., 2001). For dispatcher gadgets, we have 21 items. There are seven items belonging to Dispatcher Gadgets, seven for Best Dispatcher Gadgets, and seven for Other Dispatcher Gadgets, and each of these represents seven registers. The included registers are EAX, EBX, ECX, EDX, ESI, EDI, and EBP. There are facets as well for each operation performed by functional gadgets, such as ADD, SUB, MUL, DIV, etc., and each of these contains 8 items, each selected specifically due to modification of EAX, EBX, ECX, EDX, ESI, EDI, or EBP. There is also a final item that includes all the above. A user may aggregate queries from many different facets to obtain the desired data that may be useful for their unique needs for analysis.

With faceted classification, having a baseline is important, as this is the standard by which future efficacy of the experimented can clearly be demonstrated (Zendler, et al., 2001). There should be repeated measurements that can help ensure that the baseline is consistent. In terms of establishing a baseline, for this research we could look at it from two perspectives. First, we could look at the total number of indirect jumps and calls that are made. Secondly, we can look at the "all" item under each facet to get a baseline of the total number of gadgets found. From that, we can see how these break down into individual items based on register. Some gadgets that may form

a part of the baseline may not appear under items for specific registers due to exclusion criteria. Additionally, we could look at the baseline, from the perspective of all possible gadgets that would be found by a naïve implementation of a search for a particular operation, and then we could look at the results from a more sophisticated, mature implementation. This tool does not have maintain naïve implementations of different searches, as it would require significant extra work. Such data is irrelevant to security researchers, who simply want to be able to construct JOP exploits as easily as possible.

This research supports the idea that efficacy and utility can stem from the effective use of faceted classification with JOP. Appendix B provides exhaustive results from 32 binaries that were analyzed and classified by the JOP ROCKET. The total number of gadgets found under each item, are provided, for each of the binaries analyzed. Results are provided for just an image of the executable itself and collectively as the image and its associated modules.

Table 5. Faceted classification for gadgets that perform various operations

Dispatcher Gadget	**Best Dispatcher Gadget**	**Other Dispatcher Gadget**	**ADD gadget**	**SUB gadget**	**MUL gadget**	**DIV gadget**
EAX	*EAX*	*EAX*	*EAX*	*EAX*	*EAX*	*EAX*
EBX	*EBX*	*EBX*	*EBX*	*EBX*	*EBX*	
ECX	*ECX*	*ECX*	*ECX*	*ECX*	*ECX*	
EDX	*EDX*	*EDX*	*EDX*	*EDX*	*EDX*	*EDX*
EDI	*EDI*	*EDI*	*EDI*	*EDI*	*EDI*	
ESI	*ESI*	*ESI*	*ESI*	*ESI*	*ESI*	
EBP	*EBP*	*EBP*	*EBP*	*EBP*	*EBP*	

ALL	*ALL*	*ALL*	*ALL*	*ALL*	*ALL*	*ALL*

Table 6. Faceted classification for gadgets that perform various operations

POP gadget	**PUSH gadget**	**INC gadget**	**DEC gadget**	**MOV ALL gadget**	**MOV SHUFFLE gadget**	**MOV VALUE gadget**
EAX	*EAX*	*EAX*	*EAX*	*EAX*	*EAX*	*EAX*
EBX	*EBX*	*EBX*	*EBX*	*EBX*	*EBX*	*EBX*
ECX	*ECX*	*ECX*	*ECX*	*ECX*	*ECX*	*ECX*
EDX	*EDX*	*EDX*	*EDX*	*EDX*	*EDX*	*EDX*
EDI	*EDI*	*EDI*	*EDI*	*EDI*	*EDI*	*EDI*
ESI	*ESI*	*ESI*	*ESI*	*ESI*	*ESI*	*ESI*
EBP	*EBP*	*EBP*	*EBP*	*EBP*	*EBP*	*EBP*
ALL	*ALL*	*ALL*	*ALL*	*ALL*	*ALL*	*ALL*

Table 7. Faceted classification for gadgets that perform various operations

LEA gadget	XCHG gadget	SHIFT LEFT gadget	SHIFT RIGHT gadget	ROTATE LEFT gadget	ROTATE RIGHT gadget
EAX	*EAX*				
EBX	*EBX*				
ECX	*ECX*				
EDX	*EDX*				
EDI	*EDI*				
ESI	*ESI*				

EBP	*EBP*				
ALL	*ALL*	*ALL*	*ALL*	*ALL*	*ALL*

Table 8. Faceted classification for all gadgets that jump to specific registers

JUMP EAX	JUMP EBX	JUMP ECX	JUMP EDX	JUMP EDI	JUMP ESI	JUMP EBP
ALL	*ALL*	*ALL*	*ALL*	*ALL*	*ALL*	*ALL*

Table 9. Faceted classification for all gadgets that call specific registers

CALL EAX	CALL EBX	CALL ECX	CALL EDX	CALL EDI	CALL ESI	CALL EBP
ALL	*ALL*	*ALL*	*ALL*	*ALL*	*ALL*	*ALL*

Artifact 5: A Method for Statically Enumerating and obtaining modules in the import table and obtaining their JOP gadgets, while applying exclusion criteria.

Design science is highly iterative, and this dissertation research has developed and refined a method to statically enumerate all modules that will be loaded initially by a PE, including those not in the Import Address Table (IAT), and to allow for their JOP gadgets to be obtained. As a static analysis tool, it is difficult by its very nature to obtain all the modules that will be loaded by a PE file. To do this is typically a task best left for dynamic analysis tools, as all modules will already be loaded at the time of a search. With the JOP ROCKET, a static tool, the goal was to not only identify modules that likely were to be active once the binary became a process, but to also discover their respective file locations. This would allow the text section to be extracted from each of these and then searched for JOP gadgets.

A DLL can be loaded through implicit linking as an import from the IAT, and this is the most common way DLLs are loaded. The Pefile library provides functionality to easily enumerate these, but this would not include a number of modules loaded through other means. For instance, some of these other modules would be present upon a binary being loaded in WinDbg, yet they would not be present in the IAT. Additional ways to load DLLs include forwarded and delayed imports. These are not able to be obtained through the functionality built into the Pefile library. A forwarded library involves having the call to another function or library delegated to a different library (Winitor, 2010). A forwarded library will look like a typical exported function, but in the ordinal export table, it will indicate that it is forwarded. The end effect is a forwarded DLL will not actually be loaded until or if it is called (Chen, 2006). A delay loaded library is similar and is not loaded until they are actually called upon; these are not loaded as they are rarely needed (Alexander, 2017).

Explicit linking is more dynamic in nature. This can concern techniques such as using LoadLibraryEx and GetProcAddress to dynamically load it; such dynamically linked libraries would not appear in the IAT. Additional malware techniques are possible for dynamic loading of DLLs, such as utilizing the PEB and the PE file format to obtain the address for and load a specific library and desired API functions (Stroschien, 2018). As a static analysis tool, this work does not concern itself with enumerating modules loaded in a dynamic fashion. How some of this is done, particularly with the former, can be highly obfuscated at times and often occurs in malware, which is outside the purview of this work. The latter would be difficult and complex to discover purely through a static analysis tool, such as this. It would be possible with a more sophisticated, dynamic tool.

There were several problems with getting the desired functionality to work for this tool. First, Pefile would list only those DLLs inside the IAT, failing to enumerate others that would be loaded very soon after a process was loaded. Second, there needed to be a way to obtain the file locations for DLLs, so they could be loaded and have their text sections extracted. This functionality was not provided by Pefile. Third, after finding a way to obtain the file locations for modules, some location proved to be incorrect, with many returning the the Python module. This was used for the artifact, since it is the language the JOP ROCKET is written in, but it is unrelated to the target binary. Through the iterative DSR process, all of these problems were successfully remediated.

To find the file location of the modules, so that their text section could be loaded, Windows APIs were used to find the location of the DLL. First, LoadLibraryEx was used to load the DLL, and then GetModuleHandleW was used to return a handle to the DLL. This handle was then used with GetModuleFileNameW to obtain the module file location. When it would fail to return a handle and would return null, then it would then return a handle for Python, the language of the artifact. This was how Python was improperly enumerated with its JOP gadgets extracted. In those cases, the JOP ROCKET searched in the directory of the target binary as well as in System32 or SysWow64, based on whether or not it was a 32 or 64 bit binary, as seen in Figure 16. Additionally, the application's directory would be searched as well. More often than not, if it could not be found through Windows APIs, it was able to be found this way. Occasionally, some modules could not be found through the above means, and rather than risk getting improper results by having something such as Python be inadvertently returned as a handle and thus loaded, these were simply excluded. The user was provided with a notification the module was excluded from searching. This was rare, and it stems from the limitation of working with a purely static analysis

environment, while trying to do information that would typically be obtained in a dynamic environment.

```
885         if h_module_base is None:
886             directory = r'C:\Program Files\            '
887             directory = PE_path
888             newpath = os.path.abspath(os.path.join(directory, dllName))
889             if os.path.exists(newpath):
890                 module_path.value = newpath
891                 ans = newpath
892             else:
893                 if bit32:
894                     directory = r'C:\Windows\SysWOW64'
895                     newpath = os.path.abspath(os.path.join(directory, dllName))
896                     if os.path.exists(newpath):
897                         module_path.value = newpath
898                         ans = newpath
899                     else:
900                         print "\t\tNote: " + dllName + " will be excluded. Please scan this manually if needed."
901                         Remove.append(dllName)
902                 if not bit32:
903                     directory = r'C:\Windows\System32'
904                     newpath = os.path.abspath(os.path.join(directory, dllName))
905                     if os.path.exists(newpath):
906                         module_path.value = newpath
907                         ans = newpath
908                     else:
909                         print "\t\tNote: " + dllName + " will be excluded. Please scan this manually if needed."
910                         Remove.append(dllName)
911         head, tail = os.path.split(module_path.value)
912         if tail != dllName:
913             print "\tNote: " + str(tail) + " is being searched instead of " + dllName + "."
914             PE_DLLS[index] = tail
915             Remove.append(dllName)
```

Figure 16. Excerpt from extractDLLNew function, illustrating its ability find DLL file locations when handle module base is null

As a means of addressing some of these delayed imports, the artifact then extracted the imports of other imports. As discussed previously, more libraries than what was in the IAT would frequently be loaded immediately or soon thereafter, once a process was loaded. Once the imports of the imports were obtained, these would then be added to a list of modules to be extracted and searched. This initially gave the greater coverage, but it also included some DLLs that were not likely to be loaded in the target binary. When tested in a dynamic environment, such as WinDbg, it would be found that some of these modules would not appear. However, a commonality was observed to help explain this. Numerous DLLs located in SysWow64/downlevel and System32/downlevel that began with API-MS were among these that would be found through this method, but they would not typically appear as a loaded binary. Results from downlevel were thus

excluded, with the recognition that a static analysis tool has limitations, but that it is better to exclude these than to present gadgets from unavailable DLLs to the analyst. In addition, many of those DLLs were extremely small and likely to yield minimal if any gadgets.

Initially, prior to excluding downlevel modules, the number of modules could rapidly increase, producing a very inappropriately high number of modules. Many of these came from api-ms-win modules located in downlevel. On one testing machine, there were 111 of these api-ms-win modules in the SysWOW64/downlevel directory. Excluding these results produced results that were much more in line with reality, as they modules did not appear when examined in WinDbg

```
830 def noApi_MS(DLLs):
831     global Remove
832     for dll in DLLs:
833         matchObj = re.match( r'\bapi-ms\b', dll, re.M|re.I) # This is in system32 or wow64 /downlevel - not
            typically loaded at start. This is found by recursively searching imports and more likely than not will
            not be loaded right away. Thus, it is excluded.
834         if matchObj:
835             Remove.append(dll)
836         matchObj = re.match( r'\bpython\b', dll, re.M|re.I)  # Typically this is an error from not finding the
            correct  file location
837         if matchObj:
838             Remove.append(dll)
```

Figure 17. Excerpt from function noApi_MS, that drops APIs that will be represented by ucrtbase.dll

A final anomaly noted was that although various downlevel modules might be named in the IAT, all beginning with api-ms-win, such as api-ms-win-core-debug-l1-1-0.dll, the handle for these returned would always be ucrtbase.dll. Looking to WinDbg, it would be confirmed that it was the ucrtbase.dll that was loaded instead of these tiny, numerous downlevel modules. It is important to note also that ucrtbase.dll itself is located in the downlevel directory. This ucrtcase.dll is nearly the file size of the 111 api-ms-win DLLs, with a difference of only about 200 KB between ucrtcase.dll and all the api-ms-win DLLs. Without examining the content of each, it is likely safe to conclude the entirety of these 111 are in ucrtbase.dll, and that the 200 KB difference is simply related to bytes for the structure of the PE file across the various DLLs. Thus, given that Windows returns a handle for ucrtbase.dll when these DLLs are in the IAT, extracting ucrtbase.dll instead

of downlevel api-ms-win modules is the appropriate course of action and would be what was done once the process was loaded. And again, indeed, this behavior mirrors what is seen in dynamic analysis tools such as WinDbg. The JOP ROCKET has included logic so that if a handle is returned for a different module, such as ucrtbase.dll, it will be accepted; this provides coverage for other similar actions that may occur. DLLs that compromise part of ucrtbase.dll are excluded, as shown in Figure 17.

```
6978        for dll in PE_DLLS:
6979            test = extractDLLNew(PE_DLLS[i])
6980            head, tail = os.path.split(test)
6981            modName = tail
6982            i +=1
6983        PE_DLLS = listReducer(PE_DLLS)
6984        moreDLLs()
6985        noApi_MS(PE_DLLS)
6986        Answer  = set(PE_DLLS) - set(Remove)
6987        PE DLLS = list(Answer)
```

Figure 18. Excerpt from function obtainAndExtractDlls, which drops extraneous modules

By using the above technique to provide expanded coverage to discover other modules outside the IAT, some DLLS would be listed multiple times in the list. Accordingly, there is logic to truncate the list, so that there would only be one instance of a module in the list, as illustrated in Figure 18.

The above logic for static enumeration of modules is not perfect, but it provides for improved and more accurate coverage, than what had been initially created in earlier forms of the artifact, with missing modules or improper results, such as Python being loaded when the handle returned was null. By remediating these issues through the iterative process, the JOP ROCKET is now able to find additional modules and to ensure accuracy of file locations, allowing them to be extracted and scanned for JOP gadgets. Logic was implemented to exclude modules whose correct file location could not be found. This method of providing expanded coverage to statically find additional DLLs outside of the IAT, while ensuring correct file locations are being returned, serves

as an original contribution. This method could be adapted to other reverse engineering tools, for many diverse purposes.

The JOP ROCKET provides the user with the opportunity to search the binary in three ways. They can search for just the executable itself, the executable and only the modules contained in the IAT, or the executable, the modules in the IAT, as well as expanded coverage of DLLs. This final option is made possible by the original contribution that is this method, allowing for users to get more realistic results. Without the presence of this method, users would frequently miss out on modules that would be present when the binary loads, as some of these modules would not be in the IAT. With the relative scarcity of JOP gadgets, it is critical that these modules be enumerated, as this method has allowed, so that there JOP gadgets may be included among the results.

Artifact 6: An Instantiation of the Artifact

Perhaps the most important of the artifacts for this research is the instantiation itself, the JOP ROCKET. It is the consummate realization of all ideals for the perfect JOP tool, and it accomplishes this by adhering to the functional requirements specification. The instantiation is the culmination of the DSR inquiry into the research problem, and it effectively brings to life all the various methods, in the form of a tool that is accessible and usable, providing the desired functionality to the user. The evolution of the artifact has been iteratively developed, while following well-established design science principles set forth by Hevner, et al., as well as those that were provided by Wieringa.

This instantiation was constructed to be portable, allowing for it to be used across multiple platforms. Because the JOP ROCKET is written in Python, all that is necessary is an installation of Python, access to the command line, as well as all dependencies. The primary dependencies are

the Pefile and Capstone libraries, which are available across a variety of platforms. Sometimes it can be complicated to install these in certain environments, but it is not a great level of difficulty. Thus, the JOP ROCKET can be used to some degree on platforms other than Windows. The only limitation, as previously explained in chapter 3, is that outside of Windows, the executable's modules cannot be scanned, only the image executable. This is due to the use of Windows API's in the artifact, which would be absent elsewhere, and because the DLLs would not be available in their proper locations. A user of course can still scan individual DLLs, if they are running analysis outside of Windows.

Verification of Disassembly

Verification of disassembly is paramount, as it is one of the key features of for the JOP ROCKET, as it is the disassembly that is used for the JOP gadgets. From this, we can look at it from two perspectives. First, there must be assurances that the addresses and offsets provided for each line of gadgets is accurate. Occasionally during development, there were issues that crept up with the address or offset not being properly aligned. Once it was noticed this was occurring, corrections were made. Next, the principal concern was that the disassembly was correct, regardless of address. This was especially true with opcode-splitting. After all, one could not simply go and check in IDA Pro by attempting to look at the line prefix for the intended address, as IDA Pro only displays disassembly for intended instructions, though it is possible to change this. For greater simplicity and reproducibility of results by others, to verify that the addresses and offsets were correct, both IDA Pro and WinDbg were made use of.

To ensure that the correct disassembly was produced, there were two primary approaches. First, we could go to the location in IDA Pro and look at the opcodes to see if they aligned with

what was in the Defuse Online x86 /x64 Assembler and Disassembler (Defuse, n.d.), as shown in Figure 19 and Figure 20. Thus, it could be determined if the opcodes matched up, and one could use the Defuse tool to see if the correct lines of opcodes produced the disassembly. Alternatively, one could simply take the offset and the base address of the module in question, and enter *u [address]* in WinDbg. That would indicate if the disassembly produced by the JOP ROCKET and WinDbg were a match, as shown in Figure 21 and Figure 22.

Figure 19. IDA Pro confirms that these are unintended instructions.

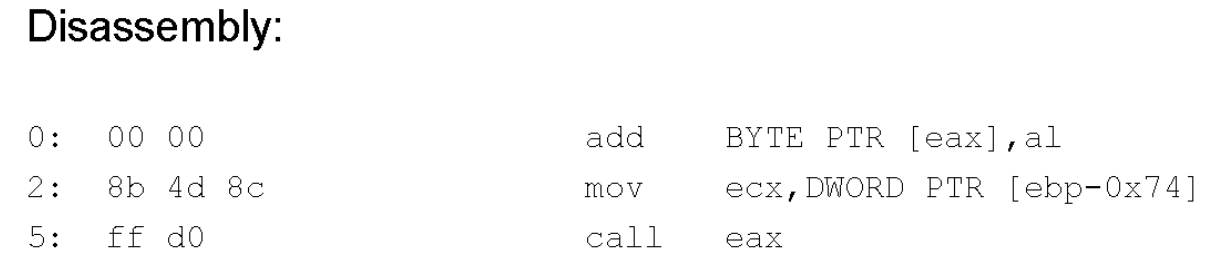

Figure 20. Defuse Assembler and Disassembler confirms that those opcodes do produce the unintended instructions provided by the JOP ROCKET.

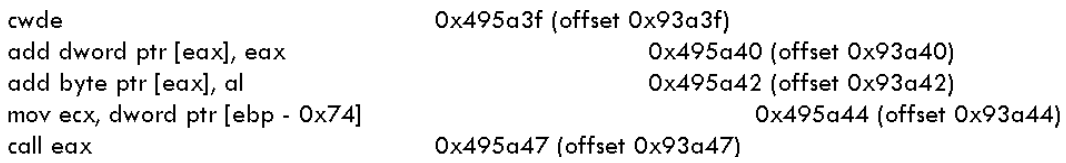

Figure 21. The JOP ROCKET provides output for a gadget. The gadget is created using unintended instructions.

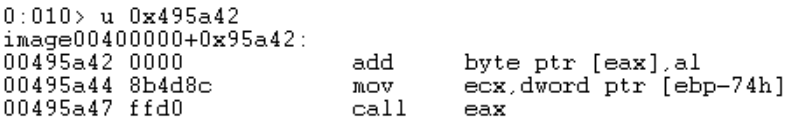

Figure 22. By using the u command in WinDbg, we can see what Assembly instructions would be executed if we began execution at the address provided.

Throughout the development process, there were numerous changes to various functions, structures, or helper functions that had an adverse effect on the disassembly, introducing different complications that caused inaccurate disassembly. Once these were identified, they were remediated. The JOP ROCKET in its current form is well tested and produces only accurate disassembly for the gadgets, as shown in Figure 21. The verification methods leave no shadow of a doubt as to the accuracy of the gadgets produced by ROCKET.

V**alidation**

Validation is a necessity before a new technology can mature sufficiently to be able to be of use in the market. New technologies such as the JOP ROCKET must be subjected to testing with appropriate simulations that replicate similar conditions that would be used in a real-world context (Wieringa, 2013). An empirical method for validation can be performed here with a single-case mechanism experiment. With validation, we endeavor to simulate its use in a context similar to what will be used, to see if it satisfies the needs of stakeholders.

Conceptual validation is the first stage of validation, where calculations and examples are worked through to try to access the validity (Wieringa, 2013). Then comes the modelling stage, where experiments are done with artifact. As the JOP ROCKET was being developed, significant informal testing was performed. Finally, we get to real-world field testing. Here the JOP ROCKET is used as intended on actual binaries.

For validation, we primarily focus on the single-case mechanism experiment as presented by Wieringa (2013; 2014). Much of what we can do also can be demonstrated via the work with Hevner, et al., and owing to their prominence, we discuss that later in chapter 4. For purposes of validating software, other similar methods can be used to achieve the same ends. Zelkowitz and

Wallace (1998) cover much of the same area in what they call a simulation or dynamic analysis, while Glass et al. (2001) cover the same province with what they refer to as field experiment, laboratory experiment – software, or simulation. While these other methods are appropriate to use for validation, what Wieringa describes as the single-case mechanism experiment provides significant potential overlap and is mutually consistent with the aforementioned validation methods.

A validation model consisting of a model of the artifact and a model of the context if fully validated should generalize to the implemented target as well as the intended real-world context (Wieringa, 2013). As the single-case mechanism utilizes contexts going from idealized to more realistic conditions, the artifact becomes more robust.

Single-Case Mechanism Experiment

Under both Wieringa and Hevner, et al., validation of new design science artifacts is critical. Validation is provided using a single-case mechanism experiment, where a simulated use of the JOP ROCKET is performed. The results it obtains are then evaluated for the applicability. They show that the JOP ROCKET produces the desired results and output via a controlled experiment. As this is very valuable information to have for a researcher wishing to do a JOP exploit, it would strongly imply that the tool would be of immense value to such researchers. Absent this tool, there would be no automated way to generate all the gadgets divided into appropriate classifications. Thus, the idea of asking users or potential users if they find value in the tool would be redundant, since without the tool, there would be no other options except for laborious, tedious manual discovery of gadgets. Here we have performed validation of the methods as well as the instantiation of the framework, and the results speak for themselves.

This single-case mechanism experiment is often used as a means to perform validation, according to Wieringa, and that has been the case here. With implementation evaluation, a tool may be tested for purposes of being able to analyze the tool's architecture. Algorithms may also be tested with a real or simulated context (Wieringa, 2014). The research context is important for the single-case mechanism experiment. We can view as part of the knowledge goals, within specific target binaries, the need to determine what are the JOP dispatcher gadgets and the functional gadgets. More broadly, the results of the single-case mechanism experiment may provide insights into the prevalence or infrequency of certain categories of gadgets.

A single-case mechanism experiment is used with the knowledge goal of validating new technology, along with the improvement goal of developing a new or better form of technology. With this research, the improvement goal has been to develop new methods to facilitate the discovery of JOP gadgets, or where some methods already exist, to iteratively develop better, more accurate, more relevant versions of those methods. In validation research, a knowledge question might concern whether or not the effects of the artifact prototype interacting with a simulation of the context can satisfy the requirements laid out. The results obtained from the single-case mechanism experiment clearly demonstrate this has been achieved with the JOP ROCKET.

JOP Dataset

The artifact was intended to allow exploit developers to analyze a target PE file, with the intent to discover all useful JOP gadgets. As part of the validation effort, to demonstrate the efficacy and utility of the artifact, it is necessary to test the tool on PE files. The PEs selected cover a broad range, from small to large executables, each with a few to numerous modules. In all, 32 PE files were subjected to analysis. A broad range were examined, to try to mimic natural working conditions. These included Windows system binaries, such as Notepad, as well as large and small

binaries that were both commercial and open source. In the design of this artifact, the choice was made for it to target 32-bit applications, and accordingly the 64-bit PE32+ files are excluded. The research here does not seek to draw conclusions on specific categories of binaries, e.g. large or small commercial binaries, binaries made using certain compilers, binaries that employ have object-oriented C++, etc. The dataset exists simply to demonstrate the instantiation and its methods work as intended and can contribute some practical utility to the intended user. Different research efforts that employ quantitative research methods could perhaps make more relevant discoveries on the nature of JOP in the 32-bit Windows environment, but that is outside our scope.

The number of viable JOP gadgets vis-à-vis ROP gadgets is much smaller, and for some smaller or even large binaries, the attack surface may not be large enough to realistically mount an attack. It is important to include these binaries in the dataset. Some binaries, due to their size, have an attack surface that is simply too narrow to realistically accommodate an attack, as there may not be enough strong gadgets in plentiful numbers. Because of the relatively small attack surface that may be the case with JOP, the JOP ROCKET has gone to extreme lengths to make sure all possible gadgets can be found, even those that while less than desirable, were still feasible. Additionally, on the same note, the artifact has gone to great lengths to provide order and classification, removing junk gadgets (some completely unrealistic, derived from opcode-splitting), so that the good ones can be found, rather than they suffer the fate of a needle lost in a haystack. After all, if there 8000 gadgets for jump to EAX and only a small number may be viable for a specific purpose, it does no favors to the analyst, when there is no simple way to make sense of the deluge of gadget and locate what is sought. Clearly, that would be suboptimal, and so this artifact has eased that burden.

The JOP ROCKET can be used in essentially three modes when searching for gadgets: the PE file itself, the PE file and the imports contained in the IAT, and the PE file and expanded imports. Having more modules than simply the executable itself potentially could greatly expand available gadgets, or make available gadgets that may lack protections. It is important to consider that many of these DLLs, as well as perhaps the PE file itself, may have protections in place, such as DEP, ASLR, SAFESEH, or CFG, which may make them unsuitable, unless a bypass exists. A bypass could include memory disclosures, to overcome ASLR, and then other protections potentially could be overcome. This research does not consider these possibilities. Like many other tools, it can indicate if a particular PE or module has any of these protections. The research is undertaken from the perspective that even if all the protections were in place, potentially they could be overcome by a dedicated attacker, given the right set of circumstances. In more practical terms though, the presence of some of these protections may mean some attacks may not be viable or the level or difficulty would be too high. Clearly, it would be impossible to tell if an attack would be feasible from a static analysis tool such as this.

The artifact can generate a .csv with a breakdown of all the numbers of different gadgets, which serves as a useful as a way of easily examining the results of the single-case mechanism experiment. The results obtained from the dataset are representative of what may be commonplace with typical PEs. Looking at the dataset for a binary may provide an indication whether or not JOP may be potentially viable. Ultimately the only way to know for certain as to examine representative gadgets. Looking at the .csv may allow an analyst to make an informed conjecture as to whether or not a binary is likely to be feasible for JOP.

Validation Model

In validation research, the Objects of Study (OoS) is the validation model. This consists of the artifact prototype as well as a model of the context (Wieringa, 2013). In this wok, the artifact prototype is the Python instantiation of the framework as well as the different methods. The context used is a real-world context. Actual binaries that an exploit developer might target serves as the real-world context. A simulated context would be toy binaries constructed solely for the purpose of validation. With OoS, there is a need that the validation model satisfies the population predicate, and in this research the validation efforts employ real-world contexts (Wieringa, 2013).

In order for a validation model to be valid, it should support inferences that are analogic, descriptive, as well as abductive. The validity of these inferences are then assessed during the empirical cycle. For descriptive inference, we must define the descriptive inference (Wieringa, 2013).

Validation requires that data preparation be addressed; there should be an answer as to whether or not data prepared data represent the same phenomena as unprepared data (Wieringa, 2013). With the JOP ROCKET, we find that unprepared data would indeed represent the same phenomena. However, it would be in a form where manual methods would need to find these, whereas the JOP ROCKET provides the prepared data in an automated fashion, as files containing gadget and as a .csv file that provides a summary of all the different JOP gadgets found, broken down into specific classifications. We also must show if interpretations formed, e.g. classification of gadgets, may be regarded as facts. Indeed, the JOP gadgets found and their classifications are performed in a highly objective fashion, where there is no room for opinion.

Repeatability is also a necessity with validation. Other scientists also need to be able to utilize this work to reproduce the validation (Wieringa, 2013). An analyst making use of this

framework with the same binaries and the same options, would obtain the same results. Another, independently formed tool would likely find the same results. Many of the results obtained have been repeated numerous times throughout the testing cycle to ensure accuracy and as the artifact has been improved, so there is no question as to repeatability.

Validation models additionally must support abductive inferences. This addresses whether the data given can support the explanations provided. It should also address treatment control. That is, it must show if the setup of the experiment or even the experiments themselves influence the validation model (Wieringa, 2013). With respect to abductive inferences, with the JOP ROCKET, the inferences drawn have been minimal and only used to better inform the design of the artifact. There is little that needs to be done with respect to treatment control and concern with the experiments influencing the validation model, because the results obtained can objectively and decisively be determined as accurate and correct, and there appears to be no undue influence on validation.

Sampling

The sampling is representative of many types and has been carefully selected, with only relevant, modern PEs are selected. These are tested on a current, updated, Windows 10 PC outside of virtualization. The results would be accurate on any operating system. The artifact is to be used on a large representative sampling of PEs using all available options (all registers selected, all dispatcher gadget registers found, printing all operations, and obtaining statistics in .csv format on all the data obtained). This is performed with all the default settings, which are deemed reasonable. Modification of defaults would produce different results, but the further we go from the defaults, the less likely they are to be truly representative of the binary. The resulting binaries are then searched for anomalies and errors in reliability.

Context

In this validation, we have the instantiation and its methods as the artifacts, and the treatment is subjected it to a realistic context—namely, actual, real PEs that feasibly would be used in the wild. The treatment can be viewed as inserting an artifact into a context (Wieringa, 2013). The treatment with this experiment is the request to identify relevant JOP gadgets in target binaries. We could also view the treatment from the perspective of a security researcher investigating target binaries (context) through the use of ROCKET. The measurement is the identification of these gadgets and the precision and accuracy with which it is done, in the form of correct addresses and offsets.

Throughout design, a variety of select real world PEs were repeatedly analyzed, some having been run through various algorithms hundreds of times, as the instantiation and different methods evolved and were continuously refined. Constructing a toy app would be tedious and unnecessary. However, in some cases, certain classifications of gadgets were rare, and in the development, these were treated with a simulated context, as finding a real-world context would be challenging, particularly with respect to testing the many variations of acceptable gadgets, not to mention implementing exclusion criteria. The simulated context consisted of real-world chunks of disassembly that were then subjected to testing in a separate, small Python program with limited functionality, whose only purpose was to test that the regular expressions worked exactly as intended. This was often a highly iterative process, with many changes being made, until refinements were made just right..

It should be stressed that the limited use of validation throughout the development process with a simulated context was not intended to be repeatable by other researchers. Once the tool had

matured and was refined, this testing was no longer necessary, and a real-world context, i.e. actual PE files, was then used.

Execution of the Validation

Sample construction was routine. A large, representative sampling, meeting the aforementioned loose criteria, was selected for final validation. Months of iterative validation had gone on previously, both using real-world as well as simulated contexts, in order to arrive at this point. A text file was used as input containing the file location for a target binary. This was run on the command line with the command *python prog.py input.txt*. By default, all binaries supplied via file is subjected to the most complete analysis possible, with all options selected, generating all output.

Unexpected Events

Once the treatment was applied and the execution began, some unexpected events occurred. A couple DLLs were included twice, and this had been overlooked. The source of the error was found and rectified. This was part of the algorithms to expand coverage of DLLs beyond what was in the IAT. There had been a reduction algorithm that removed repeated DLLs, but some logic had not been considered, allowing for some DLLs to persist.

Prior to successful execution, some bugs from the latest revisions that had not been caught were discovered. Throughout the iterative process, more and more features and functionality have been continuously added, and some last-minute additions to help streamline the validation had results in a couple minor bugs. These bugs stemmed from changes to facilitate streamlining the validation efforts, and they did not pertain to core functions. Once these were corrected, execution ran smoothly.

For a few binaries that were to be tested, memory ran out. It is believed this is due to using a 32-bit version of Capstone and Python, thus limiting available RAM. These were very large binaries, each with more than 24 modules, some fairly large. Better memory management may also alleviate this, but it is also possible they may reach the limits of what can be done with 32-bit. It is believed that moving to a 64-bit version of Python and Capstone would eliminate this issue. This would involve changes to the environment, changes that are not straightforward. As the JOP ROCKET is being developed outside of a VM, this has been deferred to a later date.

Treatment Validation

Treatment validation can be achieved through a number of means, and this is an important requirement that needs to be met under Wieringa's theories on design science. Treatment validation must determine if the context and artifact can produce the desired effects. With this research, the question has been answered as to whether or not using the JOP ROCKET with actual binaries in a simulated environment can create the necessary gadgets; this clearly has been achieved. Additionally, treatment validation should address whether these effects can then satisfy the requirements that were previously delineated in chapter 3, and indeed that has been the case.

Data Analysis

Having ran the tool against some 32 binaries, we can now engage in data analysis, by looking at the key areas described by Wieringa for data analysis. Some representative data is included in Appendix B, which will be useful to review while examining the data analysis.

While a much larger dataset could allow us to make more statistically sound generalizations about the nature of JOP in modern 32-bit binaries, that is not the point of the research, as the overarching goal here is to create a tool to facilitate pragmatic work with JOP for exploitation, not to perform

an exhaustive, quantitative study on JOP. Given these constraints, we make only tentative, very general speculations on the dataset.

Descriptions

Execution times would take up to 20 minutes or more for some binaries. This period of time involved searching through many modules for gadgets, as well as searching imports in the IAT. In some previous testing, some options were more limited, e.g. searching only for indirect calls or jumps, or for only certain registers, and this resulted in less time.

The figures obtained from the framework were in line with expectations. Indirect jumps or calls to certain registers were much more plentiful, whilst others were much less so. Those that tended to be more or less plentiful tended to be so across multiple binaries that were tested. Additionally, the application of faceted classification to the data resulted in some categories being significantly more plentiful than others, and this as well is true across multiple binaries. Some categories tend to have few or even no gadgets associated with them, due to general scarcity. Looking at the dataset, it is speculated that across a number of binaries, many categories may exist only in limited numbers. Additionally, it is to be noted that some modules have far fewer gadgets than others.

JMP EAX is far and above the most plentiful of the indirect jumps, with an average of 413.9 gadgets per binary, while JMP EDX has the next highest at 77.9. JMP EBX as the least plentiful of indirect jumps has an average of 1 gadget per binary, while indirect jumps to other registers on average are in the single digits to low tens. Indirect calls, on average, are more plentiful, with CALL EAX on average having 5276.9 and CALL EDX the next highest at 4684.5. CALL EBX is still abundant at 1665.8, and CALL EDI and CALL ESI are as well, at 2112.2 and 2277.3 respectively. CALL ECX and CALL EBP are the least plentiful at 272.3 and 86.2. While

some of the figures are relatively high, there can be wide variance across different binaries, where one binary might have gadgets for a particular indirect call in the thousands, and another might have them in the low hundreds. Some gadgets on average also tend to have very low numbers of any indirect calls or jumps, while some outliers may have vastly more, with over tens of thousands of gadgets.

Some operational gadgets are much more common, while others are much less so. Ones that are plentiful include ADD, SUB, MOV, LEA, PUSH, POP. Ones that are much less frequent but exist in sufficient quantities to be useful include INC, DEC, XCHG, shift left, and shift right. Ones that are much more limited or even rare include MUL, DIV, rotate left, and rotate right. For those operations identified as rare, there simply may be no viable gadgets in those facets, although the functionality could be emulated in other ways, e.g. shifting left or shifting right.

Some dispatcher gadgets pertaining to certain registers were limited or rare. These are the key gadgets to make JOP possible. In no case were any of the DG Best category found; these are dispatcher gadgets that would be guaranteed to work with minimal effort and an ideal setup. The DG Other subcategory also had limited numbers. Some dispatcher gadgets to certain registers had little to no representation in smaller binaries.

Often there seems to be a relatively small number of functional gadgets, compared to ROP. This coupled with the limited number of JOP dispatcher gadgets means the attack surface for JOP is much more limited than it is for ROP.

In terms of the artifact itself, all algorithms and methods performed as intended. The JOP ROCKET has been tested through repeated iterations of the design cycle; all programmatic behavior appeared was as had been anticipated. Numerous instances of unwanted or unintended behavior existed and were encountered often throughout the design cycle, but were all identified

and corrected. These unexpected phenomena, e.g. software bugs, stem from unanticipated mechanisms in the implementation (Wieringa, 2013). The only anomaly that occurred during the single-case mechanism experiment was with a RAM memory error from Python that occurred with a small number of very large binaries that were tested, and a reasonable explanation was provided.

Explanations

As described in descriptions, there exists a wide variance in the average number of gadgets for certain indirect jumps or calls, and Appendix B provides the full details, showing the total number of gadgets found for each of the categories. With some binaries represented in Appendix B, some may be close to the average for a particular indirect jump or call, and others may be only a fraction of the average, while a few outliers may instead of having have 5 to 10 times as many gadgets as the average. Again, it must be reiterated that the testing done was just to demonstrate efficacy and utility, so the dataset is not statistically large enough to make broad generalizations that withstand scrutiny. A more comprehensive study would be needed to form accurate generalizations, with respect to the nature of indirect jumps and calls across a broad spectrum.

However, we might speculate simply that there can be large variance, and that the size of the binary can be tied to it, but that other factors may come into play. With these samples, the binaries with the largest number of gadgets have been large commercial applications that were not Microsoft products. Some of the results could be related to programming choices, languages or compilers used, or whether object-oriented was employed, etc. Beyond simply size, there is insufficient data to support further informed speculation. The fact that some modules have far fewer gadgets than others can be tied to the size of the binary, as an increased size tends to correlate with a higher number of gadgets, whilst the opposite remains true as well. A large number of modules of greater size can increase the number of available gadgets as well, so this could

contribute to the overall potential attack surface for a binary. Nonetheless, at least for this dataset, there are some gadgets that are on average existing in much larger or smaller numbers than the average.

The fact that some classifications of gadgets are more plentiful can be attributed to two areas. First, some registers are used much more frequently, such as EAX, while others less so. Some Assembly instructions are also more commonly used, such as ADD, MOV, and SUB. Thus, some functional gadgets for certain operations produce more plentiful results because they are used more frequently. There is also a tendency for some unintended instructions associated with certain registers or certain operations to be more likely to result in meaningful instructions. We can distinguish this from far more impractical specimens that would have little use in any conceivable attack scenarios. This research does not have data to draw conclusions on whether certain specific operational gadgets tend to be more plentiful purely on account of opcode-splitting, but past work would lend support to this idea (Bletsch, 2011)

Some categories of gadgets have fewer numbers, and there are some categories of gadgets that jump or call to certain registers much less than to others. This may seem on the surface to preclude the use of certain registers, if a particular operation does not have appropriate gadgets that modify a certain register. However, it bears pointing out that by utilizing one category of functional gadgets, the move shuffle, it is possible to change a value from register that is more commonplace, to one where there are no available gadets. The register holding the value sought by the attacker can switched back and forth from multiple registers, as many as times as needed. Some functional gadgets with careful planning can perform more than one key operation. Thus, a move shuffle or adding a specific value to certain register could both occur in the same JOP gadget. This allows for greater flexibility, but requires more careful planning.

The number of available gadgets for a binary can be increased by scanning modules. Although this would seem readily apparent, we state this nonetheless. As more modules are analyzed, they will tend to produce more gadgets, thereby enlarging the attack surface. Thus, using the option to scan all modules in the IAT will tend to increase the attack surface, while selecting the option to include the IAT as well as expanded coverage will tend to increase it even further. Some gadgets may not be accessible, however, if ASLR is in place and there is no memory disclosure or other means to overcome it.

The fact that the category for Other DG had only limited numbers in some registers can be explained in part because of the tendency for those shifting operations to be used more frequently with EAX, and the fact that unintended combinations of opcodes involving some registers do not tend to produce examples of Other DG gadgets. Thus, it seems the Other DG will typically only be found in some registers if these instructions are intended, reducing the attack surface.

While there were not other programmatic errors encountered during the single-case mechanism experiment, this is because they were previously addressed. These errors can come from a variety of causes, such as carelessness, inadequate understanding of what we are doing, or there may be unknown mechanisms that may interfere with our capacity to comprehend what we programmed (Wieringa, 2013). Sometimes correction of errors was a time-consuming endeavor, but careful measures were taken to ensure that the algorithms in place worked step by step as intended. The anomaly with the MemoryError mentioned in discussion is due to available RAM being exhausted for certain very large binaries with more than 24 modules. In this testing environment, 32-bit Python and 32-bit Capstone are used. There is a hard limit of available RAM for 32-bit, limited to 4 GB, of which 2 GB is reserved for kernel usage. This error only occurred with some binaries that were very large and had an abnormally high number of gadgets. It is clear

that due to the size of the binaries and their modules, that memory was apparently exhausted. It does seem unusual that the binaries would consume that much memory, but there could be other nuances of memory management at play, for either Capstone or Python. It is likely that a 64-bit implementation of both Python and Capstone would resolve these.

The dataset is a testament to the utility of the framework, as the faceted classification has produced a wide array of different gadgets, organized by function and by affected registers. The high degree of understandability and learnability for users stems from the fact that the JOP ROCKET provides a robust menu and help system. The help system should be adequate for those familiar with code-reuse attacks, i.e. ROP. ROCKET trades on familiar ideas and similar territory. For those who are not familiar, the tool and its usage likely would be beyond their understanding, without previous exposure to code-reuse attacks. We can view knowledge of ROP as a necessary prerequisite, and it is unlikely someone would attempt to utilize this without that knowledge.

Analogic Generalization

Using analogy to generalize as a form of analytic induction is a valid technique that can allow statements to be made from a single case to extend to other similar cases (Yin, 2003). Thus, we can assume that similar contexts as well as similar artifacts would tend to produce very similar results. Those results from a single-case mechanism experiments could speak for many others contexts.

A single-case mechanism experiment is often run in a laboratory environment. This experiment was not run in a production environment with exploit developers using it for real-world tasks in the development of actual exploits. Instead, a laboratory environment simulated that context; this permitted various modern PE files to be subjected to analysis. All these PE files feasibly could have been used as targets, as these are much the same binaries that an attacker might

exploit, given a vulnerability. Thus, we can generalize that the efficacy and utility demonstrated in this experiment would extend to the real-world usage as well. Accordingly, all the generalizations derived from a laboratory simulation of the context likely would extend to real-world usage of the framework (Wieringa, 2014). It is not likely that any situations or conditions anticipated in the real world would cause any change whatsoever, as the binaries would remain identical and thus produce the same dataset. Here we can also employ sample-based reasoning to note that what was observed with the artifact prototype and the simulated context can be extended and generalized to the real world, thereby establishing external validity (Wieringa, 2013).

Answers to Knowledge Questions

This work was intended to provide contributions towards the knowledge goals regarding the best way in which to devise an instantiation along with appropriate methods to address the research problem. The fact that the artifact works as intended serves as evidence that this goal was satisfied. The answers to these questions are then the methods and instantiation themselves, as these are the appropriate ways to realize the goal of constructing this artifact.

As supplemental research questions, this dissertation was intended to also some additional research questions. These concerned (a.) the frequency of certain categories of gadgets, (b.) how frequently gadgets could be eliminated as impractical, and (c.) the average breakdown of indirect jumps or calls to specific registers. To reiterate, these were not knowledge questions that guided this research, but they were felt to be an interesting by-product that would be useful to security researchers.

The below diagrams illustrate the response to the first question. The second question does not have evidence in support of a response, as this data was not collected. Owing to the sheer volume of data structures, methods, and the size of the code itself, additional methods to collect

this data would consume too much space. Additionally, it would add unnecessary complexity to the logic and disrupt the flow of the results. Anecdotally, during testing, it did appear that a reasonable number of gadgets were discarded. It is likely that a majority of those discarded were unintended results. The third supplemental knowledge question (c.) can also be addressed by a graph in Figure 23. This does show that indeed certain registers used in indirect jumps or calls to registers tend to be much higher than others.

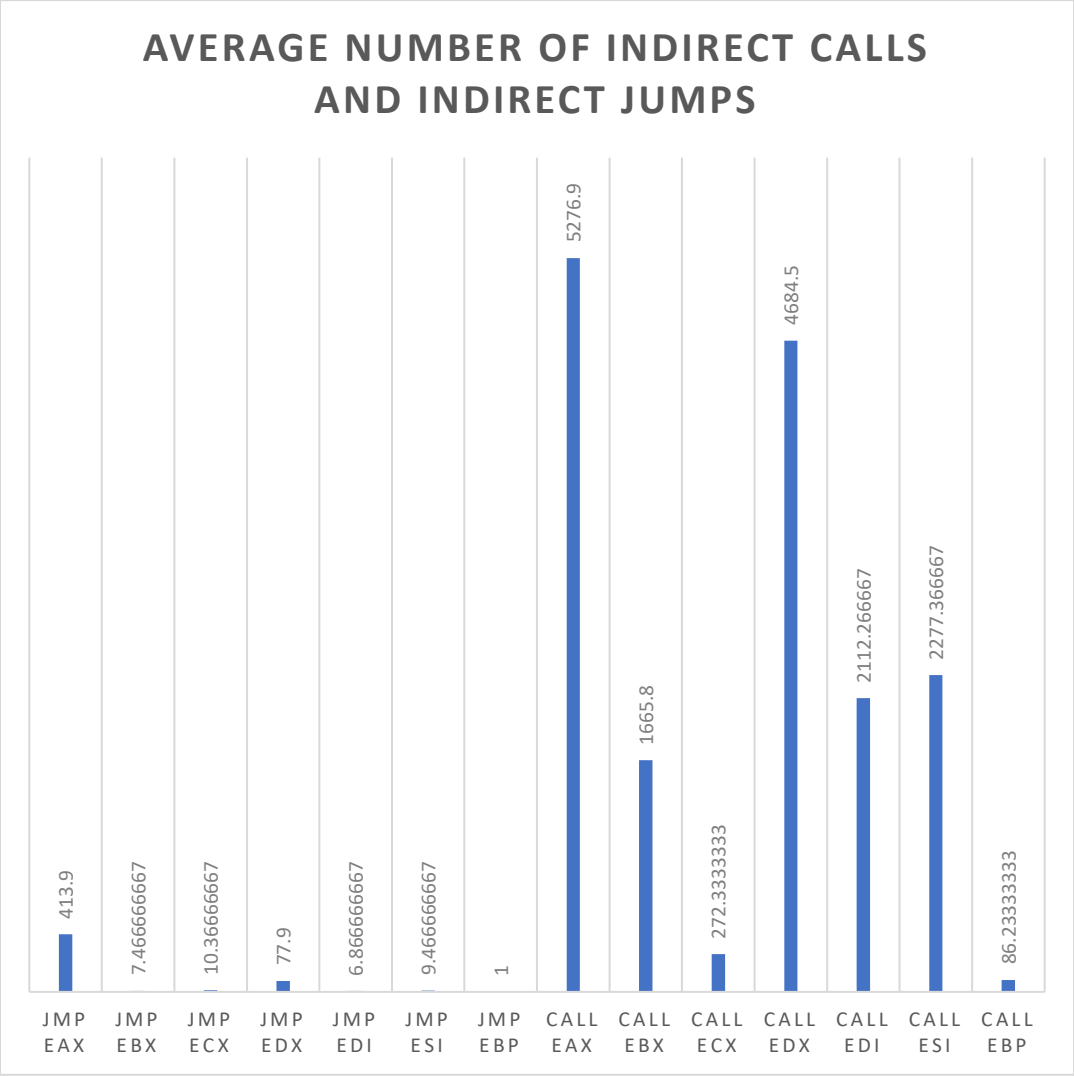

Figure 23. Average number of indirect jumps and indirect calls for the 32 binaries analyzed.

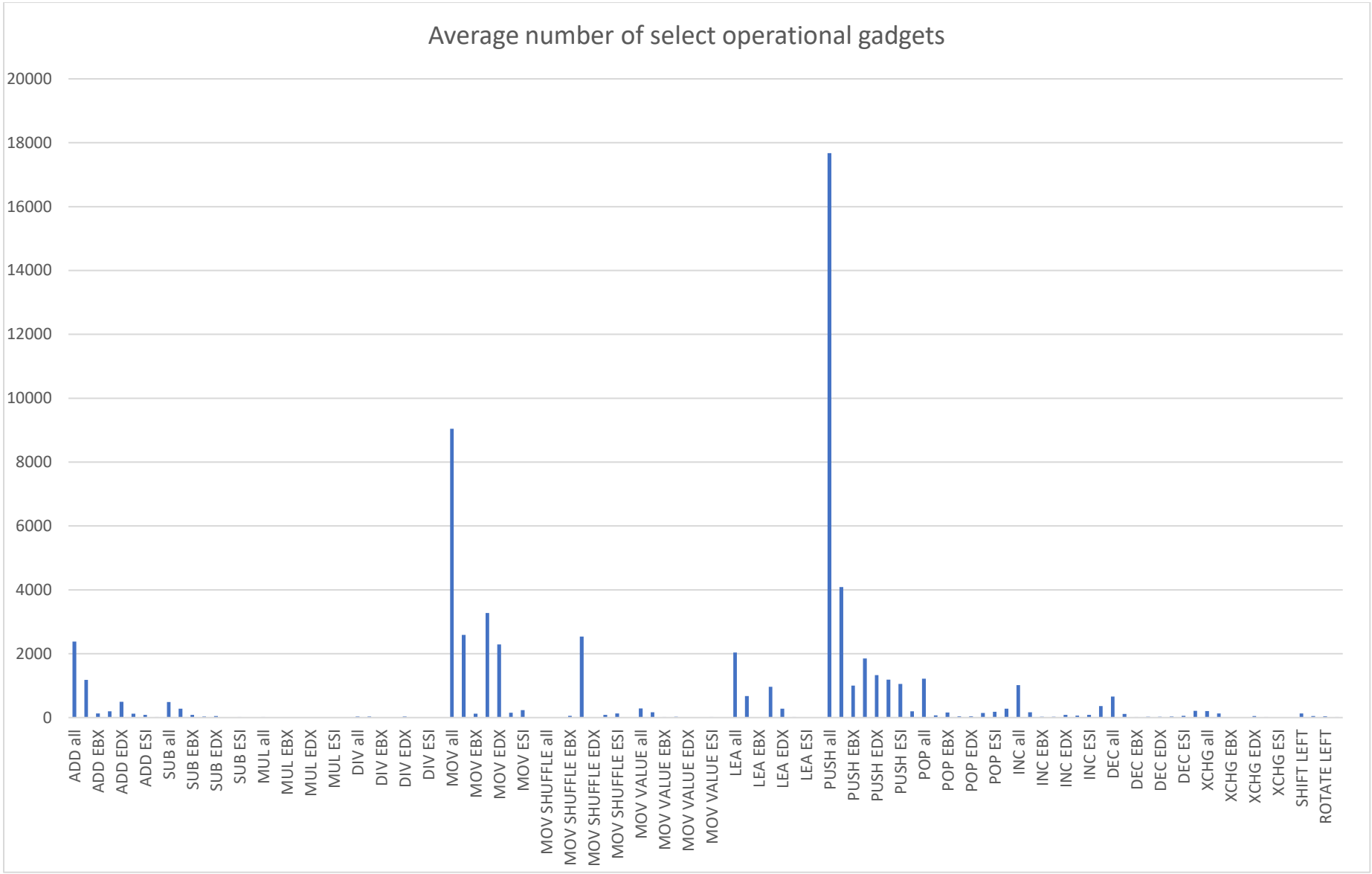

Figure 24. Selected averages for total number of operational gadgets for the 32 binaries analyzed.

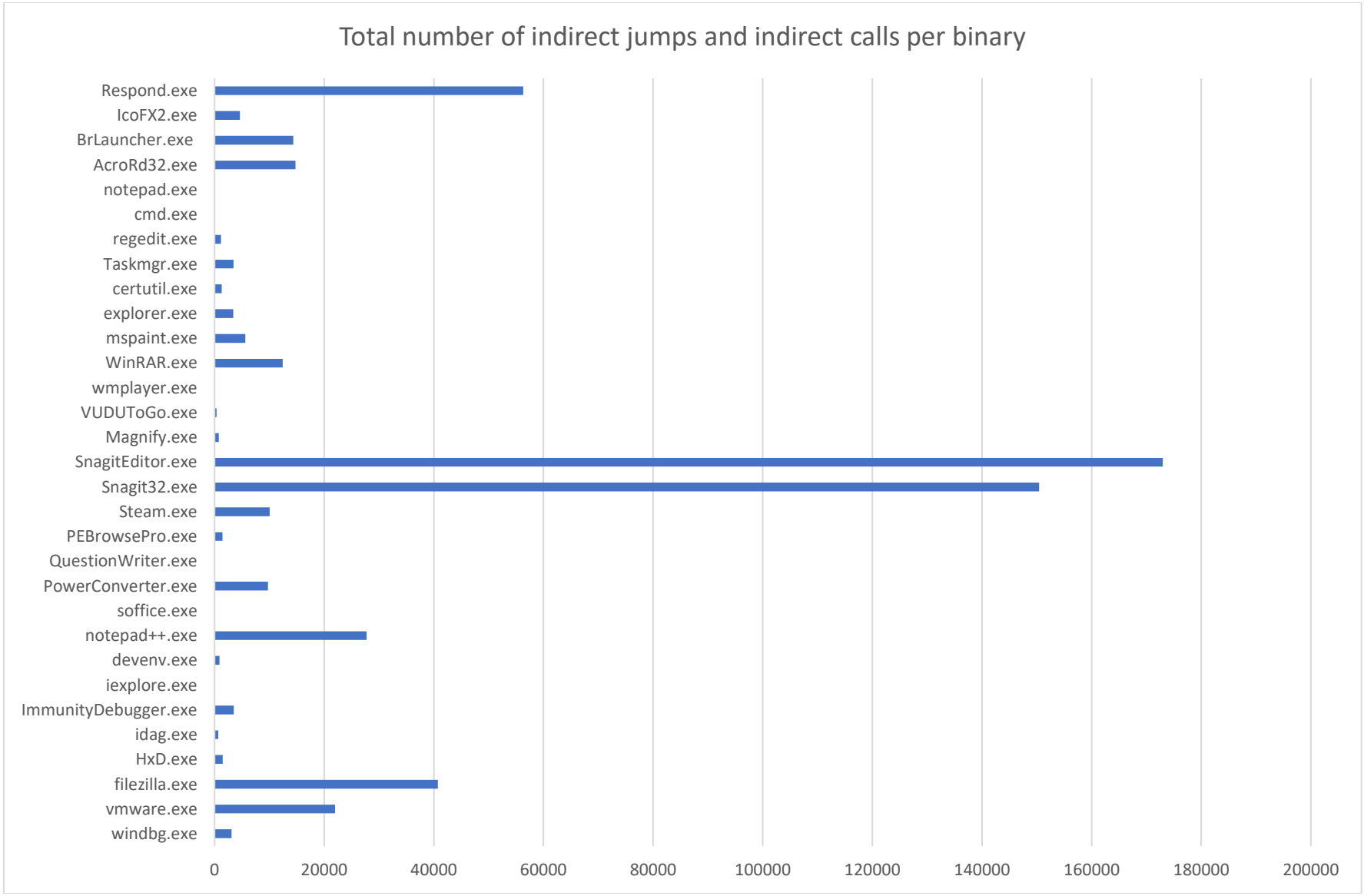

Figure 25. The total number of indirect jumps and indirect calls for the 32 binaries analyzed

Implications for the Context

This work provides a contribution to the improvement goal. Some of these methods had already existed in a more rudimentary form, but they lacked refinement and usage in a well-developed tool. Additionally, this work serves as a contribution to the practice of exploit development, by providing said tool to address the previously described unmet needs.

Discussion of Results

One of the ways in which a design science artifact can demonstrate its efficacy and utility is by validation through a single-case mechanism experiment. These results show that the artifact meets those two criteria. It produces an abundance of useful gadgets carefully classified into relevant categories and sub-categories. The JOP ROCKET improves upon as well as introduces new methods as artifacts, and their implementation has been successful as seen by the meticulously ordered data that can be generated by the tool.

The results could be enlarged or narrowed by changing search criteria, such as the number of opcodes to disassemble as well as the depth, or the to put it another way, the number of lines between the target operation and the indirect jump or call. For the discovery of JOP dispatcher gadget, additional search criteria could be modified, to further enlarge the results, though increasing the depth here would produce inferior or unusable dispatcher gadgets. For obtained gadgets in the dataset, the default settings had been employed. These settings were reasonable, but could possibly exclude useful gadgets.

Single-case mechanism experiments are often useful for the testing of a prototype of an artifact, allowing for not only the new technology to be validated, but for the evaluation of the

implementation. The results of this single-case mechanism experiment with this research does satisfy all the stated objectives.

The Iterative Approach

Hevner, et al., emphasize that design science is to be a highly iterative process, with a cycle of evaluation allowing for the artifact to improve continuously and better address the research question at hand. To simply build an artifact and declare that one is done with it, while not subjecting it to multiple iterations of validation, is to fail at design science. Thus, a flurry of activity occurred as different stages of the design cycle iterated repeatedly, to yield an artifact that embodies correctness, consistency, and completeness.

As initial algorithms were created and implemented for the JOP ROCKET, the results produced were carefully analyzed for accuracy, reliability, and appropriateness. At the most basic level, the question of whether they were right and accurate was asked. Then other considerations came into play, such as how to save and utilize the large dataset generated for each binary analyzed. Initially, an object-oriented approach was started, but that was soon jettisoned for a naïve approach that favored simpler data structures. Because of the ambitious nature of this project, it became necessary to expand these simple data structures significantly, as their numbers grew into the hundreds. Later, it came apparent that an object-oriented approach was indeed required due to the complexity of the program with respect to how it used and manipulated data, and significant reworking of the existing code occurred to facilitate this transformation. Thus, design science and its iterative approach has significantly enhanced the JOP ROCKET, allowing for features to mature and new features to be developed, resulting in a much stronger artifact.

Hevner's Design Science Guidelines

While this work was completed by following the design science principles of Wieringa, an artifact completed in that fashion would lend itself to evaluation under Hevner's design science guidelines. We briefly review how it has satisfied Hevner's guidelines. The first two of Hevner's guidelines, Design as an Artifact and Problem Relevance, have been previously in abundant detai; having completed the design of the artifact does not change these. We will look more closely at the latter five guidelines, Design Evaluation, Research Contributions, Research Rigor, Design as a Search Process, and Communication of Research, in the sections that follow.

Design Evaluation

Hevner's guidelines provide that the triad of utility, quality, and efficacy of the artifact must be shown with rigor and the use of evaluative methods. This has been rigorously demonstrated via single-case mechanism experiment. The single-case mechanism experiment has shown that the gadgets are accurate, that they produce the appropriate offsets, that the UI is clean and intuitive, and that the results are organized in a fashion that is useful and meaningful to the analyst. According to Wieringa, single-case mechanism experiments are used to implement and evaluate a design science artifact, permitting the researcher to explore the tool in a look at the cause and effect of the object of the study in an environment with its intended context. Finally, the single-case mechanism experiment also helps validate the design science artifact, serving to demonstrate its utility and efficacy. This has been done by showing the numbers of gadgets that can be classified into appropriate groupings.

Research Contributions

Design science, according to Hevner, should provide strong contributions in the area of the design artifact. This has done so via providing a verifiable contribution in the form of design artifact. The framework itself, as has been discussed in detail, is innovative and provides several novel methods that can enrich the field of exploit development. This is seen through the five artifacts that are methods as well as the primary focal point, the instantiation of the framework itself. These amount to a significant contribution, answering a need that has been unmet. What's more is they can be used to help facilitate additional research by others in the area of JOP and exploit development.

Research Rigor

According to Hevner, design science must employ rigorous methods during both the construction and evaluation of the artifact. During the highly iterative development of this tool, hundreds of tests were done to ensure the numerous different algorithms, helper functions, etc., worked as intended and to discover where changes needed to be made. This was no small task as the tool grew to be over 14,000 lines of Python code, but rigor was a focal point. Much rigor could have been eliminated if the author had taken the easy way and decided simply to integrate the JOP ROCKET with an existing tool, like WinDbg or Immunity; numerous problems and programming challenges would have been instantly eliminated. A reduced feature set and a more naïve approach to JOP also would have lowered rigor and allowed for the artifact to be created more quickly . However, rigor has been a point of pride with this dissertation research, and it would not be traded away for convenience or ease, particularly as this work should be of a very high quality to satisfy Ph.D. requirements. Additionally, this rigor has been demonstrated through validation efforts, and rgor can be shown through the validity of the results the JOP ROCKET has produced. The simple

fact that it does what it purports to do and does so accurately is a testament to the necessary rigor that the artifact has undergone. Accomplishing these benchmarks is no trivial task, given the level of complexity inherent in this artifact, but that level of complexity is commensurate with what is needed to fully address JOP.

Design as a Search Process

Hevner asserts that that the search for an artifact must employ all possible means to reach a successful outcome, whilst satisfying guidelines in the problem realm. This process has been actively engaged, as various other models were studied that were used to facilitate the discovery of code-reuse attacks. These include the Mona Python script, which must integrate with WinDbg or Immunity, or the ROPgadget python script by Jonathan Salwan, which can be run independently. Other more generic reverse engineering tools were examined as well. This included looking at the tools themselves as well as examining source code, where available. These were helpful to some extent, but this work went in a completely different direction and involved subject matter that was outside their scope. Mona and ROPgadget have very minimal JOP coverage, but they are not developed and appear to be more a placeholder for future work.

Communication of Research

Communication of research is essential, as it must be presented to both a technical as well as a management-oriented audience. This tool is highly specialized, and the relevance is quite clearly communicated to the intended audience. In fact, it is presented in a way to make the content as easily digestible as possible. As has been described, the fact that this tool is able to generate well-organized groupings of gadgets to facilitate attacks should be clearly understandable by management audiences. They need only understand that it produces the necessary gadgets that

potentially could be used for a JOP exploit. Additionally, as one of the means of fulfilling this requirement of Hevner, publication or conference presentations will be pursued as an avenue to communicate some of these findings to other researchers.

Summary

This chapter serves to detail the results obtained by this research. It confirms that the artifacts have been created as described in the methodology section, by following rigorous design science guidelines. It has been shown that this work has met the DSR guidelines established by Hevner, et al., and it has conformed to Wierigina's principles of design science. This chapter has shown in great detail how it meets the guidelines or principles described by both Wierigina and Hevner, et al. This research has culminated in the creation of six artifacts, including five new or substantially reworked methods, alongside the crown jewel, the instantiation itself. This chapter has explored each of these artifacts in detail, explaining what they do and the significance of their contributions.

CHAPTER 5

CONCLUSION

At the heart of this research, the JOP ROCKET was developed as a versatile tool that can confer strong benefits to security researchers wishing to do JOP exploits. Outside of doing JOP just as a proof of concept, JOP can be a way to overcome heuristics that may look for certain ROP behavioral patterns in a Windows 7 environment. Robust anti-ROP heuristics and defenses would be unable to detect JOP, as that is outside their scope. Thus, these mitigations that might have been too powerful for ROP to overcome, then could be defeated through JOP, making a potentially secure system, now suddenly vulnerable to attacks that utilize JOP.

This work has focused on Windows 7, as it lacks Microsoft's CFG, and so JOP can be done freely without restrictions. However, this work could be extended to Windows 8.0, which also lacks CFG, and it is important to be aware that while even Windows 10 has CFG, there are vulnerabilities with CFG discovered, from time to time, which can allow attacks to be executed on that system, and patching to remediate these can sometimes be delayed. The best defense would be new computers with hardware support for Intel's CET, but that could be years until it is out, although Microsoft and GCC have already provided compiler support. While the theoretical CET may offer protection, it too may be vulnerable to attacks, and some security researchers have already pointed out inherent weaknesses ("Close, But No Cigar," n.d.). CET would have similar forward-edge defenses against indirect calls and jumps, by simply whitelisting a series of acceptable targets. That would severely reduce the attack surface, but it would still permit JOP,

and other potential, to be discovered vulnerabilities, potentially could make more JOP attacks possible.

This work focuses on Windows as it is the dominant operating system and more relevant. While this work has focused on a Windows environment, there is nothing to stop it from being modified to include Linux or other operating systems, which may lack support for JOP defenses. In fact, much of the existing code and logic in the JOP ROCKET, after extracting executable content, would be identical. The code was not adapted only because scope for this dissertation research was limited. If the JOP ROCKET were to be adapted to Linux, it could increase its relevancy and increase the total number of machines vulnerable to JOP. While Linux does provide some limited, third party CFI solutions, such as Reuse Attack Protector (RAP) and Clang/LLVM, these are limited in scope, and neither are widely deployed.

This chapter will provide a brief review of a few of the principal contributions that have been made. Lessons learned from throughout the process are explained, and some of the limitations inherent in the application are discussed. Recommendations are made, and finally this chapter concludes with a discussion listing possible directions for future work.

Contributions

This research provides contributions in the form of an instantiation of a novel framework and five distinct artifacts consisting of new or improved methods. These have already been discussed at length in the previous chapter, so just a broad review of some of the more significant contributions will be provided.

Faceted Classification for JOP Gadgets

This work provides a faceted classification for JOP functional gadgets. While Turing complete features have often been used to demonstrate that different architectures can perform full arbitrary computation, this research entails classification into new sub categories. The point here is not to demonstrate that JOP is fully Turing-complete, but to use what is most relevant and to extended it further for other practical uses. Thus, there is classification of gadgets based also on the affected register or, as with the dispatcher gadgets, which gadgets might be perceived as best. The method by which the classification is performed itself, with filtering via regular expressions, is a novel addition.

Robust, Powerful Framework that can work across platforms

The most important contribution is the instantiation itself. To reiterate this tool addresses a gaping hole that exists, where a versatile, resilient tool could allow an analyst to easily discover JOP gadgets. In so doing, it provides a tremendous amount of granularity, allowing the user to be flexible in how they customize searching, either enlarging or narrowing results. The instantiation utilizes much exclusion filtering, so that the results obtained have been thoroughly sanitized to eliminate useless or impractical gadgets.

This powerful framework exists in Python and is not reliant on other existing tools, and this means it is highly portable across platforms. Thus, it can be used not only on multiple versions of Windows, but on other architectures as well, although there are limitations on a non-Windows OS.

Novel and Improved Methods

This work provided several novel or improved methods. In the case of improved methods, the improvements are significant enough for them to be considered as design science artifacts. Some of these methods are useful as part of algorithms that help provide necessary functionality for the framework, and some could be feasibly be taken and used outside this work in other reverse engineering tools.

One significant contribution is the method to discover dispatcher gadgets. This is the most important element to forming JOP attacks, as without this central piece, it is not possible for the JOP attack to occur. While the criteria needed for a JOP dispatcher gadget is not new, with a naïve implementation, it could turn up significant, impractical results. Additionally, this research proposes an entirely new method to form a dispatcher gadget. This is done by using shift left or shift right by 1 or 2, achieving multiplication or division. With careful planning and a significant area of the heap available, and the ability to find an intermediate operation to perform necessary subtraction or addition to go backwards or forwards, then an exploit developer could make use of this less than ideal set up. This could present options to make JOP feasible on some binaries, where there would otherwise be none.

This research has provided a new way to discover JOP functional gadgets, reworking an existing method to provide greater depth and coverage. Not only that, but it provides the user with the ability to be specific and granular in terms of what they seek. It also supports strong exclusion criteria to reduce gadgets that would not be likely to be of any value.

This work also presents a novel method for saving and recording information used to print or reproduce the disassembly for JOP gadgets. This could be extended further and used with other tools that produce any type of code-reuse attack gadgets. This work presents a multitude of data

structures, which maintains bookkeeping information for the different classifications of gadgets. These data structures maintain only a few values, so that no opcodes or no disassembly needs to be stored. Once the information for the gadgets have been saved, it can be manipulated or printed in seconds. This keeps storage requirements minimal. The artifact also includes a print menu with numerous options, allowing for flexibility.

Finally, this work presents an interesting variation on the method to statically enumerate modules. While it is simple to use the Pefile library to discover the DLLs contained in the IAT, there can be some shortcomings, as not all modules that will eventually be loaded when the PE loads, will be present in the IAT. This could have made a static analysis tool such as the JOP ROCKET not as relevant if it could not detect some of the modules outside the IAT. This work provides additional logic to address finding file locations for modules when the Windows API fails to return a handle, ensuring that nearly all DLLs can be loaded. This work utilizes Windows API's to load and obtain a handle for the module in question, which then can be used to obtain a file location and then extract the module's text section, to be analyzed for JOP gadgets.

Big Picture

In more practical terms we can view the contributions as a way of facilitating the construction of JOP exploits. This is a type of code-reuse attack rarely used in the wild. A large part of its difficulty is tied to the lack of dedicated tools, forcing the user to do it manually. By way of an analogue, we could view this almost like trying to manually edit a PE file with just the raw bytes, rather than using a template on a hex editor. A hex editor, such as 010, can allow a user to parse and view the binary data of a computer file, presenting the data in a way that humans can understand. Such a tool can map out clearly defined structures with members and values, enabling the user to quickly find the desired values. Without using a template in a hex editor, the user would

simply see raw binary data and not have a way to make sense of it. The hex editor provides order and clarity to what would otherwise be difficult to decipher, and the JOP ROCKET does much the same with the JOP gadgets it generates. The JOP ROCKET could be viewed as a way of applying a template to raw binary data, to extract something useful. The JOP gadgets would still be present in the binary blob that is an application, but without a proper tool, we just would not have a way to view them in a way that is meaningful for the intended purpose.

Ultimately, the JOP ROCKET may spur the use of JOP more in exploits, both in the wild or in the academic literature, or we may see increased hobbyist usage, such as with Capture the Flag (CTF) completions. The JOP ROCKET could do well to enhance the relevance of JOP, with more people wanting to attempt to use it. This could lead to heightened awareness of the need for strong CFI, while showcasing the shortcomings of operating systems that lack CFI, such as Windows 7 and various other Linux operating systems.

Lessons Learned

From a programming standpoint, there were numerous lessons learned throughout this research, and it would be tedious to recount them all, so we will touch on a few that standout as more significant.

One lesson was learned was the need to stick with original plans if there is a good reason for them in the first place. During development, there was some back and forth on the usage of object-oriented programming (OOP). Originally, the intent had been for the framework to be object-oriented in its approach, and for each module to be its own object, each with its own sets of gadgets. The author had familiarity with OOP from using it in both C++ and Java, but not with Python. With the JOP ROCKET, there was a need to figure out how to do some potentially

challenging work in a language not well designed for low-level code. This was complicated by the fact that Python as a language had some limitations on how OOP is implemented, so it was decided to largely abandon OOP, using it in a only a very limited sense. As such, some workarounds were used to get some of the desired functionality. The development of this artifact was highly iterative over a period of time, with the focus being on evolving the algorithms and data structures. During much of this time, most of the work was done on just the executable image itself. Later, as the artifact matured and modules were included, it became clear there was a necessity for the approach to be OOP, if the artifact were to work as had been envisioned.

To provide one example of why this was the case, we can look at the printing of operations based on faceted classification. This task was simple, when there was just the executable image to process, but when there were also modules, it then became necessary to produce a separate output for each module. This could quickly produce a staggering number of files. The reason for that was, at that given time, all the various data structures were used to generate the needed gadgets and disassembly on the fly, after they had been found through searching algorithims. With this tool, everything is very modular, and so it would print out all of one operation at a time and move onto the next operation to be printed.

The values in the data structure are used to carve out parts of the text section, which is also stored in a data structure. Thus, we can see if there was only one data structure that could be possible for each operation and gadget type and only one list, comprised of the text section, then the program would need to loop through each of the DLLs. This was highly inefficient, resulting in an unwanted deluge of many files. There was no storage mechanism in place to hold data for more than one module, and appending modules to the same data structure would not work.

The only conceivable solution was to go back to the original plan for having objects for each module and all the myriad data structures that would be associated with it, including the text section for each module. This would seem like a simple solution, but at this point the code was over 10,000 lines, and this meant rewriting larges portions of it to fully utilize OOP. This also required additional, careful testing, to ensure all the many functions and helper functions worked as intended. At the end, after OOP was fully implemented, the JOP ROCKET worked exactly as intended. Now we could have one file for an operation with all modules contained in it; this was a tremendous improvement.

Having a fully realized OOP approach came late into the maturation of the JOP ROCKET, but once it did come, it did enable other minor improvements to more rapidly be implemented, and it allowed for removing some less efficient code. The lesson learned here would have been to more rigidly stick to the original functional requirements.

We will touch on final lesson learned. In hindsight, waiting to implement the UI until late in the development of the tool was not the best choice. During much of the development, efforts centered on getting algorithms and methods to work with no attention being given to UI. Once many the appropriate functions and data structures were working well together, then energy was expended on creating a UI. That is not to say the UI was a secondary consideration, as there had been a clear idea of what was sought. It was left to be developed later because of the size of the datasets and the operations being tested. Informal tests were not instantaneous, and they could take a few minutes or more, so avoiding repetitive interaction with the UI was done to expedite development. This also permitted focused testing on specific functions and their interactions with different data structures as needed. When it came time to create the UI though, there were some bugs to be worked out; these could have been avoided, if the UI had been built and tested

concurrently. Allowing the program to have a UI also provided clarity from the standpoint of how the JOP ROCKET would look and feel for users. Once a UI was created, it helped give new insights to different issues, some not previous considered, that come into play with how the user might interact with the UI.

Limitations

Throughout the project various obstacles were encountered. These limitations must be properly understood when looking at the results.

Time bore an influence on this research, as time was somewhat limited and there was a desire to not unnecessarily prolong this research and to complete it quickly.

Some ideas were and thoughts were explored at great length, and some work and thoughts were ultimately scrapped due to time constraints. Time was a factor with regard to scope, as initially the intent had been to provide a hodgepodge of different code-reuse attack tools. Design and evaluation efforts for the JOP artifact took so long, that this was impossible, unless graduation would be delayed a year. Thus, it was decided to narrow the scope to work exclusively with a JOP tool. Often with Ph.D. dissertation work, the goal is to be very narrow in scope, so this research is in line with what is typical, and its contributions are significant. Of course, a tool could have had greater utility if scope were broader and encompassed other areas, but doing so can greatly increase the number of man hours if down properly.

One limitation, by design, is that the JOP ROCKET only processes 32-bit binaries. This was a decision made early on. Extending this to both 64-bit and 32-bit would add unreasonable complexity during development. Moreover, because there is a significant amount of regular expressions used, a lot of those would need to be rewritten to accommodate 64-bit. A 32-bit

limitation seemed reasonable as many binaries are 32-bit, and that is far and above the most prevalent architecture for ROP.

The author had curiosity about prevalence of other indirect jumps or indirect calls to the new 64-bit registers, including r8 through r15, but there were very few found in some preliminary tests. This was disappointing, as it had been hoped that unintended instructions possibly might result in a large number of indirect jumps or calls to some of the new registers. This might give JOP more flexibility and allow it to work better. After the completion of a significant portion of the development for the JOP ROCKET, some time was spent attempting to create an alternate 64-bit version. However, it soon became clear that adapting it to 64-bit would not be straightforward, as even after some initial changes and small tests were made, there appeared to be serious issues at play. These have yet to be worked through. Thus, it was concluded that it would be a much more time-consuming endeavor to convert to 64-bit than had been anticipated.

It Is not presently known how JOP would work with 64-bit binaries. It is felt, however, that even with the absence of increased attack surface on the new 64-bit registers, it was at least possible that JOP might enjoy an increased attack surface in 64-bit. The x64 calling convention could also introduce limitations that could affect the practicality of some registers. Without a tool to examine it closely, only speculation would be possible.

Another minor limitation is that this work does not address binaries that are packed, heavily obfuscated, or self-modifying. This work only addresses standard, well-formed PEs. Some binaries, often malware, may exhibit some of the aforementioned characteristics to make analysis more difficult. As this tool is intentionally restricted to static analysis, we cannot dynamically access the memory of a PE that deobfuscates itself or has modified its memory, as we can only do analysis on the text section as it exists on disk. Secondly, some of the imports might not be apparent

until dynamic analysis can occur, due to dynamic loading of modules via LoadLibrarayEx and GetProcAddress, as well as other malware techniques, such as traversing the PE file format by using the PEB. Unfortunately, a static analysis tool such as this could not provide functionality to discover these, not without very significant efforts.

The limitations described above bring up the topic of dynamic analysis versus static analysis, and this certainly is one of the most significant limitations present, for several reasons. The type of work being done by this tool is better served by dynamic analysis, rather than static analysis. This was not pursued because a dynamic analysis approach, while technically better, would r with an require integration with an existing debugger, such as WinDbg or Immunity, and there was a strong desire to avoid doing so. There was a desire to make this as much a standalone tool as possible, and this certainly created a good deal more work and effort that could have been avoided, if we had taken the simpler, easier choice of integrating with an existing tool. But, again, it was felt that for doctoral research, integrating with an existing tool would be too easy. Thus, with reluctance, we accepted the limitations inherent with static analysis.

One limitation lies in the fact that, although this tool provides the functionality to search for functional gadgets that perform a specific operation, e.g. adding a value to the register EAX, it is possible that the contents of EAX could be clobbered or destroyed in the next line of instruction. This tool employs no logic to search for this behavior and then provide exclusion on the basis of it. Additionally, if that clobbering were to occur, it is also possible the desired value in EAX could have been moved to another register and then moved back to EAX, and logic could address this as well. This would increase the complexity a great deal given the way that the artifact is designed and operates. If there were more time available, likely this would have been implemented.

There is a limitation with respect to different platforms. Because the required dependencies are just Python as well as the Pefile and Capstone libraries, this artifact can be used across platforms other than Windows. As has been described more fully elsewhere in the dissertation, if the JOP ROCKET is used outside of Windows, it will be unable to scan any of the DLLs, unless loaded and scanned separately.

Perhaps the elephant in the room is that JOP is still limited in terms of practical application. The JOP ROCKET cannot change that or somehow generate gadgets that do not exist. The simple fact is often there is a dearth of JOP gadgets available, relative to ROP. This is in part why this artifact has been so meticulous in its approach to provide all available assistance to the analyst, such as discovering all possible gadgets, the use of faceted classification, and the exclusion of impractical gadgets. The fact remains thought that despite these efforts, some binaries may not have enough of the right gadgets for JOP to be feasible, or other protections in place such as DEP or ASLR may make some needed gadgets unusable. There may not be obvious ways to overcome these hurdles with certain binaries. One might just have to accept that with respect to certain applications, JOP may not be practical.

Recommendations

The primary recommendation is that if individuals or enterprises by necessity or choice use an operating system, such as Windows 7, that does not have protection against code-reuse attacks, then they should make use of available tools. EMET reached end of life in July 2018, but it still may be downloaded and used, offering some protection. Of course, much of its protections are built into Windows 10, but we can regard that as a relatively secure operating system. EMET does not, however, extend protection against JOP. If a user wishes to use an older operating system with

valuable assets, then they should consider an implementation of control flow integrity, depending on the value of the target. The leading implementation of CFI available is Microsoft's CFG, but that is not available below Windows 8.1. Various other CFI solutions are available, some discussed in the literature in chapter 2, some of which could offer some limited protection against JOP. Still, many of these solutions have unacceptably high performance costs or are in some ways overly limited or impractical. Thus, it may not be feasible or worthwhile to employ such limited CFI solutions.

The only possible recommendation then can be that if an organization or individual must use a Windows operating system unprotected by CFG, then they should greatly limit what is available on that machine to only what is necessary. The least amount of privileges should be used. Network isolation and segmentation should be employed, so that if the machine were compromised using a code-reuse attack, then the attacker would be limited in their ability to do lateral movement on the network. All other relevant precautions should take place. In short, the user should accept that it is reasonable that such a machine could be unsafe, and they should act accordingly.

Recommendations for attackers are to attempt to utilize the JOP ROCKET. Some binaries, as discussed, many not lend themselves as easily to JOP. That is not a limitation of the ROCKET, but of JOP itself. The only want to get good at a new attack methodology is to try to utilize it and understand it through study and practical experience. One should also understand that if ROP is an option, because of lack of mitigations or because mitigations can be overcome, then it is going to be the better choice.

Future Work

This research presents many opportunities for expansion and additional work to be performed. The JOP ROCKET is powerful and versatile at the tasks it was designed to perform, but if we were to go broader and look at code-reuse attacks, we might find other ways in which to expand functionality and improve what is there.

The most obvious choice is to add functionality for 64-bit binaries. As it is, it is not known how much easier or harder JOP would be with 64-bit, if there would even be any appreciable difference in the level of difficulty. One hope is that with 64-bit the attack surface would be larger. Even if this not the case, providing an artifact with the capability of finding gadgets with 64-bit binaries would open many applications to potential exploitation with JOP.

This tool has been designed to work with JOP in the Windows environment. The tool could be expanded to any other architecture where JOP exists. Doing this could involve significant new work if the instruction set differs greatly. In one architecture, JOP is done in a very different fashion, so this would involve little reuse of existing code. Adapting this to work with x86 Linux, however, would likely not involve significant effort.

The framework could expand to cover other advanced code-reuse attacks, beyond simply JOP. One area that would be far more promising than JOP for modern binaries is Counterfeit Object-oriented Programming (Schuster, et al.,2015). This would enable a user to even overcome CFI. Implementing this in a tool would be far more challenging than JOP, and likely it would need to integrate with an existing tool, such as IDA Pro, as doing it as a standalone tool would not be possible, even with supporting libraries. There also would be the question as to whether it would be even feasible to automate parts of the process for constructing an exploit that utilizes Counterfeit Object-oriented Programming.

Future work on the framework could include data persistence. This would avoid needing to perform analysis each time there was an input file. This could be performed by saving the contents of all data structures to a binary file. This binary file, properly delimited, could then be parsed with its contents loaded into the data structures, thereby not requiring rescanning of the binary or its modules. It is felt this would be a simple feature to implement, given the design of the artifact.

On a related topic to implementation of the instantiation, we will discuss the prospect of a GUI. It is often felt that a power user can be more adept at using merely just a keyboard and keystroke shortcuts, to more quickly utilize a program. The GUI, rather than making work with software easier, can slow down users, as they are forced to interact with the mouse, when it might be more efficient to not do so. Even still, not all users will be power users, and for them a GUI would be a more apposite choice for more novice users. A GUI can also provide convenient functionality for drag and drop, which would be an excellent feature. Thus, if a user set reasonable default settings, and they could drag and drop files. That arguably could be faster and more efficient. Thus, there is a reasonable case for a GUI.

Future work could be done to add additional categories to the faceted classification. Additional sub-categories for those that are presently exiting could allow the user to be more granular still. To be very fine-grained, perhaps a user could even specify specific values or a range of specific values that would be sought. While in theory that may sound useful, it could also be unintentionally limiting, as the attack surface of JOP is far more restrictive than ROP, and it might be better to rethink some attack strategies, rather than to limit a search unnecessarily to only a certain range. Along these lines, future work could be given to thinking of alternative ways to emulate some desired functionality, which may or may not be scarce. For instance, in this

dissertation research, we considered using left shift and right shift as part of a unique strategy to expand viable dispatcher gadgets. Similar efforts could be applied elsewhere.

Finally, future work could involve rebuilding the artifact in C++. This language is much better suited for low-level programming. Additionally, OOP is far less restrictive in C++ than with Python. Python is useful for rapid development, but a compiled program that provides this functionality, including an optional GUI, would make the artifact stronger. It would also eliminate the need to have an environment with certain dependencies. There would be various programming challenges associated with rebuilding the artifact in C++. Better and more efficient data structures could be used than what is available with Python. Some present data structures are inefficient owing to Pythonic limitations. Given the proposed changes, it is not believed that it would in any way improve the results that would be found. It would just result in an application that would otherwise be more pleasing.

Conclusions

This work constitutes a significant contribution to the discipline. The JOP ROCKET enables the user to construct an exploit using a form of advanced code-reuse attacks rarely encountered in the wild. This research adds to the overall body of knowledge as it pertains to code-reuse attacks and exploit development. JOP was an area where there was limited knowledge and limited academic publications, but there were not publicly available, well-developed tools appropriate for JOP. Through this research, we have identified an important gap that exists with lack of tooling available for JOP, and we have met that need.

One unstated goal has been to help make JOP more practical accessible to non-specialists. It is hoped that other researchers can continue research in this vein, by developing other relevant, fully-featured tools to help facilitate not just JOP, but other less used code-reuse attacks.

Summary

This chapter has provided a brief survey of the various contributions made by the JOP ROCKET. Lessons learned have been discussed, as have opportunities for improvement that have arisen throughout this research. Possibilities for future work were explored, as with a framework such as the JOP ROCKET, there are numerous ways it could be improved or expanded. It was discussed that some other advanced code-reuse attacks could be incorporated into the framework. This could include the addition of an automated way to facilitate Counterfeit Object-oriented Programming. Other possible future work included the additional coverage for 64-bit binaries, expansion to other architectures or operating systems, adding additional categories or subcategories to the faceted classification, and as the addition of a GUI. Next, recommendations were made to users from both a defensive as well as an offensive standpoint. Finally, this chapter concluded by reaffirming the contributions made by the JOP ROCKET.

REFERENCES

Abadi, M., Budiu, M., Erlingsson, Ú., & Ligatti, J. (2009). Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC), 13*(1), 4.

Alexander, B. (2017, September). Abusing delay load dlls for remote code injection. Blog. Retrieved from http://hatriot.github.io/blog/2017/09/19/abusing-delay-load-dll/

Bacchus, A. (2019, February 2). Here's what it will cost to stay on Windows 7 when extended support ends in 2020. *Digital Trends.* Retrieved from https://www.digitaltrends.com/computing/microsoft-pricing-plan-staying-on-windows-7/

Bania, P. (2010). Security mitigations for return-oriented programming attacks. *arXiv preprint arXiv:1008.4099*.

Bilge, L., & Dumitraş, T. (2012, October). Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security* (pp. 833-844). ACM.

Bletsch, T., Jiang, X., Freeh, V. W., & Liang, Z. (2011, March). Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (pp. 30-40). ACM.

Bo Qu, T. & Lu, R. (2014, Jul. 16). Is it the beginning of the end for use-after-free exploitation? Unit 42. Retrieved from https://unit42.paloaltonetworks.com/beginning-end-use-free-exploitation/

Brandon, J. (2016.). Control flow integrity gadget-finder. Retrieved from https://github.com/jdbrandon/ControlFlowIntegrity/tree/master/gadget-finder

Bright, P. (2017, Jun. 27). *Ars Technica.* Retrieved from https://arstechnica.com/information-technology/2017/06/microsoft-bringing-emet-back-as-a-built-in-part-of-windows-10/

Caballero, J., Grieco, G., Marron, M., & Nappa, A. (2012, July). Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (pp. 133-143). ACM.

Carlini, N., & Wagner, D. (2014). {ROP} is Still Dangerous: Breaking Modern Defenses. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)* (pp. 385-399).

Carlini, N., Barresi, A., Payer, M., Wagner, D., & Gross, T. R. (2015). Control-flow bending: On the effectiveness of control-flow integrity. In *24th {USENIX} Security Symposium ({USENIX} Security 15)* (pp. 161-176).

Carrera, E. (n.d.) Pefile. GitHub repository. Retrieved from https://github.com/erocarrera/pefile

Checkoway, S., & Shacham, H. (2010). *Escape from return-oriented programming: Return-oriented programming without returns (on the x86)*. [Department of Computer Science and Engineering], University of California, San Diego.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A. R., Shacham, H., & Winandy, M. (2010, October). Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security* (pp. 559-572). ACM.

Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., & Yin, X. (2011, March). Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (pp. 20-29). ACM.

Chen, R. (2006, July). Exported functions that are really forwarders. MSDN. Retrieved from https://blogs.msdn.microsoft.com/oldnewthing/20060719-24/?p=30473

Close, but no cigar; on the effectiveness of Intel's CET against code reuse attacks. (n.d.) Grsecurity. Retrieved from https://grsecurity.net/effectiveness_of_intel_cet_against_code_reuse_attacks.php

Doubly freeing memory. (2018, Dec. 29). OWASP. Retrieved from https://www.owasp.org/index.php/Doubly_freeing_memory

Intel. Control-flow enforcement technology preview. (2017, Jun.) Retrieved from https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

Creswell, J., & Creswell, J. (2018). *Research design: Qualitative, quantitative, and mixed methods approaches*. Los Angeles. SAGE Publications, Inc.

Dai Zovi, D. (2010). Return-oriented exploitation. *Black Hat*.

Data execution prevention. (2009, Oct. 7). Retrieved from https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc738483(v=ws.10)

Davi, L. V. (2015). *Code-reuse attacks and defenses* (Doctoral dissertation, Technische Universität).

Davi, L., Dmitrienko, A., Sadeghi, A. R., & Winandy, M. (2010). *Return-oriented programming without returns on ARM*. Technical Report HGI-TR-2010-002, Ruhr-University Bochum.

Davi, L., Sadeghi, A. R., Lehmann, D., & Monrose, F. (2014). Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)* (pp. 401-416).

de Clercq, R., & Verbauwhede, I. (2017). A survey of hardware-based control flow integrity (CFI). *arXiv preprint arXiv:1706.07257*.

DeMott, J. (2015). Bypassing EMET 4.1. *IEEE Security & Privacy*, *13*(4), 66-72.

Engebretson, P. (2013). *The basics of hacking and penetration testing: ethical hacking and penetration testing made easy*. Elsevier.

Erdődi, L. (2013, June). Attacking x86 windows binaries by jump oriented programming. In *Intelligent Engineering Systems (INES), 2013 IEEE 17th International Conference o*n (pp. 333-338). IEEE.

Erdődi, L. (2013, May). Finding dispatcher gadgets for jump oriented programming code reuse attacks. In *Applied Computational Intelligence and Informatics (SACI), 2013 IEEE 8th International Symposium on* (pp. 321-325). IEEE.

Erdődi, L. (2015). Applying Return Oriented and Jump Oriented Programming Exploitation Techniques with Heap Spraying. *Acta Polytechnica Hungarica*, *12*(5).

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Glass, R. L., Vessey, I., & Ramesh, V. (2001). *Research in software engineering: an empirical study*. Technical Report TR105-1, Information Systems Department, Indiana University.

Göktas, E., Athanasopoulos, E., Bos, H., & Portokalidis, G. (2014, May). Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy* (pp. 575-589). IEEE.

Göktas, E., Athanasopoulos, E., Bos, H., & Portokalidis, G. (2014, May). Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy* (pp. 575-589). IEEE.

Hevner, A., March, S. T., Park, J., & Ram, S. (2004). Design science research in information systems. *MIS quarterly*, *28*(1), 75-105.

Homescu, A., Stewart, M., Larsen, P., Brunthaler, S., & Franz, M. (2012, August). Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on Offensive Technologies* (pp. 7-7). USENIX Association.

Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., & Liang, Z. (2016, May). Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)* (pp. 969-986). IEEE.

Keizer, G. (2013, Oct. 18). Small biz admins squawk over Windows 8.1. updates. Retrieved from https://www.computerworld.com/article/2486335/small-biz-admins-squawk-over-windows-8-1-updates.html

Kennedy, J., & Satran, M. (2018, May 30.) Control flow guard. MSDN. Retrieved from https://docs.microsoft.com/en-us/windows/desktop/SecBP/control-flow-guard

Kennedy, J., & Satran, M. (2018, May 30.) File system redirector. MSDN. Retrieved from https://docs.microsoft.com/en-us/windows/desktop/WinProg64/file-system-redirector

Kornau, T. (2010). *Return oriented programming for the ARM architecture* (Doctoral dissertation, Master's thesis, Ruhr-Universität Bochum).

Krsul, I. V. (1998). *Software vulnerability analysis*. West Lafayette, IN: Purdue University.

March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. *Decision support systems*, *15*(4), 251-266.

Mashtizadeh, A. J., Bittau, A., Mazieres, D., & Boneh, D. (2014). Cryptographically enforced control flow integrity. *arXiv preprint arXiv:1408.1451*.

Min, J. W., Jung, S. M., Lee, D. Y., & Chung, T. M. (2012). Jump oriented programming on windows platform (on the x86). In *Computational Science and Its Applications–ICCSA 2012* (pp. 376-390). Springer Berlin Heidelberg.

Nelißen, J. (2002, May 1.) Buffer Overflows for Dummies. SANS Institute Information Security Reading Room. Retreived from https://www.sans.org/reading-room/whitepapers/threats/buffer-overflows-dummies-481

NSA. 2015. Hardware control flow integrity for an IT ecosystem. Retrieved from https://github.com/nsacyber/Control-Flow-Integrity/tree/master/paper

Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., & Kirda, E. (2010, December). G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference* (pp. 49-58). ACM.

One, A. (1996). Smashing the stack for fun and profit. *Phrack magazine*, *7*(49), 14-16.

Payer, M., Barresi, A., & Gross, T. R. (2015, July). Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 144-164). Springer, Cham.

Prieto-Diaz, R. (1990, March). Implementing faceted classification for software reuse. In *[1990] Proceedings. 12th International Conference on Software Engineering* (pp. 300-304). IEEE.

ProcessMitigations. (2018.) Microsoft. Retrieved from https://docs.microsoft.com/en-us/powershell/module/processmitigations/?view=win10-ps

Protect devices from exploits. (2018.) Microsoft. Retrieved from https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-exploit-guard/exploit-protection-exploit-guard

Qiao, R., Zhang, M., & Sekar, R. (2015, December). A Principled Approach for ROP Defense. *Proceedings of the 31st Annual Computer Security Applications Conference* (pp. 101-110).

Roemer, R. G. (2009). *Finding the bad in good code: Automated return-oriented programming exploit discovery* (Doctoral dissertation, UC San Diego).

Roemer, R., Buchanan, E., Shacham, H., & Savage, S. (2012). Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, *15*(1), 2.

Roemer, R., Buchanan, E., Shacham, H., & Savage, S. (2012). Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, *15*(1), 2.

Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A. R., & Holz, T. (2015, May). Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy* (pp. 745-762). IEEE.

Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A. R., & Holz, T. (2015, May). Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy* (pp. 745-762). IEEE.

Shacham, H. (2007, October). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security* (pp. 552-561).

Showcases – Capstone – The Ultimate Disassembler. Retrieved from https://www.capstone-engine.org/showcase.html

Spisak, M. (2017, April 25). Disarming control flow guard using advanced code reuse attacks. Endgame. Retrieved from https://www.endgame.com/blog/technical-blog/disarming-control-flow-guard-using-advanced-code-reuse-attacks

Stroschein, J. (2017). *Binary Analysis Framework* (Doctoral dissertation, Dakota State University).

Tang, J., & Team, T. M. T. S. (2015). Exploring control flow guard in windows 10. Retrieved from https://sjc1-te-ftp.trendmicro.com/.../exploring-control-flow-guard-in-windows10.pd

The enhanced mitigation experience toolkit. (2018). Microsoft Support. Retrieved from https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit

Van Eeckhoutte, P. (2010, Jun. 16). Exploit writing tutorial part 10: chaining DEP with ROP – the rubik's[TM} cube. *Corelan Team*. Retrieved from https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/

Wieringa, R. (2013). Empirical research methods for technology validation: Scaling up to practice. *Journal of systems and software, 95*, 19-31.

Wieringa, R. (2014). *Design science methodology for information systems and software engineering.* Berlin: Springer.

Williams, C. (2016, Jun. 10). Rip ROP: Intel's cunning plot to kill stack-hopping exploits at CPU level. *The Register.* Retrieved from https://www.theregister.co.uk/2016/06/10/intel_control_flow_enforcement/

Winitor. (2010, November). Windows dynamic-link libraries. Retrieved from https://winitor.com/pdf/DynamicLinkLibraries.pdf

Wojtczuk, R., & DeMott, J. (2015). Gadgets Zoo: Bypassing control flow guard in Windows 10. Iron Geek. Retrieved from http://www.irongeek.com/i.php?page=videos/derbycon5/break-me02-gadgets-zoo-bypassing-control-flow-guard-in-windows-10-rafal-wojtczuk-jared-demott

Ye, T., Zhang, L., Wang, L., & Li, X. (2016, April). An empirical study on detecting and fixing buffer overflow bugs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (pp. 91-101). IEEE.

Yin, R. K. (2003). Case study research: design and methods, (3rd) Sage Publications. *Thousand Oaks, California*.

Zatko, P. (1995, Oct. 20). How to write buffer overflows. Retrieved from https://insecure.org/stf/mudge_buffer_overflow_tutorial.html

Zelkowitz, M. V., & Wallace, D. R. (1998). Experimental models for validating technology. *Computer*, *31*(5), 23-31.

Zendler, A., Horn, E., Schwärtzel, H., & Plödereder, E. (2001). Demonstrating the usage of single-case designs in experimental software engineering. *Information and Software Technology*, *43*(12), 681-691.

Zendler, A., Horn, E., Schwärtzel, H., & Plödereder, E. (2001). Demonstrating the usage of single-case designs in experimental software engineering. *Information and Software Technology*, *43*(12), 681-691.

Zetter, K. (2014). *Countdown to Zero Day: Stuxnet and the launch of the world's first digital weapon*. Broadway books.

Zowghi, D., & Gervasi, V. (2002, September). The Three Cs of requirements: consistency, completeness, and correctness. In *International Workshop on Requirements Engineering: Foundations for Software Quality, Essen, Germany: Essener Informatik Beitiage* (pp. 155-164). Retrieved from http://islp.di.unipi.it/~gervasi/Papers/refsq02.pdf

APPENDIX A: FREQUENCY OF JOP GADGETS FOUND IN SELECT BINARIES

Binaries Tested

Table 10 depicts the binaries that were scanned for purposes of this research to enumerate the number of gadgets according to classification.

Table 10. Select binaries scanned for JOP GADGETS

Acronym	Software	File Location
ACR	**Adobe** Acrobat Reader	C:\Program Files (x86)\Adobe\Reader 11.0\Reader\AcroRd32.exe
BRL	BrLauncher.exe	C:\Program Files (x86)\Brother\BrLauncher\BrLauncher.exe
WDB	WinDbg	C:\Program Files (x86)\Debugging Tools for Windows\windbg.exe
FLX	Filezilla	C:\Program Files (x86)\FileZilla FTP Client\filezilla.exe
HXD	HxD	C:\Program Files (x86)\HxD\HxD.exe
ICO	IcoFX 2	C:\Program Files (x86)\IcoFX 2\IcoFX2.exe
IDA	IDA Free	C:\Program Files (x86)\IDA Free\idag.exe
IMM	Immunity Debugger	C:\Program Files (x86)\Immunity Inc\Immunity Debugger\ImmunityDebugger.exe
IEX	Internet Explorer	C:\Program Files (x86)\Internet Explorer\iexplore.exe
DEV	Microsoft Visual Studio 14	C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\devenv.exe
NOT	Notepad++	C:\Program Files (x86)\Notepad++\notepad++.exe
SOFF	Open Office 4	C:\Program Files (x86)\OpenOffice 4\program\soffice.exe
POW	Power MP3 WMA Converter WMA Converter	C:\Program Files (x86)\Power MP3 WMA Converter\PowerConverter.exe
QUE	Question Writer HTML5	C:\Program Files (x86)\Question Writer HTML5\QuestionWriter.exe

Acronym	Software	File Location
PEB	PEBrowse Pro PEBrowse Pro 86	C:\Program Files (x86)\SmidgeonSoft\PEBrowsePro\PEBrowsePro.exe
STA	Steam	C:\Program Files (x86)\Steam\Steam.exe
SNA	TechSmith Snaggit	C:\Program Files (x86)\TechSmith\Snagit 12\Snagit32.exe
SNE	TechSmith Snaggit Editor	C:\Program Files (x86)\TechSmith\Snagit 12\SnagitEditor.exe
VMW	VMware Workstation	C:\Program Files (x86)\VMware\VMware Workstation\vmware.exe
VUD	VUDUToGo VUDUToGo\VUDUTToGo	C:\Program Files (x86)\VUDUToGo\VUDUToGo.exe
WM	Windows Media Player	C:\Program Files (x86)\Windows Media Player\wmplayer.exe
WIN	WinRARin WinRAR\WinRAR.	C:\Program Files (x86)\WinRAR\WinRAR.exe
MSP	Microsoft Paint	C:\Windows\SysWOW64\mspaint.exe
EXP	Explorer	C:\Windows\SysWOW64\explorer.exe
CER	Cert Util	C:\Windows\SysWOW64\certutil.exe
TAS	Task Manager	C:\Windows\SysWOW64\Taskmgr.exe
MAG	Magnify	C:\Windows\SysWOW64\Magnify.exe
REG	Regedit	C:\Windows\SysWOW64\regedit.exe
CMD	Cmd	C:\Windows\SysWOW64\cmd.exe
NOT	Notepad	C:\Windows\SysWOW64\notepad.exe
RES	Respondus	C:\Program Files (x86)\RespondusCampus40\Respond.exe

JOP Gadgets for Scanned Applications – Image Only

Table 11 and Table 12 depict the number of gadgets obtained from each of the binaries that was scanned; the results are spread across two tables. The average number of gadgets from all binaries scanned is provided. Standard default settings were used. Had the settings been modified, the numbers produced likely would increase or decrease.

Table 11. JOP Gadgets for 31 applications (image only) – Part 1.

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOFF	POW	QUE	PEB	STA	SNA
JMP EAX	413.9	0	111	7560	291	18	635	12	18	25	0	100	2	245	331	1053
JMP EBX	7.466667	0	6	2	0	1	0	0	0	0	0	0	13	0	0	1
JMP ECX	10.36667	0	7	24	3	0	82	0	0	0	0	0	0	6	32	4
JMP EDX	77.9	0	0	412	7	0	113	0	0	0	0	25	7	8	10	912
JMP EDI	6.866667	0	0	22	0	0	0	0	0	3	0	0	6	0	0	0
JMP ESI	9.466667	6	0	9	0	0	7	0	0	6	0	6	2	18	11	2
JMP EBP	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
CALL EAX	5276.9	1524	2081	25690	984	120	1139	0	97	9607	0	2786	0	1039	6845	42757
CALL EBX	1665.8	0	4730	1033	65	40	239	0	173	4224	0	535	0	51	444	14863
CALL ECX	272.3333	571	377	263	31	52	123	0	0	1275	0	1427	0	20	562	1378
CALL EDX	4684.5	977	467	3183	11	83	459	0	8	41	0	2698	4	10	298	58547
CALL EDI	2112.267	0	4536	862	45	139	301	46	257	3847	0	1057	0	5	483	20777
CALL ESI	2277.367	23	9654	1147	20	220	227	7	323	8681	34	1074	14	15	737	10060
CALL EBP	86.23333	0	0	539	28	0	185	0	0	0	0	28	2	1	276	45
DG EAX	92.33333	0	217	412	272	5	152	0	23	29	0	37	0	217	134	264
DG EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EAX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOFF	POW	QUE	PEB	STA	SNA
DG BEST EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EAX	0.533333	0	0	6	0	0	0	0	0	0	0	0	0	0	0	5
DG Other EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADD all	2379.633	217	4103	6053	517	89	824	15	189	3018	0	1448	13	418	1442	18249
ADD EAX	1183.8	154	1729	3328	376	42	528	13	103	1014	0	857	7	348	764	8013
ADD EBX	128.3333	2	131	1209	38	11	63	0	12	149	0	64	0	9	88	785
ADD ECX	198.1333	56	1445	201	31	4	50	0	9	203	0	119	5	8	108	1379
ADD EDX	497.2667	0	465	63	26	4	12	0	51	1217	0	261	0	29	202	4786
ADD EDI	122.1333	1	95	66	9	0	15	2	4	206	0	50	0	5	85	1176
ADD ESI	83.6	0	57	36	9	9	7	0	7	148	0	23	0	2	59	805
ADD EBP	20.03333	4	79	28	4	0	0	0	0	33	0	23	0	3	27	145
SUB all	486.5333	78	1261	1281	20	28	258	3	32	342	0	231	2	14	326	4425
SUB EAX	279.1667	9	621	692	9	17	226	2	27	220	0	137	2	5	218	2421
SUB EBX	81.53333	61	463	274	0	6	27	1	4	69	0	42	0	2	45	598
SUB ECX	33.7	19	17	57	9	1	26	0	1	53	0	25	0	7	11	303
SUB EDX	47.3	0	77	24	2	0	4	0	2	43	0	35	0	2	64	519
SUB EDI	13.56667	0	13	4	2	0	0	0	2	11	0	18	0	0	40	122
SUB ESI	15.8	50	8	0	0	4	4	0	0	10	0	22	0	0	27	129

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOF F	POW	QUE	PEB	STA	SNA
SUB EBP	1.833333	0	0	1	0	0	0	0	0	3	0	3	0	0	1	11
MUL all	15.76667	0	4	5	12	0	35	0	4	0	0	25	1	5	17	67
MUL EAX	2.833333	0	2	2	0	0	23	0	0	0	0	7	0	0	9	17
MUL EBX	0.133333	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0
MUL ECX	0.666667	0	0	0	0	0	2	0	0	0	0	0	0	0	2	2
MUL EDX	4.566667	0	2	0	0	0	0	0	0	0	0	5	0	0	0	16
MUL EDI	0.133333	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MUL ESI	0.133333	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MUL EBP	0.933333	0	1	0	0	0	4	0	0	0	0	0	0	0	0	3
DIV all	36.6	8	32	31	3	2	3	0	0	7	0	19	7	0	34	296
DIV EAX	36.6	8	32	31	3	2	3	0	0	7	0	19	7	0	34	296
DIV EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EDX	36.6	8	32	31	3	2	3	0	0	7	0	19	7	0	34	296
DIV EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV all	9042.433	2737	3516	14379	160	273	2252	25	178	4331	7	6970	5	477	3144	93558
MOV EAX	2586.033	780	1041	4801	54	114	145	0	34	958	0	2203	0	207	2244	28555
MOV EBX	119.6333	4	146	193	16	23	47	0	11	245	0	74	5	12	51	972
MOV ECX	3278.167	965	1464	3097	4	20	8	23	7	2318	1	2288	0	219	540	32024
MOV EDX	2292.467	986	173	279	31	35	53	0	9	85	0	2086	0	10	77	27234
MOV EDI	152.4333	0	232	39	22	31	98	0	58	166	0	69	0	13	35	1524
MOV ESI	235.9333	0	299	47	3	25	113	2	24	451	0	100	0	6	90	2217
MOV EBP	10.6	2	18	23	10	10	16	0	0	10	0	22	0	0	8	48
MOV SHUFFLE all	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOFF	POW	QUE	PEB	STA	SNA
MOV SHUFFLE EAX	13.16667	0	7	134	15	31	4	0	3	10	0	4	0	4	6	45
MOV SHUFFLE EBX	59.1	0	93	89	11	20	26	0	7	79	0	25	0	8	11	470
MOV SHUFFLE ECX	2534.367	36	1119	2741	0	0	0	23	0	2106	0	608	0	0	414	24638
MOV SHUFFLE EDX	8.166667	0	0	10	19	11	9	0	0	0	0	0	0	6	0	79
MOV SHUFFLE EDI	88.4	0	164	14	7	28	12	0	7	120	0	54	0	3	14	901
MOV SHUFFLE ESI	130.2333	0	80	34	3	24	8	0	18	174	0	49	0	6	19	1509
MOV SHUFFLE EBP	4.466667	2	0	9	5	10	16	0	0	2	0	20	0	0	5	0
MOV VALUE all	283.7333	11	292	2658	30	9	17	0	18	237	1	65	5	219	122	1847
MOV VALUE EAX	166	11	174	2485	16	5	5	0	9	88	0	15	0	15	61	767
MOV VALUE EBX	21.56667	0	8	27	0	0	0	0	1	58	0	22	5	1	20	168
MOV VALUE ECX	25.16667	0	64	12	1	2	1	0	6	33	1	7	0	195	6	223
MOV VALUE EDX	7.066667	0	7	30	0	1	2	0	0	22	0	3	0	1	9	43
MOV VALUE EDI	13.46667	0	1	24	8	0	0	0	0	2	0	4	0	4	2	157
MOV VALUE ESI	15.8	0	13	12	0	1	8	0	2	6	0	5	0	0	8	155
MOV VALUE EBP	5.4	0	14	14	0	0	0	0	0	8	0	1	0	0	2	44

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOF F	POW	QUE	PEB	STA	SNA
LEA all	2039.533	10	9313	2817	46	15	121	0	60	588	1	733	0	53	756	19056
LEA EAX	675.1	4	2410	57	40	12	48	0	51	389	1	364	0	53	499	5898
LEA EBX	10.5	0	11	22	0	0	0	0	0	8	0	6	0	0	0	79
LEA ECX	961.4	0	6799	124	0	0	30	0	8	148	0	260	0	0	167	9419
LEA EDX	280.1333	3	74	78	6	3	41	0	0	27	0	86	0	0	42	3387
LEA EDI	16.46667	0	7	3	0	0	0	0	0	3	0	4	0	0	28	190
LEA ESI	10.33333	0	11	11	0	0	2	0	1	13	0	0	0	0	15	77
LEA EBP	1.1	0	1	1	0	0	0	0	0	0	0	10	0	0	5	0
PUSH all	17677.07	1818	18457	997	619	587	269	8	1134	40861	30	13540	14	672	15531	166423
PUSH EAX	4085.867	470	5885	147	342	72	12	0	166	4172	4	1926	0	423	2104	42170
PUSH EBX	1002.5	8	1242	56	45	22	23	0	48	956	0	850	0	21	1246	8506
PUSH ECX	1848.967	489	545	3	5	36	3	0	11	280	0	1183	0	4	1001	22182
PUSH EDX	1333.333	523	278	38	4	70	17	0	1	126	0	842	0	2	509	16072
PUSH EDI	1189.5	7	613	7	12	177	0	0	131	1063	0	998	1	31	1386	11338
PUSH ESI	1057.367	0	498	11	30	14	0	0	77	980	0	2269	9	3	2981	9840
PUSH EBP	194.9	86	28	155	24	11	58	0	2	48	0	488	0	1	571	1369
POP all	1219.467	129	499	17164	70	15	765	28	57	2069	3	330	7	54	283	6273
POP EAX	67	18	43	156	16	1	3	0	1	501	0	15	0	6	20	388
POP EBX	160.5	0	12	4111	8	3	110	0	0	174	0	10	0	4	1	221
POP ECX	43.6	0	4	5	0	2	0	12	14	55	3	141	0	3	1	177
POP EDX	37.03333	0	2	2	12	1	0	12	14	100	0	40	0	10	1	189
POP EDI	141.2667	0	0	3327	3	2	34	0	0	43	0	9	0	3	96	495
POP ESI	181.5333	87	83	3588	8	3	38	0	5	24	0	12	0	5	18	943
POP EBP	282.1	0	120	5748	10	2	540	0	4	23	0	14	0	5	51	838
INC all	1014.267	183	1453	6531	173	58	422	0	32	849	3	651	4	105	958	7143
INC EAX	167.3667	11	93	1671	52	12	52	0	3	106	1	113	0	25	302	1002
INC EBX	27.5	0	32	44	7	7	28	0	3	59	0	18	0	17	37	181
INC ECX	23.63333	29	20	20	23	6	1	0	0	36	0	57	3	1	103	92

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOF F	POW	QUE	PEB	STA	SNA
INC EDX	87.7	15	20	111	7	0	12	0	0	30	0	105	0	0	13	752
INC EDI	64.7	0	52	400	1	1	4	0	4	94	0	36	0	9	78	364
INC ESI	85.33333	0	217	20	8	6	5	0	2	244	0	76	0	5	208	679
INC EBP	361.8667	105	963	2517	48	22	54	0	15	188	2	171	0	37	63	2742
DEC all	657.2667	270	1573	1035	155	39	86	0	13	343	0	356	6	130	290	5940
DEC EAX	116.1333	29	331	154	6	1	5	0	0	84	0	43	0	0	94	986
DEC EBX	14.6	0	43	6	6	2	3	0	0	11	0	6	0	20	8	100
DEC ECX	22.83333	8	271	15	26	0	2	0	0	9	0	4	0	9	8	126
DEC EDX	22.3	23	106	2	0	4	1	0	0	20	0	14	2	3	6	151
DEC EDI	32.43333	5	77	3	3	0	3	0	0	13	0	13	2	1	3	272
DEC ESI	53.53333	0	37	5	2	3	0	0	0	104	0	41	0	1	14	592
DEC EBP	209.0333	168	504	139	97	16	18	0	1	45	0	132	0	82	31	2078
XCHG all	202.4333	25	88	136	11	21	14	0	5	219	0	87	11	6	82	2290
XCHG EAX	126.3333	25	36	95	9	20	5	0	4	90	0	30	5	4	38	1295
XCHG EBX	12.83333	0	2	13	0	0	0	0	0	9	0	0	0	4	2	166
XCHG ECX	5.666667	5	7	7	0	0	0	0	0	11	0	8	0	0	18	58
XCHG EDX	46.76667	0	8	25	0	2	1	0	0	5	0	2	5	0	6	546
XCHG EDI	18.63333	0	15	12	0	0	6	0	0	41	0	2	0	2	0	287
XCHG ESI	12.5	0	3	11	0	1	0	0	0	5	0	19	0	0	0	196
XCHG EBP	12.83333	17	2	16	2	0	4	0	2	10	0	5	0	0	5	129
SHIFT LEFT	126.6333	3	1034	37	0	3	3	0	36	375	5	117	6	0	113	696
SHIFT RIGHT	51.23333	3	163	131	0	0	0	2	0	45	0	36	6	4	16	437
ROTATE LEFT	37.93333	10	89	66	14	2	0	0	1	66	0	20	0	2	65	332
ROTATE RIGHT	16.43333	0	13	15	1	0	5	0	6	40	0	11	0	6	42	148

Table 12. JOP Gadgets for 31 applications (image only) – Part 2

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
JMP EAX	413.9	907	0	0	10	25	3	181	42	144	3	10	0	183	35	473	332
JMP EBX	7.466667	6	0	0	0	0	0	0	0	3	0	0	0	0	0	192	11
JMP ECX	10.36667	6	0	0	0	0	0	0	0	0	0	0	0	0	0	147	41
JMP EDX	77.9	661	0	0	0	0	0	5	0	0	0	0	0	3	0	174	20
JMP EDI	6.866667	0	0	0	0	0	0	0	7	0	0	0	0	0	0	168	15
JMP ESI	9.466667	4	0	6	0	6	3	17	0	1	0	0	3	6	6	165	9
JMP EBP	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
CALL EAX	5276.9	54623	0	41	0	1283	0	8	0	7	0	3	0	4896	1505	1272	36913
CALL EBX	1665.8	16180	2	44	5	2806	14	195	19	76	0	2	0	1310	2338	586	2143
CALL ECX	272.3333	937	1	6	0	57	0	10	15	0	0	9	0	776	61	219	732
CALL EDX	4684.5	69574	0	0	0	51	0	1	0	0	0	0	0	3601	363	159	11471
CALL EDI	2112.267	20088	14	24	1	1752	96	376	73	143	0	2	0	2272	5716	456	1324
CALL ESI	2277.367	9955	736	242	1	5206	5465	2594	1161	3084	1139	14	70	1694	4324	400	2025
CALL EBP	86.23333	45	0	0	0	1233	8	3	0	0	0	0	0	0	0	194	1279
DG EAX	92.33333	487	0	4	0	20	0	11	0	5	0	1	0	87	40	353	1192
DG EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EAX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
DG BEST ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EAX	0.533333	3	0	0	0	2	0	0	0	0	0	0	0	0	0	0	3
DG Other EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADD all	2379.633	20627	165	39	13	2656	1102	1625	399	760	257	11	55	1933	1655	3497	5947
ADD EAX	1183.8	8757	164	21	11	1192	1091	1582	367	736	256	10	55	854	636	2506	3256
ADD EBX	128.3333	760	0	2	0	183	4	16	11	4	0	0	0	98	125	86	110
ADD ECX	198.1333	1873	0	6	0	170	3	7	1	8	1	1	0	110	96	50	1571
ADD EDX	497.2667	5484	0	0	0	652	0	0	1	5	0	0	0	514	455	691	284
ADD EDI	122.1333	1349	1	3	2	187	4	7	19	4	0	0	0	127	177	70	185
ADD ESI	83.6	960	0	0	0	112	0	5	0	1	0	0	0	96	102	70	99
ADD EBP	20.03333	159	0	1	0	36	0	0	0	1	0	0	0	18	26	14	43

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
SUB all	486.5333	5008	9	11	1	355	25	27	3	32	10	0	0	350	389	75	530
SUB EAX	279.1667	2874	9	8	1	227	25	18	2	32	10	0	0	265	246	52	433
SUB EBX	81.53333	682	0	1	0	89	0	6	0	0	0	0	0	33	36	7	43
SUB ECX	33.7	398	0	0	0	22	0	5	0	0	0	0	0	7	33	17	25
SUB EDX	47.3	485	0	3	0	43	0	3	1	0	0	0	0	48	63	1	40
SUB EDI	13.56667	140	0	1	0	12	0	3	0	0	0	0	0	3	36	0	13
SUB ESI	15.8	139	0	0	0	40	0	0	0	0	0	0	0	13	28	0	30
SUB EBP	1.833333	19	0	0	0	7	0	0	0	0	0	0	0	3	2	5	0
MUL all	15.76667	78	0	0	0	77	0	1	41	0	0	0	0	18	16	67	13
MUL EAX	2.833333	20	0	0	0	2	0	0	0	0	0	0	0	2	0	1	0
MUL EBX	0.133333	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	3
MUL ECX	0.666667	2	0	0	0	1	0	0	0	0	0	0	0	0	4	7	0
MUL EDX	4.566667	16	0	0	0	47	0	0	41	0	0	0	0	9	1	0	0
MUL EDI	0.133333	3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
MUL ESI	0.133333	3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
MUL EBP	0.933333	0	0	0	0	0	0	0	0	0	0	0	0	1	0	19	0
DIV all	36.6	288	0	0	0	28	0	63	2	0	0	0	6	201	62	6	30
DIV EAX	36.6	288	0	0	0	28	0	63	2	0	0	0	6	201	62	6	30
DIV EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EDX	36.6	288	0	0	0	28	0	63	2	0	0	0	6	201	62	6	30
DIV EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV all	9042.433	108322	717	37	0	2063	5167	1801	993	3013	1221	23	20	10582	4263	1039	61431

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
MOV EAX	2586.033	30965	0	0	0	744	1	28	3	5	0	0	0	3041	1135	523	15402
MOV EBX	119.6333	1202	4	5	0	119	2	5	0	1	0	0	0	156	226	70	105
MOV ECX	3278.167	38139	662	0	0	385	4922	1696	894	2954	1092	21	18	3240	1239	105	23902
MOV EDX	2292.467	32771	0	1	0	101	0	0	2	2	0	0	0	3621	1080	138	21363
MOV EDI	152.4333	1709	1	4	0	110	9	4	0	0	0	0	0	143	241	65	178
MOV ESI	235.9333	2540	50	0	0	129	233	49	88	47	129	2	2	218	189	25	129
MOV EBP	10.6	53	0	2	0	70	0	2	6	0	0	0	0	11	5	2	16
MOV SHUFFLE all	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EAX	13.16667	47	0	0	0	2	0	0	0	0	0	0	0	19	5	59	14
MOV SHUFFLE EBX	59.1	592	0	0	0	61	0	0	0	0	0	0	0	79	154	48	44
MOV SHUFFLE ECX	2534.367	30289	632	0	0	315	4832	1673	864	2854	1092	16	18	1226	527	8	267
MOV SHUFFLE EDX	8.166667	88	0	0	0	0	0	0	0	0	0	0	0	0	0	23	28
MOV SHUFFLE EDI	88.4	985	0	0	0	88	0	0	0	0	0	0	0	90	151	14	94
MOV SHUFFLE ESI	130.2333	1684	0	0	0	91	0	0	0	0	0	0	0	84	103	21	85
MOV SHUFFLE EBP	4.466667	0	0	2	0	58	0	0	0	0	0	0	0	2	2	1	11

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
MOV VALUE all	283.7333	2040	8	7	0	171	2	48	6	10	0	0	0	253	233	203	196
MOV VALUE EAX	166	979	0	0	0	56	1	25	0	4	0	0	0	84	68	112	128
MOV VALUE EBX	21.56667	191	4	5	0	41	1	5	0	0	0	0	0	40	42	8	4
MOV VALUE ECX	25.16667	137	4	0	0	13	0	4	0	0	0	0	0	7	7	32	6
MOV VALUE EDX	7.066667	42	0	0	0	8	0	0	0	2	0	0	0	22	5	15	3
MOV VALUE EDI	13.46667	123	0	0	0	5	0	1	0	0	0	0	0	7	49	17	8
MOV VALUE ESI	15.8	164	0	0	0	19	0	0	0	0	0	0	0	41	38	2	3
MOV VALUE EBP	5.4	48	0	0	0	11	0	2	6	0	0	0	0	9	3	0	5
LEA all	2039.533	22766	10	16	0	861	20	86	0	27	0	0	0	1322	2114	395	2048
LEA EAX	675.1	7288	0	6	0	697	0	0	0	0	0	0	0	371	1883	182	981
LEA EBX	10.5	109	0	0	0	2	0	0	0	0	0	0	0	65	12	1	1
LEA ECX	961.4	10975	10	10	0	123	20	86	0	27	0	0	0	466	149	21	573
LEA EDX	280.1333	4091	0	0	0	13	0	0	0	0	0	0	0	344	20	189	457
LEA EDI	16.46667	204	0	0	0	3	0	0	0	0	0	0	0	20	31	1	13
LEA ESI	10.33333	95	0	0	0	10	0	0	0	0	0	0	0	56	19	0	10
LEA EBP	1.1	2	0	0	0	13	0	0	0	0	0	0	0	0	0	1	12
PUSH all	17677.07	195840	12	618	3	24893	140	135	47	131	41	3	3	23160	22976	1350	29801
PUSH EAX	4085.867	49791	9	91	0	3501	40	37	0	18	0	0	0	6098	4416	682	6409

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
PUSH EBX	1002.5	10712	0	63	0	2226	1	12	0	4	37	0	0	1672	2273	52	1206
PUSH ECX	1848.967	25408	1	19	0	855	8	2	4	3	0	0	0	2623	772	32	3368
PUSH EDX	1333.333	18847	0	0	0	90	5	3	23	8	0	0	3	2312	194	33	2748
PUSH EDI	1189.5	12800	0	90	0	4020	2	5	0	12	3	1	0	1791	1162	35	1675
PUSH ESI	1057.367	10541	0	26	0	1430	2	3	0	0	0	0	0	2018	923	66	3491
PUSH EBP	194.9	1532	0	0	0	978	0	12	2	0	0	0	0	429	26	27	3238
POP all	1219.467	6567	4	20	12	317	52	324	75	264	6	13	4	461	361	358	233
POP EAX	67	502	0	1	0	67	9	1	2	15	0	0	0	59	141	45	61
POP EBX	160.5	128	0	0	0	5	0	0	0	0	0	0	0	7	6	15	1
POP ECX	43.6	287	0	17	6	72	3	165	36	125	3	6	0	74	77	20	21
POP EDX	37.03333	313	0	0	6	8	3	143	36	119	3	6	0	58	6	27	4
POP EDI	141.2667	170	0	0	0	13	0	1	0	0	0	0	0	20	14	8	4
POP ESI	181.5333	583	0	0	0	8	0	0	0	0	0	0	0	8	2	31	8
POP EBP	282.1	934	0	0	0	20	1	1	0	1	0	0	0	101	36	14	31
INC all	1014.267	8411	5	16	2	687	37	187	15	17	0	5	6	804	923	748	5004
INC EAX	167.3667	1133	0	16	1	97	22	110	4	3	0	0	0	70	60	62	238
INC EBX	27.5	307	0	0	0	11	0	9	1	0	0	0	0	22	25	17	37
INC ECX	23.63333	160	0	0	0	13	0	6	0	0	0	0	6	51	62	20	89
INC EDX	87.7	1086	0	0	0	1	0	1	7	5	0	4	0	134	57	271	84
INC EDI	64.7	754	0	0	0	29	1	4	1	0	0	0	0	14	81	14	100
INC ESI	85.33333	805	0	0	0	29	0	4	0	0	0	1	0	106	95	50	204
INC EBP	361.8667	2963	0	0	1	63	2	12	2	1	0	0	0	340	437	108	3835
DEC all	657.2667	7563	2	3	0	257	83	46	5	49	0	4	1	570	540	359	5672
DEC EAX	116.1333	1480	1	1	0	31	6	5	0	11	0	0	0	33	172	11	122
DEC EBX	14.6	175	0	0	0	12	1	1	0	26	0	0	0	2	5	11	4

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
DEC ECX	22.83333	136	0	0	0	7	0	0	0	0	0	0	0	13	30	21	16
DEC EDX	22.3	139	0	0	0	6	13	7	0	0	0	0	0	28	90	54	17
DEC EDI	32.43333	537	0	0	0	4	0	0	0	1	0	0	0	18	16	2	12
DEC ESI	53.53333	695	0	0	0	26	0	0	0	0	0	0	0	43	33	10	19
DEC EBP	209.0333	2546	0	1	0	26	28	0	1	0	0	0	0	258	57	43	5164
XCHG all	202.4333	2407	0	4	0	283	0	3	0	0	0	0	0	82	157	142	294
XCHG EAX	126.3333	1686	0	0	0	247	0	3	0	0	0	0	0	38	67	93	186
XCHG EBX	12.83333	176	0	0	0	2	0	0	0	0	0	0	0	1	0	10	6
XCHG ECX	5.666667	41	0	0	0	1	0	0	0	0	0	0	0	5	1	8	33
XCHG EDX	46.76667	731	0	2	0	15	0	3	0	0	0	0	0	4	12	36	18
XCHG EDI	18.63333	165	0	0	0	6	0	0	0	0	0	0	0	3	2	18	8
XCHG ESI	12.5	115	0	0	0	1	0	0	0	0	0	0	0	7	0	17	19
XCHG EBP	12.83333	70	0	0	0	93	0	0	0	0	0	0	0	9	1	20	2
SHIFT LEFT	126.6333	625	0	10	0	393	1	0	0	0	0	0	0	82	242	18	142
SHIFT RIGHT	51.23333	488	0	3	0	56	0	0	0	12	0	0	0	31	92	12	50
ROTATE LEFT	37.93333	408	0	4	0	20	0	0	1	4	0	1	0	22	9	2	41
ROTATE RIGHT	16.43333	177	0	0	0	11	0	0	1	0	0	0	0	3	12	2	49

JOP Gadgets for Scanned Applications, Image and Modules

Table 13 and Table 14 present the results obtained from scanning different binaries and their modules; the results are spread across two tables. The average number of gadgets from all binaries scanned is provided. Results have been aggregated for all the gadgets obtained from the image along with each binary's modules, with one number representing the total gadgets found from each. Standard default settings were used. Had the settings been modified, the numbers produced likely would increase or decrease.

Table 13. JOP Gadgets for 31 applications (image and associated modules) – Part 1.

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOFF	POW	QUE	PEB	STA	SNA
JMP EAX	2417.533	1138	23077	9049	1913	1672	1890	198	1351	1863	860	2620	42	1698	1363	3182
JMP EBX	21.33333	20	57	21	8	2	19	22	9	2	13	1	13	1	1	31
JMP ECX	25	34	77	24	16	9	89	25	0	0	0	42	0	6	32	79
JMP EDX	101.4667	23	53	435	15	34	136	36	22	2	15	82	7	10	11	1022
JMP EDI	26.06667	34	38	42	21	13	23	54	20	16	21	13	6	13	13	17
JMP ESI	49.33333	60	69	57	46	32	49	36	38	35	31	55	8	57	40	51
JMP EBP	13.5	5	27	5	56	13	8	9	7	5	5	5	1	5	5	9
CALL EAX	8058.567	12044	42154	25796	1061	1124	12928	78	198	9678	194	4442	14	2152	6905	53684
CALL EBX	4403.933	502	55035	1301	715	525	1968	246	687	4605	372	2250	18	478	660	23812
CALL ECX	1257.167	5189	7828	382	181	198	10202	64	103	1367	63	2028	12	450	635	4424
CALL EDX	7359.967	6478	4007	3321	145	290	18072	176	160	179	135	3172	4	845	424	67728
CALL EDI	6600.133	976	65618	2013	1316	965	4591	507	1335	7585	878	3909	24	1001	1485	38659
CALL ESI	27484.43	11250	238800	16146	19155	11356	15095	1895	28351	37409	10603	22672	643	16773	13297	42386
CALL EBP	131.4333	3	65	543	39	18	185	20	4	0	0	272	2	47	280	673
DG EAX	195	101	1684	431	300	21	531	13	39	85	19	225	1	232	145	470
DG EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOFF	POW	QUE	PEB	STA	SNA
DG ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EAX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EAX	1.2	0	16	6	0	0	0	0	0	0	0	1	0	0	0	8
DG Other EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADD all	10837.8	5260	73224	10621	6216	4101	12599	1190	7875	11845	3384	9124	358	5380	5244	34224
ADD EAX	7370.8	4291	27238	7626	5692	3577	7586	1007	7400	9267	3195	7078	338	4883	4379	18165
ADD EBX	287.9667	86	1976	1245	121	50	624	28	56	190	17	165	3	67	109	1275
ADD ECX	552.1	543	7985	272	115	78	376	25	197	312	58	286	7	77	167	2030
ADD EDX	834.9667	150	5740	92	83	230	318	23	89	1282	32	559	0	134	226	6406
ADD EDI	561.9333	110	7326	185	144	72	1433	100	110	550	67	343	5	130	162	1824
ADD ESI	225.2333	19	1168	53	34	28	1727	2	17	162	7	75	4	20	69	1445
ADD EBP	50.23333	54	605	33	8	7	19	6	8	40	3	47	0	8	28	250

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOFF	POW	QUE	PEB	STA	SNA
SUB all	1550.3	621	13699	1657	461	860	2018	80	829	935	346	1048	7	443	672	7324
SUB EAX	981.7667	482	7982	1038	404	794	1098	51	558	748	330	672	6	404	553	3892
SUB EBX	250.6333	112	2506	292	31	26	97	18	235	103	10	77	0	27	57	1507
SUB ECX	79.63333	46	507	61	13	11	49	4	5	60	2	204	1	9	12	535
SUB EDX	155.3667	19	1660	45	24	13	57	22	15	66	13	90	0	18	71	1121
SUB EDI	86.43333	8	1051	17	23	12	28	2	10	28	7	36	0	14	46	552
SUB ESI	74.9	57	435	2	3	4	10	2	225	10	2	24	0	0	27	451
SUB EBP	4.6	0	42	1	6	0	1	0	0	3	0	3	0	0	1	17
MUL all	483.9	592	858	597	598	612	638	11	591	588	586	653	2	590	602	123
MUL EAX	7.066667	0	42	2	0	0	27	1	2	0	0	20	0	0	9	27
MUL EBX	0.8	0	12	0	0	0	2	0	0	0	0	2	0	0	0	3
MUL ECX	16.36667	19	42	19	19	20	22	0	19	19	19	22	0	19	21	6
MUL EDX	6.133333	0	29	0	0	0	0	1	0	0	0	7	0	0	0	27
MUL EDI	2.266667	1	32	1	1	3	1	0	1	1	1	1	0	1	1	4
MUL ESI	2.266667	1	32	1	1	3	1	0	1	1	1	1	0	1	1	4
MUL EBP	301.7333	391	402	391	391	391	395	6	391	391	391	391	0	391	391	3
DIV all	94.63333	86	550	64	37	32	65	0	29	45	21	76	8	38	64	576
DIV EAX	94.63333	86	550	64	37	32	65	0	29	45	21	76	8	38	64	576
DIV EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EDX	94.63333	86	550	64	37	32	65	0	29	45	21	76	8	38	64	576
DIV EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV all	31935.83	31479	127408	26679	16322	10147	48028	1550	25944	32618	9086	26259	267	15510	14236	129953
MOV EAX	5005.233	6783	27894	4902	202	330	11225	104	169	1057	85	2475	2	646	2316	35178
MOV EBX	465.8667	68	6714	212	49	57	402	4	44	297	30	152	7	44	62	1744
MOV ECX	20179.2	16139	78291	14406	14972	8929	22844	1246	23751	28989	8364	18153	230	13185	10804	51900
MOV EDX	4152.567	7914	4682	395	153	232	10015	99	157	216	97	2303	0	701	175	32016

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOFF	POW	QUE	PEB	STA	SNA
MOV EDI	359.2333	51	2809	89	65	91	816	18	94	294	49	196	0	69	83	2410
MOV ESI	1112.3	401	4598	676	813	411	801	25	1621	1608	354	1009	19	765	667	3933
MOV EBP	49.96667	50	216	63	23	37	59	37	25	25	24	56	2	11	14	171
MOV SHUFFLE all	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EAX	51.1	34	200	134	15	59	788	0	3	10	1	4	0	4	6	124
MOV SHUFFLE EBX	143.2667	7	1646	89	11	35	290	0	7	79	10	38	0	12	11	839
MOV SHUFFLE ECX	16799.93	9460	35694	13711	14123	8700	9587	1144	23066	28151	8198	15765	226	11982	10363	41523
MOV SHUFFLE EDX	20.2	0	26	10	21	32	25	0	10	0	0	0	0	6	0	134
MOV SHUFFLE EDI	183.5667	17	1513	19	12	46	561	2	12	125	14	77	0	9	19	1368
MOV SHUFFLE ESI	208.4333	0	1411	34	3	28	167	0	18	174	1	79	0	7	19	2023
MOV SHUFFLE EBP	15.8	18	14	21	15	34	26	6	8	8	6	30	2	10	11	47
MOV VALUE all	3926.433	386	53829	2933	430	338	4055	244	270	512	231	565	9	572	314	2796
MOV VALUE EAX	1124.7	139	7902	2567	130	110	1535	90	83	167	68	125	0	85	125	980
MOV VALUE EBX	184.5333	25	3964	38	24	8	32	2	26	87	11	73	5	16	26	328
MOV VALUE ECX	1327.8	6	37015	35	122	34	115	6	9	55	2	144	4	316	8	332
MOV VALUE EDX	1095.5	93	3347	121	92	131	2233	92	96	118	89	103	0	91	98	165
MOV VALUE EDI	39.6	27	93	39	34	24	31	16	14	18	23	39	0	35	17	255

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOFF	POW	QUE	PEB	STA	SNA
MOV VALUE ESI	29.36667	2	157	13	0	7	46	0	3	6	0	23	0	0	8	256
MOV VALUE EBP	31.6	32	176	42	3	3	33	31	17	17	18	13	0	1	2	99
LEA all	7558.167	535	146078	3357	701	520	1688	20	816	1419	537	1724	0	976	1185	26951
LEA EAX	7558.167	535	146078	3357	701	520	1688	20	816	1419	537	1724	0	976	1185	26951
LEA EBX	19.1	0	132	22	0	0	0	0	0	8	0	6	0	0	0	170
LEA ECX	5132.5	486	115140	654	641	460	823	17	751	946	441	1031	0	673	591	11020
LEA EDX	418.9333	15	2192	78	9	14	424	0	2	27	0	134	0	201	42	4520
LEA EDI	33.06667	4	192	7	4	4	16	0	4	8	4	8	0	8	32	332
LEA ESI	22.66667	3	163	11	0	0	28	0	1	13	0	0	0	0	15	209
LEA EBP	4.3	3	15	2	0	0	1	1	2	0	1	16	0	0	5	52
PUSH all	36746.2	16255	307395	2877	2726	2366	58244	253	2456	43408	2716	21084	32	7277	16087	266054
PUSH EAX	8259.567	4767	75670	293	583	345	14067	34	389	4604	407	2881	0	2098	2195	61506
PUSH EBX	2207.833	131	16801	92	83	133	4298	13	88	1025	243	1431	1	200	1273	16569
PUSH ECX	2726	3906	10207	48	63	110	1854	0	92	321	71	1560	0	385	1041	29148
PUSH EDX	2414.7	4464	6477	993	974	143	2459	0	47	1106	41	1957	0	1457	527	22757
PUSH EDI	2290.8	247	12734	59	51	228	6637	29	189	1133	221	1563	6	480	1413	18867
PUSH ESI	2669.2	212	17826	27	38	64	15314	8	119	1022	351	2728	15	682	2989	18639
PUSH EBP	353.8	874	949	167	36	38	131	7	41	66	12	622	0	117	582	3098
POP all	5302.233	2790	43268	20256	3349	2792	4595	739	2787	5297	1857	5003	65	3240	2551	10972
POP EAX	161.2333	279	1121	181	79	37	64	1	40	541	18	140	0	74	30	597
POP EBX	305.3	153	801	4268	175	171	273	0	8	354	16	345	0	161	158	429
POP ECX	1147.633	1020	1752	1361	1460	1023	1141	178	1185	1379	874	1813	27	1439	994	1143
POP EDX	1105.1	1025	1797	1357	1451	1013	1163	184	1180	1423	809	1600	25	1294	992	796
POP EDI	212.2667	2	823	3333	8	7	50	10	6	73	1	243	0	8	98	933
POP ESI	409.8667	96	6384	3602	16	19	68	21	13	40	10	165	0	13	26	1024
POP EBP	1046.4	17	20187	5761	20	22	562	281	40	56	19	368	5	20	61	2067
INC all	3007.4	1822	39342	6655	414	699	1910	88	251	1220	147	1459	11	372	1043	11861
INC EAX	350.8333	218	2297	1704	161	512	244	15	66	187	40	263	0	81	329	1610

Op.	Average	WDB	VMW	FLX	HXD	IDA	IMM	IEX	DEV	NOT	SOFF	POW	QUE	PEB	STA	SNA
INC EBX	100.6	11	1360	50	12	23	329	0	8	74	6	78	0	28	42	352
INC ECX	70.1	94	860	25	55	12	40	3	5	42	6	74	3	13	108	233
INC EDX	139.0333	244	726	122	9	3	48	12	10	34	8	168	0	2	15	987
INC EDI	166.4667	13	1533	412	9	19	91	8	29	242	10	143	2	14	81	934
INC ESI	402.0333	20	2819	29	20	15	190	7	12	258	11	136	0	16	217	1233
INC EBP	1376.267	1117	27814	2549	82	64	318	13	62	232	46	276	5	72	84	3566
DEC all	2172.4	2523	19812	1475	1181	482	1263	75	381	763	203	1602	17	962	626	9288
DEC EAX	224.8667	205	1597	170	36	17	102	4	52	141	12	107	0	24	105	1472
DEC EBX	140.7667	18	1388	26	39	20	229	0	18	29	16	36	0	38	24	254
DEC ECX	75.7	93	1100	18	29	42	47	1	8	18	3	20	0	12	11	258
DEC EDX	53.93333	82	730	7	7	13	22	0	12	26	4	22	2	8	10	209
DEC EDI	139.7333	25	1982	69	74	20	71	7	26	33	18	82	2	62	27	582
DEC ESI	122.2	9	1103	13	33	12	196	20	9	117	7	57	0	9	21	997
DEC EBP	638.6333	1489	9306	259	214	240	210	11	74	116	56	295	0	158	139	2332
XCHG all	651.2667	354	3947	316	215	210	431	19	41	409	31	351	11	225	251	3177
XCHG EAX	264.1667	198	2447	115	40	39	222	14	29	120	14	109	5	28	52	1514
XCHG EBX	34.13333	4	298	16	10	3	13	2	3	14	3	27	0	7	5	185
XCHG ECX	34.36667	122	249	18	11	11	172	10	13	22	11	20	0	11	29	87
XCHG EDX	61.9	3	235	26	4	4	8	0	2	10	3	5	5	3	7	620
XCHG EDI	34.03333	1	342	13	7	1	27	0	3	42	1	5	0	3	1	314
XCHG ESI	32.8	0	414	11	7	2	5	0	2	5	0	21	0	0	0	300
XCHG EBP	25.1	36	255	17	3	1	11	1	4	11	1	7	0	1	6	152
SHIFT LEFT	784.9	48	17696	63	26	206	120	76	60	390	33	271	10	38	128	1748
SHIFT RIGHT	154.8	50	2090	159	11	14	44	25	22	53	23	84	8	14	25	723
ROTATE LEFT	100.6	52	1183	71	25	14	28	13	5	74	2	91	0	13	66	510
ROTATE RIGHT	40.8	13	448	16	2	2	31	0	12	41	2	17	0	7	43	230

Table 14. JOP Gadgets for 31 applications (image and associated modules) – Part 2.

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
JMP EAX	2417.533	1735	1631	904	122	1446	1665	2026	1843	1816	1546	22	1731	390	1621	2112	2319
JMP EBX	21.33333	34	9	1	0	3	19	26	20	31	19	18	19	0	3	218	21
JMP ECX	25	60	7	1	7	5	7	25	7	12	12	7	7	0	0	160	59
JMP EDX	101.4667	701	18	1	0	6	23	37	37	40	27	21	22	3	2	203	32
JMP EDI	26.06667	7	23	14	3	13	23	32	30	25	26	10	23	0	13	196	36
JMP ESI	49.33333	56	42	29	20	41	57	70	57	52	48	13	52	20	47	212	56
JMP EBP	13.5	7	10	10	4	14	10	68	11	15	15	4	8	1	7	56	65
CALL EAX	8058.567	57507	108	85	43	1344	209	206	219	1396	113	60	128	4927	1566	1394	37031
CALL EBX	4403.933	21284	1278	730	34	3356	1084	1952	468	908	440	43	1665	1330	2991	1381	4621
CALL ECX	1257.167	1887	138	64	56	156	126	177	167	215	111	33	110	811	169	369	900
CALL EDX	7359.967	71019	3508	127	3496	180	3498	11923	3510	3524	3501	3372	3498	3725	489	293	11608
CALL EDI	6600.133	33238	2833	939	436	3003	2666	3805	1972	3368	1210	38	2176	2350	7116	1992	5010
CALL ESI	27484.43	26749	45812	12243	1032	32232	39619	26057	19832	36661	18480	253	19721	2305	35511	22195	40805
CALL EBP	131.4333	246	11	4	0	1246	13	24	4	20	9	0	6	0	4	205	1708
DG EAX	195	654	34	13	9	34	31	76	29	87	19	6	25	95	55	386	1222
DG EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
DG BEST EAX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EAX	1.2	3	0	0	0	2	0	0	0	0	0	0	0	0	0	0	3
DG Other EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
DG Other EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADD all	10837.8	27050	14236	6091	1132	10348	10979	11579	7519	10980	5701	522	5850	2501	10320	9681	17356
ADD EAX	7370.8	12900	13666	5564	935	8584	10443	10729	7035	9912	5264	373	5385	1361	8965	8286	13608
ADD EBX	287.9667	1036	111	125	42	212	93	177	90	101	59	36	89	104	183	169	225
ADD ECX	552.1	2181	193	64	23	238	138	211	121	231	93	28	100	111	163	140	1669
ADD EDX	834.9667	6289	60	71	14	694	61	115	39	422	88	9	53	519	503	748	427
ADD EDI	561.9333	1693	155	185	79	334	169	210	167	228	157	41	163	175	321	220	426
ADD ESI	225.2333	1168	19	73	10	130	26	52	17	65	19	6	29	97	120	96	545
ADD EBP	50.23333	235	9	4	2	40	8	6	6	3	5	1	5	19	30	18	47
SUB all	1550.3	6690	1009	497	88	1124	609	739	514	867	497	67	713	404	1156	535	1624
SUB EAX	981.7667	3695	725	447	83	720	531	610	421	788	430	56	456	292	737	450	1021
SUB EBX	250.6333	1257	244	21	7	331	38	56	22	30	25	4	34	35	277	40	313
SUB ECX	79.63333	502	15	2	2	23	6	17	41	9	6	3	169	7	40	28	199
SUB EDX	155.3667	946	14	22	1	63	29	58	19	31	27	7	51	49	79	31	97
SUB EDI	86.43333	533	10	12	1	31	15	23	7	17	18	0	15	4	51	22	44
SUB ESI	74.9	217	224	0	0	261	2	5	5	8	2	2	2	13	249	5	254
SUB EBP	4.6	24	0	0	0	7	8	7	1	1	0	0	0	3	2	11	6
MUL all	483.9	104	594	586	14	663	593	609	634	595	593	6	603	27	601	654	611

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
MUL EAX	7.066667	24	7	0	4	2	4	6	4	5	4	4	15	2	0	1	11
MUL EBX	0.8	4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	3
MUL ECX	16.36667	4	19	19	0	20	19	19	19	19	19	0	19	0	23	26	19
MUL EDX	6.133333	17	1	0	0	47	0	4	41	0	0	0	0	9	1	0	0
MUL EDI	2.266667	7	1	1	0	2	1	1	1	1	1	0	1	0	1	1	1
MUL ESI	2.266667	7	1	1	0	2	1	1	1	1	1	0	1	0	1	1	1
MUL EBP	301.7333	0	391	391	6	391	392	392	391	391	391	0	392	7	391	410	391
DIV all	94.63333	366	38	20	0	65	37	89	39	82	31	0	40	201	100	40	68
DIV EAX	94.63333	366	38	20	0	65	37	89	39	82	31	0	40	201	100	40	68
DIV EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EDX	94.63333	366	38	20	0	65	37	89	39	82	31	0	40	201	100	40	68
DIV EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV all	31935.83	118917	46240	9665	4416	26331	38291	36090	20889	36424	19705	3854	19201	10945	31917	19704	96056
MOV EAX	5005.233	32618	1705	143	1502	852	1563	5267	1566	1924	1599	1457	1558	3105	1239	691	15604
MOV EBX	465.8667	1789	69	52	16	378	55	313	84	490	257	14	49	158	263	103	382

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
MOV ECX	20179.2	43494	39647	8907	575	22687	33018	21567	15919	29983	14706	174	14595	3400	27015	17486	56042
MOV EDX	4152.567	33809	2338	102	2223	263	2318	7760	2314	2416	2258	2132	2271	3718	1237	263	21557
MOV EDI	359.2333	2325	75	31	44	159	60	83	45	172	47	5	41	159	287	110	273
MOV ESI	1112.3	3124	2315	363	35	1471	1163	959	826	1328	738	17	578	224	1668	859	1754
MOV EBP	49.96667	115	31	11	10	86	53	58	59	48	49	35	47	19	23	42	38
MOV SHUFFLE all	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EAX	51.1	60	0	0	0	2	6	0	0	0	0	0	0	19	5	59	14
MOV SHUFFLE EBX	143.2667	881	1	0	0	61	0	0	0	0	0	0	0	79	154	48	44
MOV SHUFFLE ECX	16799.93	34808	38413	8714	416	21852	32078	20368	15361	28705	14177	84	13943	1384	25474	16528	30728
MOV SHUFFLE EDX	20.2	122	16	0	6	10	41	43	40	10	6	6	7	0	10	25	40
MOV SHUFFLE EDI	183.5667	1284	8	5	5	93	8	8	8	8	8	3	8	92	156	19	99
MOV SHUFFLE ESI	208.4333	1981	0	0	0	91	3	3	0	0	0	0	3	84	103	21	85
MOV SHUFFLE EBP	15.8	31	10	6	6	64	14	10	14	10	10	4	10	6	12	11	17

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
MOV VALUE all	3926.433	2701	4071	318	3810	447	3971	13135	3992	4390	4023	3686	4088	427	605	645	698
MOV VALUE EAX	1124.7	1115	1640	132	1484	129	1512	5184	1523	1874	1561	1434	1523	145	142	237	262
MOV VALUE EBX	184.5333	313	44	45	11	76	27	63	38	48	28	11	25	40	70	32	56
MOV VALUE ECX	1327.8	228	104	3	78	37	125	102	95	142	128	78	222	7	129	153	145
MOV VALUE EDX	1095.5	135	2219	93	2211	105	2222	7672	2222	2224	2220	2124	2224	115	101	109	107
MOV VALUE EDI	39.6	192	17	14	16	20	19	24	18	22	17	2	27	21	71	43	36
MOV VALUE ESI	29.36667	225	2	1	2	20	3	4	3	11	3	2	3	41	38	2	5
MOV VALUE EBP	31.6	79	21	5	4	21	39	48	45	38	39	31	37	13	11	30	21
LEA all	7558.167	27307	951	448	12	1634	1115	681	532	749	671	7	688	1322	3035	1086	3136
LEA EAX	7558.167	27307	951	448	12	1634	1115	681	532	749	671	7	688	1322	3035	1086	3136
LEA EBX	19.1	155	0	0	0	2	0	0	0	0	0	0	0	65	12	1	8
LEA ECX	5132.5	11485	925	437	11	883	1096	653	519	731	660	5	677	466	1056	697	1582
LEA EDX	418.9333	4303	10	0	0	15	4	11	4	5	0	0	0	344	22	192	480

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
LEA EDI	33.06667	270	4	4	0	7	4	4	4	4	4	0	4	20	35	5	26
LEA ESI	22.66667	152	0	0	0	10	0	0	0	0	0	0	0	56	19	0	10
LEA EBP	4.3	8	1	0	0	13	1	1	1	1	1	1	1	0	0	2	12
PUSH all	36746.2	255231	1803	1130	167	27071	3099	1636	1353	4620	2590	192	2206	23226	25273	3559	36864
PUSH EAX	8259.567	60507	281	146	17	3723	407	261	161	246	182	30	248	6098	4701	940	7890
PUSH EBX	2207.833	16297	97	88	12	2265	92	80	52	598	85	16	87	1675	2314	96	1439
PUSH ECX	2726	27688	94	33	2	909	111	102	85	314	31	2	44	2623	840	96	3603
PUSH EDX	2414.7	20147	65	14	8	1073	1003	88	79	89	962	8	1014	2312	1174	1003	3875
PUSH EDI	2290.8	16579	107	114	24	4073	141	119	94	327	75	44	66	1795	1212	97	2009
PUSH ESI	2669.2	14383	31	33	3	1443	32	43	23	481	505	10	21	2018	934	82	4331
PUSH EBP	353.8	2093	32	19	2	993	36	55	65	23	14	3	27	429	42	41	3305
POP all	5302.233	9262	3476	1866	283	3428	3604	4172	3882	3910	3862	104	3587	800	3593	3677	4409
POP EAX	161.2333	653	106	12	3	110	86	70	35	81	34	2	92	60	183	108	215
POP EBX	305.3	289	163	32	2	162	165	41	160	161	155	2	163	7	163	182	171
POP ECX	1147.633	1165	1461	854	112	1342	1475	1756	1679	1603	1381	9	1569	244	1507	1483	1769
POP EDX	1105.1	852	1454	826	115	1272	1459	1694	1759	1564	1379	18	1533	220	1424	1475	1701
POP EDI	212.2667	591	14	2	9	19	13	19	10	10	11	8	11	20	23	13	17

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
POP ESI	409.8667	623	11	8	2	23	10	16	14	12	11	2	10	8	10	39	19
POP EBP	1046.4	1009	33	23	3	37	24	20	17	23	514	4	20	101	53	25	55
INC all	3007.4	10910	322	154	46	891	2120	2442	256	357	260	56	2073	818	1212	1011	5567
INC EAX	350.8333	1475	93	49	7	169	87	241	42	69	125	6	63	70	131	171	412
INC EBX	100.6	389	11	5	3	17	14	27	67	12	8	3	13	22	31	23	46
INC ECX	70.1	207	17	5	4	18	10	48	9	10	7	2	21	53	67	52	149
INC EDX	139.0333	1146	14	3	4	7	18	24	21	20	13	15	17	136	63	282	90
INC EDI	166.4667	982	27	6	2	43	25	36	31	23	17	10	17	17	188	30	221
INC ESI	402.0333	1040	19	10	7	38	1871	1887	16	19	15	7	1876	107	104	62	222
INC EBP	1376.267	3543	57	31	13	95	42	63	43	71	29	7	31	346	471	146	3887
DEC all	2172.4	8782	1264	207	324	659	2015	2258	821	1715	570	232	2062	607	1510	1493	7040
DEC EAX	224.8667	1675	54	26	3	78	51	53	25	363	33	3	36	40	220	42	203
DEC EBX	140.7667	184	195	16	178	31	203	221	194	220	196	178	202	2	24	44	39
DEC ECX	75.7	210	26	8	19	15	30	96	26	26	26	18	30	19	38	24	24
DEC EDX	53.93333	165	13	4	2	11	24	24	7	11	7	2	10	28	95	61	25
DEC EDI	139.7333	585	31	18	7	30	91	57	43	28	22	2	25	18	86	69	49
DEC ESI	122.2	763	11	10	2	35	19	46	17	11	10	2	11	43	42	41	62
DEC EBP	638.6333	2667	130	49	48	97	126	148	77	180	78	9	94	262	128	167	5326
XCHG all	651.2667	2851	2210	28	30	466	244	233	225	2039	201	19	224	93	340	347	513

Op.	Average	SNE	MAG	VUD	WM	WIN	MSP	EXP	CER	TAS	REG	CMD	NOT	ACR	BRL	ICO	RES
XCHG EAX	264.1667	1775	216	19	22	269	60	170	58	45	32	11	41	49	88	124	224
XCHG EBX	34.13333	184	76	3	3	5	11	97	9	5	5	2	9	2	3	20	18
XCHG ECX	34.36667	55	15	11	12	12	14	20	16	15	13	2	14	15	12	19	44
XCHG EDX	61.9	749	28	3	3	18	7	27	4	10	5	3	7	4	14	40	24
XCHG EDI	34.03333	170	4	1	2	7	7	18	4	3	4	2	8	3	3	25	17
XCHG ESI	32.8	126	2	0	2	1	11	25	2	2	2	2	11	7	0	24	26
XCHG EBP	25.1	81	3	3	2	94	5	10	4	3	3	2	5	9	2	21	5
SHIFT LEFT	784.9	1391	28	22	13	418	41	46	32	95	30	8	108	90	269	43	267
SHIFT RIGHT	154.8	697	46	14	4	72	43	45	35	74	29	20	35	33	109	43	71
ROTATE LEFT	100.6	559	18	4	12	26	19	45	65	23	16	13	21	22	15	13	70
ROTATE RIGHT	40.8	269	7	2	0	22	4	9	7	9	6	0	1	3	18	3	60

JOP Gadgets Results from Two Applications

Table 15, Table 16, Table 17, and Table present the results obtained from scanning two sample applications, Snaggit and Filezilla, and their modules; the results are spread across two tables. Here we see the results for the image and all its modules, the results for the image itself, and the results for each individual module. The intent with these tables is to provide a representative

sample of how the full results may appear. Both Snaggit and Filezilla present a higher than average number of gadgets. Standard default settings were used. Had the settings been modified, the numbers produced likely would increase or decrease.

Table 15. JOP Gadgets for Snaggit.exe and its associated modules – Part 1

Op.	Snagit32.exe all	Snagit 32.exe	gdiplus.dll	imagehlp.dll	CRYPT32.dll	shlwapi.dll	Opencv_core249.dll	PDFLib.dll	KERNEL32.DLL	IMM32.DLL	Scrolling Capture.dll	MSVCP100.dll	IPHLPAPI.DLL	mfc100u.dll
JMP EAX	3182	1053	14	19	63	123	1097	113	0	19	13	5	53	72
JMP EBX	31	1	2	0	0	0	0	0	0	0	0	0	0	16
JMP ECX	79	4	0	0	0	0	0	21	0	0	0	0	0	1
JMP EDX	1022	912	0	0	0	0	0	77	0	0	0	0	0	5
JMP EDI	17	0	0	0	0	0	10	0	0	0	0	0	0	0
JMP ESI	51	2	0	0	0	0	0	0	6	6	0	0	3	8
JMP EBP	9	1	2	0	0	0	0	0	0	0	0	0	0	3
CALL EAX	53684	42757	0	13	0	0	1715	5651	9	0	217	19	0	506
CALL EBX	23812	14863	255	0	3	14	95	2415	3	13	0	50	3	4058
CALL ECX	4424	1378	6	0	8	0	202	1330	16	6	0	0	1	205
CALL EDX	67728	58547	0	0	12	0	2105	5284	96	0	261	0	0	49
CALL EDI	38659	20777	344	4	9	1	229	2373	5	1	92	82	7	10401
CALL ESI	42386	10060	11967	14	31	38	125	1528	60	23	264	168	61	9206

Op.	Snagit32.exe all	Snagit 32.exe	gdiplus.dll	imagehlp.dll	CRYPT32.dll	shlwapi.dll	Opencv_core249.dll	PDFLib.dll	KERNEL32.DLL	IMM32.DLL	Scrolling Capture.dll	MSVCP100.dll	IPHLPAPI.DLL	mfc100u.dll
CALL EBP	673	45	0	0	0	0	0	66	0	0	0	0	0	5
DG EAX	470	264	0	3	1	0	22	54	3	0	22	0	0	5
DG EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EAX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Op.	Snagit32.exe all	Snagit 32.exe	gdiplus.dll	imagehlp.dll	CRYPT32.dll	shlwapi.dll	Opencv_core249.dll	PDFLib.dll	KERNEL32.DLL	IMM32.DLL	Scrolling Capture.dll	MSVCP100.dll	IPHLPAPI.DLL	mfc100u.dll
DG BEST EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EAX	8	5	0	0	0	0	0	3	0	0	0	0	0	0
DG Other EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADD all	34224	18249	3317	29	48	30	604	3425	81	41	100	72	87	2785
ADD EAX	18165	8013	3276	23	43	30	192	1716	75	35	55	50	80	1419
ADD EBX	1275	785	5	0	4	0	32	85	2	3	1	0	0	212
ADD ECX	2030	1379	4	0	0	0	99	185	0	0	8	3	0	173
ADD EDX	6406	4786	11	0	0	0	183	365	1	0	16	0	7	486
ADD EDI	1824	1176	17	4	0	0	15	182	2	2	2	2	0	231

Op.	Snagit32.exe all	Snagit 32.exe	gdiplus.dll	imagehlp.dll	CRYPT32.dll	shlwapi.dll	Opencv_core249.dll	PDFLib.dll	KERNEL32.DLL	IMM32.DLL	Scrolling Capture.dll	MSVCP100.dll	IPHLPAPI.DLL	mfc100u.dll
ADD ESI	1445	805	3	0	0	0	21	147	1	0	1	2	0	118
ADD EBP	250	145	0	0	1	0	7	14	0	0	12	1	0	24
SUB all	7324	4425	360	0	2	26	71	672	3	1	25	12	3	1265
SUB EAX	3892	2421	137	0	0	1	21	469	3	1	25	9	3	554
SUB EBX	1507	598	222	0	0	0	17	68	0	0	0	3	3	475
SUB ECX	535	303	0	0	2	0	17	71	0	0	0	3	0	81
SUB EDX	1121	519	1	0	0	0	26	74	0	0	0	0	0	428
SUB EDI	552	122	1	0	0	0	7	20	0	0	0	0	0	380
SUB ESI	451	129	221	0	0	0	0	12	0	0	0	0	0	69
SUB EBP	17	11	0	0	0	0	0	1	0	0	0	0	0	3
MUL all	123	67	0	0	0	0	8	21	0	0	0	0	0	23
MUL EAX	27	17	0	0	0	0	4	1	0	0	0	0	0	4
MUL EBX	3	0	0	0	0	0	0	0	0	0	0	0	0	3
MUL ECX	6	2	0	0	0	0	0	2	0	0	0	0	0	2
MUL EDX	27	16	0	0	0	0	3	7	0	0	0	0	0	1
MUL EDI	4	0	0	0	0	0	0	0	0	0	0	0	0	4
MUL ESI	4	0	0	0	0	0	0	0	0	0	0	0	0	4

Op.	Snagit32.exe all	Snagit 32.exe	gdiplus.dll	imagehlp.dll	CRYPT32.dll	shlwapi.dll	Opencv_core249.dll	PDFLib.dll	KERNEL32.DLL	IMM32.DLL	Scrolling Capture.dll	MSVCP100.dll	IPHLPAPI.DLL	mfc100u.dll
MUL EBP	3	3	0	0	0	0	0	0	0	0	0	0	0	0
DIV all	576	296	4	0	0	0	39	141	0	0	0	0	0	59
DIV EAX	576	296	4	0	0	0	39	141	0	0	0	0	0	59
DIV EBX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ECX	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EDX	576	296	4	0	0	0	39	141	0	0	0	0	0	59
DIV EDI	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ESI	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EBP	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV all	129953	93558	11771	1	9	51	3305	10774	126	2	643	44	1	2491
MOV EAX	35178	28555	23	1	1	0	856	4213	44	0	238	5	0	530
MOV EBX	1744	972	20	0	0	0	21	135	2	0	2	10	1	481
MOV ECX	51900	32024	10956	0	6	33	776	2712	7	0	203	7	0	311
MOV EDX	32016	27234	37	0	0	4	648	3264	69	0	173	3	0	85
MOV EDI	2410	1524	10	0	0	0	48	160	0	0	4	8	0	466
MOV ESI	3933	2217	719	0	0	2	67	239	0	0	11	3	0	351
MOV EBP	171	48	3	0	0	4	14	9	2	2	0	0	0	7

Op.	Snagit32.exe all	Snagit32.exe	gdiplus.dll	imagehlp.dll	CRYPT32.dll	shlwapi.dll	Opencv_core249.dll	PDFLib.dll	KERNEL32.DLL	IMM32.DLL	Scrolling Capture.dll	MSVCP100.dll	IPHLPAPI.DLL	mfc100u.dll
MOV SHUFFLE all	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EAX	124	45	0	0	0	0	3	50	0	0	0	0	0	7
MOV SHUFFLE EBX	839	470	0	0	0	0	7	69	0	0	0	0	0	229
MOV SHUFFLE ECX	41523	24638	10522	0	6	33	708	1205	7	0	203	0	0	118
MOV SHUFFLE EDX	134	79	10	0	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EDI	1368	901	0	0	0	0	38	88	0	0	1	1	0	261
MOV SHUFFLE ESI	2023	1509	0	0	0	0	43	118	0	0	11	1	0	234
MOV SHUFFLE EBP	47	0	0	0	0	0	0	2	2	2	0	0	0	1
MOV VALUE all	2796	1847	32	1	2	8	35	181	112	0	6	2	1	398

Op.	Snagit32.exe all	Snagit 32.exe	gdiplus.dll	imagehlp.dll	CRYPT32.dll	shlwapi.dll	Opencv_core249.dll	PDFLib.dll	KERNEL32.DLL	IMM32.DLL	Scrolling Capture.dll	MSVCP100.dll	IPHLPAPI.DLL	mfc100u.dll
MOV VALUE EAX	980	767	4	1	0	0	6	53	42	0	0	0	0	62
MOV VALUE EBX	328	168	19	0	0	0	4	11	0	0	0	0	1	107
MOV VALUE ECX	332	223	1	0	0	0	0	17	0	0	0	0	0	69
MOV VALUE EDX	165	43	3	0	0	4	0	25	68	0	0	0	0	2
MOV VALUE EDI	255	157	0	0	0	0	0	21	0	0	0	0	0	53
MOV VALUE ESI	256	155	0	0	0	0	8	27	0	0	0	1	0	51
MOV VALUE EBP	99	44	3	0	0	4	14	6	0	0	0	0	0	6
LEA all	26951	19056	240	0	0	0	1035	1155	0	0	178	49	0	3621
LEA EAX	26951	5898	2	0	0	0	411	434	0	0	18	19	0	3220
LEA EBX	170	79	0	0	0	0	5	17	0	0	0	0	0	39
LEA ECX	11020	9419	236	0	0	0	240	274	0	0	126	30	0	185
LEA EDX	4520	3387	2	0	0	0	347	334	0	0	26	0	0	60
LEA EDI	332	190	0	0	0	0	23	36	0	0	0	0	0	60
LEA ESI	209	77	0	0	0	0	3	56	0	0	8	0	0	57

Op.	Snagit32.exe all	Snagit 32.exe	gdiplus.dll	imagehlp.dll	CRYPT32.dll	shlwapi.dll	Opencv_core249.dll	PDFLib.dll	KERNEL32.DLL	IMM32.DLL	Scrolling Capture.dll	MSVCP100.dll	IPHLPAPI.DLL	mfc100u.dll
LEA EBP	52	0	0	0	0	0	6	4	0	0	0	0	0	0
PUSH all	266054	166423	377	47	15	26	6594	21071	5	8	628	321	10	41855
PUSH EAX	61506	42170	89	0	0	0	1540	4268	0	1	167	78	0	7977
PUSH EBX	16569	8506	12	26	0	0	281	1372	1	1	52	40	0	3326
PUSH ECX	29148	22182	12	0	0	0	1020	1995	0	0	81	10	0	1030
PUSH EDX	22757	16072	25	0	0	0	1195	2260	0	0	51	0	3	296
PUSH EDI	18867	11338	21	7	0	0	420	2418	2	0	53	53	0	2021
PUSH ESI	18639	9840	3	12	0	0	511	3039	0	0	32	34	0	1618
PUSH EBP	3098	1369	4	0	0	0	52	489	0	0	3	9	0	31
POP all	10972	6273	83	24	137	171	1165	594	2	36	14	29	83	1206
POP EAX	597	388	17	0	0	0	9	29	1	0	0	4	0	105
POP EBX	429	221	0	0	0	0	0	42	0	0	0	0	0	4
POP ECX	1143	177	12	7	60	89	3	9	0	18	2	23	40	294
POP EDX	796	189	15	5	60	81	0	8	0	18	0	0	40	60
POP EDI	933	495	4	0	0	0	0	13	0	0	0	0	0	406
POP ESI	1024	943	0	0	0	0	7	26	0	0	7	0	0	16
POP EBP	2067	838	7	12	2	0	1078	69	0	0	0	2	0	35

Op.	Snagit32.exe all	Snagit 32.exe	gdiplus.dll	imagehlp.dll	CRYPT32.dll	shlwapi.dll	Opencv_core249.dll	PDFLib.dll	KERNEL32.DLL	IMM32.DLL	Scrolling Capture.dll	MSVCP100.dll	IPHLPAPI.DLL	mfc100u.dll
INC all	11861	7143	92	0	5	3	304	1462	4	0	26	36	13	1843
INC EAX	1610	1002	34	0	0	0	67	145	0	0	5	1	5	243
INC EBX	352	181	0	0	0	0	5	79	0	0	0	0	0	54
INC ECX	233	92	0	0	0	0	21	37	0	0	1	0	0	40
INC EDX	987	752	4	0	0	0	64	97	0	0	0	0	0	32
INC EDI	934	364	7	0	0	3	5	320	0	0	0	6	0	45
INC ESI	1233	679	0	0	0	0	43	244	1	0	9	22	0	162
INC EBP	3566	2742	11	0	5	0	68	160	3	0	2	3	0	459
DEC all	9288	5940	66	1	8	12	365	1351	4	4	44	2	4	540
DEC EAX	1472	986	24	0	0	7	48	187	0	0	0	0	0	148
DEC EBX	254	100	1	0	0	0	9	125	0	0	0	0	0	5
DEC ECX	258	126	0	0	0	5	2	69	1	0	0	0	0	39
DEC EDX	209	151	0	0	0	0	1	15	0	0	0	0	0	2
DEC EDI	582	272	5	0	0	0	2	255	0	4	0	0	0	15
DEC ESI	997	592	1	0	0	0	22	301	0	0	0	0	0	23
DEC EBP	2332	2078	0	0	0	0	22	104	0	0	0	1	0	70
XCHG all	3177	2290	2	0	0	0	17	199	0	0	7	0	0	217

Op.	Snagit32.exe all	Snagit 32.exe	gdiplus.dll	imagehlp.dll	CRYPT32.dll	shlwapi.dll	Opencv_core249.dll	PDFLib.dll	KERNEL32.DLL	IMM32.DLL	Scrolling Capture.dll	MSVCP100.dll	IPHLPAPI.DLL	mfc100u.dll
XCHG EAX	1514	1295	1	0	0	0	7	96	0	0	7	0	0	60
XCHG EBX	185	166	0	0	0	0	0	1	0	0	0	0	0	4
XCHG ECX	87	58	0	0	0	0	0	15	0	0	0	0	0	7
XCHG EDX	620	546	1	0	0	0	0	57	0	0	0	0	0	9
XCHG EDI	314	287	0	0	0	0	1	7	0	0	0	0	0	3
XCHG ESI	300	196	0	0	0	0	9	31	0	0	0	0	0	11
XCHG EBP	152	129	0	0	0	0	4	8	0	0	0	0	0	11
SHIFT LEFT	1748	696	4	0	0	0	19	100	4	4	10	9	0	570
SHIFT RIGHT	723	437	8	0	0	0	12	50	2	0	1	0	0	153
ROTATE LEFT	510	332	1	0	0	0	7	30	0	0	20	1	0	49
ROTATE RIGHT	230	148	5	0	0	0	4	12	0	0	11	1	0	21

Table 16. JOP Gadgets for Snaggit.exe and its associated modules – Part 2

Op.	Ltkrn15u.dll	WINTRUST.dll	Lttwn15u.dll	VideoCommon.dll	RPCRT4.dll	COMCTL32.dll	Ltdis15u.dll	Trackerbird.dll	GDI32.dll	Ltimgclr15u.dll	ADVAPI32.dll	MSVCR100.dll	MSIMG32.dll
JMP EAX	28	49	5	21	43	0	3	155	154	0	57	23	0
JMP EBX	0	0	0	0	0	0	0	0	0	0	0	12	0
JMP ECX	0	0	0	0	0	0	0	53	0	0	0	0	0
JMP EDX	0	0	0	0	0	0	0	14	0	0	0	14	0
JMP EDI	0	0	0	0	0	0	0	0	0	0	0	7	0
JMP ESI	0	0	0	0	6	6	0	6	0	0	0	8	0
JMP EBP	0	0	0	0	2	0	0	1	0	0	0	0	0
CALL EAX	291	0	472	204	71	7	226	1177	0	25	4	320	0
CALL EBX	585	60	213	14	143	7	253	500	0	61	0	204	0
CALL ECX	162	0	301	6	6	8	104	640	0	0	4	41	0
CALL EDX	133	0	236	96	0	0	185	682	0	19	0	23	0
CALL EDI	1104	73	226	14	935	23	381	915	0	69	3	591	0
CALL ESI	1624	387	93	116	4077	228	224	1047	0	103	39	903	0
CALL EBP	241	0	13	0	0	0	79	196	0	28	0	0	0
DG EAX	8	0	20	7	6	1	1	18	0	0	1	34	0
DG EBX	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ECX	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EDX	0	0	0	0	0	0	0	0	0	0	0	0	0

Op.	Ltkrn15u.dll	WINTRUST.dll	Lttwn15u.dll	VideoCommon.dll	RPCRT4.dll	COMCTL32.dll	Ltdis15u.dll	Trackerbird.dll	GDI32.dll	Ltimgclr15u.dll	ADVAPI32.dll	MSVCR100.dll	MSIMG32.dll
DG EDI	0	0	0	0	0	0	0	0	0	0	0	0	0
DG ESI	0	0	0	0	0	0	0	0	0	0	0	0	0
DG EBP	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EAX	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBX	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ECX	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDX	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EDI	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST ESI	0	0	0	0	0	0	0	0	0	0	0	0	0
DG BEST EBP	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EAX	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EBX	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ECX	0	0	0	0	0	0	0	0	0	0	0	0	0

Op.	Ltkrn15u.dll	WINTRUST.dll	Lttwn15u.dll	VideoCommon.dll	RPCRT4.dll	COMCTL32.dll	Ltdis15u.dll	Trackerbird.dll	GDI32.dll	Ltimgclr15u.dll	ADVAPI32.dll	MSVCR100.dll	MSIMG32.dll
DG Other EDX	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EDI	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other ESI	0	0	0	0	0	0	0	0	0	0	0	0	0
DG Other EBP	0	0	0	0	0	0	0	0	0	0	0	0	0
ADD all	1554	126	395	99	1803	61	244	770	0	48	47	209	0
ADD EAX	430	111	223	75	1744	56	90	224	0	20	43	142	0
ADD EBX	56	0	9	3	5	5	27	31	0	0	2	8	0
ADD ECX	35	8	12	1	26	0	10	70	0	0	0	17	0
ADD EDX	135	5	67	15	3	0	93	213	0	10	0	10	0
ADD EDI	64	2	26	3	4	0	6	68	0	2	2	14	0
ADD ESI	206	0	35	0	0	0	15	73	0	11	0	7	0
ADD EBP	8	0	22	0	0	0	2	10	0	2	0	2	0
SUB all	97	5	29	9	19	3	81	183	0	6	1	26	0
SUB EAX	64	1	20	3	14	1	32	96	0	1	0	16	0
SUB EBX	11	0	5	2	4	2	8	85	0	0	1	3	0
SUB ECX	9	0	3	0	1	0	24	15	0	5	0	1	0

Op.	Ltkrn15u.dll	WINTRUST.dll	Lttwn15u.dll	VideoCommon.dll	RPCRT4.dll	COMCTL32.dll	Ltdis15u.dll	Trackerbird.dll	GDI32.dll	Ltimgclr15u.dll	ADVAPI32.dll	MSVCR100.dll	MSIMG32.dll
SUB EDX	14	4	5	0	3	0	22	20	0	0	1	4	0
SUB EDI	7	4	0	0	0	0	2	6	0	0	1	2	0
SUB ESI	9	0	1	4	0	0	1	3	0	0	0	2	0
SUB EBP	0	0	0	0	1	0	0	1	0	0	0	0	0
MUL all	0	0	0	0	1	0	1	2	0	0	0	0	0
MUL EAX	0	0	0	0	0	0	1	0	0	0	0	0	0
MUL EBX	0	0	0	0	0	0	0	0	0	0	0	0	0
MUL ECX	0	0	0	0	0	0	0	0	0	0	0	0	0
MUL EDX	0	0	0	0	0	0	0	0	0	0	0	0	0
MUL EDI	0	0	0	0	0	0	0	0	0	0	0	0	0
MUL ESI	0	0	0	0	0	0	0	0	0	0	0	0	0
MUL EBP	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV all	16	0	0	1	2	8	3	5	0	1	0	1	0
DIV EAX	16	0	0	1	2	8	3	5	0	1	0	1	0
DIV EBX	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ECX	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV EDX	16	0	0	1	2	8	3	5	0	1	0	1	0
DIV EDI	0	0	0	0	0	0	0	0	0	0	0	0	0
DIV ESI	0	0	0	0	0	0	0	0	0	0	0	0	0

Op.	Ltkrn15u.dll	WINTRUST.dll	Lttwn15u.dll	VideoCommon.dll	RPCRT4.dll	COMCTL32.dll	Ltdis15u.dll	Trackerbird.dll	GDI32.dll	Ltimgclr15u.dll	ADVAPI32.dll	MSVCR100.dll	MSIMG32.dll
DIV EBP	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV all	261	362	329	355	3947	88	361	941	0	8	2	523	0
MOV EAX	52	1	164	107	6	0	70	266	0	0	0	46	0
MOV EBX	14	1	3	2	3	0	8	30	0	0	0	39	0
MOV ECX	37	354	41	126	3787	79	91	215	0	2	2	131	0
MOV EDX	52	0	71	101	42	0	70	149	0	0	0	14	0
MOV EDI	29	0	0	4	11	1	39	59	0	0	0	47	0
MOV ESI	37	6	2	4	95	6	70	80	0	0	0	24	0
MOV EBP	25	0	5	0	3	2	7	20	0	0	0	20	0
MOV SHUFFLE all	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EAX	2	0	3	0	0	0	8	4	0	0	0	2	0
MOV SHUFFLE EBX	12	0	0	0	0	0	7	26	0	0	0	19	0
MOV SHUFFLE ECX	9	354	4	21	3609	79	0	5	0	0	2	0	0
MOV SHUFFLE EDX	0	0	4	0	34	0	7	0	0	0	0	0	0

Op.	Ltkrn15u.dll	WINTRUST.dll	Lttwn15u.dll	VideoCommon.dll	RPCRT4.dll	COMCTL32.dll	Ltdis15u.dll	Trackerbird.dll	GDI32.dll	Ltimgclr15u.dll	ADVAPI32.dll	MSVCR100.dll	MSIMG32.dll
MOV SHUFFLE EDI	19	0	0	0	0	0	29	13	0	0	0	17	0
MOV SHUFFLE ESI	15	0	0	4	0	0	30	55	0	0	0	3	0
MOV SHUFFLE EBP	10	0	0	0	2	2	6	18	0	0	0	2	0
MOV VALUE all	15	1	7	21	6	1	12	43	0	0	0	65	0
MOV VALUE EAX	9	1	2	0	0	0	4	24	0	0	0	5	0
MOV VALUE EBX	0	0	3	0	2	0	0	1	0	0	0	12	0
MOV VALUE ECX	0	0	0	19	0	0	0	0	0	0	0	3	0
MOV VALUE EDX	1	0	0	0	2	0	0	3	0	0	0	14	0
MOV VALUE EDI	0	0	0	0	0	1	8	9	0	0	0	6	0
MOV VALUE ESI	3	0	2	0	1	0	0	4	0	0	0	4	0
MOV VALUE EBP	1	0	0	0	1	0	0	2	0	0	0	18	0
LEA all	194	0	447	33	66	0	120	444	0	1	0	312	0
LEA EAX	40	0	162	5	0	0	43	146	0	1	0	243	0

Op.	Ltkrn15u.dll	WINTRUST.dll	Lttwn15u.dll	VideoCommon.dll	RPCRT4.dll	COMCTL32.dll	Ltdis15u.dll	Trackerbird.dll	GDI32.dll	Ltimgclr15u.dll	ADVAPI32.dll	MSVCR100.dll	MSIMG32.dll
LEA EBX	0	0	22	0	0	0	1	0	0	0	0	7	0
LEA ECX	95	0	99	22	62	0	26	172	0	0	0	34	0
LEA EDX	46	0	118	6	4	0	48	119	0	0	0	23	0
LEA EDI	5	0	10	0	0	0	2	2	0	0	0	4	0
LEA ESI	8	0	0	0	0	0	0	0	0	0	0	0	0
LEA EBP	0	0	36	0	0	0	0	5	0	0	0	1	0
PUSH all	5900	23	3478	669	168	19	2809	11864	22	438	13	3271	0
PUSH EAX	1665	0	502	163	17	0	407	1589	0	118	0	755	0
PUSH EBX	406	0	227	38	8	1	235	1451	0	60	1	525	0
PUSH ECX	852	0	310	60	37	0	382	992	0	61	0	124	0
PUSH EDX	919	0	476	39	20	0	432	894	0	31	0	44	0
PUSH EDI	404	0	241	66	20	0	296	1222	0	37	0	248	0
PUSH ESI	571	0	431	73	6	0	414	1704	0	45	0	306	0
PUSH EBP	214	0	323	6	4	0	75	473	0	13	0	33	0
POP all	76	98	24	42	91	153	27	75	308	17	116	128	0
POP EAX	4	4	0	0	7	0	5	9	0	1	0	14	0
POP EBX	2	3	0	0	0	150	3	4	0	0	0	0	0
POP ECX	9	44	9	21	32	0	9	20	154	9	57	45	0
POP EDX	2	46	0	17	32	0	3	1	154	0	57	8	0

Op.	Ltkrn15u.dll	WINTRUST.dll	Lttwn15u.dll	VideoCommon.dll	RPCRT4.dll	COMCTL32.dll	Ltdis15u.dll	Trackerbird.dll	GDI32.dll	Ltimgclr15u.dll	ADVAPI32.dll	MSVCR100.dll	MSIMG32.dll
POP EDI	0	1	0	0	0	0	0	3	0	0	0	11	0
POP ESI	8	0	0	0	0	0	0	8	0	0	0	9	0
POP EBP	1	0	0	3	0	0	1	7	0	0	0	12	0
INC all	180	153	80	21	18	2	53	299	0	34	0	90	0
INC EAX	9	8	2	1	3	0	2	60	0	4	0	19	0
INC EBX	0	0	0	7	0	0	5	20	0	0	0	1	0
INC ECX	0	1	35	0	1	0	0	4	0	0	0	1	0
INC EDX	8	0	2	6	1	0	1	14	0	0	0	6	0
INC EDI	6	139	1	2	5	0	4	19	0	0	0	8	0
INC ESI	22	2	0	0	0	0	4	39	0	0	0	6	0
INC EBP	2	1	4	4	2	2	7	45	0	2	0	44	0
DEC all	197	80	38	18	253	9	55	186	22	31	16	42	0
DEC EAX	14	4	1	2	2	1	17	26	0	1	0	4	0
DEC EBX	4	0	0	1	0	0	6	2	0	0	0	1	0
DEC ECX	1	1	0	4	0	0	0	10	0	0	0	0	0
DEC EDX	2	0	16	7	1	0	0	14	0	0	0	0	0
DEC EDI	0	0	0	0	22	0	0	7	0	0	0	0	0
DEC ESI	7	0	1	1	7	0	5	25	0	0	0	12	0
DEC EBP	2	0	0	3	1	0	6	13	22	0	0	10	0

Op.	Ltkrn15u.dll	WINTRUST.dll	Lttwn15u.dll	VideoCommon.dll	RPCRT4.dll	COMCTL32.dll	Ltdis15u.dll	Trackerbird.dll	GDI32.dll	Ltimgclr15u.dll	ADVAPI32.dll	MSVCR100.dll	MSIMG32.dll
XCHG all	88	0	139	0	10	150	6	28	0	0	0	24	0
XCHG EAX	6	0	16	0	8	0	1	13	0	0	0	4	0
XCHG EBX	2	0	8	0	4	0	0	0	0	0	0	0	0
XCHG ECX	0	0	0	0	0	0	0	7	0	0	0	0	0
XCHG EDX	0	0	0	0	0	0	0	3	0	0	0	4	0
XCHG EDI	0	0	14	0	1	0	0	1	0	0	0	0	0
XCHG ESI	1	0	52	0	0	0	0	0	0	0	0	0	0
XCHG EBP	0	0	0	0	0	0	0	0	0	0	0	0	0
SHIFT LEFT	162	0	8	9	5	3	12	68	0	4	0	61	0
SHIFT RIGHT	7	0	2	0	7	0	4	23	0	0	0	17	0
ROTATE LEFT	37	0	2	0	1	0	1	22	0	7	0	0	0
ROTATE RIGHT	2	0	4	1	3	0	2	16	0	0	0	0	0

Table 17. JOP Gadgets for Filezilla.exe and its associated modules – Part 1

Op.	filezilla.exe all	filezilla.exe	COMCTL32.DLL	MPR.DLL	COMDLG32.DLL	WINMM.DLL	GDI32.dll	ADVAPI32.dll	KERNEL32.DLL	msvcrt.dll
JMP EAX	9049	7560	0	15	154	66	154	57	0	12
JMP EBX	21	2	0	0	0	0	0	0	0	18
JMP ECX	24	24	0	0	0	0	0	0	0	0
JMP EDX	435	412	0	0	0	0	0	0	0	21
JMP EDI	42	22	0	0	0	0	0	0	0	7
JMP ESI	57	9	6	6	0	6	0	0	6	7
JMP EBP	5	1	0	0	0	0	0	0	0	0
CALL EAX	25796	25690	7	0	0	0	0	4	9	45
CALL EBX	1301	1033	7	0	8	1	0	0	3	27
CALL ECX	382	263	8	9	9	6	0	4	16	6
CALL EDX	3321	3183	0	0	0	0	0	0	96	0
CALL EDI	2013	862	23	44	25	4	0	3	5	9
CALL ESI	16146	1147	228	19	1534	86	0	39	60	107
CALL EBP	543	539	0	0	0	0	0	0	0	0
DG EAX	431	412	1	0	3	0	0	1	3	4
DG EBX	0	0	0	0	0	0	0	0	0	0
DG ECX	0	0	0	0	0	0	0	0	0	0
DG EDX	0	0	0	0	0	0	0	0	0	0
DG EDI	0	0	0	0	0	0	0	0	0	0
DG ESI	0	0	0	0	0	0	0	0	0	0
DG EBP	0	0	0	0	0	0	0	0	0	0
DG BEST EAX	0	0	0	0	0	0	0	0	0	0
DG BEST EBX	0	0	0	0	0	0	0	0	0	0
DG BEST ECX	0	0	0	0	0	0	0	0	0	0
DG BEST EDX	0	0	0	0	0	0	0	0	0	0

Op.	filezilla.exe all	filezilla.exe	COMCTL32.DLL	MPR.DLL	COMDLG32.DLL	WINMM.DLL	GDI32.dll	ADVAPI32.dll	KERNEL32.DLL	msvcrt.dll
DG BEST EDI	0	0	0	0	0	0	0	0	0	0
DG BEST ESI	0	0	0	0	0	0	0	0	0	0
DG BEST EBP	0	0	0	0	0	0	0	0	0	0
DG Other EAX	6	6	0	0	0	0	0	0	0	0
DG Other EBX	0	0	0	0	0	0	0	0	0	0
DG Other ECX	0	0	0	0	0	0	0	0	0	0
DG Other EDX	0	0	0	0	0	0	0	0	0	0
DG Other EDI	0	0	0	0	0	0	0	0	0	0
DG Other ESI	0	0	0	0	0	0	0	0	0	0
DG Other EBP	0	0	0	0	0	0	0	0	0	0
ADD all	10621	6053	61	31	220	94	0	47	81	94
ADD EAX	7626	3328	56	28	189	84	0	43	75	75
ADD EBX	1245	1209	5	2	1	6	0	2	2	0
ADD ECX	272	201	0	1	0	1	0	0	0	6
ADD EDX	92	63	0	1	3	0	0	0	1	0
ADD EDI	185	66	0	0	20	3	0	2	2	13
ADD ESI	53	36	0	0	7	0	0	0	1	0
ADD EBP	33	28	0	0	0	0	0	0	0	0
SUB all	1657	1281	3	0	14	0	0	1	3	9
SUB EAX	1038	692	1	0	6	0	0	0	3	1
SUB EBX	292	274	2	0	6	0	0	1	0	0
SUB ECX	61	57	0	0	0	0	0	0	0	1
SUB EDX	45	24	0	0	6	0	0	1	0	7

Op.	filezilla.exe all	filezilla.exe	COMCTL32.DLL	MPR.DLL	COMDLG32.DLL	WINMM.DLL	GDI32.dll	ADVAPI32.dll	KERNEL32.DLL	msvcrt.dll
SUB EDI	17	4	0	0	6	0	0	1	0	0
SUB ESI	2	0	0	0	0	0	0	0	0	2
SUB EBP	1	1	0	0	0	0	0	0	0	0
MUL all	597	5	0	6	0	0	0	0	0	1
MUL EAX	2	2	0	0	0	0	0	0	0	0
MUL EBX	0	0	0	0	0	0	0	0	0	0
MUL ECX	19	0	0	0	0	0	0	0	0	0
MUL EDX	0	0	0	0	0	0	0	0	0	0
MUL EDI	1	0	0	0	0	0	0	0	0	0
MUL ESI	1	0	0	0	0	0	0	0	0	0
MUL EBP	391	0	0	0	0	0	0	0	0	0
DIV all	64	31	8	0	3	0	0	0	0	0
DIV EAX	64	31	8	0	3	0	0	0	0	0
DIV EBX	0	0	0	0	0	0	0	0	0	0
DIV ECX	0	0	0	0	0	0	0	0	0	0
DIV EDX	64	31	8	0	3	0	0	0	0	0
DIV EDI	0	0	0	0	0	0	0	0	0	0
DIV ESI	0	0	0	0	0	0	0	0	0	0
DIV EBP	0	0	0	0	0	0	0	0	0	0
MOV all	26679	14379	88	12	580	7	0	2	126	140
MOV EAX	4902	4801	0	1	5	1	0	0	44	19
MOV EBX	212	193	0	2	6	0	0	0	2	0
MOV ECX	14406	3097	79	6	544	0	0	2	7	66
MOV EDX	395	279	0	0	7	0	0	0	69	2
MOV EDI	89	39	1	0	0	0	0	0	0	0
MOV ESI	676	47	6	1	15	0	0	0	0	13
MOV EBP	63	23	2	2	0	2	0	0	2	29

Op.	filezilla.exe all	filezilla.exe	COMCTL32.DLL	MPR.DLL	COMDLG32.DLL	WINMM.DLL	GDI32.dll	ADVAPI32.dll	KERNEL32.DLL	msvcrt.dll
MOV SHUFFLE all	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EAX	134	134	0	0	0	0	0	0	0	0
MOV SHUFFLE EBX	89	89	0	0	0	0	0	0	0	0
MOV SHUFFLE ECX	13711	2741	79	6	520	0	0	2	7	66
MOV SHUFFLE EDX	10	10	0	0	0	0	0	0	0	0
MOV SHUFFLE EDI	19	14	0	0	0	0	0	0	0	0
MOV SHUFFLE ESI	34	34	0	0	0	0	0	0	0	0
MOV SHUFFLE EBP	21	9	2	2	0	2	0	0	2	2
MOV VALUE all	2933	2658	1	4	32	2	0	0	112	42
MOV VALUE EAX	2567	2485	0	1	5	1	0	0	42	11
MOV VALUE EBX	38	27	0	2	3	0	0	0	0	0
MOV VALUE ECX	35	12	0	0	21	0	0	0	0	0
MOV VALUE EDX	121	30	0	0	0	0	0	0	68	2

Op.	filezilla.exe all	filezilla.exe	COMCTL32.DLL	MPR.DLL	COMDLG32.DLL	WINMM.DLL	GDI32.dll	ADVAPI32.dll	KERNEL32.DLL	msvcrt.dll
MOV VALUE EDI	39	24	1	0	0	0	0	0	0	0
MOV VALUE ESI	13	12	0	1	0	0	0	0	0	0
MOV VALUE EBP	42	14	0	0	0	0	0	0	0	27
LEA all	3357	2817	0	0	74	0	0	0	0	1
LEA EAX	3357	57	0	0	2	0	0	0	0	0
LEA EBX	22	22	0	0	0	0	0	0	0	0
LEA ECX	654	124	0	0	72	0	0	0	0	0
LEA EDX	78	78	0	0	0	0	0	0	0	0
LEA EDI	7	3	0	0	0	0	0	0	0	0
LEA ESI	11	11	0	0	0	0	0	0	0	0
LEA EBP	2	1	0	0	0	0	0	0	0	1
PUSH all	2877	997	19	9	999	20	22	13	5	71
PUSH EAX	293	147	0	0	20	0	0	0	0	13
PUSH EBX	92	56	1	1	0	1	0	1	1	7
PUSH ECX	48	3	0	0	0	5	0	0	0	0
PUSH EDX	993	38	0	0	937	0	0	0	0	0
PUSH EDI	59	7	0	0	2	0	0	0	2	23
PUSH ESI	27	11	0	0	0	0	0	0	0	8
PUSH EBP	167	155	0	0	0	0	0	0	0	1
POP all	20256	17164	153	30	261	121	308	116	2	36
POP EAX	181	156	0	0	5	0	0	0	1	0
POP EBX	4268	4111	150	0	0	0	0	0	0	0
POP ECX	1361	5	0	14	121	60	154	57	0	1
POP EDX	1357	2	0	14	118	57	154	57	0	7
POP EDI	3333	3327	0	0	0	3	0	0	0	0

Op.	filezilla.exe all	filezilla.exe	COMCTL32.DLL	MPR.DLL	COMDLG32.DLL	WINMM.DLL	GDI32.dll	ADVAPI32.dll	KERNEL32.DLL	msvcrt.dll
POP ESI	3602	3588	0	0	0	0	0	0	0	0
POP EBP	5761	5748	0	0	0	0	0	0	0	1
INC all	6655	6531	2	1	7	0	0	0	4	20
INC EAX	1704	1671	0	0	4	0	0	0	0	0
INC EBX	50	44	0	0	0	0	0	0	0	0
INC ECX	25	20	0	0	0	0	0	0	0	0
INC EDX	122	111	0	0	0	0	0	0	0	9
INC EDI	412	400	0	0	0	0	0	0	0	8
INC ESI	29	20	0	0	0	0	0	0	1	0
INC EBP	2549	2517	2	1	1	0	0	0	3	3
DEC all	1475	1035	9	7	21	45	22	16	4	10
DEC EAX	170	154	1	0	4	0	0	0	0	0
DEC EBX	26	6	0	2	2	0	0	0	0	0
DEC ECX	18	15	0	0	0	0	0	0	1	0
DEC EDX	7	2	0	0	1	0	0	0	0	0
DEC EDI	69	3	0	0	2	40	0	0	0	0
DEC ESI	13	5	0	0	0	0	0	0	0	0
DEC EBP	259	139	0	0	3	0	22	0	0	7
XCHG all	316	136	150	0	7	0	0	0	0	0
XCHG EAX	115	95	0	0	2	0	0	0	0	0
XCHG EBX	16	13	0	0	0	0	0	0	0	0
XCHG ECX	18	7	0	0	0	0	0	0	0	0
XCHG EDX	26	25	0	0	0	0	0	0	0	0
XCHG EDI	13	12	0	0	0	0	0	0	0	0
XCHG ESI	11	11	0	0	0	0	0	0	0	0
XCHG EBP	17	16	0	0	0	0	0	0	0	0
SHIFT LEFT	63	37	3	4	0	4	0	0	4	3

Op.	filezilla.exe all	filezilla.exe	COMCTL32.DLL	MPR.DLL	COMDLG32.DLL	WINMM.DLL	GDI32.dll	ADVAPI32.dll	KERNEL32.DLL	msvcrt.dll
SHIFT RIGHT	159	131	0	0	0	0	0	0	2	19
ROTATE LEFT	71	66	0	0	4	0	0	0	0	0
ROTATE RIGHT	16	15	0	0	0	0	0	0	0	0

Table 18. JOP Gadgets for Filezilla.exe and its associated modules – Part 2

Op.	WSOCK32. DLL	NETAPI32. dll	powrprof.dll	SHELL32.dll	Normaliz.dll	ole32.dll	CRYPT32.dll	WS2_32.dll	USER32.dll	OLEAUT32.dll
JMP EAX	0	56	21	753	0	70	63	28	27	13
JMP EBX	0	0	0	1	0	0	0	0	0	0
JMP ECX	0	0	0	0	0	0	0	0	0	0
JMP EDX	0	0	0	1	0	1	0	0	0	0
JMP EDI	0	0	0	13	0	0	0	0	0	0
JMP ESI	0	0	0	9	0	0	0	0	8	0
JMP EBP	0	0	0	4	0	0	0	0	0	0
CALL EAX	0	0	0	17	0	1	0	0	18	5
CALL EBX	0	0	0	183	0	13	3	9	3	11
CALL ECX	0	0	3	19	0	5	8	3	15	8
CALL EDX	0	0	0	2	0	0	12	0	28	0
CALL EDI	0	0	2	520	0	56	9	331	69	51
CALL ESI	0	1	8	9462	0	653	31	326	474	1971
CALL EBP	0	0	0	0	0	0	0	0	0	4
DG EAX	0	0	0	1	0	0	1	0	4	1
DG EBX	0	0	0	0	0	0	0	0	0	0
DG ECX	0	0	0	0	0	0	0	0	0	0
DG EDX	0	0	0	0	0	0	0	0	0	0
DG EDI	0	0	0	0	0	0	0	0	0	0
DG ESI	0	0	0	0	0	0	0	0	0	0
DG EBP	0	0	0	0	0	0	0	0	0	0
DG BEST EAX	0	0	0	0	0	0	0	0	0	0
DG BEST EBX	0	0	0	0	0	0	0	0	0	0
DG BEST ECX	0	0	0	0	0	0	0	0	0	0
DG BEST EDX	0	0	0	0	0	0	0	0	0	0
DG BEST EDI	0	0	0	0	0	0	0	0	0	0
DG BEST ESI	0	0	0	0	0	0	0	0	0	0

Op.	WSOCK32. DLL	NETAPI32. dll	powrprof.dll	SHELL32.dll	Normaliz.dll	ole32.dll	CRYPT32.dll	WS2_32.dll	USER32.dll	OLEAUT32.dll
DG BEST EBP	0	0	0	0	0	0	0	0	0	0
DG Other EAX	0	0	0	0	0	0	0	0	0	0
DG Other EBX	0	0	0	0	0	0	0	0	0	0
DG Other ECX	0	0	0	0	0	0	0	0	0	0
DG Other EDX	0	0	0	0	0	0	0	0	0	0
DG Other EDI	0	0	0	0	0	0	0	0	0	0
DG Other ESI	0	0	0	0	0	0	0	0	0	0
DG Other EBP	0	0	0	0	0	0	0	0	0	0
ADD all	0	2	12	2733	0	265	48	164	410	306
ADD EAX	0	2	12	2643	0	250	43	159	359	280
ADD EBX	0	0	0	9	0	2	4	0	2	1
ADD ECX	0	0	0	53	0	4	0	1	1	4
ADD EDX	0	0	0	17	0	1	0	0	4	2
ADD EDI	0	0	0	11	0	6	0	1	44	17
ADD ESI	0	0	0	4	0	0	0	3	0	2
ADD EBP	0	0	0	0	0	3	1	0	1	0
SUB all	0	0	0	299	0	5	2	1	24	15
SUB EAX	0	0	0	292	0	4	0	1	23	15
SUB EBX	0	0	0	7	0	0	0	1	1	0
SUB ECX	0	0	0	1	0	0	2	0	0	0
SUB EDX	0	0	0	6	0	1	0	0	0	0
SUB EDI	0	0	0	5	0	1	0	0	0	0
SUB ESI	0	0	0	0	0	0	0	0	0	0
SUB EBP	0	0	0	0	0	0	0	0	0	0
MUL all	0	0	0	576	0	0	0	0	9	0
MUL EAX	0	0	0	0	0	0	0	0	0	0
MUL EBX	0	0	0	0	0	0	0	0	0	0
MUL ECX	0	0	0	19	0	0	0	0	0	0

Op.	WSOCK32. DLL	NETAPI32. dll	powrprof.dll	SHELL32.dll	Normaliz.dll	ole32.dll	CRYPT32.dll	WS2_32.dll	USER32.dll	OLEAUT32.dll
MUL EDX	0	0	0	0	0	0	0	0	0	0
MUL EDI	0	0	0	1	0	0	0	0	0	0
MUL ESI	0	0	0	1	0	0	0	0	0	0
MUL EBP	0	0	0	385	0	0	0	0	6	0
DIV all	0	0	0	20	0	0	0	0	0	2
DIV EAX	0	0	0	20	0	0	0	0	0	2
DIV EBX	0	0	0	0	0	0	0	0	0	0
DIV ECX	0	0	0	0	0	0	0	0	0	0
DIV EDX	0	0	0	20	0	0	0	0	0	2
DIV EDI	0	0	0	0	0	0	0	0	0	0
DIV ESI	0	0	0	0	0	0	0	0	0	0
DIV EBP	0	0	0	0	0	0	0	0	0	0
MOV all	0	0	0	8619	0	460	9	413	184	1660
MOV EAX	0	0	0	6	0	2	1	0	20	2
MOV EBX	0	0	0	9	0	0	0	0	0	0
MOV ECX	0	0	0	8236	0	423	6	361	118	1461
MOV EDX	0	0	0	4	0	9	0	0	24	1
MOV EDI	0	0	0	8	0	2	0	23	16	0
MOV ESI	0	0	0	342	0	23	0	29	4	196
MOV EBP	0	0	0	0	0	1	0	0	2	0
MOV SHUFFLE all	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EAX	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EBX	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE ECX	0	0	0	8073	0	423	6	289	116	1383

Op.	WSOCK32. DLL	NETAPI32. dll	powrprof.dll	SHELL32.dll	Normaliz.dll	ole32.dll	CRYPT32.dll	WS2_32.dll	USER32.dll	OLEAUT32.dll
MOV SHUFFLE EDX	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EDI	0	0	0	3	0	0	0	0	2	0
MOV SHUFFLE ESI	0	0	0	0	0	0	0	0	0	0
MOV SHUFFLE EBP	0	0	0	0	0	0	0	0	2	0
MOV VALUE all	0	0	0	24	0	1	2	0	54	1
MOV VALUE EAX	0	0	0	3	0	0	0	0	19	0
MOV VALUE EBX	0	0	0	6	0	0	0	0	0	0
MOV VALUE ECX	0	0	0	1	0	0	0	0	0	1
MOV VALUE EDX	0	0	0	0	0	0	0	0	21	0
MOV VALUE EDI	0	0	0	0	0	0	0	0	14	0
MOV VALUE ESI	0	0	0	0	0	0	0	0	0	0
MOV VALUE EBP	0	0	0	0	0	1	0	0	0	0
LEA all	0	0	0	423	0	36	0	6	0	0
LEA EAX	0	0	0	1	0	2	0	0	0	0
LEA EBX	0	0	0	0	0	0	0	0	0	0
LEA ECX	0	0	0	418	0	34	0	6	0	0
LEA EDX	0	0	0	0	0	0	0	0	0	0
LEA EDI	0	0	0	4	0	0	0	0	0	0
LEA ESI	0	0	0	0	0	0	0	0	0	0
LEA EBP	0	0	0	0	0	0	0	0	0	0
PUSH all	0	8	3	388	0	199	15	6	22	81

Op.	WSOCK32. DLL	NETAPI32. dll	powrprof.dll	SHELL32.dll	Normaliz.dll	ole32.dll	CRYPT32.dll	WS2_32.dll	USER32.dll	OLEAUT32.dll
PUSH EAX	0	0	0	54	0	22	0	0	0	37
PUSH EBX	0	0	0	22	0	0	0	0	1	1
PUSH ECX	0	0	0	14	0	0	0	0	0	26
PUSH EDX	0	0	0	14	0	0	0	0	0	4
PUSH EDI	0	0	0	15	0	0	0	0	2	8
PUSH ESI	0	0	0	7	0	0	0	1	0	0
PUSH EBP	0	0	0	11	0	0	0	0	0	0
POP all	0	114	48	1556	0	77	137	48	50	35
POP EAX	0	0	0	9	0	10	0	0	0	0
POP EBX	0	0	0	3	0	0	0	0	0	4
POP ECX	0	56	21	723	0	30	60	23	24	12
POP EDX	0	56	21	721	0	31	60	23	24	12
POP EDI	0	0	0	1	0	1	0	1	0	0
POP ESI	0	0	6	8	0	0	0	0	0	0
POP EBP	0	0	0	9	0	0	2	0	0	1
INC all	0	1	1	66	0	4	5	2	7	4
INC EAX	0	0	0	27	0	2	0	0	0	0
INC EBX	0	0	1	5	0	0	0	0	0	0
INC ECX	0	0	0	3	0	0	0	0	2	0
INC EDX	0	0	0	0	0	0	0	0	2	0
INC EDI	0	0	0	3	0	1	0	0	0	0
INC ESI	0	0	0	8	0	0	0	0	0	0
INC EBP	0	0	0	7	0	1	5	2	3	4
DEC all	0	8	3	159	0	2	8	81	5	40
DEC EAX	0	0	0	10	0	1	0	0	0	0
DEC EBX	0	0	0	16	0	0	0	0	0	0
DEC ECX	0	0	0	2	0	0	0	0	0	0
DEC EDX	0	0	0	4	0	0	0	0	0	0

Op.	WSOCK32. DLL	NETAPI32. dll	powrprof.dll	SHELL32.dll	Normaliz.dll	ole32.dll	CRYPT32.dll	WS2_32.dll	USER32.dll	OLEAUT32.dll
DEC EDI	0	0	0	18	0	0	0	5	0	1
DEC ESI	0	0	0	7	0	1	0	0	0	0
DEC EBP	0	0	2	40	0	0	0	42	4	0
XCHG all	0	0	0	8	0	4	0	0	11	0
XCHG EAX	0	0	0	3	0	4	0	0	11	0
XCHG EBX	0	0	0	2	0	0	0	0	1	0
XCHG ECX	0	0	0	1	0	0	0	0	10	0
XCHG EDX	0	0	0	1	0	0	0	0	0	0
XCHG EDI	0	0	0	1	0	0	0	0	0	0
XCHG ESI	0	0	0	0	0	0	0	0	0	0
XCHG EBP	0	0	0	1	0	0	0	0	0	0
SHIFT LEFT	0	0	0	3	0	0	0	0	4	1
SHIFT RIGHT	0	0	0	6	0	0	0	1	0	0
ROTATE LEFT	0	0	0	0	0	0	0	0	0	1
ROTATE RIGHT	0	0	0	1	0	0	0	0	0	0

APPENDIX B: SAMPLE OUTPUT OF GADGETS

Appendix B provides sample output of gadgets for different operations. The purpose of Appendix B is simply to provide an illustration of how the output may appear. The examples chosen are brief or truncated, whereas in actual practice the quantity of gadgets produced could number in the hundreds or thousands. Figure 1 illustrates MOV Val output from WinRAR.exe, where EDI is the target for the MOV operation. Figure 2 depicts JMP EDX output from Respond.exe.

Figures 3 and 4 showcase dispatcher gadget output. Dispatcher gadgets are necessities for JOP to function, and both depict excellent gadgets, although EAX is the least desirable register for dispatcher gadgets. This is because EAX is the most commonly used register. Figure 3 illustrates truncated Dispatcher Gadget EAX output from Snagit.exe, while figure 4 depicts Dispatcher Gadget Other EAX output from Filezilla.exe.

Figure 26. MOV Val EDI output from WinRAR.exe .

```
@MOV Val*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#1      Ops: 13     Mod: WinRAR.exe
mov edi, 0x8b000001                 0x4abd87 (offset 0xa9d87)
sbb eax, 0x516608                   0x4abd8c (offset 0xa9d8c)
push 0x65                     0x4abd91 (offset 0xa9d91)
push ebp                      0x4abd93 (offset 0xa9d93)
call ebx                      0x4abd94 (offset 0xa9d94)

@MOV Val*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#2      Ops: 13     Mod: WinRAR.exe
mov edi, 1                    0x40c981 (offset 0xa981)
```

push 0x69 0x40c986 (offset 0xa986)

push ebx 0x40c988 (offset 0xa988)

call esi 0x40c989 (offset 0xa989)

@MOV Val*^

#3 Ops: 12 Mod: WinRAR.exe

mov edi, 1 0x40c981 (offset 0xa981)

push 0x69 0x40c986 (offset 0xa986)

push ebx 0x40c988 (offset 0xa988)

call esi 0x40c989 (offset 0xa989)

@MOV Val*^

#4 Ops: 10 Mod: WinRAR.exe

mov edi, 1 0x40c981 (offset 0xa981)

push 0x69 0x40c986 (offset 0xa986)

push ebx 0x40c988 (offset 0xa988)

call esi 0x40c989 (offset 0xa989)

@MOV Val*^

#5 Ops: 8 Mod: WinRAR.exe

mov edi, 1 0x40c981 (offset 0xa981)

push 0x69 0x40c986 (offset 0xa986)

push ebx 0x40c988 (offset 0xa988)

call esi 0x40c989 (offset 0xa989)

@MOV Val*^

#6 Ops: 10 Mod: COMCTL32.dll

mov edi, 0xff5750c3 0x5bfb7d1b (offset 0x35d1b)

adc eax, 0x5bff8654 0x5bfb7d20 (offset 0x35d20)

call esi 0x5bfb7d25 (offset 0x35d25)

@MOV Val*^

#7 Ops: 13 Mod: USER32.dll

mov edi, 0x448b69e9 0x69e8038d (offset 0x7e38d)

and al, 0x20 0x69e80392 (offset 0x7e392)

call eax 0x69e80394 (offset 0x7e394)

@MOV Val*^

#8 Ops: 11 Mod: USER32.dll

mov edi, 0x448b69e9 0x69e8038d (offset 0x7e38d)

and al, 0x20 0x69e80392 (offset 0x7e392)

call eax 0x69e80394 (offset 0x7e394)

@MOV Val*^

#9 Ops: 8 Mod: USER32.dll

mov edi, 0x448b69e9 0x69e8038c (offset 0x7e38c)

and al, 0x20 0x69e80392 (offset 0x7e392)

call eax 0x69e80394 (offset 0x7e394)

@MOV Val*^

#10 Ops: 7 Mod: USER32.dll

mov edi, 0x448b69e9 0x69e8038d (offset 0x7e38d)

and al, 0x20 0x69e80392 (offset 0x7e392)

call eax 0x69e80394 (offset 0x7e394)

@MOV Val*^

#11 Ops: 12 Mod: USER32.dll

mov edi, 0x4d8b69e9 0x69e87512 (offset 0x85512)

or byte ptr [ebx + 0x98b0451], cl 0x69e87518 (offset 0x85518)

call esi 0x69e8751e (offset 0x8551e)

@MOV Val*^

#12 Ops: 11 Mod: USER32.dll

mov edi, 0xcf8b69e9 0x69e590f0 (offset 0x570f0)

call esi 0x69e590f5 (offset 0x570f5)

@MOV Val*^

#13 Ops: 11 Mod: USER32.dll

mov edi, 0xcf8b69e9 0x69e59160 (offset 0x57160)

call esi 0x69e59165 (offset 0x57165)

@MOV Val*^

#14 Ops: 11 Mod: USER32.dll

mov edi, 0xcf8b69e9 0x69e591d0 (offset 0x571d0)

call esi 0x69e591d5 (offset 0x571d5)

@MOV Val*^

#15 Ops: 11 Mod: USER32.dll

mov edi, 0xcf8b69e9 0x69e59315 (offset 0x57315)

call esi 0x69e5931a (offset 0x5731a)

@MOV Val*^

#16 Ops: 11 Mod: USER32.dll

mov edi, 0xcf8b69e9 0x69e595a1 (offset 0x575a1)

call esi 0x69e595a6 (offset 0x575a6)

@MOV Val*^

#17 Ops: 11 Mod: USER32.dll

mov edi, 0xcf8b69e9 0x69e59621 (offset 0x57621)

call esi 0x69e59626 (offset 0x57626)

@MOV Val*^

#18 Ops: 11 Mod: USER32.dll

mov edi, 0xcf8b69e9 0x69e596a1 (offset 0x576a1)

call esi 0x69e596a6 (offset 0x576a6)

@MOV Val*^

#19 Ops: 11 Mod: USER32.dll

mov edi, 0xcf8b69e9 0x69e5977b (offset 0x5777b)

call esi 0x69e59780 (offset 0x57780)

@MOV Val*^

#20 Ops: 11 Mod: USER32.dll

mov edi, 0x4d8b69e9 0x69e87513 (offset 0x85513)

or byte ptr [ebx + 0x98b0451], cl 0x69e87518 (offset 0x85518)

call esi 0x69e8751e (offset 0x8551e)

MOV Value EDI WinRAR.exe total: 5

MOV Value EDI COMCTL32.dll total: 1

MOV Value EDI USER32.dll total: 14

Grand total MOV Value EDI : 20

Figure 27. JMP EDX output from Respond.exe

*^

#1 Ops: 13 Mod: Respond.exe

push ecx 0x5c165a (offset 0x1bf65a)

or byte ptr [ecx - 0x1276b], cl 0x5c165b (offset 0x1bf65b)

inc dword ptr [ecx - 0x12743] 0x5c1661 (offset 0x1bf661)

jmp edx 0x5c1667 (offset 0x1bf667)

*^

#2 Ops: 13 Mod: Respond.exe

inc edx 0x5c214d (offset 0x1c014d)

or byte ptr [ecx - 0x1237b], cl 0x5c214e (offset 0x1c014e)

inc dword ptr [ecx - 0x12343] 0x5c2154 (offset 0x1c0154)

jmp edx 0x5c215a (offset 0x1c015a)

*^

#3 Ops: 13 Mod: Respond.exe

push ecx 0x5c2a6a (offset 0x1c0a6a)

or byte ptr [ecx - 0x11f6b], cl 0x5c2a6b (offset 0x1c0a6b)

inc dword ptr [ecx - 0x11f43] 0x5c2a71 (offset 0x1c0a71)

jmp edx 0x5c2a77 (offset 0x1c0a77)

*^

#4 Ops: 13 Mod: Respond.exe

mov dword ptr [esp + 4], ecx 0xc923ed (offset 0x8903ed)

jmp edx 0xc923f1 (offset 0x8903f1)

*^

#5 Ops: 12 Mod: Respond.exe

or byte ptr [ecx - 0x1276b], cl 0x5c165b (offset 0x1bf65b)

inc dword ptr [ecx - 0x12743] 0x5c1661 (offset 0x1bf661)

jmp edx 0x5c1667 (offset 0x1bf667)

*^

#6 Ops: 12 Mod: Respond.exe

or byte ptr [ecx - 0x1237b], cl 0x5c214e (offset 0x1c014e)

inc dword ptr [ecx - 0x12343] 0x5c2154 (offset 0x1c0154)

jmp edx 0x5c215a (offset 0x1c015a)

*^

#7 Ops: 12 Mod: Respond.exe

or byte ptr [ecx - 0x11f6b], cl 0x5c2a6b (offset 0x1c0a6b)

inc dword ptr [ecx - 0x11f43] 0x5c2a71 (offset 0x1c0a71)

jmp edx 0x5c2a77 (offset 0x1c0a77)

*^

#8 Ops: 12 Mod: Respond.exe

ror byte ptr [edi], cl 0x6c9241 (offset 0x2c7241)

test dword ptr [ebp - 0x73000000], ecx 0x6c9243 (offset 0x2c7243)

dec ebp 0x6c9249 (offset 0x2c7249)

shr al, 1 0x6c924a (offset 0x2c724a)

push eax 0x6c924c (offset 0x2c724c)

jmp edx 0x6c924d (offset 0x2c724d)

*^

#9 Ops: 12 Mod: Respond.exe

add dword ptr [ebx - 0x2d7ad7b0], ecx 0xc923e5 (offset 0x8903e5)

je 0x8903f3 0xc923eb (offset 0x8903eb)

mov dword ptr [esp + 4], ecx 0xc923ed (offset 0x8903ed)

jmp edx 0xc923f1 (offset 0x8903f1)

*^

#10 Ops: 12 Mod: Respond.exe

rcl dword ptr [edx - 0x17aa0000], 0x53 0xd6ac91 (offset 0x968c91)

arpl word ptr [eax], ax 0xd6ac98 (offset 0x968c98)

add byte ptr [esi - 0x18], dl 0xd6ac9a (offset 0x968c9a)

jmp edx 0xd6ac9d (offset 0x968c9d)

*^

#11 Ops: 11 Mod: Respond.exe

mov dword ptr [esp + 4], ecx 0xc923ed (offset 0x8903ed)

jmp edx 0xc923f1 (offset 0x8903f1)

*^

#12 Ops: 10 Mod: Respond.exe

test dword ptr [ebp - 0x73000000], ecx 0x6c9243 (offset 0x2c7243)

dec ebp 0x6c9249 (offset 0x2c7249)

shr al, 1 0x6c924a (offset 0x2c724a)

push eax 0x6c924c (offset 0x2c724c)

jmp edx 0x6c924d (offset 0x2c724d)

*^

#13 Ops: 10 Mod: Respond.exe

push eax 0xc923e7 (offset 0x8903e7)

sub byte ptr [ebp - 0x76f98b2e], al 0xc923e8 (offset 0x8903e8)

dec esp 0xc923ee (offset 0x8903ee)

and al, 4 0xc923ef (offset 0x8903ef)

jmp edx 0xc923f1 (offset 0x8903f1)

*^

#14 Ops: 9 Mod: Respond.exe

sub byte ptr [ebp - 0x76f98b2e], al 0xc923e8 (offset 0x8903e8)

dec esp 0xc923ee (offset 0x8903ee)

and al, 4 0xc923ef (offset 0x8903ef)

jmp edx 0xc923f1 (offset 0x8903f1)

*^

#15 Ops: 9 Mod: Respond.exe

add byte ptr [esi - 0x18], dl 0xd6ac94 (offset 0x968c94)

push ebx 0xd6ac97 (offset 0x968c97)

arpl word ptr [eax], ax 0xd6ac98 (offset 0x968c98)

add byte ptr [esi - 0x18], dl 0xd6ac9a (offset 0x968c9a)

jmp edx 0xd6ac9d (offset 0x968c9d)

*^

#16 Ops: 8 Mod: Respond.exe

add byte ptr [eax], al 0x6c91d3 (offset 0x2c71d3)

add byte ptr [ebp - 0x3d1733b3], cl 0x6c91d5 (offset 0x2c71d5)

jmp edx 0x6c91db (offset 0x2c71db)

*^

#17 Ops: 8 Mod: Respond.exe

add byte ptr [eax], al 0x6c9232 (offset 0x2c7232)

add byte ptr [ebp + 0x63e8cc4d], cl 0x6c9234 (offset 0x2c7234)

jmp edx 0x6c923a (offset 0x2c723a)

*^

#18 Ops: 8 Mod: Respond.exe

add byte ptr [eax], al 0x6c9245 (offset 0x2c7245)

add byte ptr [ebp + 0x50e8d04d], cl 0x6c9247 (offset 0x2c7247)

jmp edx 0x6c924d (offset 0x2c724d)

*^

#19 Ops: 8 Mod: Respond.exe

test edx, edx 0xc923e9 (offset 0x8903e9)

je 0x8903f3 0xc923eb (offset 0x8903eb)

mov dword ptr [esp + 4], ecx 0xc923ed (offset 0x8903ed)

jmp edx 0xc923f1 (offset 0x8903f1)

*^

#20 Ops: 7 Mod: Respond.exe

sal byte ptr [esi + eax - 0x77], cl 0xc923ea (offset 0x8903ea)

dec esp 0xc923ee (offset 0x8903ee)

and al, 4 0xc923ef (offset 0x8903ef)

jmp edx 0xc923f1 (offset 0x8903f1)

JMP EDX total: 20

*^

#1 Ops: 12 Mod: UxTheme.dll

jmp eax 0x71b3839e (offset 0x3639e)

add al, byte ptr [eax] 0x71b383a0 (offset 0x363a0)

add byte ptr [edi], cl 0x71b383a2 (offset 0x363a2)

test byte ptr [ebp - 0x7efffffd], bl 0x71b383a4 (offset 0x363a4)

jmp edx 0x71b383aa (offset 0x363aa)

*^

#2 Ops: 10 Mod: UxTheme.dll

add al, byte ptr [eax] 0x71b383a0 (offset 0x363a0)

add byte ptr [edi], cl 0x71b383a2 (offset 0x363a2)

test byte ptr [ebp - 0x7efffffd], bl 0x71b383a4 (offset 0x363a4)

jmp edx 0x71b383aa (offset 0x363aa)

*^

#3 Ops: 8 Mod: UxTheme.dll

jmp dword ptr [esi - 0x2d001f3e] 0x71b0888b (offset 0x688b)

loope 0x6883 0x71b08891 (offset 0x6891)

jmp edx 0x71b08893 (offset 0x6893)

*^

#4 Ops: 8 Mod: UxTheme.dll

add byte ptr [edi], cl 0x71b383a2 (offset 0x363a2)

test byte ptr [ebp - 0x7efffffd], bl 0x71b383a4 (offset 0x363a4)

jmp edx 0x71b383aa (offset 0x363aa)

JMP EDX total: 4

*^

#1 Ops: 13 Mod: WININET.dll

cmp ebp, dword ptr [edx] 0x631ad3ac (offset 0x1ab3ac)

add byte ptr [eax], al 0x631ad3ae (offset 0x1ab3ae)

lcall 0x2a:0x3c3be2ab 0x631ad3b0 (offset 0x1ab3b0)

add byte ptr [ecx], bl 0x631ad3b7 (offset 0x1ab3b7)

jmp edx 0x631ad3b9 (offset 0x1ab3b9)

*^

#2 Ops: 12 Mod: WININET.dll

sub al, byte ptr [eax] 0x631ad3ad (offset 0x1ab3ad)

add byte ptr [edx + 0x3c3be2ab], bl 0x631ad3af (offset 0x1ab3af)

sub al, byte ptr [eax] 0x631ad3b5 (offset 0x1ab3b5)

add byte ptr [ecx], bl 0x631ad3b7 (offset 0x1ab3b7)

jmp edx 0x631ad3b9 (offset 0x1ab3b9)

*^

#3 Ops: 11 Mod: WININET.dll

add byte ptr [eax], al 0x631ad3ae (offset 0x1ab3ae)

lcall 0x2a:0x3c3be2ab 0x631ad3b0 (offset 0x1ab3b0)

add byte ptr [ecx], bl 0x631ad3b7 (offset 0x1ab3b7)

jmp edx 0x631ad3b9 (offset 0x1ab3b9)

*^

#4 Ops: 10 Mod: WININET.dll

add byte ptr [edx + 0x3c3be2ab], bl 0x631ad3af (offset 0x1ab3af)

sub al, byte ptr [eax] 0x631ad3b5 (offset 0x1ab3b5)

add byte ptr [ecx], bl 0x631ad3b7 (offset 0x1ab3b7)

jmp edx 0x631ad3b9 (offset 0x1ab3b9)

*^

#5 Ops: 9 Mod: WININET.dll

lcall 0x2a:0x3c3be2ab 0x631ad3b0 (offset 0x1ab3b0)

add byte ptr [ecx], bl 0x631ad3b7 (offset 0x1ab3b7)

jmp edx 0x631ad3b9 (offset 0x1ab3b9)

*^

#6 Ops: 7 Mod: WININET.dll

loope 0x1ee886 0x631f08d4 (offset 0x1ee8d4)

add byte ptr [eax], al 0x631f08d6 (offset 0x1ee8d6)

jmp dword ptr [eax] 0x631f08d8 (offset 0x1ee8d8)

push edi 0x631f08da (offset 0x1ee8da)

jmp edx 0x631f08db (offset 0x1ee8db)

JMP EDX total: 6

*^

#1 Ops: 8 Mod: SHELL32.dll

in eax, 0xe2 0x69a69dfd (offset 0x267dfd)

dec dword ptr [ebp - 0x1a17ebb2] 0x69a69dff (offset 0x267dff)

jmp edx 0x69a69e05 (offset 0x267e05)

JMP EDX total: 1

*^

#1 Ops: 8 Mod: ole32.dll

add byte ptr [edi], cl 0x68f2699f (offset 0x2499f)

test dword ptr [edx - 0x7efffec2], edi 0x68f269a1 (offset 0x249a1)

jmp edx 0x68f269a7 (offset 0x249a7)

JMP EDX total: 1

Grand total JMP EDX total: 32

Figure 28. Truncated Dispatcher Gadget EAX output from Snagit.exe

```
*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#1	Ops: 9	Mod: Snagit32.exe
or cl, byte ptr [ebx - 0x743774f0]		0x6fff8b (offset 0x2fdf8b)
inc edx		0x6fff91 (offset 0x2fdf91)
sub al, 0x5e		0x6fff92 (offset 0x2fdf92)
jmp eax		0x6fff94 (offset 0x2fdf94)

*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#2	Ops: 8	Mod: Snagit32.exe
add al, ch		0x7fac19 (offset 0x3f8c19)
and bh, byte ptr [edx + 0x595affdc]		0x7fac1b (offset 0x3f8c1b)
jmp eax		0x7fac21 (offset 0x3f8c21)

*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#3	Ops: 8	Mod: Snagit32.exe
add al, ch		0x88c3d8 (offset 0x48a3d8)
arpl word ptr [edx + 0x595affd3], sp		0x88c3da (offset 0x48a3da)
jmp eax		0x88c3e0 (offset 0x48a3e0)

*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#4	Ops: 8	Mod: Snagit32.exe
add al, ch		0x88c4af (offset 0x48a4af)
mov word ptr [ecx + 0x595affd3], fs		0x88c4b1 (offset 0x48a4b1)
jmp eax		0x88c4b7 (offset 0x48a4b7)

*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
#5	Ops: 13		Mod: Snagit32.exe
sal byte ptr [ebx + ebx - 0x75], 0x4b		0x404ea0 (offset 0x2ea0)
add al, 0x8b		0x404ea5 (offset 0x2ea5)
adc dword ptr [ebx + 0x6a0442], ecx		0x404ea7 (offset 0x2ea7)
```

call eax 0x404ead (offset 0x2ead)

*^

#6 Ops: 13 Mod: Snagit32.exe

add al, byte ptr [ebx - 0x3074fb3c] 0x409ce9 (offset 0x7ce9)

mov dword ptr [ebp - 0x1c], 0 0x409cef (offset 0x7cef)

call eax 0x409cf6 (offset 0x7cf6)

*^

#7 Ops: 13 Mod: Snagit32.exe

test ecx, ecx 0x44177a (offset 0x3f77a)

add dword ptr [eax], eax 0x44177c (offset 0x3f77c)

add byte ptr [eax + 0x428b178b], dl 0x44177e (offset 0x3f77e)

add al, 0x8b 0x441784 (offset 0x3f784)

iretd 0x441786 (offset 0x3f786)

call eax 0x441787 (offset 0x3f787)

*^

#8 Ops: 13 Mod: Snagit32.exe

adc byte ptr [ebx - 0x3474c9], cl 0x46292b (offset 0x6092b)

rcl byte ptr [eax - 0x75], cl 0x462931 (offset 0x60931)

inc esi 0x462934 (offset 0x60934)

add al, 0x8b 0x462935 (offset 0x60935)

iretd 0x462937 (offset 0x60937)

call eax 0x462938 (offset 0x60938)

*^

#9 Ops: 13 Mod: Snagit32.exe

dec dword ptr [ebx + 0x5411e8d8] 0x4fccc7 (offset 0xfacc7)

sti 0x4fcccd (offset 0xfaccd)

call dword ptr [eax - 0x75] 0x4fccce (offset 0xfacce)

inc edi 0x4fccd1 (offset 0xfacd1)

sbb al, 0x53 0x4fccd2 (offset 0xfacd2)

call eax 0x4fccd4 (offset 0xfacd4)

[truncated]

Dispatcher Gadgets for EAX Snagit32.exe total: 264

Dispatcher Gadgets for EAX imagehlp.dll total: 3

Dispatcher Gadgets for EAX CRYPT32.dll total: 1

Dispatcher Gadgets for EAX opencv_core249.dll total: 22

Dispatcher Gadgets for EAX PDFLib.dll total: 54

Dispatcher Gadgets for EAX KERNEL32.DLL total: 3

Dispatcher Gadgets for EAX ScrollingCapture.dll total: 22

Dispatcher Gadgets for EAX mfc100u.dll total: 5

Dispatcher Gadgets for EAX Ltkrn15u.dll total: 8

Dispatcher Gadgets for EAX Lttwn15u.dll total: 20

Dispatcher Gadgets for EAX VideoCommon.dll total: 7

Dispatcher Gadgets for EAX RPCRT4.dll total: 6

Dispatcher Gadgets for EAX COMCTL32.dll total: 1

Dispatcher Gadgets for EAX Ltdis15u.dll total: 1

Dispatcher Gadgets for EAX Trackerbird.dll total: 18

Dispatcher Gadgets for EAX ADVAPI32.dll total: 1

Dispatcher Gadgets for EAX MSVCR100.dll total: 34

Grand total Dispatcher Gadgets for EAX: 470

Figure 29. Dispatcher Gadget Other EAX output from Filezilla.exe

*^

#1 Ops: 13 Mod: filezilla.exe

pop es 0x7a02d7 (offset 0x39e2d7)

shl eax, 2 0x7a02d8 (offset 0x39e2d8)

lea eax, dword ptr [eax + eax*2 + 0x7a02f2] 0x7a02db (offset 0x39e2db)

shr edx, 1 0x7a02e2 (offset 0x39e2e2)

jmp eax 0x7a02e4 (offset 0x39e2e4)

*^

#2 Ops: 13 Mod: filezilla.exe

pop es 0x7a0387 (offset 0x39e387)

shl eax, 2 0x7a0388 (offset 0x39e388)

lea eax, dword ptr [eax + eax*2 + 0x7a03a2] 0x7a038b (offset 0x39e38b)

shr edx, 1 0x7a0392 (offset 0x39e392)

jmp eax 0x7a0394 (offset 0x39e394)

*^

#3 Ops: 12 Mod: filezilla.exe

shl eax, 2 0x7a02d8 (offset 0x39e2d8)

lea eax, dword ptr [eax + eax*2 + 0x7a02f2] 0x7a02db (offset 0x39e2db)

shr edx, 1 0x7a02e2 (offset 0x39e2e2)

jmp eax 0x7a02e4 (offset 0x39e2e4)

*^

#4 Ops: 12 Mod: filezilla.exe

shl eax, 2 0x7a0388 (offset 0x39e388)

lea eax, dword ptr [eax + eax*2 + 0x7a03a2] 0x7a038b (offset 0x39e38b)

shr edx, 1 0x7a0392 (offset 0x39e392)

jmp eax 0x7a0394 (offset 0x39e394)

*^

#5 Ops: 12 Mod: filezilla.exe

shr al, 1 0xd8d116 (offset 0x98b116)

and eax, 1 0xd8d118 (offset 0x98b118)

ret 0xd8d11b (offset 0x98b11b)

lea esi, dword ptr [esi] 0xd8d11c (offset 0x98b11c)

jmp eax 0xd8d120 (offset 0x98b120)

APPENDIX C: DEFINITIONS

Code-Reuse Attack: A code-reuse attack makes use of existing instructions in the process virtual memory in an unintended fashion. An attacker can utilize ROP or JOP gadgets to achieve arbitrary computation, to disable protections, or to set up shellcode to be executed

JOP: JOP is short for Jump-Oriented Programming. It is an advanced code-reuse attack that makes use of chunks of instructions existing in the virtual memory of the process that end with an indirect jump or call. JOP requires a dispatch table and both dispatcher and functional gadgets.

ROP: ROP is short for Return Oriented Programming. It is a code-reuse attack that makes use of chunks of instructions existing in the virtual memory of a process that terminates with a RET instruction or return. The return functions as the "glue" that binds the instructions together, allowing for control flow to be ordered.

Gadget: A gadget is a carefully selected chunk of instructions that terminates in a control flow instruction, such as a ret or an indirect jump or call. These are "strung" together to allow for an attacker to execute arbitrary computation.

Dispatcher Gadget: The dispatcher gadget is a carefully selected gadget that modifies the dispatch table in a predictable manner. This allows an attacker to order the control flow in a JOP exploit. Functional gadgets will typically go to the dispatcher gadget, which then will move the instruction pointer to the next functional gadget within the dispatch table.

Dispatch Table: The dispatch table provides the virtual memory addresses for all functional gadgets used in JOP.

Disassembly: Disassembly is the end result of machine language being translated into Assembly language.

Assembly: Assembly is a low-level programming language that is specific to the architecture it is to be run on.

Opcodes: Opcodes are representations of machine code that exist in hexadecimal format.

Opcode-splitting: Opcode-splitting is the process of starting execution in the middle of a line of machine code instructions on architectures that lack alignment. This can result in machine code being executed in a manner unintended by the compiler, resulting in unintended Assembly instructions being executed.