

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

TREND MICRO, INC.
Petitioner,

v.

WEBROOT, INC.
Patent Owner.

IPR2023-00692

U.S. Patent No. 10,284,591

DECLARATION OF SETH JAMES NIELSON

TABLE OF CONTENTS

I.	INTRODUCTION	1
II.	QUALIFICATIONS	1
III.	MATERIALS CONSIDERED	12
IV.	LEGAL STANDARDS	13
V.	Technical Background that Would Have Been Known to A POSITA	16
A.	Software Execution and Call Stacks	16
B.	System Calls and APIs.....	32
C.	Processes and Memory Address Spaces	34
D.	Call-backs, Filter Drivers, and Code Hooking.....	36

E.	Buffer-Overflow-Based Software Vulnerabilities	39
VI.	OVERVIEW OF THE CHALLENGED PATENT	45
A.	THE '591 PATENT SPECIFICATION AND CLAIMS.....	45
B.	PROSECUTION HISTORY.....	57
VII.	LEVEL OF SKILL IN THE ART	58
VIII.	Overview of the Prior art	58
A.	OVERVIEW OF US20130275981A1 PUBLISHED APPLICATION TO DALCHER (“DALCHER”).....	58
B.	OVERVIEW OF TRANSPARENT ROP EXPLOIT MIGITGATION USING INDIRECT BRANCH TRACING (“PAPPAS”)	69
IX.	GROUND 1: Claims 1-19 are rendered obvious by DALCHER IN VIEW OF PAPPAS.....	72
A.	Motivation to Combine Dalcher in view of Pappas.....	72
B.	Claim 1 is rendered obvious by Dalcher in view of Pappas	76
1.	Limitation 1[a]	76
2.	Limitation 1[b]	79
3.	Limitation 1[c]	86
4.	Limitation 1[d]	92
5.	Limitation 1[e]	102
6.	Limitation 1[f]	110
7.	Limitation 1[g]	115
C.	Claim 2 is rendered obvious by Dalcher in view of Pappas	118
1.	Limitation 2[a]	118
2.	Limitation 2[b]	121
D.	Claim 3 is rendered obvious by Dalcher in view of Pappas	126
E.	Claim 4 is rendered obvious by Dalcher in view of Pappas	129

F.	Claim 5 is rendered obvious by Dalcher in view of Pappas	131
G.	Claim 6 is rendered obvious by Dalcher in view of Pappas	136
H.	Claim 7 is rendered obvious by Dalcher in view of Pappas	139
I.	Claim 8 is rendered obvious by Dalcher in view of Pappas	144
J.	Claims 9-16 are rendered obvious by Dalcher in view of Pappas	147
K.	Claim 17 is rendered obvious by Dalcher in view of Pappas	155
	1. Limitation 17[a]	158
	2. Limitation 17[b]-17[g](i) and 17[h]	159
	2. Limitation 17[g](ii)	160
L.	Claim 18 renders obvious by Dalcher in view of Pappas	169
	1. Limitation 18[a]	169
	2. Limitation 18[b]	172
X.	There are no objective indicia of nonobviousness.....	174
XI.	Conclusion	174

1. I, Seth James Nielson, submit the following declaration (the “Declaration”) in connection with the proceeding identified above.

I. INTRODUCTION

2. I have been retained by counsel for Trend Micro, Inc. (“Trend Micro” or “Petitioner”) as a technical expert in connection with the proceeding identified above. I submit this Declaration in support of Petitioner Trend Micro’s Petition for Inter Partes Review (the “Petition”) of United States Patent No. 10,284,591 (“the ’591 patent”) against Webroot, Inc. (“Webroot” or “Patent Owner”). All “Ex. 10XX” cites herein are to the Exhibits to the Petitions. All “Ex. _” cites followed by a letter are to exhibits to this Declaration. All citations to “Section XX” are internal citations to the sections of this Declaration. All citations to “Paragraph XX” are internal citations to the paragraphs of this Declaration.

3. I have been asked to consider whether the claims of the ’591 patent are obvious in light of the prior art discussed below. The ’591 patent is directed to the identification and prevention of malware.

4. In summary, my conclusion is that the claims of the ’591 patent are obvious in light of the prior art discussed below. As set forth in detail in this declaration, I have reached the following conclusions:

- a. Claims 1-19 are rendered obvious by Dalcher in view of Pappas.

II. QUALIFICATIONS

5. I am a subject matter expert in cybersecurity, computer networks, and software engineering. I am the Founder and Chief Scientist of Crimson Vista, a computer security research

and engineering company. Furthermore, I hold appointments at the University of Texas at Austin as an Adjunct Associate Professor in the department of Computer Science and as a Cybersecurity Fellow in the Robert Strauss Center for International Security and Law.

6. I have been working with computer networks and associated technology, including parallel processing, since January 2000. My experience includes graduate-level teaching, academic research, industry employment, and consulting practice.

7. A copy of my current C.V. is attached hereto as **Exhibit 1003**. My C.V. includes all of my relevant education and work experience, a list of all publications that I have authored in the previous ten years, as well as a list of all cases in which, during the previous four years, I have testified as an expert at trial or by deposition. My experience most relevant to the opinions offered in this report is summarized below.

8. I received my B.S. in Computer Science in April of 2000. During my final undergraduate semester, I worked both as a teaching assistant for a Computer Networking course and as a researcher in the Networked Computing Lab. In these capacities, I assisted students in debugging and designing TCP/IP protocol stacks, Address Resolution Protocol implementations, and Remote Procedure Call projects. In the research lab, I collaborated on work related to resource reservations in high performance networking and MPLS analyses. Our investigations into statistical traffic engineering for bandwidth allocation resulted in a published paper entitled, “Effective Bandwidth for Traffic Engineering.”

9. During the Fall of 2000, I studied for one semester at Brandeis University in the Boston area and, as part of my research, I developed a parallel-processing system for DNA string matching. The system that I created had a central node that managed the search space and solution aggregation. A cluster of available “worker” machines registered with the central node to indicate

their availability for parallel processing. The central node would identify subdivisions of the search space that could be farmed out to the worker nodes for processing. I also studied artificial intelligence and machine learning.

10. From 2001 through 2003, I worked as a software engineer at Metrowerks (formerly Lineo, Inc.), where I had substantial responsibilities relating to software architecture, computer networking, and technical project management. In particular, I developed and maintained the GUI for the Embedix SDK (Software Development Kit), created a network communication system between a Windows front-end and Linux back-end, developed an automated system to forward Linux python scripts to a Windows GUI over the network, and developed a network-deployed packaging and automated updating system for client software. To complete these assignments, I wrote tens of thousands of lines of computer code in C++, C, Python, and Perl.

11. As part of my technical training and development at Lineo/Metrowerks, I also gained and employed an increased exposure to computer networking and network security. I tested and evaluated a prototype firewall product, built a custom VPN solution, and trained in the use of web server and mail server administration (including security). I ran my own personal MTA with an IMAP and POP3 server based on these tools under development. I also deployed a Squid web proxy for both caching and content filtering.

12. Another major networking project involved porting the Embedix SDK from Linux to Windows. For this project I used a virtual-machine/networked solution. The underlying SDK engine remained running in a Linux VM while the GUI operated in native Windows. I developed all of the code for this remote communication system.

13. While working at Lineo/Metrowerks, I also returned to BYU to pursue my Master's degree in Computer Science. In addition to the graduate level course work in wireless computer

networks and compilers, I pursued graduate research in software engineering topics, with a special emphasis on how programmers think while creating and modifying code. During my course work, I took a special topics class called “Programmer Cognition” as well as a graduate-level neuroscience class from the Psychology department.

14. My research included a study of computer architectural patterns and how those patterns might need to change as programming languages change and evolve. Based on my research, I proposed a concept called “Design Dysphasia,” wherein a programmer or software developer becomes trapped in their approach to solving problems based on the paradigms and design approaches of the programming language. My research was published as “Design dysphasia and the pattern maintenance cycle,” in the Journal Information and Software Technology August 2006. This work also was a major component of my Master’s thesis.

15. Another part of my Master’s thesis was the identification of how certain programming language concepts can be “mixed” together. I investigated practical mechanisms whereby the Python programming language could be extended to support features known as “functional programming” and “logic (or declarative) programming.” Languages with this mix of features are known as “multi-paradigm” programming languages.

16. After finishing my Master’s degree, I moved to Houston TX in 2004 to begin a PhD program at Rice University. I chose to focus my research into topics of network security including the security of peer-to-peer cooperative storage and computation technologies.

17. During the 2004 fall semester of my Ph.D. program at Rice University, I identified a security vulnerability in the Google Desktop Search that could have allowed hackers to compromise users’ computers and obtain private information. After contacting Google and assisting them in closing the vulnerability, we published the details of our investigation.

18. In 2005, I completed an internship at Google, where I designed and implemented a solution to privacy loss in Google Web Accelerator. The Google Web Accelerator was designed to increase the speed of browsing the Internet. Once installed on a user's computer, the browser would request all content through a Google Proxy. The proxy performed pre-fetching and extensive caching in order to provide fast and responsive service to the user. At the time of my internship, news reports had identified odd problems in which users of the Accelerator were accessing other individual's private pages. During my internship, I designed and implemented a prototype solution for this issue in C++.

19. In 2005, I published a paper entitled, "A Taxonomy of Rational Attacks." This paper categorized and described the various types of attacks that one might see in a decentralized, peer-to-peer network. When there is no centralized authority, users have to cooperate to obtain service. The term "rational attacks" refers to the economic incentives to not cooperate while still exploiting the system for service.

20. My Ph.D. Thesis was entitled "Designing Incentives for Peer-to-Peer Systems," and it built on this concept. Given a network where participants cannot be forced to cooperate, the operation of said network must induce cooperation by design of the outcomes. In other words, it must be in each participant's best interest to contribute to the cooperative operation. I conducted experiments including simulated extensions to the BitTorrent peer-to-peer protocol for long-term identities and mechanisms for cooperative anonymity. These systems were predecessors to modern cryptocurrency and distributed ledger technologies.

21. From 2005 through 2008, with the approval of my PhD adviser, I worked as a Security Analyst for Independent Security Evaluators (ISE). Much of my early work was spent developing a software encryption library, including the necessary tests and procedures for FIPS-

certification. The encryption library provided advanced operations such as secure data splitting and recovery.

22. In 2009, I went to work full time for ISE as a Security Analyst and later as a Senior Security Analyst. I built a number of advanced projects including a parallel-program for massive code coverage analysis, GPU hardware-accelerated AES encryption, and distributed file-system prototypes.

23. In addition to the software development, I also performed security evaluation services that included port-scanning analyses, security protocol analysis using formal and exploratory methods, and investigated security breaches.

24. I also designed and managed the implementation of a secure communication technology that splits trust between multiple SSL Certificate Authorities (CA), so that if one CA is compromised, the communication stream can still be safely authenticated. My work on the secure communications technology project led to the issuance of multiple patents. In total, I wrote hundreds of thousands of lines of code in C, C++, and Python, including projects where I had to implement the same functionality in two separate languages.

25. In 2011, I began work as a Research Scientist at Harbor Labs and continued with that consulting firm until fall 2015. I worked with a wide range of clients, specializing in network security, network communications, software architecture, and programming languages. I analyzed an extensive collection of commercial software, including software related to secure email, cloud-based multimedia delivery, document signing, anti-virus and anti-intrusion, high-performance routing, networking protocol stacks in mobile devices, PBX telecommunications software, VoIP, and peer-to-peer communications. I also analyzed security considerations for potential technology acquisitions.

26. Also at Harbor Labs, I reviewed technology and source code for multiple clients related to accusations of theft and/or misappropriation of trade secrets. These engagements included an analysis of C, C++, Java, Python, and other source code languages in high-frequency trading, e-commerce, and other similar systems.

27. I also assessed the security and privacy technologies and policies provided by a third-party vendor to the Center for Copyright Infringement (CCI). CCI represents content owners, such as the RIAA and the MPAA, in finding and reducing piracy online. Because this process necessarily involves collecting information about private individuals by scanning a network for illegal activity, I was asked to investigate and determine that the information collected from online computing devices was adequately safeguarded and protected.

28. For other clients, I have “resurrected” or re-created legacy software systems. For example, I helped one client make remote desktop-sharing/viewing systems from the mid 90’s operational. I helped them identify the most compatible components from an old CVS repository, obtain the necessary legacy hardware and software to rebuild the source code, and diagnose why the separate components weren’t completely compatible with each other. Using tools from the era (i.e., the mid-90’s), I identified and fixed these issues in C++ and Java code, and successfully demonstrated the operational system across a small cluster of networked virtual machines.

29. During my final year at Harbor Labs, I was engaged as the principal consultant with a large biomedical device firm in a twelve-month analysis of the security of their products. Notably, medical devices were for some time not considered significant threats in terms of computer security. However, recent demonstrations by security researchers of the various ways in which a malicious individual might harm a person using a medical device has shifted the thinking in the industry. Accordingly, I was engaged to assist this company in the analysis of their products,

their process, and their future roadmap in order to ensure that patients are not harmed. I and my team analyzed design documents, hardware, and a broad range of additional resources in order to expose potential problems. The security of these systems depends, in part, on the architecture and deployment of the networks in which they operate.

30. In December 2015, I left Harbor Labs to assist Ironwood Experts, LLC., as the transitional managing partner. In three months, I helped to establish a new direction, streamline operations, and wrap up difficult negotiations.

31. After handing off management responsibilities at Ironwood, I founded Crimson Vista, Inc. as a boutique computer security engineering company. Similar to the work that I did at Harbor Labs, I continue to provide technical expertise to a wide range of clients in areas of programming languages, computer networks, and network security. My expertise in the area of “security engineering” provides comprehensive analysis, design, and insight into cybersecurity concerns before, during, and after development.

32. For example, I have been retained by a start-up in telecommunications security to provide cryptography expertise and evaluations of their protocols and architectures. My team and I have prototyped new protocols, written up analyses, and presented to potential partners and investors.

33. Another start-up company retained me for guidance in matters relating to Blockchain and Smart Contracts. This technology is very much dealing with a “fad” phase where there is a lot of misinformation and hype. I guided the start-up company through analyzing where these kinds of technologies would help and where they would not. I have also provided training on Blockchain at the Data Architecture Summit and Enterprise Data World conferences.

34. I have also provided technical guidance to an antitrust team in the United States Department of Justice. Although the technologies and parties are confidential, I can disclose that I provided in-person training on technical topics and analyses of competing security products.

35. More recently, I been retained by clients, including a Fortune 100 financial institution, to provide them with post-data-breach analyses of what went wrong, the impact of the lost data, and guidance on resolution. In these engagements, I provided reverse engineering of the data to demonstrate how an attacker can or would use the compromised information, analyzed software development to determine when the system became vulnerable, and helped identify impacted customers that had been missed in the investigations.

36. I have also been retained as a vCISO (virtual Chief Information Security Officer) for a small company that needs security expertise and guidance in protecting their newly developed intellectual property. I advise the company on security policy, operations, implementations, and training.

37. Through Crimson Vista, I also invest in research and development. Recent projects include engaging with a partner to implement prototypes of communications security protocols for next-generation automobiles. I also gave a talk on “Detecting Malicious Sandboxes” at the Workshop on Defensive Deception and Trust in Autonomy, in association with the 2018 Naval Applications of Machine Learning Workshop.

38. I was the primary investigator on a government funded research project. The project dealt with automatic recovery from ransomware attacks and was funded by the United States Army. I led a team of student researchers from Johns Hopkins and Brigham Young University in developing my design and implementing a prototype.

39. I am also a contributor to another small-business research project begun by a former student and funded by the National Science Foundation. We are researching the development of a new system for secure data storage that is protected even against “insider” threats, such as a rogue IT administrator. As part of this work, I am managing several interns for Summer 2022 and directing their work on performance, analysis, and vulnerability detection.

40. I also continue to perform a wide range of code reviews for diverse technologies including CAD software, video game systems, digital mobile radios (DMRs), video streaming, and digital rights management (DRM). I am often retained for my expertise in software forensics and have been instrumental in multiple cases to discovering and identifying stolen source code or misappropriated trade secrets.

41. Moreover, I maintain ties to academia. I held adjunct appointments at Johns Hopkins University from 2014 to 2019. From July 2016 to July 2019, I also held an appointment as the Director of Advanced Research Projects in the Johns Hopkins University Information Security Institute.

42. At Johns Hopkins University I taught Network security and Advanced Network Security. I created a custom curriculum and lab experience wherein students developed their own security protocols as a class and then attempted to break their own creations. Students learned how hard it is to get security right, and how easy it is to find something wrong. I published a paper about the labwork in the Journal of Computer Science Education entitled, “PLAYGROUND: Preparing Students for the Cyber Battleground.”

43. Beyond course instruction, I also mentored Masters students at Johns Hopkins in their capstone projects. These projects included networking security and privacy concerns across a wide range of technologies including cryptography, drone security, iOS security, BitCoin, SSL

vulnerabilities, and Twitter “botnets.” My students and I have published multiple papers from these capstone projects.

44. During my tenure as the Director of Advanced Research Projects, I was tasked with developing collaborative research opportunities. Through my efforts, a wide range of student capstones have been executed with partners from the Johns Hopkins Applied Physics Lab or outside corporate partners.

45. For example, we coordinated with the company OnBoard Security to develop better security for anti-collision protocols for air traffic. Students demonstrated the potential issues related to leaving the protocol unsecured and built a working prototype of a secured variant. OnBoard and Johns Hopkins published press releases which were picked up by aviation-focused news sources.

46. I am now an adjunct professor at the University of Texas at Austin. I have taught the undergraduate Network Security and Privacy class in the Computer Science department. I also teach the Introduction to Cybersecurity Technology class in the Law School

47. I am also the co-founder and current director of the Crypto Done Right project. This project is currently hosted by Johns Hopkins University and funded by a grant from Cisco. I am transitioning the project to be hosted by my company Crimson Vista or spun out into a completely independent non-profit entity. Crypto Done Right is designed to bridge the gap between cryptography SME’s and the IT professionals that use it. It provides authoritative guidance on deployment, lifecycle, and management of cryptography in IT systems, software development, and technical management.

48. Finally, I am the author of “Cryptography in Python: Learning Correct Cryptography by Example.” I am authoring a second book on computer security that will be published later this year.

III. MATERIALS CONSIDERED

49. I have considered information from various sources in forming my opinions. Besides drawing from approximately two decades of experience in the computer industry, I also have reviewed the '591 patent and its file history, the other documents and references as cited herein, and general technical references. The exhibits are:

- Ex. 1001: U.S. Patent No. 10,284,591 to Giuliani et al. ('591 patent)
- Ex. 1002: Expert Declaration of Dr. Seth James Nielson
- Ex. 1003: Curriculum Vitae of A.L. Dr. Seth Nielson
- Ex. 1004: U.S. Patent Publication No. 2013/0275981 to Dalcher. (“Dalcher”)
- Ex. 1005: Vasilis Pappas, Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, 22nd USENIX Security Symposium, (August 14-16, 2013). (“Pappas”)
- Ex. 1006: Prosecution History of the '591 patent.
- Ex. 1007: Provisional patent application claimed by '591 patent for priority.
- Ex. 1008: David A. Patterson, Computer Organization & Design The Hardware/Software Interface, Morgan Kaufmann Publishers, Inc., (1994).
- Ex. 1009: ORS Open Systems Resources, Inc., Collecting Detailed Performance Data with Xperf, The NT Insider 17(1), (2010).
- Ex. 1010: Aleph One, Smashing the Stack for Fun and Profit, (1996).

- Ex. 1011: Jacob R. Lorch, The VTrace Tool: Building a System Tracer for Windows NT and Windows 2000, MSDN Magazine, (2000)
- Ex. 1012: Marco Pistoia, Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model, IEE, (2007).
- Ex. 1013: Galen Hunt, Detours: Binary Interception of Win32 Functions, Microsoft Research, (1999).
- Ex. 1014: Sandra Loosemore, The GNU C Library Reference Manual, 188-193, (1999).
- Ex. 1015: Jonathan Corbet, Expanding the Kernel Stack, LWN.net, (2014).
- Ex. 1016: OpenSSL documentation API listing, <https://www.openssl.org/docs/man1.0.2/man3/>.

IV. LEGAL STANDARDS

50. I am not an attorney, and I have relied on instructions from counsel as to the applicable legal standards to use in arriving at my opinions in this Declaration.

51. I have been informed that in a parallel district court litigation the owner of the '591 patent is arguing that the '591 patent claims should be afforded their plain and ordinary meaning. For example, I have been informed that patent owner has argued in the parallel litigation that the plain and ordinary meaning of the term "stack walk processing" is processing that involves using a stack walk. I have also been requested to adopt patent owner's position on claim construction for the purposes of rendering my opinions set forth in this declaration.

52. I understand that a patent may include independent claims and dependent claims. An independent claim stands by itself and only includes the limitations it recites. A dependent claim can depend on (a) an independent claim, or (2) another dependent claim. I understand that a

dependent claims include all of the limitations it recites plus the limitations recited in the claim from which it depends.

53. I have been informed and understand that the subject matter of a patent claim is obvious if the differences between the subject matter of the claim and the prior art are such that the subject matter of the claim as a whole would have been obvious at the effective filing date to a person having ordinary skill in the art to which the subject matter pertains. I have also been informed that the framework for determining obviousness involves considering the following factors: (i) the scope and content of the prior art; (ii) the differences between the prior art and the claimed subject matter; (iii) the level of ordinary skill in the art; and (iv) any objective evidence of non-obviousness.

54. I further understand that the claimed subject matter would have been obvious to one of ordinary skill in the art if, for example, it results from the combination of known elements according to known methods to yield predictable results, the simple substitution of one known element for another to obtain predictable results, the use of a known technique to improve similar devices in the same way, the application of a known technique to a known device ready for improvement to yield predictable results, or choosing from a finite number of identified, predictable solutions with a reasonable expectation of success. I have also been informed that the analysis of obviousness may include recourse to logic, judgment, and common sense available to the person of ordinary skill in the art that does not necessarily require explication in any reference. I understand that so-called “secondary considerations of non-obviousness,” must be considered in an obviousness analysis. I understand that an analysis cannot use hindsight in determining whether a patent claim is obvious. I understand that secondary considerations of non-obviousness include: (a) a long-felt but unresolved need for the invention; (b) commercial success of the invention; (c)

copying of the invention; (d) praise and recognition of the invention by others; (e) licensing of the rights to the invention; and (f) unexpected results.

55. I understand that obviousness must be determined with respect to the challenged claim as a whole.

56. I understand that a challenged claim can be invalid for obviousness over a combination of prior art references if a reason existed (at the effective filing date of the Patent in question) that would have prompted a person of ordinary skill in the art to combine prior art elements in the way required by the claim in question. I understand that this requirement is sometimes called “motivation to combine.” I further understand that the prior art references themselves might provide a motivation to combine, but that sometimes common sense as would have been known to a POSITA can provide a reason to link two or more prior art references. I also understand that market demand, rather than scientific literature, often drives innovation, so that market forces can give rise to a motivation to combine. I also understand that known work in a field of endeavor might prompt variations of it to use in the same field or a different one based on design incentives or market forces if the variations would have been predictable to a POSITA. I also understand that a need or problem known in the field of endeavor can provide a reason for combining elements in the manner claimed.

57. I understand that in order for a claim to be found invalid based upon a modification or combination of prior art references, a POSITA must have had a reasonable expectation of success in modifying or combining the prior art.

58. I further understand that claims are invalid as obvious if they merely recite combinations of, or obvious variations of, prior art teachings employing the understanding and knowledge of a POSITA and motivated by the general problem that the inventor faced at the time.

The prior art references themselves, or any need or problem known in the field of endeavor at the time of the invention can provide a motivation to combine the elements from prior art references or to attempt obvious variations on them in the claimed manner.

59. I understand that something is “inherent in” the prior art—and thus taught by it—it if necessarily follows from the prior art’s explicit disclosures.

60. I understand that the filing date of a U.S. patent is generally the filing date of the U.S. application that issued as a patent. I also understand, though, that a U.S. patent may be granted an earlier effective filing date if the patent owner properly claimed priority to an earlier patent application.

V. TECHNICAL BACKGROUND THAT WOULD HAVE BEEN KNOWN TO A POSITA

61. In this section of my Declaration, I discuss some of the background and context for the alleged invention in the ’591 patent that would have been known to a POSITA. To be clear, these background sections are the state of the art as of the earliest claimed priority date for the ’591 patent.

A. Software Execution and Call Stacks

62. Computer software is essentially a collection of instructions to a computer processor and optionally additional data values. Running a program typically involves “loading” a program (or components thereof) from some kind of long-term storage, like a hard disk, into the computer’s memory. A recognized rendezvous point in the collection of instructions serves as a starting point and the processor begins executing the start-up instructions. (*See* Ex. 1008 at p. 16). A processor executes the instructions one after another until it hits a decision point. Decision points come in various forms such as an “if” statement. These statements evaluate some true-or-false condition and, depending on the result, execute one set of instructions or another. (*See id.* at

pp. 6-9). This allows the processor to “jump around” in the set of instructions, executing different sets of instructions based on user inputs and other data fed into these decisions points. For my students, I have sometimes compared this process to a “choose-your-own-adventure” book.

63. An important feature of computer programs is that instructions and data are all represented with binary numbers stored on disk or in memory. This is sometimes called the “stored-program concept.” “Treating instructions in the same way as data greatly simplifies the memory hardware and the software of computer systems.” (See *id.* at p. 4). Memory in a computer system is basically just a big long list of storage locations and the index of this list (i.e., index 0 for the first element, index 1 for the second element, etc.) is the “address” of the memory location. (*Id.* at p. 3). Because data and instructions are handled the same within the memory, an address in memory could hold an instruction or data. In fact, because they are just numbers, only *context* indicates whether it is one or the other (or sometimes both!).

64. Computer program instructions are typically loaded into the processor in sequential order. That is, an initial instruction at some memory address is loaded. After the computer executes the instruction, the next instruction at the next address is loaded. As explained above, however, a decision point instruction can check some condition and, based on the condition, execute one set of instructions or another. This is typically performed by “jumping” to some other address of an instruction. For example, an instruction to a computer processor might say something like, “if these two values are equal jump to address XXX, otherwise just go on to the next instruction.” In some processors, this type of statement is called a “branch equal” or “beq” instruction. (*Id.* at p. 7). The fact that every instruction has an address means that any instruction in the set can be the “next” instruction if there is an appropriate jump to that address.

65. Computer program instructions can also jump around to different instructions without a branch. One reason to have such an operation is to support “subroutines.” “A procedure or subroutine is one way that programmers structure programs both to make them easier to understand and to allow code to be reused.” (*Id.* at p. 10). The idea of a subroutine is to capture some subset of instructions that perform some kind of identifiable task within the code. For example, suppose that a statistics program needs to calculate the average of a sequence of data at many points in a program. If there were not subroutines the program would have to include the instructions for averaging (e.g., summing all of the numbers, dividing the total by the count of the numbers) at each point in the program that needs an average. Alternatively, the programmer can create an “average” subroutine that acts like a mini-program within the program. Any time the program wants to compute an average, it loads the appropriate data into memory and then jumps to the address of the subroutine. The subroutine executes instructions for summing the data that was pre-loaded before the jump, computes a result, and terminates. The trick, of course, is that the program needs to **jump back** to where it came from. “An instruction set must provide a way to jump to a procedure and then return from the procedure to the instruction [afterward]...” (*Id.* at p. 10).

66. In order to be able to jump **to** a subroutine and then jump **back** when finished, computers have to keep track of the address of instructions that have been called. Typically, the address of the current instruction is held in a memory area called the “Program Counter” (PC). Normally, executing an instruction just increases the PC by one¹. But when jumping to a procedure, the PC is loaded with the new address and the old address is erased. In order to jump

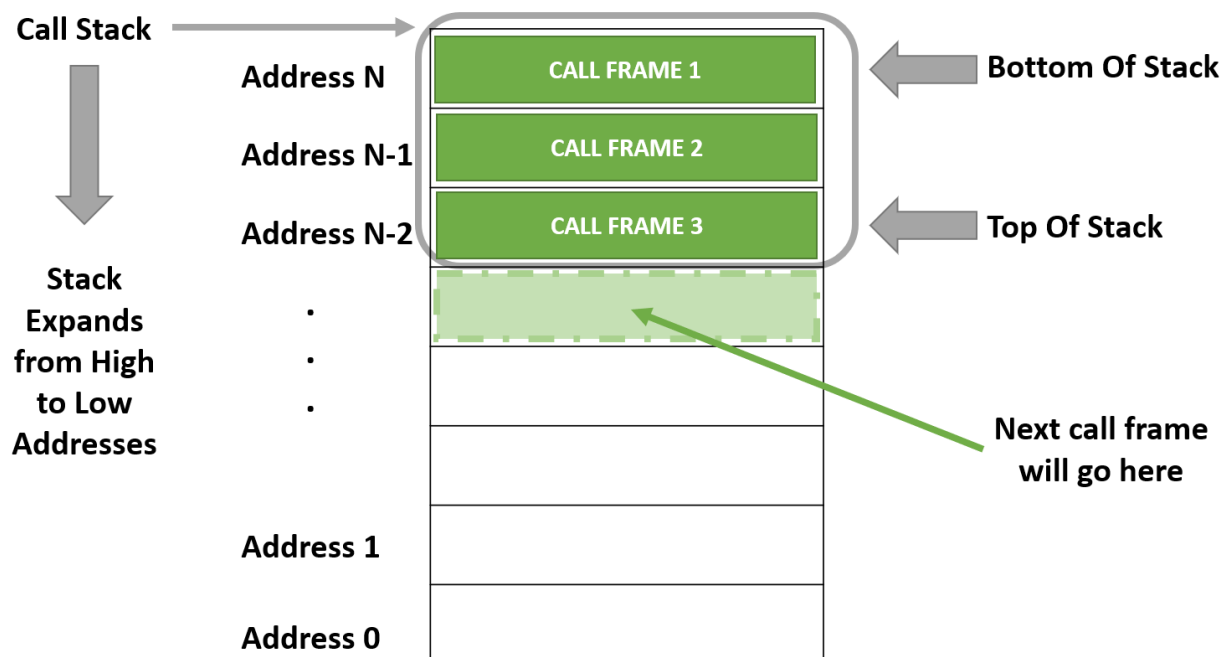
¹ It may actually increase by numbers like 4 or 8 because the size of the instruction might be 4 bytes or 8 bytes. But it usually increases by one logical instruction.

backward to where the jump forward happened, the old address must be copied out of the PC and into memory somewhere. (*Id.* at p. 10).

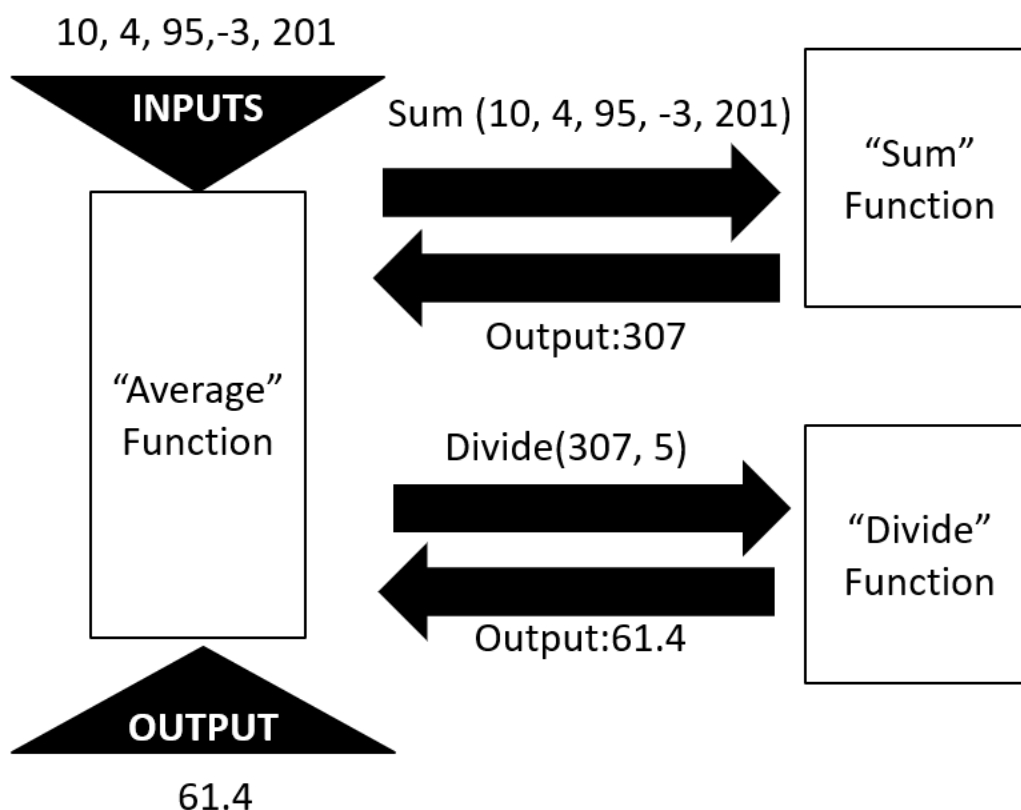
67. However, a subroutine may wish to use **another subroutine** which requires jumping to the next subroutine. And, obviously, it would need to **jump back from that**. The terms used in the art are to “call” a subroutine (also known as a procedure or function) and “return” from the call. But if subroutine A calls subroutine B, the address of the next instruction in A is lost unless it is stored. And if subroutine B calls subroutine C, the address of the next instruction in B is lost. The addresses must be stored in such a way that subroutine C will return to subroutine B, and subroutine B will return to subroutine A. To solve this problem, computers use a **stack** (also called a “call stack”) to store the data in memory. A stack is a concept of “last-in-first-out” memory storage (*Id.* at pp. 10-11). Basically, the idea is to mark some place in memory as the “top” of the stack. Data is added to the top of the stack and the top marker is increased accordingly. When it is time to remove data, data is removed from the top of the stack and the top marker is reduced by the appropriate amount. It is common to refer to putting data on a stack as a “push” operation and removing data from a stack as a “pop” operation. (*Id.*)

68. However, in a confusing twist, many systems have the stack grow “downward” in terms of memory address. This means that the bottom (or origin) of a stack is at some high address in memory but as data is added to the “top” of the stack, these new allocations are at a **lower** address in memory. Dalcher (prior art), for example, refers to “top” and “bottom” in this way. “For example, the call stack may grow **from top to bottom** as functions are called, and may shrink from bottom to top as functions return.” (Ex. 1004 at [0037]). A POSITA would have understood that Dalcher is here referring to “top” as the highest memory address and “bottom” as the lowest memory address. Because stacks operate only by adding and removing stack frames from the

conceptual “top” of the stack, it would have been obvious to one skilled in the art that Dalcher is referring to high versus low memory addresses, rather than the conceptual top and bottom of the stack. Therefore, even though Dalcher’s terminology appears to be opposite that of the general terminology in the art when describing the “top” and “bottom” of a stack, a POSITA would understand Dalcher’s description to be compatible with the common understanding. In this declaration, I will disambiguate these usages by referring to **“top” of memory** (i.e., the highest or largest memory address) and **“bottom” of the stack** (i.e., the oldest stack frame, the origin of the stack), and I will treat these two as synonymous. I will further use **“bottom” of memory** (i.e., the lowest memory address) and **“top” of the stack** (i.e., the most recently added stack frame) as synonymous. To illustrate these concepts I have created the figure below. Dalcher’s “top” of memory is illustrated with the highest memory address being at the top of the diagram, and the conceptual “bottom” of the stack is illustrated in the same place:



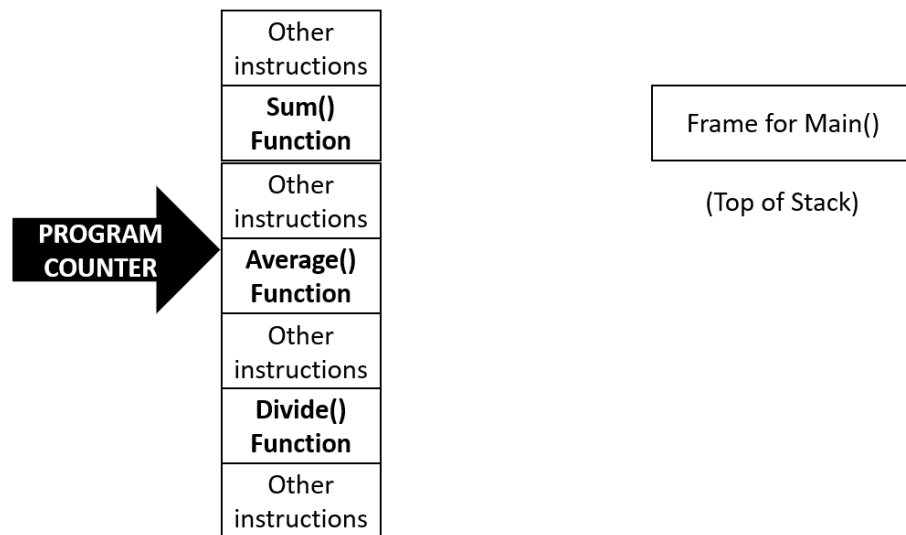
69. Using a stack, it is possible to keep track of function calls². Any time a call is made, the current address is “pushed” onto the stack. Any time a return happens, the software pops the most recent address off the stack and returns to that address. To illustrate this concept, imagine a computer function called “Average()” that takes a set of numbers as an input and returns the arithmetic mean (average) as output. The Average() function uses two sub-functions: “Sum()”, which adds a sequence of numbers together, and “Divide()”, which divides one number by another. The Average() function uses the Sum() function to add the numbers together, then uses the Divide() function to divide the sum by the total number of terms added together. I have visualized this in the image below:



² I will use “function” from here on out because that is the term I use most in my professional work. This term “function” is synonymous in this context with “procedure” or “subroutine” as used in my cited reference.

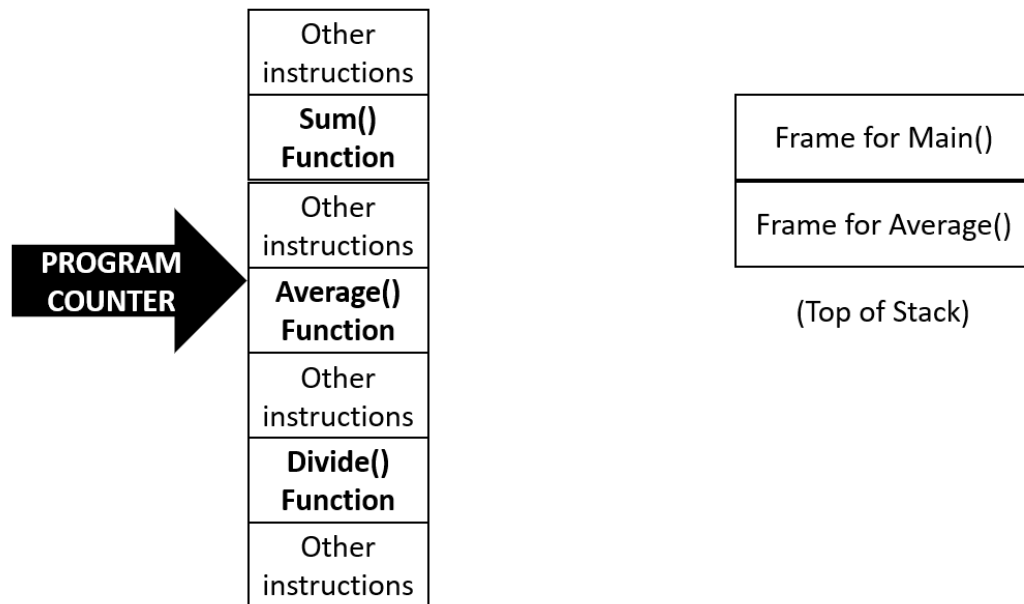
70. The call stack keeps track of these function calls by creating a stack frame for each function call. The starting function for a program created with the “C” language is typically “main()”. In the image below, the call stack has a call frame for main() when the average() function starts:

Average() Function Start (called from Main())



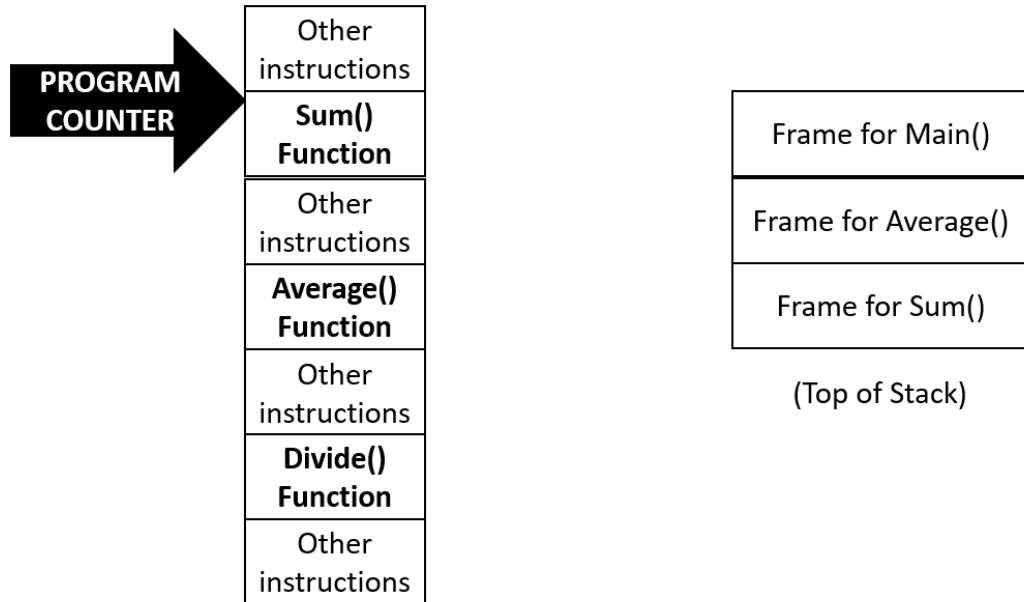
71. As explained above, the “Program Counter” is the way the computer keeps track of what instruction it is executing in memory. In this visualization, instructions are shown spread out throughout memory. The call stack is depicted as a separate memory, but that is for visual simplicity. In reality, instructions and data (including call stack) are all part of the process’s memory. As shown in the next figure, when the Average() function starts, a new call frame must be created for it and placed on the top of the stack:

Average() Function Call Frame Added



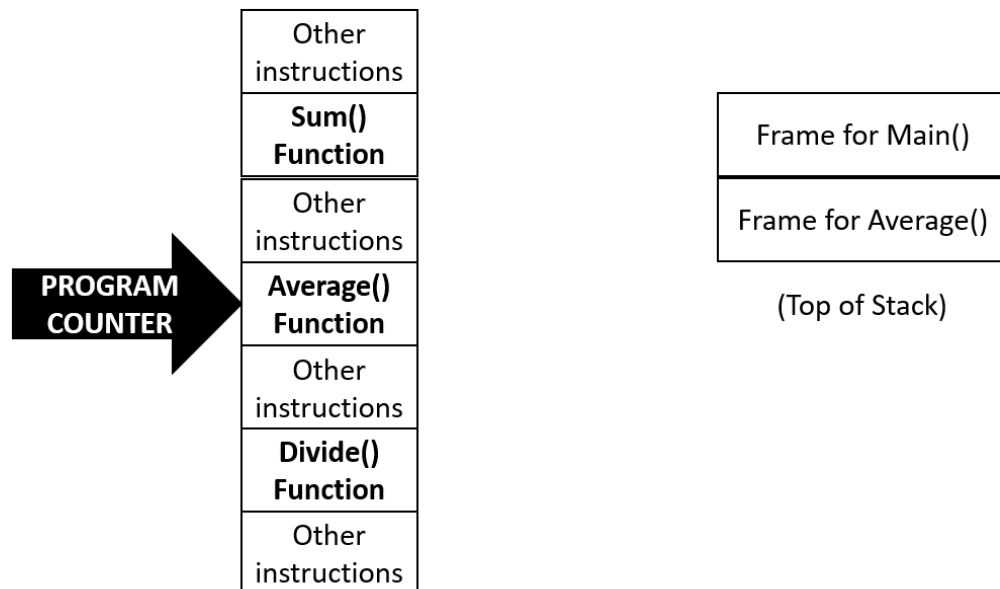
72. The call frame for Average() includes information about the instruction in “main()” that called it. It needs this information because, when it is finished, it needs to return to exactly that spot and continue on. However, before that happens, Average(), itself, calls to the Sum() function as depicted in the next image:

Sum() Function Start/Create Frame



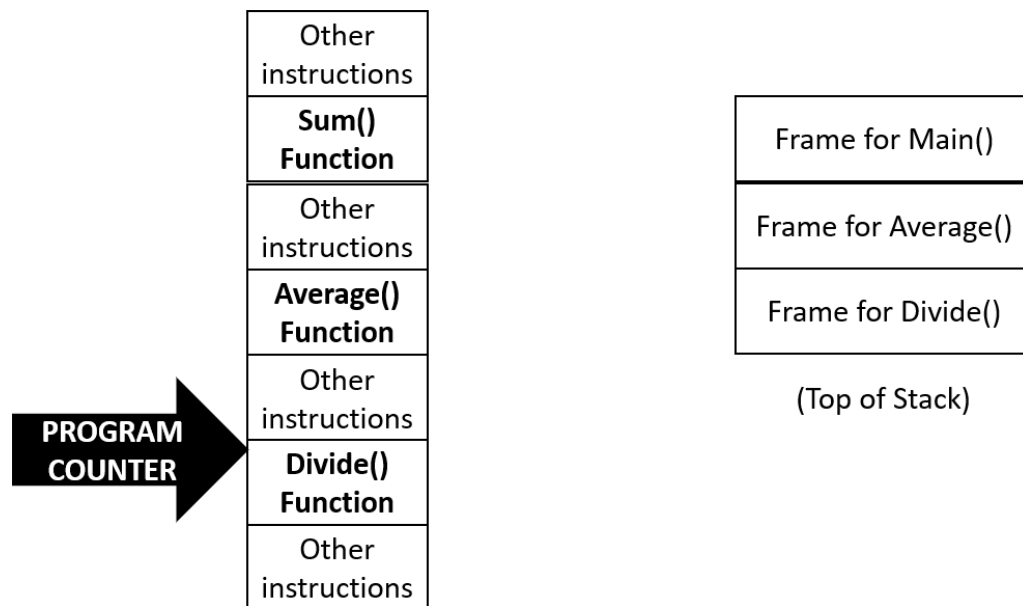
73. The Program Counter is now pointing to instructions in memory that are part of the `Sum()` function. When the `Sum()` function is finished, it will have to return to where it was in the `Average()` function. Fortunately, that information is contained within its call frame. As depicted below, when the `Sum()` function finishes, it returns to `Average()`, popping its call frame off the stack in the process.

Return to Average()/Pop Sum() Frame



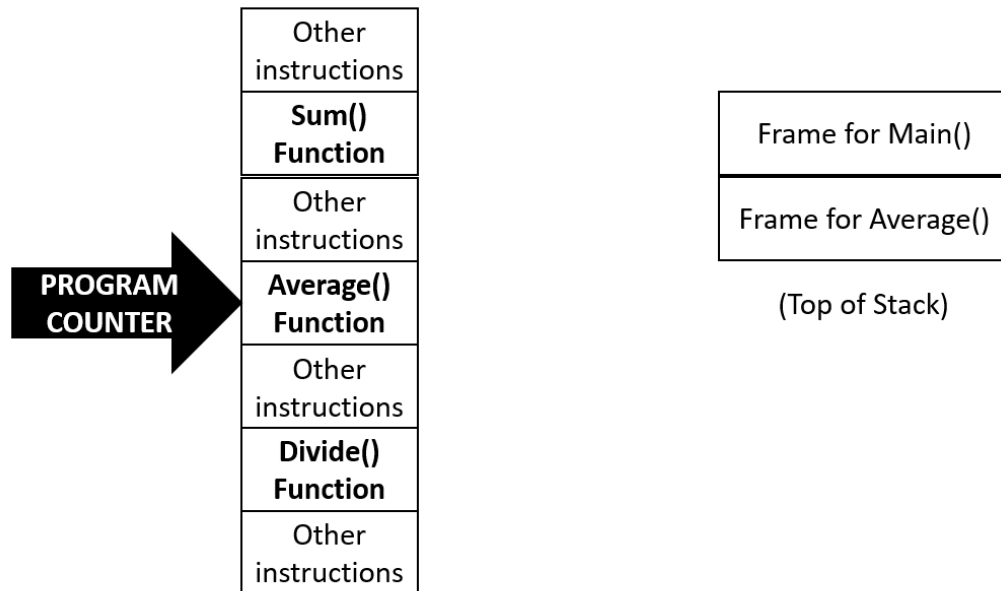
74. As depicted above, the Program Counter is **not** back at the **start** of Average(). Instead, it has returned to the exact point it was when the call to Sum() was made. Average now makes a call to Divide(), which again grows the stack (downward).

Divide() Function Start/Create Frame



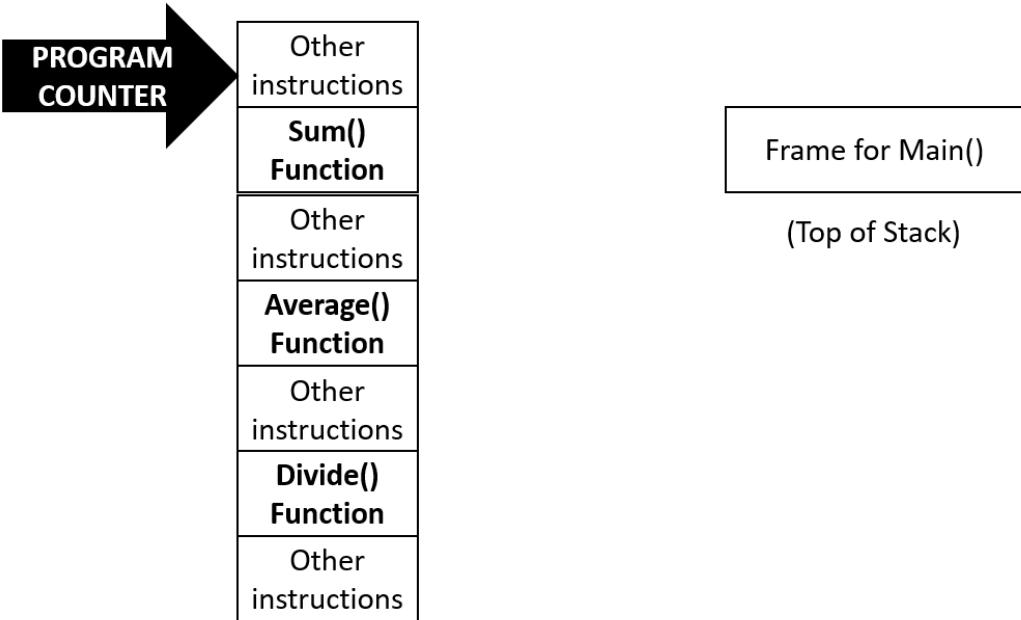
75. As with `Sum()`, when `Divide()` is over, it returns to where execution left off in `Average()`.

Return to Average()/Pop Divide() Frame



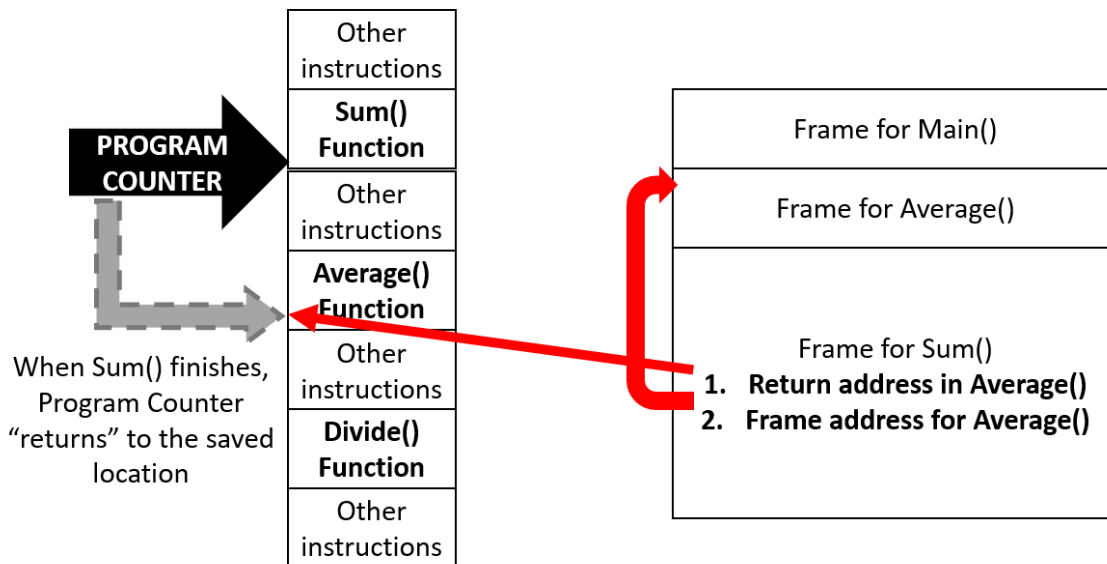
76. At this point, Average finishes its calculations and prepares to terminate. As with both Sum() and Divide(), it will return to the address stored in its stack frame and then pop its frame off the stack. Please note that the Main() function is not specifically identified in memory and I have simply pointed the Program Counter to “Other instructions”.

Return to Main()/Pop Average() Frame



77. To better visualize the storage of the return address in the stack frame, I have reproduced the visualization of the return from Sum() but this time with the stack frame for Sum() expanded to show this value:

Returning from Sum()



78. As depicted above, the call frame stores two addresses. The first is known as the “return address” and it is where the Program Counter must jump to when Sum() is finished. The second is the starting point of the previous frame. This is necessary in order to reset the call stack after Sum() is over.

79. Because each frame includes the address of the previous frame, it is straightforward to “walk” from one frame back to the previous in an iterative fashion until some originating function is reached. However, every single function is an “originating” function for the subsequent functions in the chain. That is, Main() is an originating function for Average() and Sum() in the chain depicted above, but Average() is an originating function for Sum().

80. Stack walking is a known concept used for performance analysis and security enforcement. The Microsoft tool “Xperf”, for example, can perform stack walks for performance analysis. (Ex. 1009 at pp. 7-8). As an example of security uses, programs written using Microsoft’s .NET framework (specifically, programs that use the .NET Common Language

Runtime, or CLR), “have a primitive function walk the execution stack and verify that all the callers currently on that stack have been granted the permission q guarding access to that resource.” (Ex. 1012 at p. 2).

81. In addition to saving a return address, it is often necessary to save additional context for a function. For example, functions typically store and use “scratch” data in temporary memory locations. This scratch data is kind of like a whiteboard. But when a new function is called, the newly called function will erase the whiteboard and use it for its own scratch calculations. Much like a snapshot of a real whiteboard taken with a cell phone camera, a snapshot of the local scratch data for each function must be captured and saved so that it can be restored upon return from the call. This snapshot of function information is stored along with the return address on the stack. “Most of the bookkeeping associated with a call is centered around a block of memory called a *procedure call frame*... these blocks of memory are sometimes called *stack frames*.” (Ex. 1008 at 17, italics in the original).

82. A visualization of a stack of frames is depicted in the image below:

Figure A.11 shows the stack at the call: `fact(7)`. `main` runs first, so its frame is deepest on the stack. `main` calls `fact(10)`, whose stack frame is next on the stack. Each invocation recursively invokes `fact` to compute the next-lowest factorial. The stack frames parallel the LIFO order of these calls. What does the stack look like when the call to `fact(6)` returns?

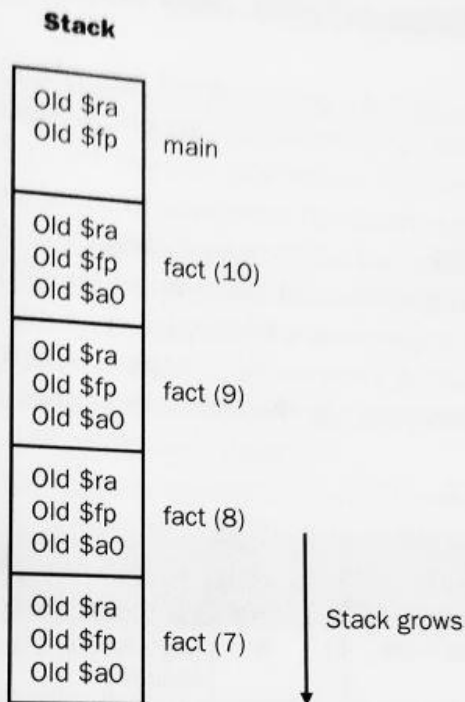


FIGURE A.11 Stack frames during the call of `fact(7)`.

(Ex. 1008 at p. 18, Figure A.11).

83. Even though the memory for holding this data is conceptually thought of as a stack, it is still just memory and can be processed by “walking” the stack. As Dalcher explains,

As shown in operation 502, stack analysis of a call stack is performed. In one embodiment, performing the stack analysis may include identifying one or more call frames of the call stack by performing a stack walk. For example, stack frames may be discovered by walking back in the stack when stack frame pointers are not present... In another embodiment, performing the stack analysis may include extracting a call return address from an

identified call frame. Further, in yet another embodiment, performing the stack analysis may include identifying an origin of code containing the return address.
(Ex. 1004 at [0036]).

84. It should also be noted that when a stack frame is “popped” (e.g., after a function call returns), the data from that stack frame remains in the block of memory reserved for the stack, even though that stack frame is no longer conceptually “on” the stack. Dalcher explains:

For example, the function may push a first call stack entry onto the call stack, and the first call stack entry may then be popped (e.g., by another function, etc.), ***where the popped first call stack entry becomes the call stack frame in freed memory in the stack.***
(Ex. 1004 at [0022], emphasis added).

85. Thus, the “freed memory in the stack” represents data that formerly represented legitimate stack frames. That memory may later be overwritten by, for example, new stack frames being pushed onto the stack, or by malicious software attempting to write data in the “freed memory”. (See, e.g., Ex. 1004 at [0022]-[0023]).

B. System Calls and APIs

86. A special class of function calls are called “system calls.” To understand their importance, it is first necessary to understand the role of an operating system (OS). The process of executing a program described above is a simplified explanation that suggests a program is loaded directly into a processor and memory. In fact, this is not usually the case. Rather, a special program, called the *operating system*, controls and manages the loading of programs. (Ex. 1008 at 16).

87. By having a special operating system that controls the execution of any other program, the system can be protected from various kinds of accidental or malicious damage caused by a program. For example, programs typically do not get to view all of memory. If they did, they could change another program’s data ***or instructions*** (because, as described above, data and

instructions are stored in the same memory and in the same format). Instead, each program is allocated a virtual (or pretend) memory address space that starts at “0” no matter what its actual address in the “real” memory. “Because multiple user programs share a single physical memory, which the operating system must also share, ***we must be able to protect the programs from one another***. Both the translation of each program’s address space and the protection of the address space from other programs are provided by virtual memory.” (Ex. 1008 at pp. 12-13, emphasis added).

88. Another way an operating system protects the computer from harmful accidental or malicious program actions is to restrict certain functions to only be executable by the operating system. In order for other application software to access these functions, they must make requests of the operating system that the OS only conditionally permits. These special functions, called ***system calls*** or ***syscalls***, are often related to sensitive system functions. As Pappas explains, “User-level programs interact with the underlying system mainly through system calls.” (Ex. 1005 at p. 5). Pappas further explains that some operating systems, such as Unix, allow the ability for a program to directly call a system call. On the other hand, the “Windows [OS] does not expose the system call interface directly to user-level programs. Instead, programs interact with the OS through the Windows API.” (*Id.*).

89. A POSITA would have known that API stands for “Application Programming Interface,” and that an API is the mechanism by which computer programs interface with other software and hardware systems. A POSITA would have known that whatever form system operations are provided to a user-space (user-level) program (i.e., through syscalls directly or a wrapping API), the provided operations are the system’s API.

90. However, API is a very general and broad term that can be applied to any software or hardware system that has a programmatic interface, and not just system or operating system calls. Software libraries incorporated into application programs include APIs. As an example, the OpenSSL is a library I have used repeatedly in my career for cryptographic operations, such as encryption. The interface for programming with the library is called an API. (Ex. 1016 at p. 1). More foundationally, the GNU Standard C library (“libc”) refers to its functions as APIs. (Ex. 1014 at p. 9³). In other words, every computer program makes use of APIs across user-space and all the way down to the kernel.

C. Processes and Memory Address Spaces

91. When a program is run on a computer, instructions and data are copied out of some form of long-term storage (e.g., hard drive) and into “memory” (e.g., RAM). The running program is known as a “process.”⁴ As explained in the previous section, each program gets its own memory address space. To repeat the explanation, each process can only see a view of memory that is limited to space allocated for that particular process. Even if the address space in true memory is starting at 10,000, the process sees a starting point at 0. In this way, a process does not have the capability to read or write to memory that is not assigned to it. These “fake” addresses are called “virtual addresses”, and “real” addresses are called “physical addresses.” (See Ex. 1008 at pp. 12-13).

92. Hardware components working in conjunction with the operating system (specifically, the **kernel** of the operating system) are responsible for handling the conversion of virtual addresses to physical addresses. Although I do not provide all the details here, the actual

³ The citation refers to the “gettext API”. In this case, “API” is referring not to the entire API (the collection of the interface function) but the specific API function “gettext”.

⁴ Actually, it might be many processes. For example, many modern Web browsers run each individual tab as a separate process. This provides certain security and performance benefits.

implementation of this process involves mapping chunks (called pages) of virtual memory to chunks (or pages) of physical memory in a “page table.” An overview of a process and the memory mapping process is:

The page table, together with the program counter... specifies the state of a [running] program. If we want to allow another [running] program to use the CPU, we must save this state. Later, after restoring this state, the program can continue execution. We often refer to this state as a *process*. The process is considered *active* when it is in possession of the CPU and, otherwise, it is considered *inactive*. The operating system can make a process active by loading the process’s state, including the program counter... The process’s address space, and hence all the data it can access in memory, is defined by its page table, which resides in memory. Rather than save the entire page table, the operating system simply loads the page table [pointer] to point to the page table of the process it wants to make active.

(Ex. 1008 at pp. 14)

93. In order to maintain the protection of the system, the operation of the operating system kernel and all other programs are treated differently. Basically, the OS (kernel) is considered to be running with *privileged* operations while normal programs are considered *unprivileged*. More commonly, these are called *user* mode and *kernel* mode. However, even user mode programs must be able to switch into kernel mode for privileged operations. As explained previously, this is often done with a *syscall*.

To enable the operating system to implement protection in the virtual memory system, the hardware must provide...

1. Support at least two modes that indicate whether the running process is a user process or an operating system process, variously called a *kernel* process, a *supervisor* process, or an *executive* process...

3. Provide mechanisms whereby the CPU can go from user mode to supervisor mode, and vice versa. The first direction is typically accomplished by a *system call* exception, implemented as a special instruction (*syscall* in the MIPS instruction set) that transfers control to a dedicated location in supervisor code space...

(Ex. 1008 at pp. 15).

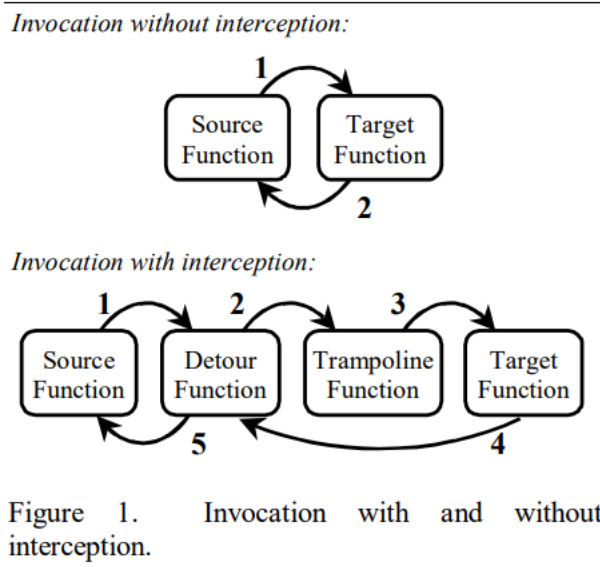
94. Notably, the ***call stack*** discussed previously is also in the process's memory space. That means that the entire call-chain path is stored in the process's memory. However, when the process switches into kernel mode, it requires a separate call stack for the function calls in the kernel. "Since every process could conceivably be running in the kernel at the same time, each must have its own kernel stack area." (Ex. 1015 at p. 1). Both of these memory spaces are process specific.

D. Call-backs, Filter Drivers, and Code Hooking

95. Computer security systems often function by *monitoring* programs for bad behavior or other signs of harmful programs. One such system/method is called "hooking." Researchers at Microsoft in 1999 described a systematic way to be able to "hook" a function call. The basic approach involved replacing one of the earliest instructions of a function with a jump instruction to a monitoring subroutine first. The monitoring code can determine that the function call is safe; if so, it executes the instruction that was removed and replaced, then jumps back to the correct place in the original function where execution can continue normally. Because of this jumping around outside of the normal code path, the researchers called this system "detours."

Detours is a library for intercepting arbitrary Win32 binary functions on x86 machines. Interception code is applied dynamically at runtime. Detours replaces the first few instructions of the target function with an unconditional jump to the user-provided detour function. Instructions from the target function are preserved in a trampoline function. The trampoline function consists of the instructions removed from the target function and an unconditional branch to the remainder of the target function. The detour function can either replace the target function or extend its semantics by invoking the target function as a subroutine through the trampoline. (Ex. 1013 at p. 1).

96. A part of this process is depicted visually in Figure 1, which I have reproduced hereinbelow:



(Ex. 1013 at p. 2)

97. Although the original white paper about Detours did not explicitly use the word hooking, Detours is described as hooking in other sources. For example, Pappas uses the Detours library and describes its operations as hooking: “[t]he interposition component is implemented on top of the Detours framework [38], which provides a library call interception mechanism for the Windows platform... [Detours] selectively hooks the set of the protected Windows API functions. Each time a protected function is called, the detour code sends a control message to the kernel component...” (Ex. 1005 at p. 10). Hooking, as described herein, are user-mode hooks. That means they intercept calls for user-mode functions including Windows API functions.

98. Another example of a hooking technique is “filter drivers.” A driver is software used to handle some type of specialized process, usually a hardware device or other low-level system operation. A “filter driver” (which implements a filter device) is a driver that hooks into an existing driver of a device to intercept data passed into or out of the driver/device. One description of filter drivers is as follows:

A filter driver implements a filter device, which is extremely helpful in tracing system events in Windows NT and Windows 2000. (For more information on drivers, see the sidebar “Drivers for Windows NT and Windows 2000.”) ***After a filter device attaches to an existing device, it starts intercepting any requests sent to that device.*** Typically, it modifies the request in some way, and then passes it on to the device. This lets it add functionality to the device; for instance, it could turn a traditional file system into an encrypted file system. ***A filter can also simply record information about requests, pass those requests unchanged to the device they were meant for, then record information about the results of those requests.*** (Ex. 1011 at p. 3).

99. A filter driver is used to intercept information at a kernel level. Unlike the Detours example above, this captures function call information of the process’s kernel memory space.

100. Dalcher refers to filter drivers as a technique for monitoring systems. “For example, the monitoring may include the use of filter drivers (e.g., file system filters, etc.)...” (Ex. 1004 at [0031]).

101. A “call back” is a generic and broad term that refers to any function that is “called back” from another function, usually to indicate some kind of progress, event, or update. (See Ex. 1016 at pp. 38-39). For example, some functions that run for a very long time have call-backs that report progress of an operation, which enable the software to, for example, update the GUI to show progress updates on the screen. (*Id.*). Other call-backs are used for logging. Many monitoring functions are call-backs in one form or another. For example, code hooking is a way of forcibly inserting a call-back into a code that did not originally have one. (See Ex. 1004 at [0031]). Filter drivers often report their results to functions that would be considered call-backs. (See Ex. 1011 at p. 7). That is, these functions would register to be “called back” after the filter driver intercepts information.

102. When a call-back is used for monitoring, the basic idea is that a function being monitored calls the call-back. As explained above, this may be inserted as part of the program,

inserted through modifications like code hooking, or indirectly through other scanning processes like filter drivers. Either way, another function is configured to receive the update that the function is running and is performing some kind of operation. The called-back function inspects what is going on or, in other words, monitors the function being executed. It may simply record what it observed (i.e., logging) or it may forcibly stop the function if the behavior is not acceptable. (See Ex. 1004 at [0030]-[0034]).

E. Buffer-Overflow-Based Software Vulnerabilities

103. I have explained in Section V.A. how normal software execution works. The flow of a program from one part to another, such as one function calling another function, is known as “control flow.” Control flow describes any execution of instructions that is not sequential. (See Ex. 1005 at pp. 3-4).

104. Computer attackers often look for ways to subvert computer programs so that the computer programs run the attackers’ instructions rather than the intended instructions. In effect, these attackers would like to make the computer programs obey them rather than their authorized users. One way to achieve this is by hijacking a program’s typical control flow. (See Ex. 1005 at p. 14).

105. One way to hijack a program’s control flow is known as a “buffer overflow.” (See Ex. 1010 at p. 5). As I explained in Section V.A., call stacks are used to track which function called another function. But the stack frames also store data for each individual function. I explain it to my students as the function’s “scratch pad” where it performs its local calculations.

106. These stack frames and their local data are, unfortunately, a means of exploitation by attackers. As explained in Section V.A., it is considered fundamental to computing that instructions and data are represented the same way and stored the same way. This is equally true in call stacks. As also explained above, return addresses (which are data but are really instructions

about where to return to when the call is finished) are stored along with the scratch data for a function.

107. But the stack frames, as I just explained, also store the local scratch-pad-like data. One of the pieces of information that can be stored in the scratch data are spaces for user inputs. For example, a function might set aside 80 bytes to hold a user's password. But a malicious user could insert **more** than 80 bytes. If the program does not carefully check this and just blindly inserts the more than 80 bytes, it will **overflow** the memory and overwrite other data. (See Ex. 1010 at p. 4).

108. Attackers have used buffer overflows to **change the return address** of a function's frame on the call stack. (See Ex. 1010 at p. 5). The attacker can use this to change the whole flow of the program. I have taught classes about this problem in undergraduate and graduate classes. Preventing these kinds of problems is a subject of much research in computer security, including the study of analyzing memory to detect overwritten stack data.

109. To illustrate how a buffer overflow works, here is an illustration I use in my classes. This figure shows a normal stack frame where the data did not overwrite the buffer. In this example, the buffer is set to hold up to 64 bytes of data. Stacks are generally depicted as "growing down" (because as I already explained it goes from high memory to low memory). Accordingly, the bottom of the stack has the "return address" and another value called the "ebp," which relates to the previous call frame. Notice that the data is 7 bytes (the numbers 1-6 plus a control character represented by "\0" that represents the end of the string).

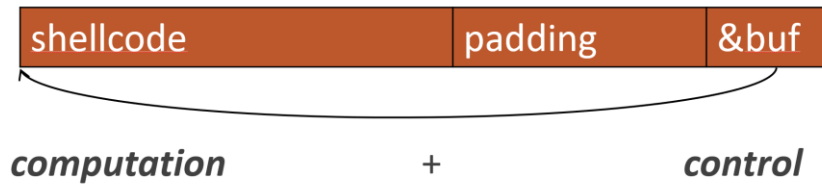


110. When a buffer overflow happens, more than 80 bytes is inserted overwriting the ebp and the return address. (See Ex. 1010 at p. 5). In the illustration below, 68 “A” characters were inserted followed by some arbitrary binary address. For illustration purposes, this address is written in hexadecimal as “DEADBEEF”, which is a commonly used example hexadecimal number used by programmers. The point is that the return address has been changed.



111. Because the return address has been overwritten, when the code attempts to pop the stack using the “ret” command, instead of returning to the correct place in memory, it will jump to whatever address was overwritten.

112. In the “old” days before there existed many defenses for this kind of attack, the evildoers would just put their instructions *in the buffer itself and have the return address be within the buffer*. I have illustrated this below (the orientation has been changed to horizontal, but the bottom of the stack with the overwritten return address is on the right-hand side):



113. The term “shellcode” refers to the fact that the attacker will almost always want to launch a command “shell” in which they can execute arbitrary commands on the system including exfiltrating data and so forth. (See Ex. 1005 at p. 2).

114. This form of buffer overflow attack does not work on most modern machines, although it does sometimes show up on weaker devices such as IOT devices. Most modern machines, however, split memory up into portions that can be written to (like buffers) and portions that can be used for executing instructions. By keeping these parts of memory separate, it is harder, if not impossible, for an attacker write their instructions to a buffer and then execute them. (See Ex. 1005 at p. 2).

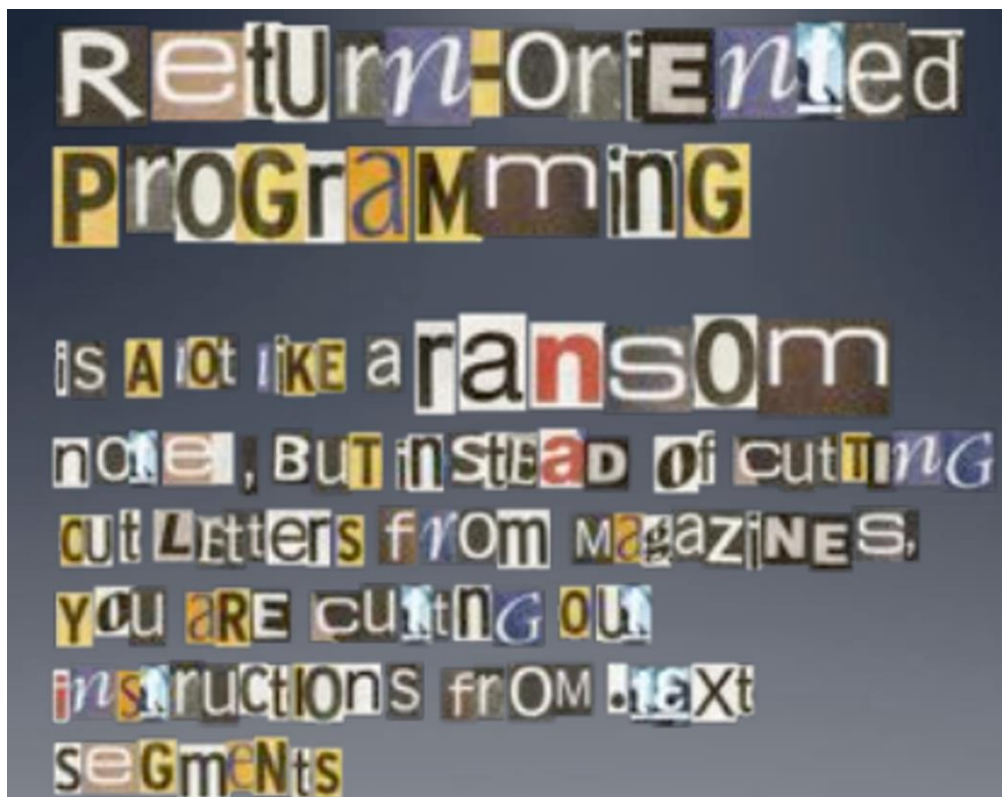
115. However, attackers can still overflow buffers and they are very creative about how they exploit it. One of the most powerful techniques in use at present is called “Return Oriented Programming,” or “ROP.” (See Ex. 1005 at p. 2).

116. ROP works when an attacker can get or figure out addresses of ***other code already in memory***. Many computer programs include libraries of functions beyond the functions they use themselves. For example, many programs include “libc,” which is short for the C library. (See, e.g., Ex. 1014 at p. 9).

117. If an attacker knows where a library like “libc” is in memory, it gives them a whole buffet of executable instructions. Once the base location of the library is found in memory it is generally easy for the attacker to know where each instruction of the library is based off this starting location. (See Ex. 1005 at p. 2).

118. In order to craft their own program, the attackers identify what have been called “gadgets.” A gadget is a very small number of computer instructions, often just one, followed by a “ret” command. Because of the way “ret” works, once the computer returns from “ret,” it automatically pops the stack. But if the buffer overflow has overwritten *past* the first return address, the attacker can queue up a whole array of return addresses. After the first one returns, it will jump to the next address the attacker inserted. When that one returns, it will move onto the next. In this way, an attacker can cause the computer to execute the array of gadgets. Even though each one is just one or a few instructions, by chaining them together, the attacker is essentially creating a new program out of the gadgets. (See Ex. 1005 at p. 3).

119. ROP has sometimes been illustrated like a ransom note created from cut-out letters from newspapers and magazines.



(Image by Dino Dai Zovi).

120. Although there are some defenses against ROP, such as Address-Space Layout Randomization (ASLR), it continues to be an effective method for attackers to subvert programs. (See Ex. 1005 at p. 2).

VI. OVERVIEW OF THE CHALLENGED PATENT

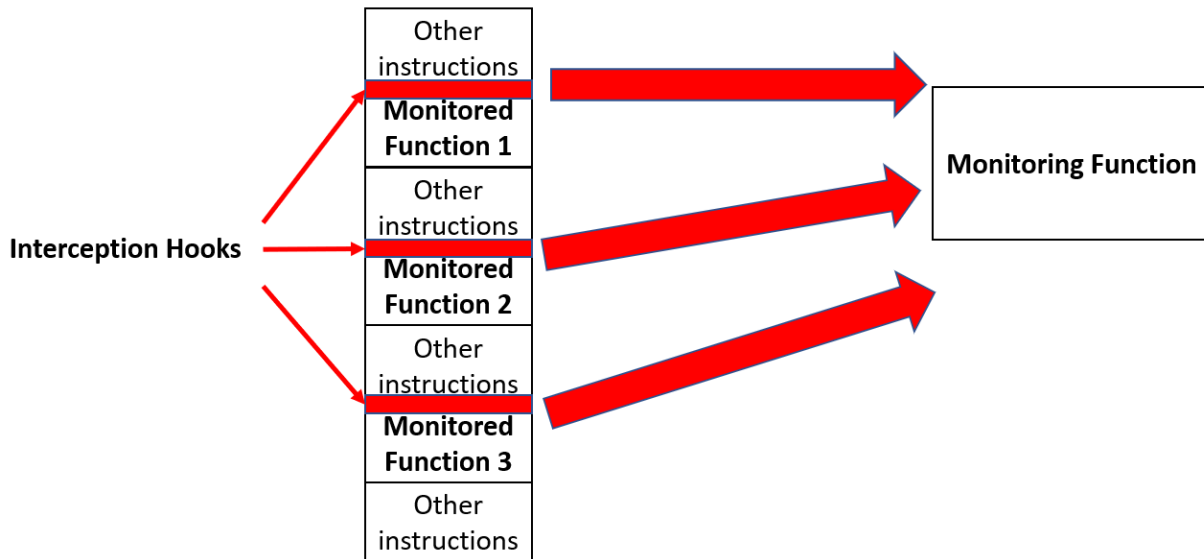
A. THE '591 PATENT SPECIFICATION AND CLAIMS

121. The '591 patent relates to “monitor[ing] a memory space of a process for execution of functions.” (Ex. 1001 at Abstract). Although the operations of the '591 patent are very familiar to security professionals including a person of ordinary skill in the art (POSITA) (see Section VII regarding POSITA), the concepts can be difficult to visualize for those unfamiliar with them. In my review of the '591 patent in this section I will both explain some of the '591 patent's concepts as well as provide some example/sample visualizations. I do not mean to imply that these examples or visualizations are limiting. Rather, they represent some common ways that a POSITA would have understood the patent at the time of the alleged invention.

122. As stated in the abstract, the '591 patent discusses the monitoring of a process's memory space. In particular, it refers to having some set of functions that are monitored within the memory space. The monitoring operation, in at least some of the claim language, comprises loading a component for evaluating the function.

123. The '591 patent offers no visualization of this. To help illustrate I have created the following figure:

Monitoring Functions



124. This is an **example** of how a memory space can be monitored and with loaded components. This particular model I have depicted follows the *Detours* code hooking paper which I have already described and cited (Ex. 1013; see Section V.D.). Basically, a “hooked” monitored functions has a small piece of code inserted at the very beginning of the function. Thus, when the function is called, instead of executing the function normally, the injected instructions redirect execution to some other function (e.g., a monitored function), which performs the evaluation/monitoring. Although not shown, when the monitoring function is complete it can return to the normal execution of the monitored function.

125. I also note that both the injected hook and the monitoring function are within the process’s memory space, meaning that each individual process (execution of the program) can be monitored and detoured differently. “The code of the target function is modified in memory, not on disk, thus facilitating interception of binary functions at a very fine granularity. For example, the procedures in a DLL can be detoured in one execution of an application, while the original

procedures are not detoured in another execution running at the same time.” (Ex. 1013 at pp. 1-2).

126. Monitoring components claimed by the ’591 patent use “stack walk processing.” The specification defines “stack walk processing” as “a process that walks a stack frame for a hooked functioned [sic] from a lowest level user mode function to an address of the invoked function/originating caller function.” (Ex. 1001 at 10:35-39).⁵ This is consistent with what was well known in the art and what I explained above, namely that a stack walk works backwards from the monitored function (i.e., lowest level user mode function) up to some previous calling function (i.e., invoked function/originating caller function) closer to the “bottom” of the stack. (See Section V.A.).

127. The purpose of this stack walk, as per the ’591 patent specification, is to check the stack memory for suspicious data/behavior. The specification states, “Generally, a non-malicious call starts from a range of pages in a zone of memory containing application or system module execution code. This zone, of memory contains data (e.g., pages) used to store a module (e.g., executable file (.exe) or dynamic-link library (DLL)) assembly code.” (Ex. 1001 at 4:5-9). On the other hand, “An anti-exploit component 102 implements anti-exploit examples described herein to detect suspicious behavior and prevent exploit code of software exploits from being executed.” (Ex. 1001 at 3:26-29).

⁵ I have been informed that in a parallel district court litigation, the owner of the ’591 patent is arguing that the ’591 patent claims should be afforded their plain and ordinary meaning. I have also been requested to adopt patent owner’s position on claim construction for the purposes of rendering my opinions set forth in this declaration. (See Section IV). Thus, prior art can disclose “stack walk processing” under patent owner’s plain and ordinary meaning even if its disclosures do not completely match this definition at 10:35-39 of the ’591 patent.

128. The '591 patent identifies some examples of well-known monitored functions that are considered sensitive (e.g., executing another process). “Functions monitored comprise processes that attempt execution of another process or loading of a module into a process memory, for example, function such as NtCreateProcess, NtCreateProcessEX, NtCreateUserProcess, LdrLoadDll, among other examples monitored functions. (Ex. 1001 at 4:61-63). As explained later, these functions, or ones similar/comparable, were identified as needing monitoring by Pappas (prior art; see Ex. 1005).

129. As mentioned above, the '591 patent discusses performing a stack walk back to “an” originating caller function. The '591 patent specification indicates what this originating calling function should be: ***the hooked/monitored function***. To be clear, in some of the citations, this is identified as the ***invoked*** function. I have included citations from the '591 patent specification that discuss this:

In the examples disclosed herein, the anti-exploit technology may ***go back through the stack_entry chain until the initial caller's stack frame is reached (e.g., back-walk the stack frame starting from the lowest level user mode function to an address where the call sequence started for the hooked function)***. Once reached, an address from where the call sequence has started for the hooked function is located and characteristics of that particular memory region are retrieved and analyzed (e.g., evaluating state information of the particular memory region). (Ex. 1001 at 5:34-43, emphasis added).

For example, stack walk processing may include performing one or more memory checks from the lowest level user function of the hooked function down through the address of the ***originating caller function (e.g., the hooked function)***. (Ex. 1001 at 6:7-11, emphasis added).

Memory checks (identified in block 210) are made continuously during the stack walk processing until the address of an ***originating caller function (e.g., hooked function)*** is reached. (Ex. 1001 at 8:10-13, emphasis added).

An example of an originating caller function is ***the hooked and monitored function*** that is associated with the stack frame being evaluated. (Ex. 1001 at 8:15-17, emphasis added).

During the stack walk processing, an address of an ***invoked function*** (e.g., ***originating caller function is identified***). (Ex. 1001 at 9:59-61, emphasis added).

Flow proceeds to operation 304 where stack walk processing is performed on a ***hooked and/or monitored function***. When one of the plurality of ***functions is invoked***, the stack walk processing is executed for a stack frame associated with an ***invoked function*** (e.g., ***originating caller function***). Execution of the stack walk processing is described in further detail in the description of FIG. 1 and is a process that ***walks a stack frame for a hooked functioned from a lowest level user mode function to an address of the invoked function/originating caller function***. Operation 304 further comprises performing one or more memory checks during the stack walk processing. Examples of memory checks that are performed are described in detail in the description of FIG. 2. ***In one example, an originating address associated with the invoked function is identified***. (Ex. 1001 at 10:30-44, emphasis added).

130. In addition, a POSITA would have understood that a monitored function is a hooked function / invoked function / originating caller (e.g., initial caller) function, based on the quotes above and also based on the following disclosures.

Examples of ***functions monitored*** by the anti-exploit technology include but are not limited to: `NtAllocateVirtualMemory`, `NtProtectVirtualMemory`, `NtCreateProcess`, `NtCreateProcessEX`, `NtCreateUserProcess`, `LdrLoadDll`, among other examples. (Ex. 1001 at 5:6-10, emphasis added).

A ***hooked function*** (including but not limited to `NtCreateProcess`, `NtCreateProcessEX`, `NtCreateUserProcess`, `LdrLoadDll`) is ***monitored***/evaluated through execution of stack walk processing. The anti-exploit component 102 executes stack walk processing that walks up a call stack for the process (function) until an address of the first function of a called function is reached. As an example, for stack walk processing, ***when one of the functions listed before is invoked***, the first step is to get the value of the call frame pointer for being able to walk the call stack frame. The call frame pointer contains the address of the stack frame for the called function. In the examples disclosed herein, the anti-exploit technology may go back through the `stack_entry` chain ***until the initial caller's stack frame is reached*** (e.g., ***back-walk the stack frame starting from the lowest level***

user mode function to an address where *the call sequence started for the hooked function*). Once reached, an address from where the ***call sequence has started for the hooked function*** is located and characteristics of that particular memory region are retrieved and analyzed (e.g., evaluating state information of the particular memory region).
(Ex. 1001 at 5:23-43, emphasis added).

131. What makes the '591 patent's explanation not straightforward is that it describes stack walk processing being initiated on a monitored/hooked/invoked function but then says that the stack walk goes back ***to this same function***. For example, the '591 patent talks of a "lowest level" function. As cited above, it states that the stack walk moves from "a lowest level user mode function to an address of the invoked function/originating caller function." (Ex. 1001 at 10:37-39; *see also id.* at 5:23-43, 6:7-11 ("For example, stack walk processing may include performing one or more memory checks from the lowest level user function of the hooked function down through the address of the originating caller function (e.g., the hooked function).")).

132. For this to be possible, there would have to be subsequent function invocations ***after*** the monitored/hooked/invoked function. This is because, as explained in Section V.A., the stack only has legitimate frames for functions that have been called and are in the current call chain.

133. In other words, the '591 Specification is describing "stack walk processing" as starting when the monitored/hooked/invoked function is invoked, but not finished until a later point wherein the actual stack walk is processed from a ***different*** (lowest-level) function back to the monitored function, consistent with its definition of "stack walk processing." The '591 patent specification provides additional explanation for how it works.

134. First, the '591 patent explains that one possible "first step" of stack walk processing is to record the address of the call stack frame for the invoked function. "As an example, for stack walk processing, when one of the functions listed before is invoked, the first step is to get the value

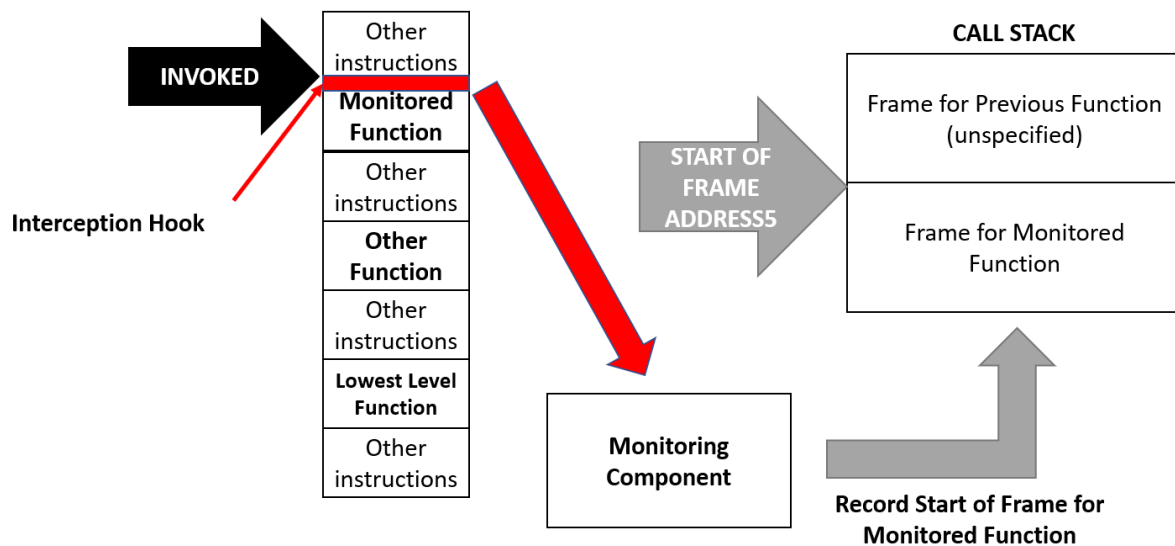
of the call frame pointer for being able to walk the call stack frame. The call frame pointer contains the address of the stack frame for the called function.” (Ex. 1001 at 5:29-32). The purpose of such a step is to basically record where in the call stack this function is so that when later functions have been called (and added to the stack), the location of the “originating” call can be easily located.

135. Second, the ’591 patent also provides an example of how the stack-walk can actually be triggered from the “lowest level” function. That is, there must be a mechanism by which the actual walk back from the lowest level function to the monitored function can be started. The ’591 patent discusses one solution for this through inserting an **additional** hook (e.g., additional to the above-discussed hooked function that is the monitored/hooked/invoked function) upon the invocation of a monitored function. “In at least one example, operation 302 further comprises applying, **upon detecting a function call of one of the plurality of functions** in the monitored memory space, **a function hooking technique to hook a lowest level user mode function** of a called process (e.g., API).” (Ex. 1001 at 10:21-26, emphasis added; see also Claim 5).

136. In this case, the monitored function is not the “lowest” level user mode function (e.g., the call right before switching to the kernel). Instead, when the monitored function is triggered, a lower-level user mode function (in fact, a lowest level, such as a syscall function), is additionally hooked for subsequent monitoring. If/when the lowest level function is subsequently invoked, the stack walk operates from that point backward instead of from the originally monitored function.

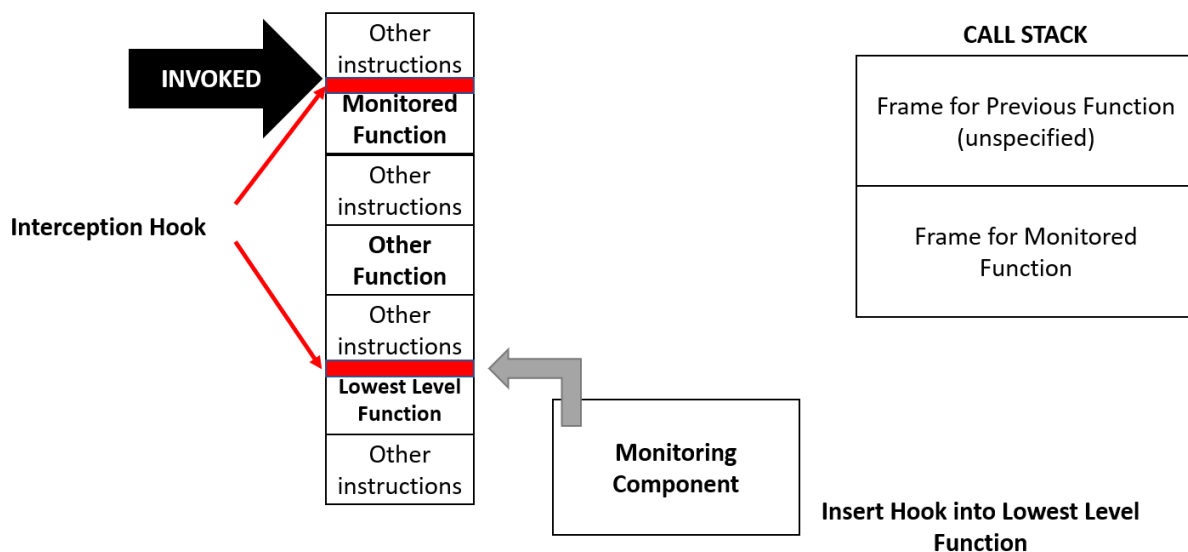
137. To help illustrate this process I have created the following figures. This figure illustrates the first step of the stack walk processing: collecting the address of the stack frame of the monitored/hooked/invoked function.

Stack Walk Processing Step 1



138. As shown in the next figure, the next step is to add a new (additional) hook to the lowest level function.

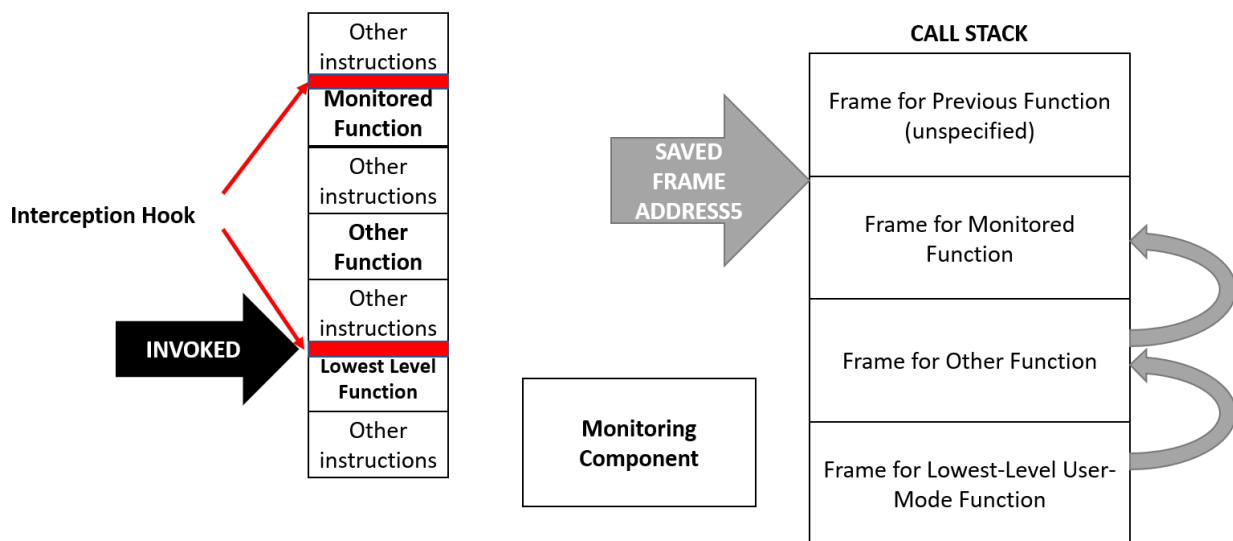
Stack Walk Processing Step 2



139. Finally, when the hooked lowest-level user mode function is invoked in a chain that started from the Monitored Function (monitored/hooks/invoked function) (with some “other”

functions thrown in), the stack walk works backwards from the lowest-level user mode function until it reaches the frame for the Monitored Function. It recognizes this using the saved address from step 1.

Stack Walk Processing Step 3



140. These explanations from the '591 patent specification are the only examples provided for how some of these components, which appear in claims, work. Thus, this is how a POSITA would understand how the above-discussed disclosures/embodiments of the '591 patent work. (See, e.g., '591 patent at 10:21-26 ("In at least one example, operation 302 further comprises applying, *upon detecting a function call of one of the plurality of functions* in the *monitored* memory space, *a function hooking technique to hook a lowest level user mode function* of a called process (e.g., API)."), emphasis added; see also Claim 5). To the extent (and even if) Patent Owner argues that the '591 patent does not involve the above-discussed process of inserting an additional hook to a lowest-level user mode function (even though such an argument is contrary to the '591 patent's disclosures), as discussed above, the '591 patent makes clear that a stack walk starts from the lowest-level user mode function, and proceeds to the above-discussed

monitored/hooked/invoked function (originating caller function). (*See id.* at 5:6-10, 5:23-43, 6:7-11, 8:10-13, 8:15-17, 9:59-61, 10:30-44.) Thus, the above-discussed monitored/hooked/invoked function is the originating caller function. (*See id.*)

141. The issue of which function is the originating caller function (of initial/invoked function) matters because several claims deal with the timing of the stack walk. That is, certain checks must be done **before** reaching the originating/invoked function in a stack walk while other checks must be done **upon** reaching the originating/invoked function in a stack walk. For example, Claim 1 recites “performing, during the executing of the stack walk processing before an address of an originating caller function is reached, a memory check . . .” (Ex. 1001 at Claim 1). Also, for example, Claim 18 identifies certain checks that must “occur[] before an address of the invoked function is reached,” while a different check “occurs upon reaching the address of the invoked function in the execution of the stack walk processing.” (*Id.* at Claim 18).

142. The ’591 patent identifies the following as examples of suspicious behavior found in the course of stack walk processing: “code execution is attempted from non-executable memory, a base pointer is identified as being invalid, an invalid stack return address is identified, attempted execution of a return-oriented programming technique is detected, the base pointer is detected as being outside a current thread stack, and a return address is detected as being inside a virtual memory area...” (Ex. 1001 at Claim 1). A POSITA would have known that these were well known memory checks for vulnerabilities (known by a POSITA).

143. Some claims of the ’591 patent also require “invoking one of the plurality of functions as a result of receiving a call from an application programming instance.” (*See*, e.g., Ex. 1001 at Claim 1). Notably, the term “application programming instance” is not a term of art. In my opinion, the claim drafter meant “application programming interface” (API), which is a term

used in the specification and a very common term in the art. (See, e.g., Ex. 1001 at 3:43-44). In my opinion, this is consistent with disclosures of the '591 patent (e.g., as to how the processes of the patent work) and a POSITA's understanding of what APIs are and how they work. (See, e.g., my explanations in Section V.B.). For example, a POSITA would have known that APIs call and invoke functions. (See *id.*). For the remainder of my analysis, I will use the term "Application Programming Instance" and "Application Programming Interface" (API) as synonyms.

144. Based on the structure of the claim, the "plurality of functions" mentioned in the above-cited limitation (e.g., functions that are invoked, as in "invoking one of the plurality of functions...") are the monitored/hooks/invoked functions that I have been discussing. The requirement for them to be invoked as a result of a call from an API is not particularly meaningful as just about every function call is invoked from an API of some sort. (See my explanations in Section V.B.). The '591 specification states, "An action of calling a system process (e.g., function of an application or an application programming interface..." (Ex. 1001 at 3:43-44). It also states, "This component intercepts monitored operating system applications **or APIs...**" (Ex. 1001 at 4:47-48, emphasis added). A POSITA would have understood the APIs referred to in these sentences refer to operating system calls (syscalls) or API wrappers, such as those used by Windows. In other words, certain APIs are monitored and comprise/call monitored system calls / monitored functions. (See *id.*; see also *id.* at 4:36-38 ("The anti-exploit component 102 executes a function hooking technique to hook to the lowest level user mode function and **intercept system routines of a called API.**"), 5:2-4 ("The anti-exploit examples of the anti-exploit component 102 are configurable to be programmed to **monitor any type of function of an application/API.**"), 9:67-10:4 ("If the anti-exploit technology intercepts **a call to API functions such as or similar to**

NtCreateProcess/NtCreateProcessEx/LdrLoadDll ¹⁶¹ from one of such entries, it automatically marks this behavior as suspicious and potentially malicious.”), emphasis added; *see also* my explanations in Section V.B.).

145. The '591 patent claims also refer to a “cache.” For example, claim 2 includes the requirement of “when the address of the invoked function address is found in a cache, preventing execution of a payload for the invoked function.” (Ex. 1001 at Claim 2). The '591 patent specification explains that the term “cache” may refer to memory set aside by the anti-exploit technology for monitoring the process: “The cache component 104 is a structure (hardware or software) that stores data related to a memory region for a monitored process. Memory used for a cache is allocated by the anti-exploit technology, for example by the anti-exploit component 102... A cache is a specific memory region allocated in each monitored process where information about monitored memory pages is tracked.” (*Id* at 6:31-57).

146. Claim 17 of the '591 patent requires checking “a low level user mode function is attempted to be executed in a cache.” (Ex. 1001 at Claim 17). A POSITA would have further understood that, if the cache is the memory for storing data about monitored memory of process, that “executing in” the cache refers to having tracked memory (e.g., memory addresses) about the function in the cache. This is also consistent with the provisional patent application of the '591 patent in which a cache is described this way:

If the address is found in a cache entry, then the call is suspicious. Every entry present in the cache structure represents a part of memory that has been recently allocated with the flag EXECUTE, thus a potential piece of memory that could host malicious code.

⁶ As discussed above, the '591 patent explains that these functions are monitored functions. (See, e.g., '591 patent at 5:6-10 (“Examples of **functions monitored** by the anti-exploit technology include but are not limited to: NtAllocateVirtualMemory, NtProtectVirtualMemory, **NtCreateProcess**, **NtCreateProcessEX**, NtCreateUserProcess, **LdrLoadDll**, among other examples.”), emphasis added.)

(Ex. 1007 at p. 15).

B. PROSECUTION HISTORY

147. I understand that the '591 patent issued as a result of the prosecution of the 14/606,604 application. I further understand that the prosecution began with a non-final Office Action on December 16, 2016, continued through two non-final Office Actions, one final Office Action, three sets of claim amendments, and one Request for Continued Examination, ending in a Notice of Allowance that issued on December 20, 2018. (Ex. 1006 at pp. 18–145).

148. During prosecution of the '591 patent, claims 1-20 were rejected in a non-final Office Action that issued on August 30, 2018. I understand that claims 1-6, 8-15 and 17-20 were rejected under 35 U.S.C. § 103(a) as unpatentable over U.S. Patent Publication 2012/0255018 to Salaam in view of U.S. Patent Publication 2007/0016914 to Yeap. (Ex. 1006 at pp. 62–74).

149. In an amendment filed October 1, 2018, Applicants responded to the non-final Office Action by amending the independent claims. Specifically, the independent claims were amended to recite “... performing, during the executing of the stack walk processing before an address of an originating caller function is reached a memory check for a plurality of stack entries identified during the stack walk processing to detect suspicious behavior, wherein an alert of suspicious behavior is triggered when the performing of the memory check detects....” (Ex. 1006 at pp. 40-47; underlines showing the amended language). In associated remarks, Applicants argued that Salaam and Yeap do not disclose the limitations added in the claim amendments. (*Id.* at pp. 48–53). In the Notice of Allowance, the Examiner did not provide an examiner’s statement of reasons for allowance and instead merely noted that the Applicant’s amendments and remarks were persuasive. (*Id.* at pp. 19–20).

VII. LEVEL OF SKILL IN THE ART

150. Based on my knowledge and experience, it is my opinion that a person of ordinary skill in the art (“POSITA”) at the time of the earliest claimed priority date for the ’591 patent would have had a bachelor’s degree in computer science, or a related field, or an equivalent technical degree or equivalent work experience, and an additional 1-2 years of education or experience in the field of computer security topics such as malware/exploit detection. More education can supplement practical experience and vice versa.

151. I meet that definition of a person of ordinary skill in the art, and I did so on, in, and before January 2014⁷. I also have worked with persons of ordinary skill in the art as part of my professional and academic experience, and I have an understanding of their skill level in and before January 2014.

VIII. OVERVIEW OF THE PRIOR ART

A. OVERVIEW OF US20130275981A1 PUBLISHED APPLICATION TO DALCHER (“DALCHER”)

152. U.S. Published Patent Application No. 20130275981 to Dalcher (“Dalcher”) discloses memory monitoring techniques that can be used, for example, to detect malware. As Dalcher explains, “A system, method, and computer program product are provided for monitoring an execution flow of a function... Further, an execution flow of the function is monitored...” (Ex. 1004 at Abstract). As for the purpose, Dalcher explains that “the execution flow of the function may be monitored for malware.” (Ex. 1004 at [0025]).

⁷ The prior art references I use in my analysis were publicly available to a POSITA prior to January 2014, the priority date of the provisional application that eventually issued as the ’591 patent. If Petitioner argues that the provisional application does not fully support the claims of the ’591 patent and that a later priority date should be used instead, this does not affect my analysis and conclusions.

153. Dalcher discloses monitoring the execution of a (potentially harmful) program running on a computer by analyzing the “flow” of the execution. As I explained in the technical background, program flow across subroutines is captured in a computer’s stack (in memory) that stores the context of each subroutine. Dalcher explains that the monitoring is achieved using stack analysis:

In one embodiment, monitoring the execution flow of the function may include monitoring the call stack frame from the freed memory in the call stack. For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame. In another example, the monitoring point may identify code involved in the handling of the test event. In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame.
(Ex. 1004 at [0023]).

154. Dalcher illustrates that this process takes place in a computer, as shown in Dalcher’s Figure 2, reproduced hereinbelow.

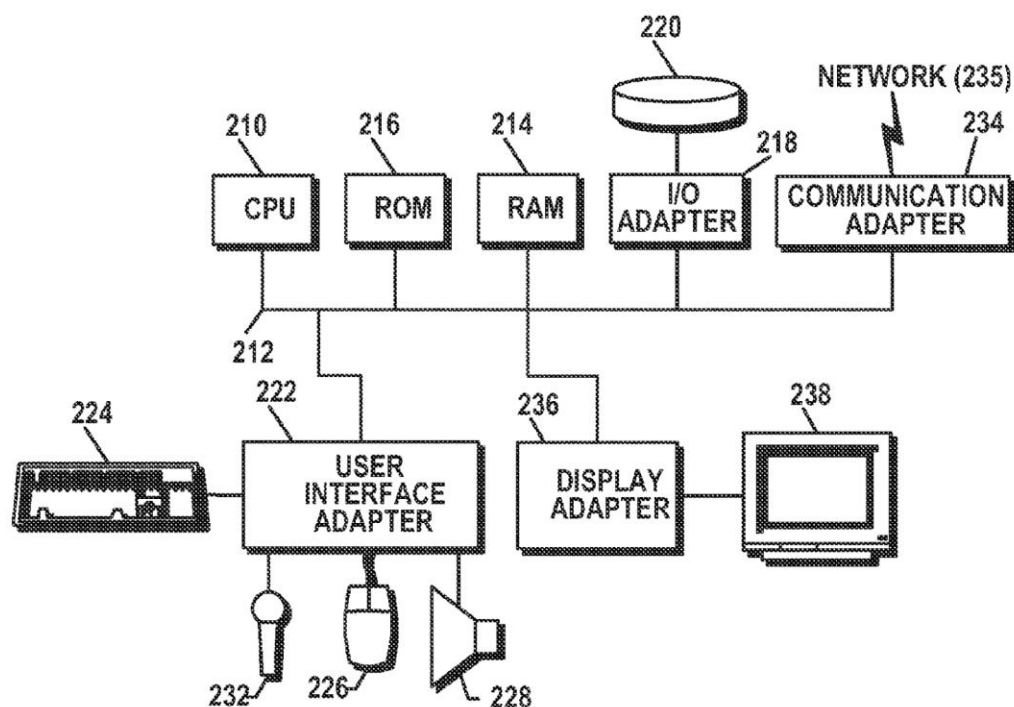
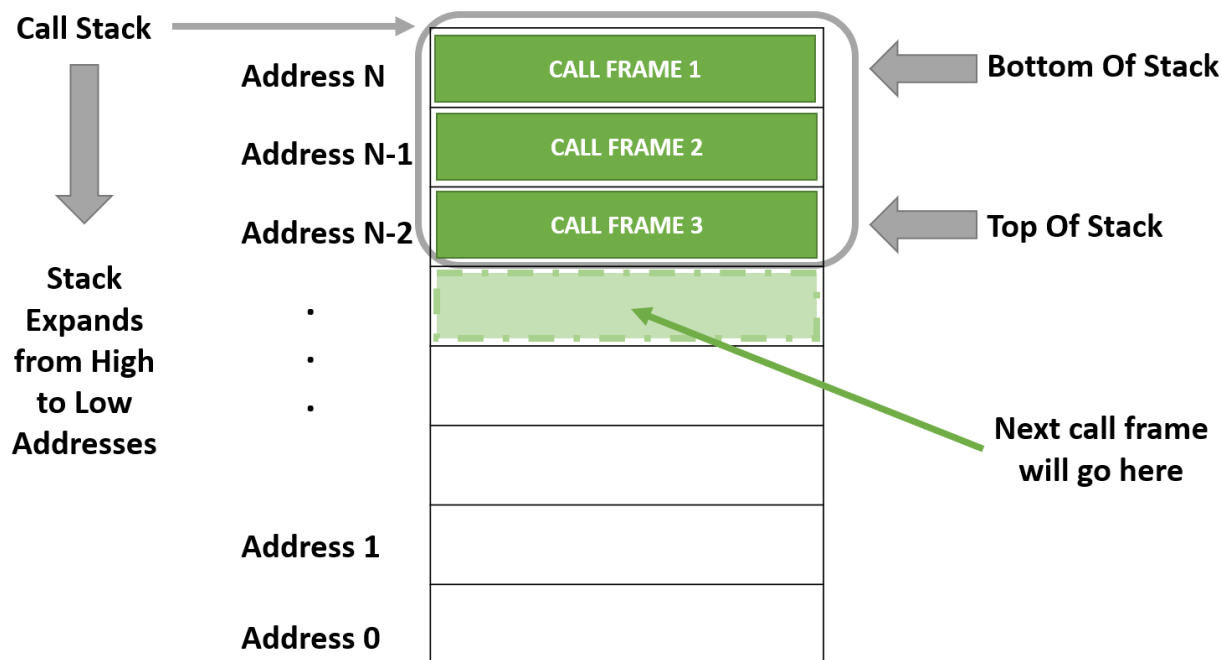


FIGURE 2

155. Dalcher explains that its disclosures/methods (e.g., including disclosures/methods about “monitoring an execution flow of a function” and stack walks (discussed further below)) take place within the context of environments like the one described in Figure 2 (which involves computers). (Ex. 1004 at [0016]).

156. In order to identify stack frames to analyze, Dalcher discloses the use of stack walking. Dalcher teaches that “performing the stack analysis may include identifying one or more call frames of the call stack **by performing a stack walk.**” (Ex. 1004 at [0036], emphasis added). Dalcher further discloses, as part of a stack walk, analyzing stack frames prior to some kind of “top” frame. In more detail, Dalcher discusses identifying a “top” part of a call stack and analyzing the stack frames that come before it. As explained by Dalcher, “Additionally, as shown in operation 504, extended call stack analysis is performed. In one embodiment, performing extended

call stack analysis may include identifying a current top of the call stack. For example, the call stack may grow from top to bottom as functions are called, and may shrink from bottom to top as functions return.” (Ex. 1004 at [0037]). A POSITA would have understood this step of “extended call stack analysis” to be used with stack walking. As I explained in the technical background, it is common in the art to refer to the “top” of the stack as the last function called. However, many systems have the stack grow “downward” in terms of memory address. This means that the bottom (or origin) of a stack is at some high address in memory but as data is added to the “top” of the stack, these new allocations are at a **lower** place in memory. A POSITA would have understood Dalcher’s disclosure about growing from top to bottom to refer to memory addresses and identifying a “top” (address) of the stack, **which is the bottom of the conceptual stack**, to be identifying some initial function’s stack frame address. For clarity, I include my diagram from the technical background (note that the “Bottom of Stack” here is the bottom of the conceptual stack, but the “top” of the addresses used in the stack):



157. Dalcher also explained that each frame in the call stack could be analyzed to determine its location in the chain, to further determine if the original frame, such as the original call in the code, had been reached. “Additionally, in another embodiment, it may be determined whether the suspected call return address points to code. Further, in yet another embodiment, the code pointed to by the suspected call return address may be analyzed. For example, the code may be examined for a preceding call that may explain the invocation of the candidate stack frame. For instance, the preceding call may be a call to the candidate stack frame, another stack frame, etc.” (Ex. 1004 at [0039]). Although Dalcher does not reuse the term “stack walk” in this citation, a POSITA would have understood determining a preceding call to a stack frame, and then another stack frame, etc. to be stack walking.

158. For each of the identified frames, Dalcher discloses analyzing it for “compromises,” which a POSITA would understand are indicative of malicious behaviors that compromise the security of a computer/system. Dalcher states that “it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of ***the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame***... includ[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024], emphasis added). Dalcher analyzes each frame to determine a “confidence” in its “genuineness.” “Further, as shown in operation 506, a candidate stack frame is identified as a result of the extended call analysis... Further still, as shown in operation 508, an amount of confidence associated with the candidate stack frame is determined.” (Ex. 1004 at [0039]; see also [0041] that explains confidence is in whether or not the stack frame is “genuine”).

159. Dalcher explains that one way the call stack is monitored is by inserting monitoring points, such as “hooks”, that trigger analysis. Dalcher teaches “... the monitoring may include the use of filter drivers (e.g., file system filters, etc.)... [or] code hooking. For example, the monitoring may include modifying code to be monitored so that it transfers control to a callback when executed.” (Ex. 1004 at [0031]). As I explained in Section V.D., filter drivers, code hooking, and other callback functions are mechanisms for monitoring code. With reference to hooking, Dalcher explains in detail that, “the data associated with the function may be identified by... hooking based monitoring of the call stack...” (Ex. 1004 at [0020]). Some systems provide support for hooking: “In yet another embodiment, the monitoring may include the use of central processing unit (CPU) debugging extensions. For example, a limited number of CPU-based monitors may be registered to invoke callbacks when indicated code is executed.” (Ex. 1004 at [0031]).

160. As cited above, Dalcher also explains that additional monitoring may be inserted as part of monitoring some initial point. Dalcher teaches, “...enabling the additional monitoring may include placing one or more additional monitoring points (e.g., hooks, probes, etc.) ***in the call stack or the freed memory in the call stack.***” (Ex. 1004 at [0033], emphasis added). A POSITA would have understood that any of Dalcher’s disclosures about “monitoring” apply to the “additional monitoring points.”

161. Although Dalcher regularly uses the terms “in another embodiment,” a POSITA would have understood that Dalcher’s disclosures are different examples and applications within the same compatible framework. None of Dalcher’s techniques are incompatible with each other and are, in fact, complementary. For example, as explained above, Dalcher describes monitoring and identifying potentially malicious behavior such as overwritten stack entries. But Dalcher also explains that monitoring can be for any purpose. “In another embodiment, the execution flow of

the function may be monitored for malware. Of course, however, the execution flow of the function may be monitored for any purpose.” (Ex. 1004 at [0025]). Nevertheless, a POSITA would have understood that nothing prevents monitoring execution flow for malware **and** for other purposes.

162. In another example of compatible embodiments, Dalcher states: “[i]n one embodiment, monitoring the execution flow of the function may include monitoring the call stack frame **from the freed memory in the call stack.**” (Ex. 1004 at [0023], emphasis added). Nevertheless, a POSITA would have understood that monitoring the freed memory (which Dalcher describes as “one embodiment”) is perfectly compatible with monitoring other parts of memory.

163. Also, Dalcher explicitly ties various embodiments together. For example, Dalcher includes Figures 3-5 and the associated specification text that, although identified as embodiments, are built on top of each other. Fig.3, reproduced herein below, is summarized by Dalcher as “FIG. 3 shows a method 300 for monitoring an execution flow of a function, in accordance with another embodiment.” (Ex. 1004 at [0016]).

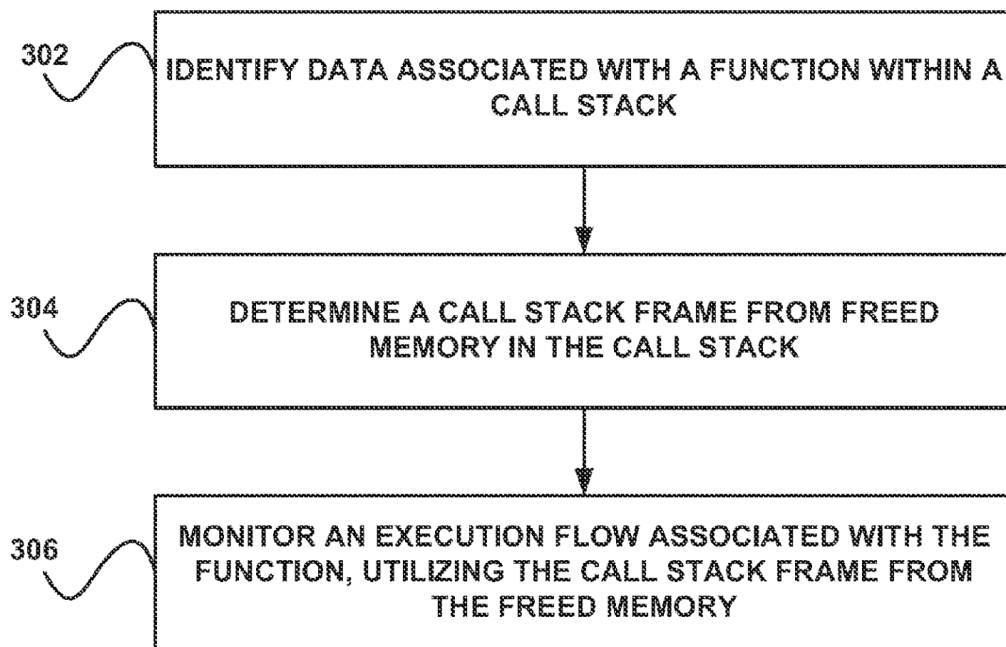


FIGURE 3

164. Figure 4, reproduced hereinbelow, is described as potentially building on top of Figure 3. Dalcher explains that “FIG. 4 shows a method 400 for performing execution flow monitoring using enhanced call stack analysis, in accordance with another embodiment. *As an option, the method 400 may be carried out in the context of the architecture and environment of FIGS. 1-3.*” (Ex. 1004 at [0027], emphasis added).

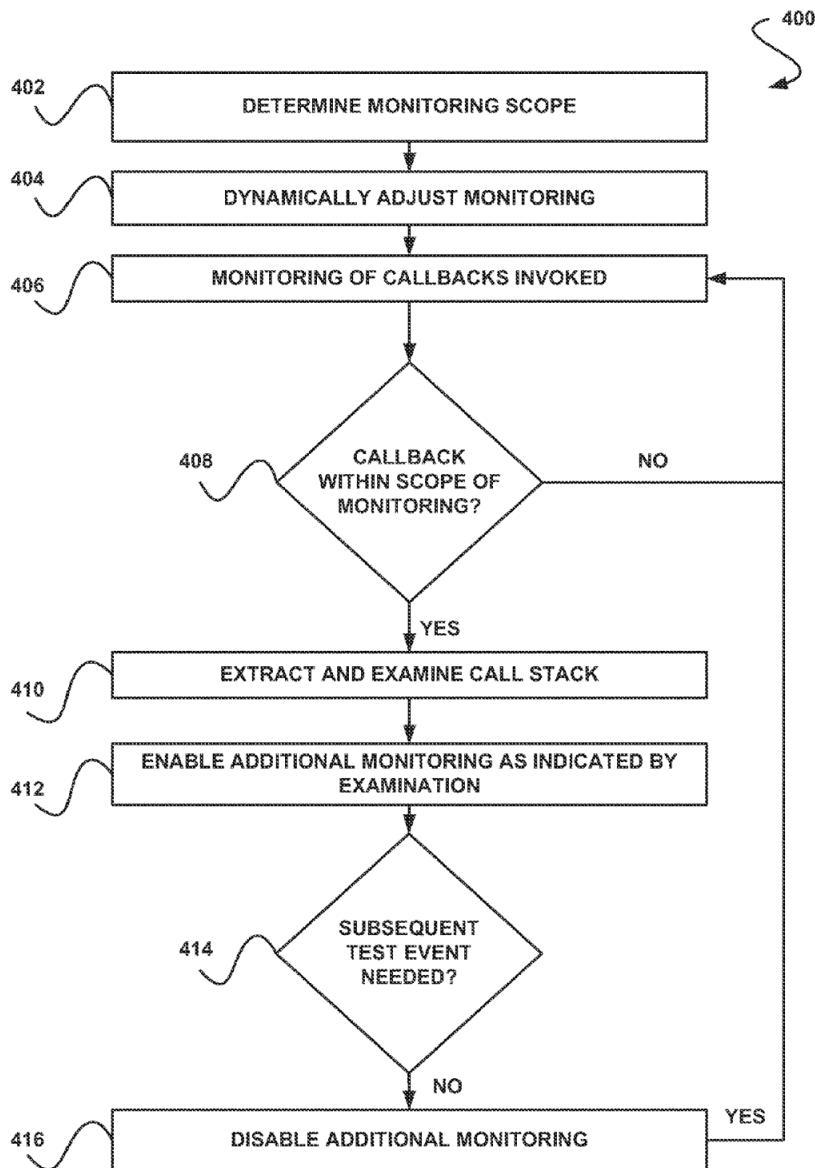


FIGURE 4

165. Thus, Dalcher teaches at least one embodiment wherein the embodiments illustrated in Figure 3 and Figure 4 are combined. As an illustration, Dalcher explains that the embodiment illustrated by Figure 4 may be used to monitor “freed memory” as discussed with respect to the embodiment illustrated in Figure 3. Dalcher teaches, “...enabling the additional monitoring may include placing one or more additional monitoring points (e.g., hooks, probes,

etc.) *in the call stack or the freed memory in the call stack.*” (Ex. 1004 at [0033], emphasis added).

166. Similarly, Figure 5, reproduced hereinbelow, is described as potentially building on top of Figures 3 and 4. Dalcher explains that “FIG. 5 shows a method 500 for performing extended call stack analysis, in accordance with another embodiment. *As an option, the method 500 may be carried out in the context of the architecture and environment of FIGS. 1-4.*” (Ex. 1004 at [0035], emphasis added).

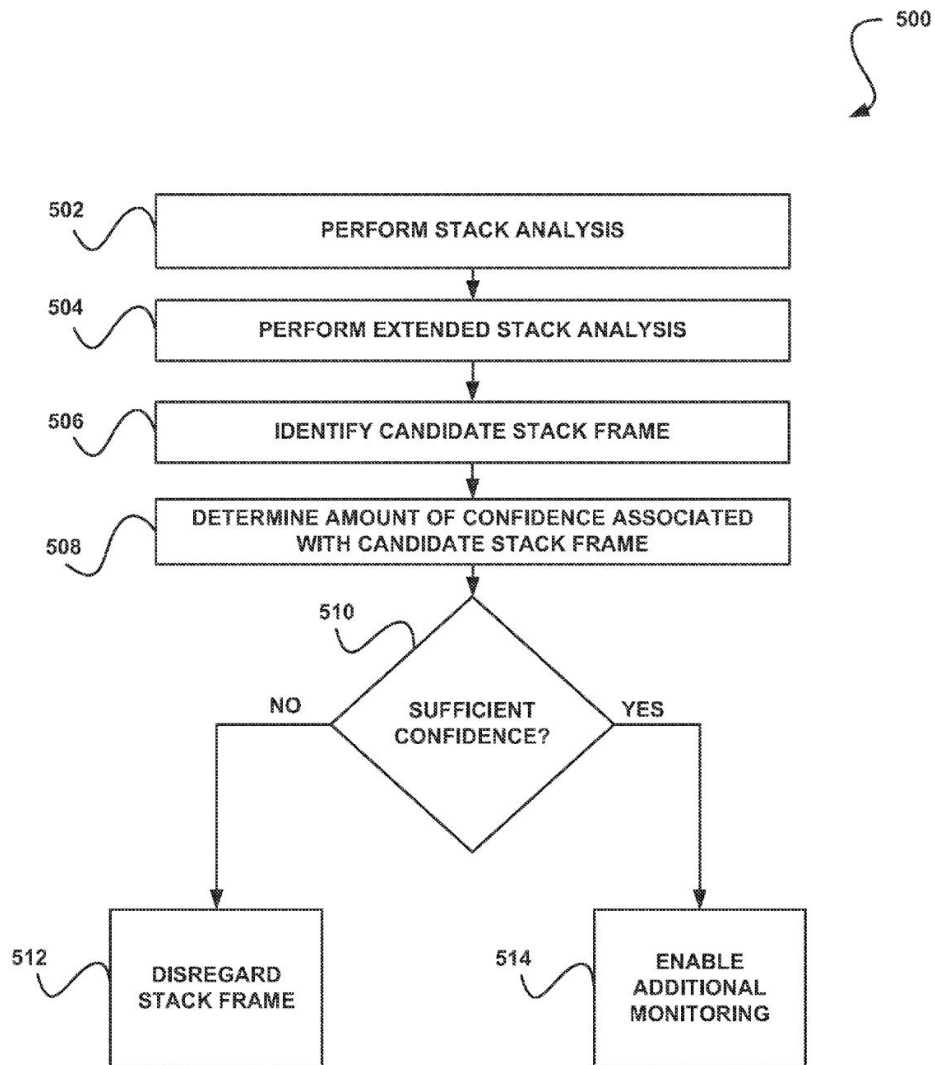


FIGURE 5

167. Accordingly, Dalcher discloses at least one embodiment wherein the embodiments illustrated in Figure 3, Figure 4, and Figure 5 are combined. More explicitly, the embodiment associated with Figure 4 explicitly discloses the use of “extended call stack analysis.” Dalcher teaches, “if in decision 408 it is determined that the callback is within the scope of the monitoring, then in operation 410, a call stack for the operation that triggered the monitoring callback is

extracted and examined.” (Ex. 1004 at [0033]). As I explained with respect to Figure 5., the embodiment associated with the figure is for “extended call stack analysis.” Thus, Dalcher explains that the extended call stack analysis of Figure 5, which identifies more candidate frames and identifies if they are genuine, is used as part of the extended monitoring of Figure 4.

168. Thus, a POSITA would have recognized that Dalcher’s “embodiments” are different examples and applications of the same techniques and are compatible with each other.

B. OVERVIEW OF TRANSPARENT ROP EXPLOIT MITIGATION USING INDIRECT BRANCH TRACING (“PAPPAS”)

169. The research paper “Transparent ROP Exploit Mitigation Using Indirect Branch Tracing” by Pappas, et. al (“Pappas”), was presented at the 22nd Usenix Security Symposium. The Usenix Security Symposium, often just called “Usenix Security”, is a competitive and highly respected security venue for academic researchers.

170. As suggested by the title of the paper, Pappas is focused on a type of attack technique known as “Return-oriented programming,” or “ROP.” As Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable [memory] page protections... we present a practical runtime ROP exploit prevention technique for the protection of third-party applications.” (Ex. 1005 at p. 2).

171. To identify ROP attacks, Pappas relies on a technique called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6). Because the LBR contains the branching instructions for “call” and “ret” a POSITA would

have recognized that the LBR is a subset of the entire call stack. Through call and ret it includes, at the very least, which function called another function.

172. Notably, Dalcher’s describes the term “call stack” in similar terms: “In one embodiment, the [call stack] entry may include a return address, and the return address may identify the function that pushed the entry onto the call stack.” (Ex. 1004 at [0018]).

173. Pappas illustrates the approach in Figure 1, reproduced hereinbelow:

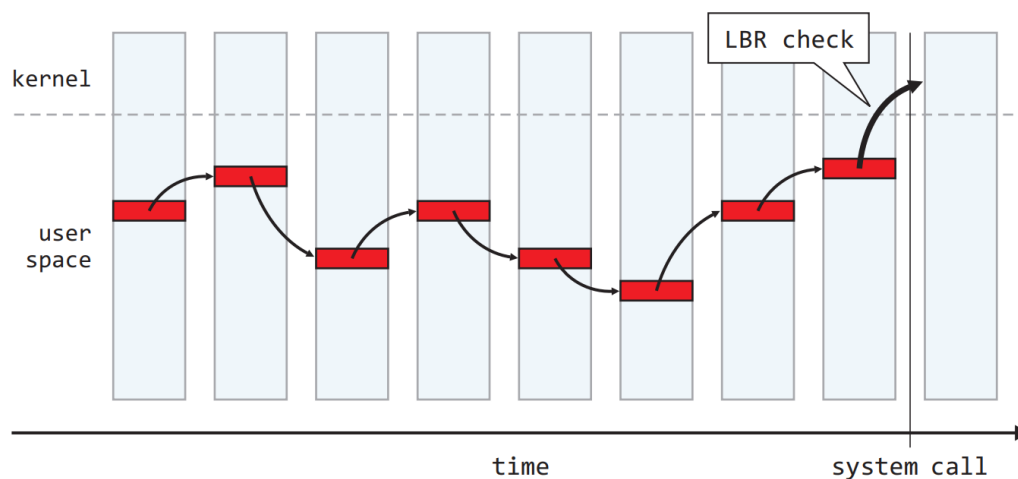


Figure 1: Illustration of a basic scheme for ROP code detection. Whenever control is transferred from user to kernel space (vertical line), the system inspects the most recent indirect branches to decide whether the system call was invoked by ROP code or by the actual program.

174. As noted in the Figure 1, the LBR check is triggered at the transition from user space to kernel space. As Pappas explains, “The vertical line denotes the point at which the flow of control is transferred from user space to kernel space through a system call. At this point, by interposing at the OS’s system call handler, the system can access the LBR stack and retrieve the targets of the indirect branches that led to the system call.” (Ex. 1005 at pp. 3-4).

175. Pappas also notes that on Windows systems there is another layer between the user space calls and the kernel. Specifically, Windows provides API calls in user space that are higher than the actual user space system calls that switch over to the Kernel. This is illustrated by Pappas in Figure 3, reproduced hereinbelow:

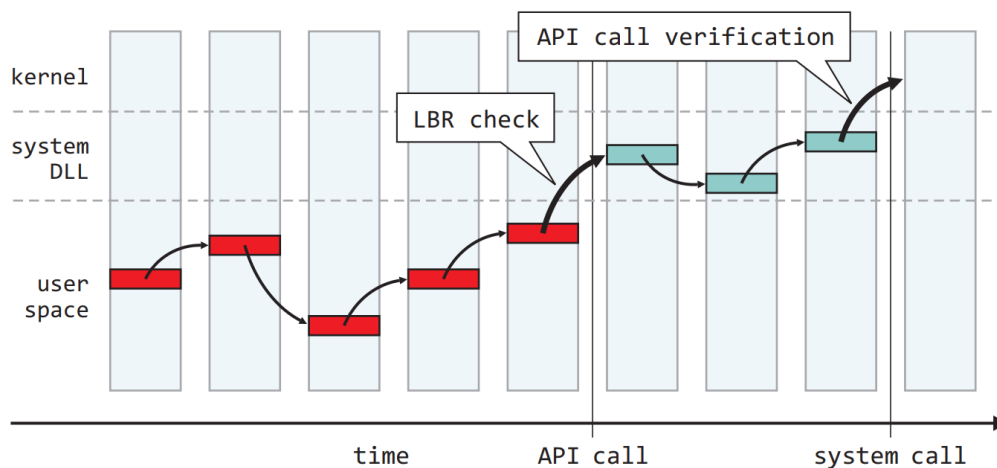


Figure 3: Overview of the detection scheme of kBouncer. Before the invocation of protected Windows API functions, the system inspects the LBR stack to identify whether the execution path that led to the call was part of ROP code, and writes a checkpoint. To account for ROP code that would bypass the check by jumping over kBouncer’s function hook, the system then verifies the entry point of the API function at the time of the corresponding system call invocation.

176. As noted above, Pappas calls the implementation of their concept “kBouncer.” As also noted in the text of the Figure above, kBouncer hooks functions in order to know when to trigger. Pappas explains that kBouncer “...selectively hooks the set of the protected Windows API functions.” (Ex. 1005 at p. 10). As shown in Figure 3 above, when the API call happens, kBouncer calls back through the hook to trigger the LBR check over the LBR stack (which, as explained

above, is a subset of the full call stack). The LBR check backs up and verifies that the API call is from the expected program and not a ROP chain. As Pappas teaches, “whenever a control request is received from the user-space component, kBouncer analyzes the contents of the LBR stack, looking for the attributes described in Sec. 3.” (Ex. 1005 at p. 10).

177. Pappas uses the LBR stack to look for gadget chains, which I explained in Section V.E. As Pappas notes, real code can have executions that look like gadgets. But a long sequence of them together is the hallmark of a ROP attack. “To account for potential future exploits of these sorts, the second attribute that kBouncer uses to identify the execution of ROP code is an inherent characteristic of its construction: the observation of several short instruction sequences chained through indirect branches... Although legitimate programs also contain an abundance of code fragments linked with indirect branches, these fragments are typically much larger than gadgets, and more importantly, they do not tend to form long uninterrupted sequences (as we show below).” (Ex. 1005 at p. 8).

IX. GROUND 1: CLAIMS 1-19 ARE RENDERED OBVIOUS BY DALCHER IN VIEW OF PAPPAS

178. As I explain in greater detail hereinbelow, a POSITA would have found claims 1-19 rendered obvious by Dalcher in view of Pappas.

A. Motivation to Combine Dalcher in view of Pappas

179. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher and Pappas are both directed toward the same goal: monitoring a program’s execution in memory for aberrations or other signs of errors or malicious activity. Dalcher declares, “A system, method, and computer program product are provided for monitoring an execution flow of a function... Further, an execution flow of the function is monitored...” (Ex. 1004 at Abstract). As for the purpose, Dalcher explains that “the execution flow of the function may be monitored for malware.”

(Ex. 1004 at [0025]). Similarly, Pappas sets forth its goal: “Return-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable [memory] page protections... we present a practical runtime ROP exploit prevention technique for the protection of third-party applications.” (Ex. 1005 at p. 2).

180. As I describe in detail in Section VIII.A., although Dalcher regularly uses the terms “in another embodiment,” a POSITA would have understood that Dalcher’s disclosures are different examples and applications within the same compatible framework. None of Dalcher’s techniques are incompatible with each other and are, in fact, complementary. Furthermore, Dalcher explicitly ties various embodiments together. Thus, a POSITA would have recognized that Dalcher’s “embodiments” are different examples and applications of the same techniques and are compatible with each other. (See my analysis in Section VIII.A.).

181. A POSITA would have been motivated to combine Pappas with Dalcher to improve Dalcher. Dalcher describes providing a “call stack analysis” that traces one call to another. Dalcher also described “in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... includ[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024]). As indicated by these citations, Dalcher provides a framework for identifying the chain of frames in a call stack using a wide range of techniques.

182. A POSITA would have recognized that Pappas relies on a call-stack tracing implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas

explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6).

183. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher. A POSITA would have been motivated to make this combination in order to achieve Dalcher’s goals of identifying “compromises.” Pappas provides implementation details that work with Dalcher’s stack tracing framework.

184. A POSITA would have been especially motivated to use Pappas as an implementation of stack tracing because Pappas identifies attacks using “Return Oriented Programming.” As Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2). A POSITA would have been motivated to combine Pappas’ teachings in order to deal with the growing ROP threat.

185. Moreover, a POSITA would have been motivated to use Pappas for its low false positive rate, which was borne out of experimentation. “This is a generic constraint that holds for both return-oriented and jump-oriented code (or potential combinations—in the rest of this section we refer to both techniques as ROP)... For the current set of protected functions we use a threshold of eight gadgets, ***which allows for increased resilience to false positives.***” (Ex. 1005 at p. 8-9, emphasis added).

186. A POSITA also would have seen that Pappas is compatible for combining with Dalcher. Both systems are directed toward identifying malware that corrupts memory in one form or another and both systems make use of call stack analyses. As explained above, Dalcher creates a framework for analyzing the relationship between stack frames. A POSITA would have

recognized that these techniques combine naturally with Pappas techniques for identifying ROP compromises using the relationship between stack frames. Moreover, Dalcher describes analyzing stack frames in memory caches, such as freed memory. As Dalcher explains:

“In one embodiment, monitoring the execution flow of the function may include monitoring the call stack frame from the freed memory in the call stack. For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame. In another example, the monitoring point may identify code involved in the handling of the test event. In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame.”
(Ex. 1004 at [0023]).

187. A POSITA would have known that the “freed memory” is a cache of recently executed stack frames. (See Section V.A., Ex. 1004 at [0022]). As Dalcher explains: “... as shown in operation 304, a call stack frame is determined from freed memory in the call stack... For example, the call stack frame may include memory in the stack that previously held an active stack entry.” (Ex. 1004 at [0021]). Moreover, as discussed in Section VI.A., a POSITA would have recognized that Dalcher is using this memory as data structures for monitoring and analysis, consistent with the definition of “cache” in the ’591 patent. (See Ex. 1001 at 6:31-57; *see also* my analysis in Section VI.A.).

188. Pappas discloses a similar use of memory as data structures for monitoring and analysis. Pappas explains “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that these “last N” branches identified call-stack information that can “...track... calls/jumps, returns, and so on.” (*Id.*). Thus, the LBR contains data about the call stack, including addresses of the functions in the

call chain from the originating function down to the API call. The ROP analysis of Pappas evaluates addresses within the LBR as described above.

189. A POSITA would have recognized the LBR data to be consistent with the definition of “cache” in the ’591 patent, as discussed in Section VI.A. (*See also* Ex. 1001 at 6:31-57). A POSITA would further have recognized that because of the similarities in content and purpose between the LBR cache of Pappas and the cache (e.g., of “freed memory”) of Dalcher in its techniques and stack-walking framework, Pappas and Dalcher could be combined in order to serve the same goal. A POSITA would have expected them to be compatible and would have expected the combination to succeed.

190. Thus, a POSITA would have expected a high chance of success in combining the two references.

B. Claim 1 is rendered obvious by Dalcher in view of Pappas

191. Each of the limitations of Claim 1 of the ’591 patent is disclosed by Dalcher in view of Pappas as described below.

1. Limitation 1[a]

192. Limitation 1[a] of the ’591 patent reads:

A computer-implemented method comprising:

193. I understand that the preamble of a patent claim is not always considered limiting. However, to the extent that the preamble of Claim 1 of the ’591 patent is considered limiting, Dalcher discloses the preamble of Claim 1.

194. Dalcher discloses “a computer-implemented method.” Dalcher discloses monitoring the execution of a (potentially harmful) program running on a computer. This monitoring is computer-implemented. As Dalcher explains:

In one embodiment, monitoring the execution flow of the function may include monitoring the call stack frame from the freed memory in the call stack. For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame. In another example, the monitoring point may identify code involved in the handling of the test event. In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame. (Ex. 1004 at [0023]).

195. A POSITA would have understood based at least on the disclosure above, “monitoring the execution flow of the **function** may include monitoring the **call stack frame** from the freed **memory** in the call stack,” that the said monitoring pertains to computers and computer processes and that Dalcher discloses a computer-implemented method. (Ex. 1004 at [0023], emphasis added).

196. Moreover, Dalcher illustrates that this process takes place in a computer, as shown in Dalcher’s Figure 2, reproduced hereinbelow.

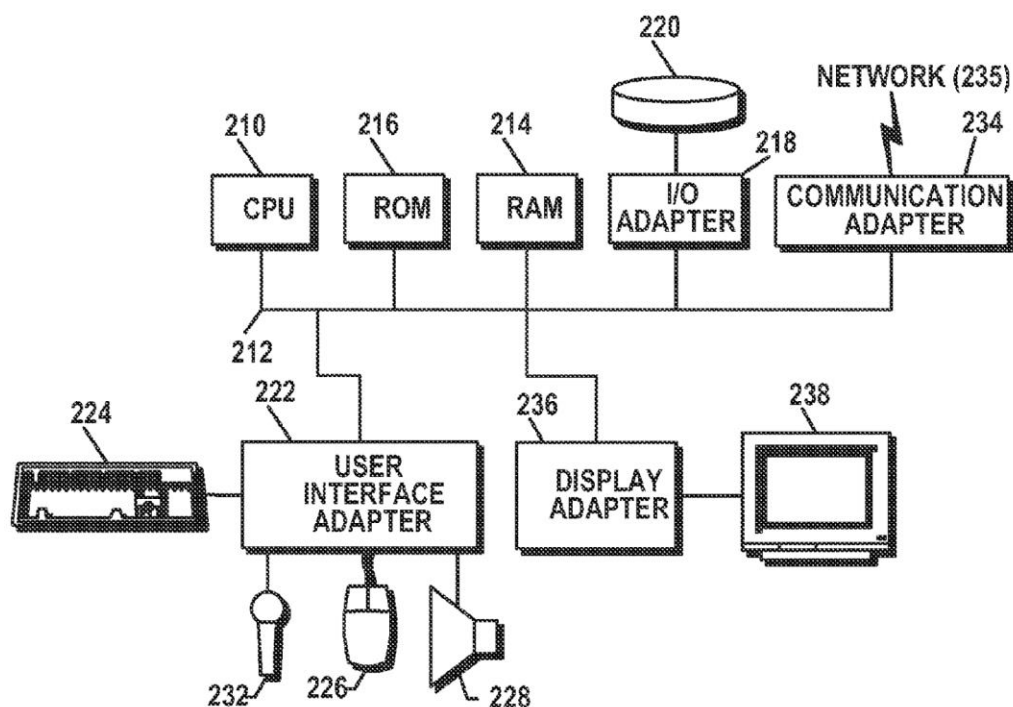


FIGURE 2

197. Dalcher explains that its disclosures/methods (e.g., including disclosures/methods about “monitoring an execution flow of a function” and stack analysis (discussed further below)) take place within the context of environments like the one described in Figure 2 (which involves computers). (Ex. 1004 at [0016]).

198. Moreover, Dalcher explains that “[a] system, **method, and computer program product** are provided for monitoring an execution flow of a function.” (Ex. 1004 at Abstract, emphasis added). Thus, a POSITA would have understood that Dalcher discloses a computer-implemented method.

199. Dalcher discloses a computer-implemented method of Claim 1 as described below for the remaining limitations. (See my analysis below for the remaining limitations of Claim 1).

200. Thus, Dalcher discloses the preamble of Claim 1.

2. Limitation 1[b]

201. Limitation 1[b] of the '591 patent reads:

monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions, wherein monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space;

202. Dalcher and Pappas each disclose limitation 1[b].

203. Dalcher discloses “monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions”. For example, Dalcher discloses monitoring a call stack, which is a memory space of a process, for monitoring execution flow of a function. A POSITA would have known that a call stack is a collection of interrelated chunks of memory representing a chain of functions. (See Section V.A.). As Dalcher explains:

In one embodiment, ***monitoring the execution flow of the function*** may include ***monitoring the call stack frame from the freed memory in the call stack***. For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame. In another example, the monitoring point may identify code involved in the handling of the test event. In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame. (Ex. 1004 at [0023], emphasis added).

204. In addition, Dalcher explains multiple examples of how the memory space of the process is involved in the monitoring. For example, Dalcher explains that a call stack and call stack frame (which, as discussed above, Dalcher’s process monitors) relate to “freed memory.” (Ex. 1004 at [0021]). A POSITA would have known that accessing data in freed memory can be associated with malware or an intrusion. (See Section V.E.). Similarly, Dalcher discusses call stack data outside the normal memory bounds expected. (Ex. 1004 at [0021]).

205. Dalcher discloses monitoring operating system operations and functions, which is a “monitored function of a plurality of functions” (e.g., the plurality of functions is the “system operations and functions” of which Dalcher monitors some. Dalcher discloses multiple mechanisms for monitoring functions: “... the monitoring may include the use of filter drivers (e.g., file system filters, etc.)... [or] code hooking. For example, the monitoring may include modifying code to be monitored so that it transfers control to a callback when executed.” (Ex. 1004 at [0031]). As I explained in Section V.D., filter drivers, code hooking, and other callback functions are mechanisms for monitoring code. With reference to hooking, Dalcher explains in detail that, “the data associated with the function may be identified by... hooking based monitoring of the call stack...” (Ex. 1004 at [0020]). Some systems provide support for hooking: “In yet another embodiment, the monitoring may include the use of central processing unit (CPU) debugging extensions. For example, a limited number of CPU-based monitors may be registered to invoke callbacks when indicated code is executed.” (Ex. 1004 at [0031]).

206. In more detail, Dalcher describes functions including “subroutine[s] of the software”. (Ex. 1004 at [0017]). Moreover, the call stack “stores the data associated with the function.” (*Id.*). The call stack also contains “one or more entries [that] are associated with an action performed by the function.” (*Id.*). Dalcher specifically identifies that, for example, “a generated keystroke test event may be expected to be handled only by a subset of OS-supplied software. If the monitoring indicates that that other software was involved in this handling, one or more actions may be taken. For example, the monitoring could flag the involvement as outside allowed system operations, possibly indicating a presence of malicious software, etc.” (Ex. 1004 at [0026]). That is, handling of certain events should only be performed by a subset of OS

software, and this handling could be monitored for use by other (e.g., unexpected or unauthorized) software, as such use might indicate malicious software.

207. Another example of monitored functions includes “callbacks.” “Further, as shown in operation 406, monitoring of callbacks is invoked.” (Ex. 1004 at [0030]). As I explained in Section V.D., a POSITA would have known that “callbacks” are a type of function that are triggered by events, such as pre-specified events. Dalcher explains that the monitoring process selects a subset of callbacks for monitoring based on a determined scope: “Further still, as shown in decision 408, it is determined whether a callback is within a scope of the monitoring.” (Ex. 1004 at [0032]).

208. Thus, as explained above, Dalcher discloses mechanisms for monitoring various functions, indicates the sorts of functions that can be monitored, and explains that a call stack (memory space of a process) is monitored for monitoring execution flow of a function. Therefore, a POSITA would have understood Dalcher’s disclosures to teach the claimed “monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions”.

209. Dalcher also discloses “wherein monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space.” For example, Dalcher explains that the above-discussed monitoring of a call stack (which, as discussed above, discloses the claimed “monitoring [a/the] memory space...”) is performed by software (e.g., filter drivers or code hooks), which discloses the claimed “a component for evaluating the at least one monitored function in the memory space.” (See Ex. 1004 at [0023]). This software is used for monitoring memory in real-time and for evaluating the monitored function within the call stack memory space. As discussed in Section V.D., filter drivers and code hooks are used to

modify the control flow of software for evaluating or monitoring a function's behavior, by temporarily redirecting execution of a function to a software component provided by the system that installed the filter driver or code hook. (For example, the technique of "hooking" works by replacing one of the earliest instructions of a function with a jump instruction to a monitoring subroutine for evaluation first. The monitoring code can determine that the function call is safe; if so, it executes the instruction that was removed and replaced, then jumps back to the correct place in the original function where execution can continue normally.) (See Section V.D.). As I cited above, Dalcher discloses "the execution flow of the function may include **monitoring the call stack frame from the freed memory in the call stack**. For example, a monitoring point (e.g., a **hook**, etc.) **may be placed on the call stack frame...**" (Ex. 1004 at [0023], emphasis added). A POSITA would have understood based on this disclosure that Dalcher discloses "monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space" (e.g., hook). (See *id.*). Dalcher explains that such a component (e.g., hook) is loaded in the call stack frame (e.g., "monitoring point (e.g., a **hook**, etc.) **may be placed on the call stack frame**"). (See *id.*, emphasis added). Dalcher further discloses, "In one embodiment, the monitoring may include operating system supported monitoring. For example, the **monitoring may include the use of filter drivers** (e.g., file system filters, etc.), event callbacks (e.g. registry monitoring callbacks, etc.), etc. In another embodiment, the **monitoring may include the use of code hooking**. For example, the monitoring may include modifying code to be monitored so that it transfers control to a callback when executed." (Ex. 1004 at [0031], emphasis added).

210. As I explained above, a call stack frame and call stack disclose the claimed "[monitored] memory space," so a POSITA would have understood that Dalcher discloses "loading a component" in the monitored memory space. (See *id.*; see also my analysis above in

this section for limitation 1[a].) Thus, to the extent this claim element is interpreted to require “loading a component . . . in the [monitored] memory space,” Dalcher also discloses this claim element under such an interpretation. (*See id.*)

211. In another example, Dalcher provides more details about how and where to place monitoring points. “For example, enabling the... monitoring may include placing one or more additional monitoring points (e.g., hooks, probes, etc.) in the call stack or the freed memory in the call stack.” (Ex. 1004 at [0033]).

212. A further example from Dalcher describes using a “test event” for tracing whether (monitored) software functions are involved with a call stack. “For example, the data may be identified within the call stack as the result of generating a test event (e.g., a trace event, etc.). For instance, a test event may be generated in order to determine one or more functions that are involved with a call stack associated with the test event...” (Ex. 1004 at [0019]).

213. As I described in detail in Section VIII.A., although Dalcher regularly uses the terms “in another embodiment,” a POSITA would have understood that Dalcher’s disclosures are different examples and applications within the same compatible framework and, as also discussed in Section VIII.A., key embodiments cited (e.g., the embodiments illustrated in Figs. 3-5) are explicitly described as being combinable. None of Dalcher’s techniques are incompatible with each other and are, in fact, complementary. Thus, a POSITA would have recognized that Dalcher’s “embodiments” are either one combined embodiment or, at the very least, different examples and applications of the same techniques and are compatible with each other. (*See my analysis in Section VIII.A.*)

214. Pappas also discloses “monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions”. Pappas discloses monitoring Windows

API calls, which are low-level user mode functions for the Windows 7 Operating System. Pappas describes the difference between system calls (which Pappas also refers to as “Native API” calls) and Windows API calls (which are available to “user-level programs”) as follows:

User-level programs interact with the underlying system mainly through system calls. Unix-like systems provide to applications wrapper functions for the available system calls (often using the same name as the system call they invoke) as part of the standard library. In contrast, Windows does not expose the system call interface directly to user-level programs. Instead, programs interact with the OS through the Windows API [13], which is organized into several DLLs according to different kinds of functionality. In turn, those DLLs call functions from the undocumented Native API [59], implemented in ntdll.dll, to invoke kernel-level services.
(Ex. 1005 at p. 5).

215. Pappas further discloses, “Fortunately, as the Native API is not exposed to user-level programs, i.e., applications never invoke Native API calls directly; we can solve this issue by ensuring that system calls are always invoked solely through their respective Windows API functions.” (Ex. 1005 at p. 6).

216. Pappas monitors these Windows API calls to look for ROP code vulnerabilities and exploits. As Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable [memory] page protections... we present a practical runtime ROP exploit prevention technique for the protection of third-party applications.” (Ex. 1005 at p. 2).

217. As I explained in Section V.A., software is executed in a call stack, which is a memory space of a process where the process keeps track of control flow (e.g., execution of a function). Thus, a POSITA would have understood Pappas to be monitoring this memory space of a process for execution of Windows API calls/functions (monitored functions) when looking for ROP exploits.

218. Pappas also discloses “wherein monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space.” To perform the monitoring, Pappas explains that the “kBouncer” module would be loaded ahead of time for monitoring the protected functions. This is another example of loading “a component” that monitors and evaluates the function within the call stack memory space.

The overall operation of the runtime system is depicted in Fig. 8. The interposition component is implemented on top of the Detours framework [38], which provides a library call interception mechanism for the Windows platform. During initialization, it requests by kBouncer’s kernel module to enable the LBR feature on the CPU. The two components communicate through control messages over a pseudo-device that is exported by the kernel module (using the DeviceIoControl API function). ***Then, it selectively hooks the set of the protected Windows API functions. Each time a protected function is called, the detour code sends a control message to the kernel component, instructing it to inspect the contents of the LBR stack for abnormal control transfers.***

(Ex. 1005 at p. 10, emphasis added).

219. A POSITA would have recognized this to teach a concept substantially similar to the following disclosure in the ’591 patent specification: “The anti-exploit component loads a component such as a custom ***module*** into the memory space of each monitored process. This component ***intercepts monitored operating system applications or APIs*** and evaluates a record (e.g., stack frame) of a call stack” (Ex. 1001 at 4:45-49, emphasis added)

220. See also, Ex. 1005 at Figure 8 (annotated), reproduced hereinbelow:

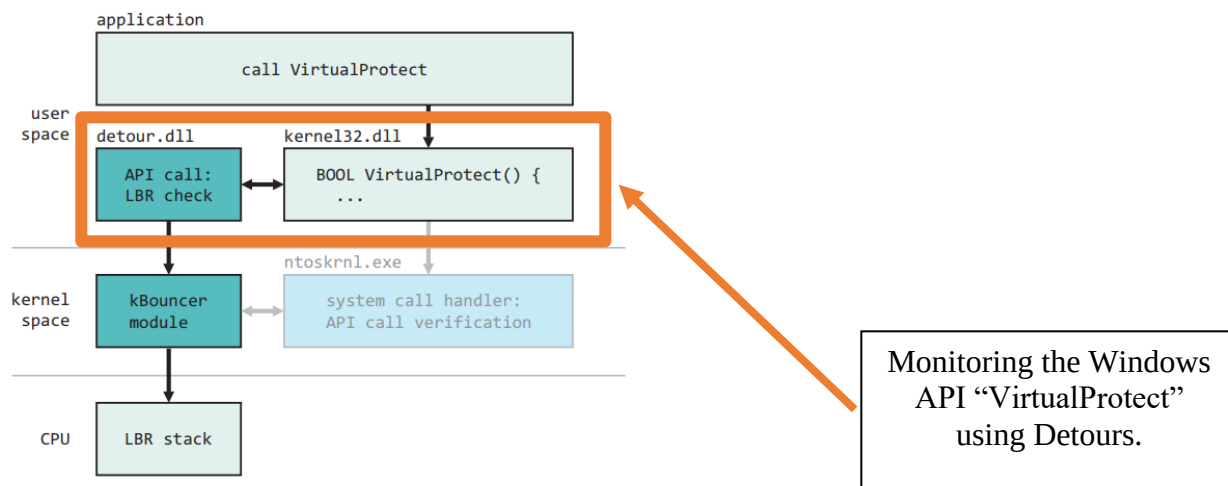


Figure 8: Overview of kBouncer’s implementation. At the entry point of Windows API functions, kBouncer detours the execution, inspects the LBR stack in kernel mode, and then returns control back to the application.

221. Thus, Pappas, in disclosing the kBouncer module which “hooks the set of protected Windows API functions” (Ex. 1005 at p. 10), “detours the execution,” and “inspects the LBR stack” (call stack relating to execution of monitored Windows API calls/functions) (*id.*; see Figure 8 hereinabove) in order to monitor and evaluate Windows API calls (monitored functions), discloses the claimed “loading a component for evaluating the at least one monitored function in the memory space”. As explained above for this limitation and in Section V.C, code hooks and detours are used to modify the control flow of software for evaluating or monitoring a function’s behavior, by temporarily redirecting execution of a function to a software component provided by the system that installed the code hook / detour.

222. A POSITA would have recognized that the disclosures of Pappas are compatible with those of Dalcher.

223. Thus, Dalcher and Pappas each disclose limitation 1[b].

3. Limitation 1[c]

224. Limitation 1[c] of the ’591 patent reads:

invoking one of the plurality of functions as a result of receiving a call from an application programming instance;

225. Dalcher in view of Pappas discloses limitation 1[c].

226. Dalcher discloses “invoking one of the plurality of functions as a result of receiving a call...” As I already explained with respect to limitation 1[b], Dalcher discloses monitoring functions that are part of “the plurality of functions”. Dalcher discloses that the monitored functions transfer control to the callback code when the monitored functions are executed. As Dalcher explains, “the monitoring may include modifying code to be monitored so that it transfers control to a callback **when executed**... [or] a limited number of CPU-based monitors may be registered to invoke callbacks **when indicated code is executed**.... [or] extensions to a virtualization system may be used to support invocation of a callback when monitored code is executed.” (Ex. 1004 at [0031], emphasis added). A POSITA would understand executing a function to mean invoking a function. Moreover, a POSITA would have understood function execution to be in response to a function call, as I explained in Section V.A. This is also clear from Dalcher’s disclosures: “the call stack may grow from top to bottom **as functions are called**, and may shrink from bottom to top as functions return.” (Ex. 1004 at [0037], emphasis added).

227. Dalcher also discloses “invoking... as a result of receiving a call from an application programming instance.” Dalcher discloses that the functions being monitored can be operating system functions. “For example, a generated keystroke test event may be expected to be handled **only by a subset of OS-supplied software**.” (Ex. 1004 at [0026], emphasis added). (As I explained above for limitation 1[b], keystroke test event relates to monitored functions.) As explained in my technical background about Operating Systems and APIs (Section V.B.), a POSITA would have known that OS-supplied functions (which include monitored functions) are

part of the operating system’s “application programming instance” or Application Programming Interface (API) for user programs. Thus, when an OS function (e.g., monitored function) is invoked, it receives the call as part of the system API. A POSITA would have understood that invoking a function (e.g., monitored function) that serves as an API means invoking a function (e.g., monitored function) as a result of an API call. ***That is, the call is to the API but what is invoked is the monitored function.*** Thus, invoking an API function (such as a monitored function) is invoking the function in response to receiving a call from an API. (See, e.g., Ex. 1011 at p. 8, which describes the inherent distinction between the Windows API and the underlying system calls).

228. However, to the extent that invoking an API function is not invoking a function in response to receiving a call from an API (which is not conceded), Dalcher still discloses this element. A POSITA would have understood that an API call typically calls other functions that are invoked by the API call. Moreover, standard library calls in almost any language, as I also explained in the background (see Section V.B.), are provided by an API and any calls triggered by it are in response to an API call. Thus, a POSITA would have understood that most, if not all, functions (including monitored functions) that can be invoked are invoked by an API. (See my analysis of software APIs, such as libc and OpenSSL, in Section V.B.).

229. Dalcher also teaches monitoring code identified as part of an interface. “In yet another embodiment, the targets of additional monitoring may be determined by the identification of a body of code associated with an observed call frame. For example, code that is ***part of an interface*** may be identified through stack analysis as having been invoked.” (Ex. 1004 at [0045], emphasis added). A POSITA would have known that a software interface is an “application programming instance.” (See Section VI.A.). Moreover, a POSITA would have known that code

that is “part of an interface” includes code and functions (e.g., monitored functions) invoked in response to the API call itself.

230. In yet another example, Dalcher discloses monitoring driver functions through “filter drivers.” A POSITA would have known that filter drivers are a mechanism in certain versions of the Windows Operating System that monitor operating system APIs. (See Section V.D.) As I explained above in Section VI.A., a POSITA would have known that an Application Programming Interface is an “application programming instance.” As Dalcher explains, “For example, the monitoring may include the use of filter drivers (e.g., file system filters, etc.), event callbacks (e.g. registry monitoring callbacks, etc.), etc.” (Ex. 1004 at [0031]). As explained in the technical background, filter drivers monitor calls to driver functions. These are monitored functions that would be invoked in response to calls to system APIs. (See Sections V.B. and V.C.).

231. Pappas also discloses this limitation. For example, as discussed above for limitation 1[b], Pappas discloses monitoring Windows API calls/functions. As I explained above for this limitation, API calls/functions are invoked as a result of receiving a call from an application programming instance / API.

232. Moreover, Pappas discloses monitoring Windows system calls which are triggered by Windows API calls.

User-level programs interact with the underlying system mainly through system calls. Unix-like systems provide to applications wrapper functions for the available system calls (often using the same name as the system call they invoke) as part of the standard library. In contrast, Windows does not expose the system call interface directly to user-level programs. ***Instead, programs interact with the OS through the Windows API*** [13], which is organized into several DLLs according to different kinds of functionality. In turn, ***those DLLs call functions from the undocumented Native API [59], implemented in ntdll.dll, to invoke kernel-level services.*** (Ex. 1005 at p. 5, emphasis added).

233. Pappas further discloses, “Fortunately, as the Native API is not exposed to user-level programs, i.e., applications never invoke Native API calls directly; we can solve this issue by ensuring that system calls are always invoked solely through their respective Windows API functions.” (Ex. 1005 at p. 6).

234. To perform the monitoring, Pappas explains that the “kBouncer” module would be loaded ahead of time for monitoring the protected functions (i.e., when the functions are invoked).

The overall operation of the runtime system is depicted in Fig. 8. The interposition component is implemented on top of the Detours framework [38], which provides a library call interception mechanism for the Windows platform. During initialization, it requests by kBouncer’s kernel module to enable the LBR feature on the CPU. The two components communicate through control messages over a pseudo-device that is exported by the kernel module (using the API function). ***Then, it selectively hooks the set of the protected Windows API functions. Each time a protected function is called, the detour code sends a control message to the kernel component, instructing it to inspect the contents of the LBR stack for abnormal control transfers.***

(Ex. 1005 at p. 10, emphasis added).

235. See also, Ex. 1005 at Figure 8, reproduced hereinbelow:

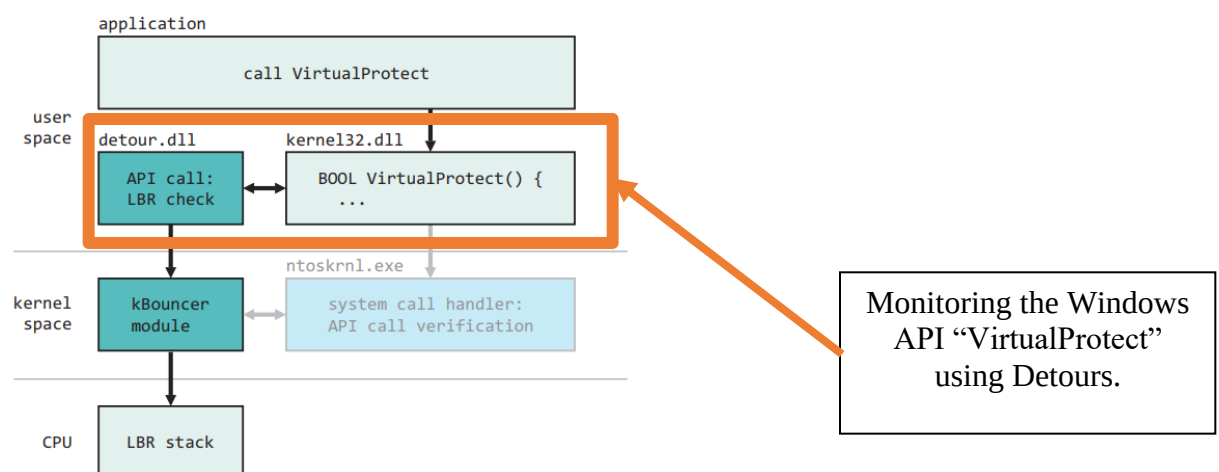


Figure 8: Overview of kBouncer’s implementation. At the entry point of Windows API functions, kBouncer detours the execution, inspects the LBR stack in kernel mode, and then returns control back to the application.

236. Pappas also explains that both the API call and the system call (the lowest level call) are monitored. As shown in Figure 3 below, an LBR check is initiated upon the API call. A follow-up check is performed at the system call boundary (marked “API Call verification” in the figure).

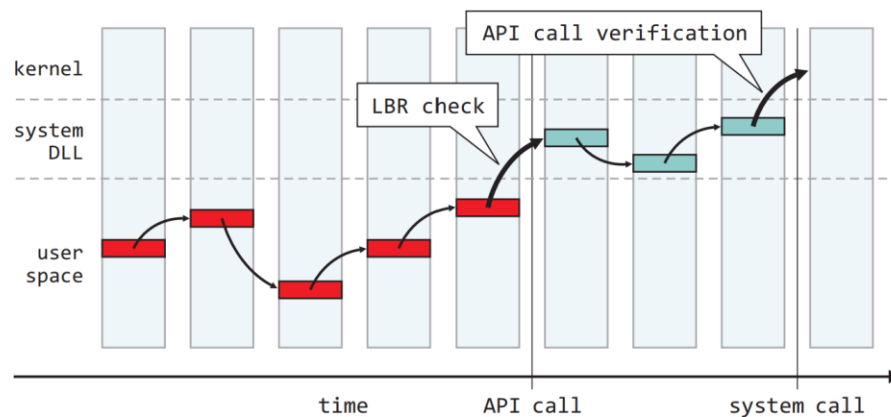


Figure 3: Overview of the detection scheme of kBouncer. Before the invocation of protected Windows API functions, the system inspects the LBR stack to identify whether the execution path that led to the call was part of ROP code, and writes a checkpoint. To account for ROP code that would bypass the check by jumping over kBouncer’s function hook, the system then verifies the entry point of the API function at the time of the corresponding system call invocation.

237. As shown in this Figure 3 of Pappas, the system call (e.g., monitored function) is invoked as a result of a call to the API call. To the extent that Dalcher does not explicitly disclose this limitation, a POSITA would have found it obvious to combine Pappas with Dalcher. Specifically, in the Windows operating system, a POSITA would have found it obvious in view of Pappas to have Dalcher monitor the system calls that are invoked as a result of the Windows API calls. As I explained above, Dalcher already discloses monitoring system operations/functions.

And as also explained above, Dalcher monitors “interface” code. A POSITA would have known that system operations and interface code include the system calls to the kernel.

238. A POSITA would have been motivated to also use the teachings of Pappas. As I explained in Section IX.A., a POSITA would have been motivated to combine Dalcher and Pappas. A POSITA would have been motivated to use the additional insights and techniques from Pappas to better defend a computer system against malicious programs.

239. Thus, Dalcher in view of Pappas discloses limitation 1[c].

4. Limitation 1[d]

240. Limitation 1[d] of the '591 patent reads:

executing stack walk processing upon the invocation of one of the plurality of functions in the monitored memory space; and

241. Dalcher discloses limitation 1[d]. Or, alternatively, Dalcher in view of Pappas discloses limitation 1[d].

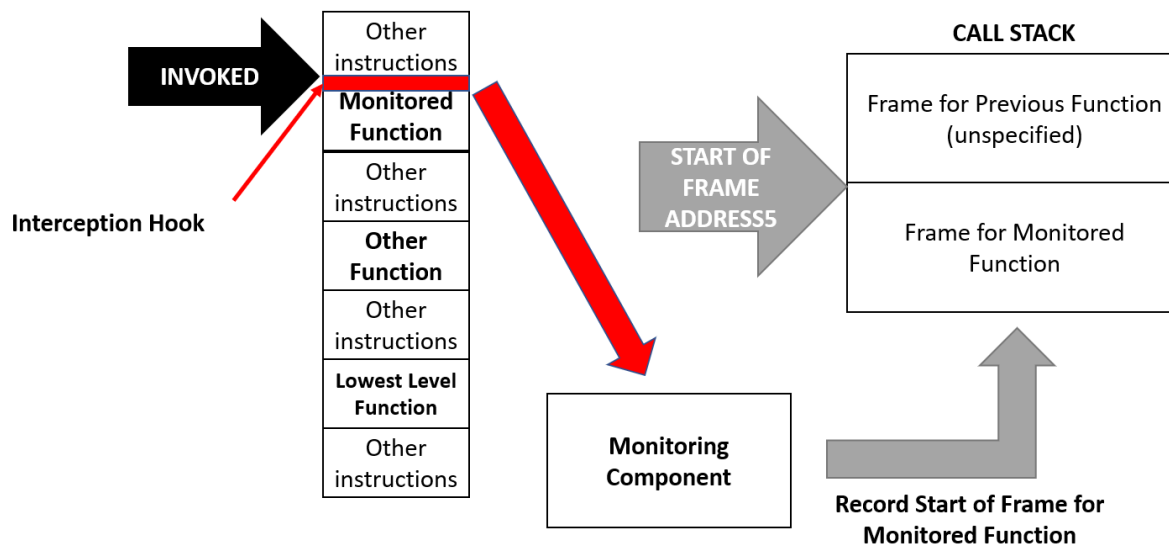
242. As I explained in my overview of the '591 patent (Section VI.A.), “stack walk processing” may include setup and configuration for an actual stack walk that begins later. The '591 patent explains that a monitored function can trigger the start of processing and mark the invoked function as the “original” function. Subsequent stack walks move backward through the stack to this marked point.

243. The specification defines “stack walk processing” as “a process that walks a stack frame for a hooked functioned [sic] from a lowest level user mode function to an address of the invoked function/originating caller function.” (Ex. 1001 at 10:35-39).⁸

⁸ I have been informed that in a parallel district court litigation, the owner of the '591 patent is arguing that the '591 patent claims should be afforded their plain and ordinary meaning. I have (footnote continued)

244. For clarity, I repeat part of my explanation of the '591 patent specification. This first figure illustrates getting stack walk processing started: collecting the address of the stack frame of the invoked/monitored function.

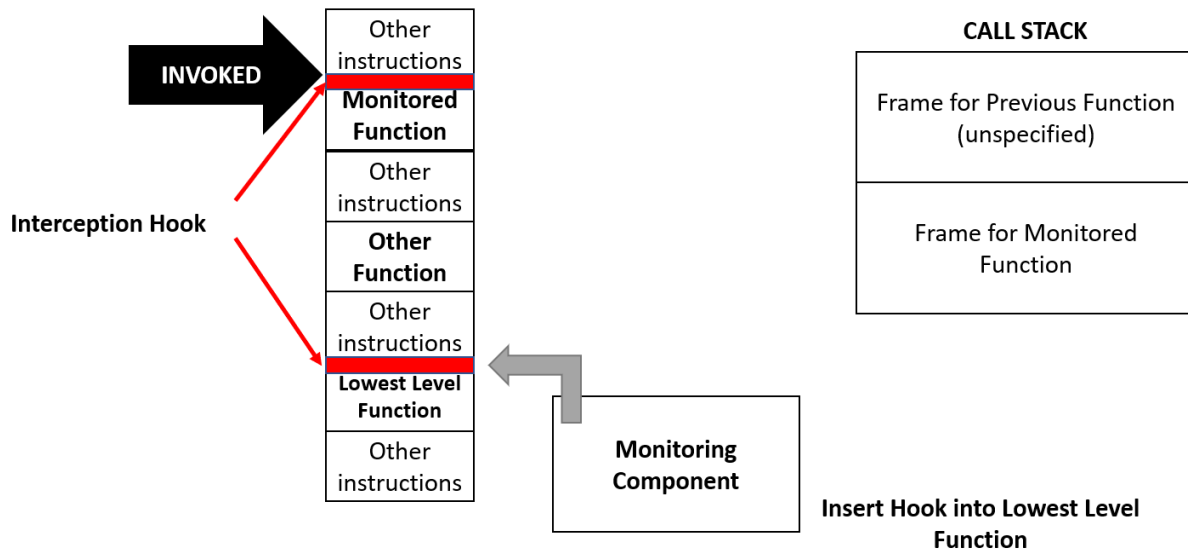
Stack Walk Processing Step 1



245. As shown in the next figure, the next step is to add a new hook to a lowest level function.

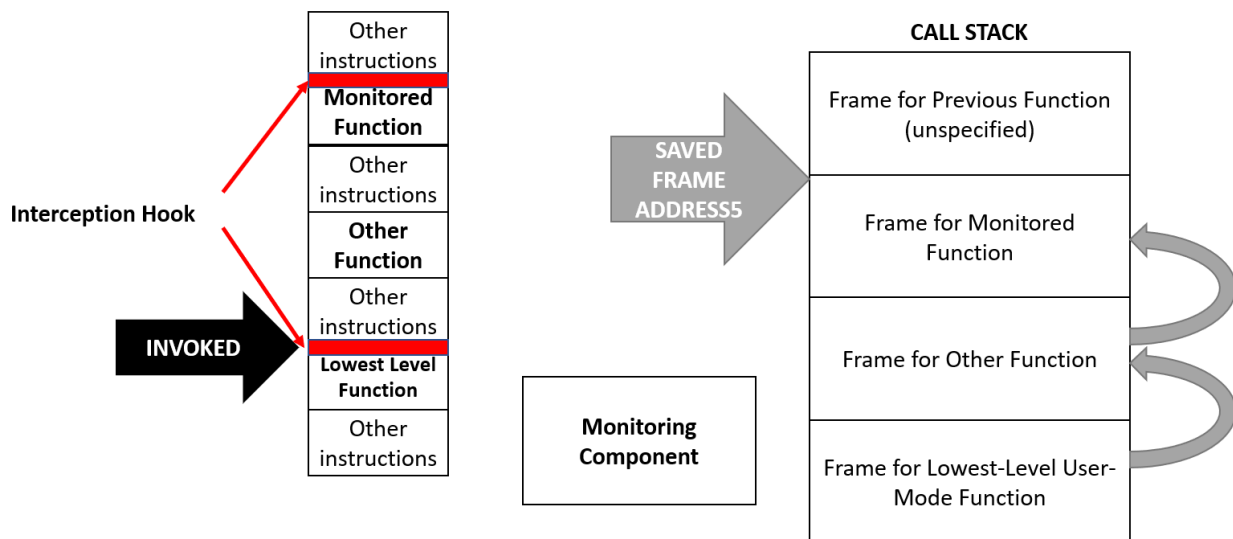
also been requested to adopt patent owner's position on claim construction for the purposes of rendering my opinions set forth in this declaration. (See Section IV). Thus, prior art can disclose "stack walk processing" under patent owner's plain and ordinary meaning even if its disclosures do not completely match this definition at 10:35-39 of the '591 patent.

Stack Walk Processing Step 2



246. Finally, when the lowest-level user mode function is invoked in a chain that started from the invoked/monitored function (with some “other” functions thrown in), the stack walk works backwards from the lowest-level user mode function until it reaches the frame for the invoked/monitored function. It recognizes this using the saved address from step 1.

Stack Walk Processing Step 3



247. Thus, a POSITA would have understood that the start of stack walk processing can include designating the invoked/monitored function as an “original” calling function (e.g., upon invocation of the invoked/monitored function). Dalcher in view of Pappas discloses or render obvious this process.

248. Dalcher discloses “executing stack walk processing upon the invocation of one of the plurality of functions in the monitored memory space.” Dalcher discloses performing a stack walk. “FIG. 5 shows a method 500 for performing extended call stack analysis... As shown in operation 502, stack analysis of a call stack is performed. In one embodiment, performing the stack analysis may include identifying one or more call frames of the call stack by performing a stack walk. For example, stack frames may be discovered by walking back in the stack when stack frame pointers are not present.” (Ex. 1004 at [0035]-[0036]).

249. Dalcher discloses executing the stack walk as part of the monitoring described in 1[b] and 1[c], which as I explained in my analysis of those limitations, includes the execution of one of many monitored functions wherein the monitoring is in the call stack memory space. As Dalcher explains, “However, if in decision 408 it is determined that the callback is within the scope of the monitoring, then in operation 410, a call stack for the operation that triggered the monitoring callback is extracted and examined. In one embodiment, examining the call stack may include performing an extended call stack analysis on the call stack.” (Ex. 1004 at [0033]). The extended call stack analysis, Dalcher explains, includes stack walking. “As shown in operation 502, stack analysis of a call stack is performed... stack frames may be discovered by walking back in the stack when stack frame pointers are not present.” (Ex. 1004 at [0036]).

250. Dalcher also discloses that it can begin stack walk processing by triggering **additional monitoring** of another, related piece of code (e.g., subsequently called functions) such

that the initially called functions and subsequently called functions are included in (e.g., and checked during) stack walk. Dalcher explains:

Additionally, as shown in operation 412, ***additional monitoring is enabled as indicated by the examination of the call stack***. For example, enabling the ***additional monitoring may include placing one or more additional monitoring points (e.g., hooks, probes, etc.) in the call stack or the freed memory in the call stack***. In one embodiment, the additional monitoring may include dynamically adjusted monitoring. In this way, a more comprehensive view of call stack frames not visible from the callbacks of the initial level of monitoring may be provided. (Ex. 1004 at [0033], emphasis added).

251. In this way, Dalcher indicates that when one monitored function is invoked, it may result in ***other*** functions being monitored, similar to the disclosures of the '591 patent. As also cited above, the goal of this additional monitoring is additional stack processing / extended call stack analysis in order to get a “comprehensive view of call stack frames not visible from... the initial level of monitoring.”

252. A POSITA would have understood that any of Dalcher's teachings about monitoring, including extended call stack analysis and stack walking, also apply to Dalcher's disclosure of “additional monitoring”. In other words, Dalcher teaches that call stack processing is triggered by a monitored function by hooking other, related functions for monitoring that will be part of extended call stack analysis and stack walking. A POSITA would have understood that Dalcher's stack walk and other stack analysis teachings are applied to these additional monitoring points in order to obtain the “more comprehensive view of call stack frames.” In other words, a POSITA would have understood that Dalcher's stack walk includes the initially called functions and the above-discussed subsequently called functions (e.g., by utilizing the above-discussed additional monitoring points).

253. A POSITA would have understood Dalcher’s “additional monitoring” to include functions called by the originally monitored function. A POSITA would have known that functions that are subsequently called from the first monitored function would be in the same call stack. (*See, e.g.*, my analysis in Section V.A.). Performing a stack walk from those functions would contribute to a “comprehensive view of call stack frames not visible from... the initial level of monitoring.” (Ex. 1004 at [0033]). But to the extent it is found that a POSITA did not have such understanding, Pappas discloses this teaching explicitly.

254. Pappas discloses an originally monitored function in the form of a Windows API call/function. When the API call/function is invoked, Pappas adds monitoring for a system call (function) that is meant to be triggered by the API. This is illustrated by Pappas in Figure 3, reproduced hereinbelow:

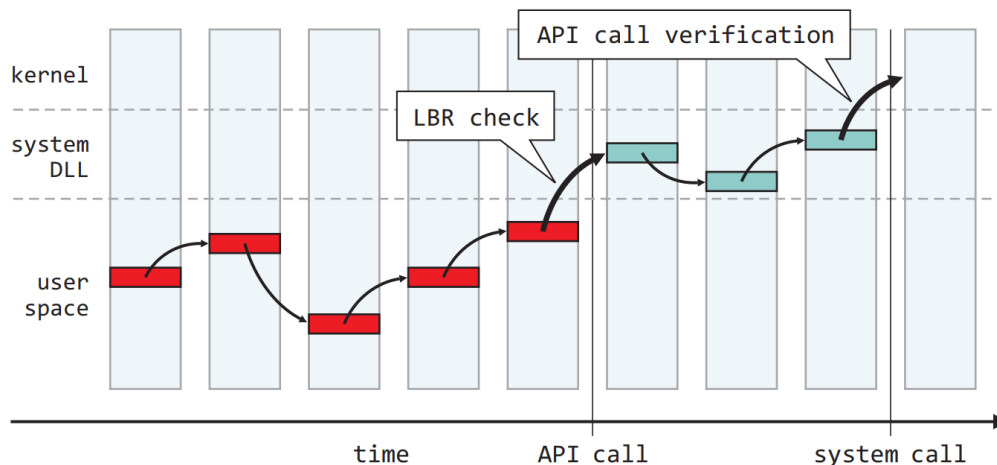


Figure 3: Overview of the detection scheme of kBouncer. Before the invocation of protected Windows API functions, the system inspects the LBR stack to identify whether the execution path that led to the call was part of ROP code, and writes a checkpoint. To account for ROP code that would bypass the check by jumping over kBouncer’s function hook, the system then verifies the entry point of the API function at the time of the corresponding system call invocation.

255. As also noted in the text of the Figure above, kBouncer hooks functions in order to know when to trigger. Pappas explains that kBouncer “...selectively hooks the set of the protected Windows API functions.” (Ex. 1005 at p. 10). The “API Call verification” in the figure above is monitoring on a system call that is triggered by the API call. Pappas explains “[a]fter a clear LBR check at an API function’s entry point, kBouncer writes a checkpoint that denotes a legitimate invocation of that particular function. When the respective system call is later invoked, the system call handler verifies that a proper checkpoint was previously set by the expected API function...” (Ex. 1005 at p. 6). As I explain in more detail below, Pappas discloses a stack walk that includes such initially called Windows API function and subsequently called system calls.

256. It would have been obvious to a POSITA in view of Pappas to extend Dalcher to place additional monitoring at system calls after monitored API calls/functions have been triggered (e.g., to include subsequently called functions in the above-discussed extended call stack analysis and stack walk). A POSITA would have understood that Pappas' teachings with respect to the monitored Windows API call/function are an "original" function (as it is called first) and that when the original function is invoked, the system call is also configured to be monitored. A POSITA would have recognized that this is similar to Dalcher disclosing that after an initial function is invoked, "additional monitoring" can be placed in order to have a "comprehensive view of call stack frames not visible from... the initial level of monitoring." A POSITA would have understood that Dalcher's techniques and Pappas' techniques are similar and compatible.

257. A POSITA would have understood that stack walks that take place from monitored calls/functions that are called after an initially invoked function would begin with the subsequent function and then walk backwards toward the initial function. As explained previously, Dalcher analyzes each "candidate frame" for potential compromises. Thus, a POSITA would have understood that each frame in the relevant call stack is analyzed starting with the frame of the subsequent call/function, then working backwards through the stack frames of every intermediate function call, until the stack frame of the initial call/function is reached. Moreover, a POSITA would have known that the stack walks and stack analysis from monitored subsequent function calls would become part of the "call stack for the [initial] operations" taught by Dalcher. This is also consistent with the '591 patent specification, as I explained in the overview. (See Section VI.A.).

258. Pappas also discloses "executing stack walk processing upon the invocation of one of the plurality of functions in the monitored memory space." Pappas discloses a call-stack tracing

(stack walk) implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame of a stack to another. (See Ex. 1005 at p. 6). To analyze gadget chaining, Pappas performs a reverse stack walk: “Gadget chaining patterns are identified as follows: starting from the most recent branch in the LBR stack, the number of consecutive targets that point to gadgets are counted. Any ret targets are looked up in the call-preceded gadgets hash table, whereas call or jmp targets are looked up in both hash tables, call-preceded or not.” (Ex. 1005 at p. 10).

259. As explained above, Pappas initiates stack walk processing when a monitored Windows API call/function is invoked and adds in monitoring on a Windows native system call (syscall). A POSITA would have been motivated to combine Dalcher and Pappas such that Dalcher’s stack walk processing involving additional monitoring points (e.g., as extended call stack analysis) is configured to monitor a subsequent call (like the syscall) and an original call (like the Windows API call/function) such that stack frames of the subsequent call/function and the original (initially invoked) call/function and the stack frames of the calls/functions in between those calls/functions are included in (and analyzed during) stack walk.

260. Summarizing Dalcher’s teachings (in view of Pappas), I reiterate the following for clarity.

261. **First**, an initial point/function is monitored. When this initial monitored function is invoked, the monitoring code triggers an analysis of at least that function’s stack frame.

262. **Second**, as part of extended stack analysis, additional code is hooked for additional monitoring. As I explained previously, a POSITA would have understood this to include subsequent calls/functions triggered by the initial function. But to the extent it is found that a POSITA did not have such understanding, Pappas discloses such an approach, and thus it was obvious in view of Pappas.

263. **Third**, the additionally hooked code is invoked and Dalcher explains that these functions also trigger monitoring code that analyzes their stack frames. A POSITA would have understood that because these functions are subsequent calls from the initial function, they are within the same call stack. Stack frames from subsequent calls are higher than the frame for the initial call in the conceptual stack, but, in terms of memory addresses (as Dalcher describes it), lower (i.e., closer to the bottom; at a lower memory address) than the frame for the initial call.

264. **Fourth**, Dalcher explains that extended stack analysis may search for additional stack frames to analyze using a stack walk, and a POSITA would have understood that a stack walk starts at the current stack frame for the current function and walks backwards. (See Section V.A.). Dalcher explains that each frame is a “candidate frame” that is analyzed to determine if it has been compromised. Because the stack frames are analyzed in reverse, the stack frames of the additional monitored code are analyzed first, then the stack frames for the functions that called them, and then the stack frames for the functions that called those, backwards until the stack frame for the initial (or “original”) function is reached.

265. A POSITA would have been motivated to also use the teachings of Pappas. As I explained in Section IX.A., a POSITA would have been motivated to combine Dalcher and Pappas. A POSITA would have been motivated to use the additional insights and techniques from Pappas to better defend a computer system against malicious programs. Moreover, as I explained above

for this limitation, Dalcher and Pappas disclose techniques and approaches relating to stack walk that are similar. Thus, a POSITA would have understood that Dalcher and Pappas' disclosures relating to stack walk are compatible, and thus would have been motivated to incorporate techniques of one into the other for implementing a robust and effective stack analysis process involving stack walk.

266. Thus, Dalcher discloses limitation 1[d]. Or, alternatively, Dalcher in view of Pappas discloses limitation 1[d].

5. Limitation 1[e]

267. Limitation 1[e] of the '591 patent reads:

performing, during the executing of the stack walk processing before an address of an originating caller function is reached, a memory check for a plurality of stack entries identified during the stack walk processing to detect suspicious behavior,

268. Dalcher discloses limitation 1[e]. Or, alternatively, Dalcher in view of Pappas discloses limitation 1[e].

269. Dalcher discloses “performing, during the executing of the stack walk processing before an address of an originating caller function is reached, a memory check for a plurality of stack entries identified during the stack walk processing to detect suspicious behavior.” As discussed with reference to limitation 1[d], Dalcher discloses stack walks associated with monitored functions, including functions that are monitored and triggered after the initial (or “original”) invoked function is triggered. As Dalcher explains,

As shown in operation 502, stack analysis of a call stack is performed. In one embodiment, performing the ***stack analysis may include identifying one or more call frames of the call stack by performing a stack walk***. For example, stack frames may be discovered by walking back in the stack when

stack frame pointers are not present... In another embodiment, performing the stack analysis may include extracting a call return address from an identified call frame. Further, in yet another embodiment, performing the stack analysis may include identifying an origin of code containing the return address.
(Ex. 1004 at [0036], emphasis added).

270. As explained previously and as disclosed by the above quote, Dalcher discloses that the stack walk identifies stack frames (stack entries) for analysis (e.g., memory check). For each of the identified frames, Dalcher discloses analyzing it for “compromises,” which a POSITA would have understood are indicative of malicious or suspicious behaviors. Dalcher states that “it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... include[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024]). Dalcher analyzes each frame (e.g., during stack walk) to determine a “confidence” in its “genuineness.” “Further, as shown in operation 506, a candidate stack frame is identified as a result of the extended call analysis... Further still, as shown in operation 508, an amount of confidence associated with the candidate stack frame is determined.” (Ex. 1004 at [0039]; see also [0041] that explains confidence is in whether or not the stack frame is “genuine”).

271. Dalcher discloses, as part of the stack walk, analyzing stack frames (memory checks for stack entries) prior to some kind of “top” frame is reached (where “top” refers to a high memory address of the stack (which is also referred to as the bottom of the stack, in terms of a conceptual stack) as explained above, which corresponds to an initially called function). Dalcher also discusses identifying a top part of a call stack (referred to as “bottom” in terms of memory address) and analyzing the stack frames that come before it. A POSITA would have known that the “stack” refers to the chain of function/subroutine calls, the top of the stack (bottom in terms of

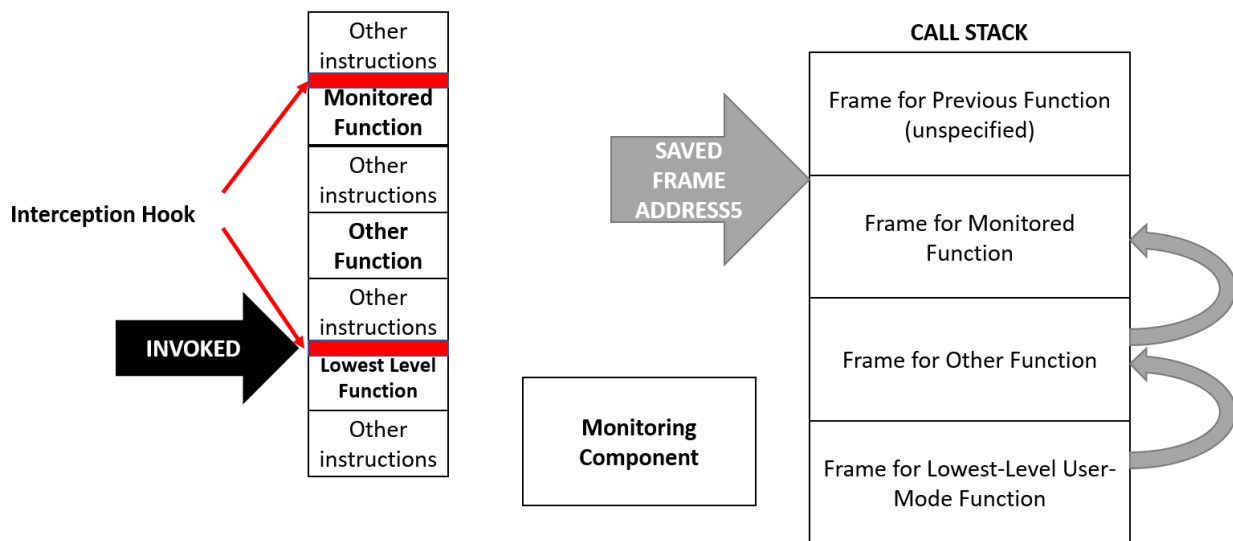
memory address) is the most recent call in the chain (e.g., subsequently called function), and the bottom of the stack (top in terms of memory address) is the call that originated the stack walk (i.e., initially invoked function / originating caller function that called subsequent functions). (See Section V.A.). However as explained in the technical background, stacks are generally described as growing “downward” in memory (from the top of memory to the bottom, or from higher memory addresses toward lower memory addresses). As explained by Dalcher, “Additionally, as shown in operation 504, extended call stack analysis is performed. In one embodiment, performing extended call stack analysis may include identifying a current top of the call stack. For example, the call stack may grow from top to bottom as functions are called, and may shrink from bottom to top as functions return.” (Ex. 1004 at [0037]). (As explained above, Dalcher here uses “top” to mean top memory address (and bottom part of a stack) and “bottom” to mean bottom memory address (and top part a stack).) Thus, Dalcher describes being able to trace/walk the stack frames (of a call stack) representing a function call chain from the most recent function call back to any previous function call, including the initial monitored function (e.g., originating called function).

272. As I explained with respect to limitation 1[d], a POSITA would have understood executing stack walk processing to include designating the **initial** invoked/monitored function as the “original” caller function (original calling function). When performing stack walk processing, stack frames are walked backwards from the lowest level user mode function back to the initial invoked/monitored function (invoked/originating caller function). The specification defines “stack walk processing” as “a process that walks a stack frame for a hooked functioned [sic] from

a lowest level user mode function to an address of the invoked function/originating caller function.” (Ex. 1001 at 10:35-39).⁹

273. Each stack frame has its own address so when walking backwards, each frame is processed until the address of the frame of the original call is reached. For clarity, I include my previous illustration for this process.

Stack Walk Processing Step 3



274. As explained with respect to limitation 1[d], a monitored function can trigger additional monitoring and is also included in the stack walk processing. Dalcher explains:

Additionally, as shown in operation 412, ***additional monitoring is enabled as indicated by the examination of the call stack.*** For example, enabling the ***additional monitoring may include placing one or more additional monitoring points (e.g., hooks, probes, etc.) in the call stack or the freed memory in the call stack.*** In one embodiment, the additional monitoring may include dynamically adjusted monitoring. ***In this way, a more***

⁹ I have been informed that in a parallel district court litigation, the owner of the '591 patent is arguing that the '591 patent claims should be afforded their plain and ordinary meaning. I have also been requested to adopt patent owner's position on claim construction for the purposes of rendering my opinions set forth in this declaration. (See Section IV). Thus, prior art can disclose "stack walk processing" under patent owner's plain and ordinary meaning even if its disclosures do not completely match this definition at 10:35-39 of the '591 patent.

comprehensive view of call stack frames not visible from the callbacks of the initial level of monitoring may be provided.
(Ex. 1004 at [0033], emphasis added).

275. As I explained with respect to limitation 1[d], Dalcher teaches that call stack processing is triggered by a monitored function by hooking other, related functions for monitoring. A POSITA would have understood that Dalcher's stack walk and other stack analysis teachings are applied to these additional monitoring points in order to obtain the "more comprehensive view of call stack frames." To the extent that a POSITA would not already have understood from the disclosures of Dalcher, a POSITA would have found it obvious, in view of Pappas (as explained for limitation 1[d]), to include subsequent calls that are invoked in response to the original call for the above-discussed monitoring (such as monitoring a system call after the monitored Windows API function is invoked) and stack walk.

276. A POSITA would have understood the initially or originally monitored function to be an "originating caller function." Thus, it would have been obvious to a POSITA to perform Dalcher's stack walk processing from a subsequently called function back to the originating caller function.

277. Dalcher also explained that each frame in the call stack could be analyzed to determine its location in the chain, to further determine if the original frame, such as the original call (the originally monitored function / originating caller function) in the code, had been reached. "Additionally, in another embodiment, it may be determined whether the suspected call return address points to code. Further, in yet another embodiment, the code pointed to by the suspected call return address may be analyzed. For example, the code may be examined for a preceding call that may explain the invocation of the candidate stack frame. For instance, the preceding call may be a call to the candidate stack frame, another stack frame, etc." (Ex. 1004 at [0039]). As I

explained in Section V.A., stacks are organized with the originating function at the bottom of the stack, but at the top of memory. Stack walking moves backwards toward the originating calling function. Thus, a POSITA would have understood that Dalcher's memory checks made during call stack walking from additionally monitored functions to the originally monitored functions are memory checks that are performed before the stack frame address of the originating caller function is reached in the stack walk.

278. Pappas also discloses “performing, during the executing of the stack walk processing before an address of an originating caller function is reached, a memory check for a plurality of stack entries identified during the stack walk processing to detect suspicious behavior.” Pappas discloses a call-stack tracing (stack walk) implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame in a stack to another. (Ex. 1005 at p. 6). To analyze gadget chaining, Pappas performs a reverse stack walk: “Gadget chaining patterns are identified as follows: starting from the most recent branch in the LBR stack, the number of consecutive targets that point to gadgets are counted. Any ret targets are looked up in the call-preceded gadgets hash table, whereas call or jmp targets are looked up in both hash tables, call-preceded or not.” (Ex. 1005 at p. 10).

279. Pappas performs this memory check before reaching an original caller (e.g., monitored Windows API function). Pappas discloses stack walk processing backwards toward the bottom of the stack (e.g., with a cap of 20 calls). (See Ex. 1005 at p. 10). A POSITA would have known that the bottom of the stack is the function call that started the stack chain and constitutes

the originating call. “The maximum gadget [call stack] length is given as a parameter—in our experiments we conservatively used a length of 20 instructions... [the] code analysis explores all possible paths from every offset within a code segment and follows recursively any conditional branches.” (Ex. 1005 at p. 10). Thus, Pappas discloses performing memory checks of stack frames/entries (during stack walk) before an address of an originating caller function (e.g., monitored Windows API function that was initially invoked) is reached.

280. Pappas explains that its stack walk walks backwards to determine if the call chain is a ROP gadget chain, which is an indication of suspicious activity/behavior. It does this by checking memory for “illegal” return statements (which are not paired with a corresponding “call” statement) or gadgets of length 8. These memory checks can trigger alerts before reaching the bottom of the stack. In other words, each stack frame (stack entry) checked before reaching the bottom stack address of the originating function call can trigger an alert if suspicious behavior is detected. “Gadget chaining patterns are identified as follows: starting from the most recent branch [frame] in the LBR stack, the number of consecutive targets that point to gadgets are counted. Any targets are looked up in the call-preceded gadgets hash table, whereas or targets are looked up in both hash tables, call-preceded or not. The most recent branch target is not considered, as it does not point to a gadget, but to the protected API function.” (Ex. 1005 at p. 10). This is similar to disclosures in the ’591 patent specification about ROP gadgets (suspicious behaviors). For example:

Some exploit scenarios describe previously relate to injecting code into an API and getting the injected code executed. In evaluating a ROP gadget, this memory check evaluates whether API code (without injecting code) is to be executed out of sequence, for example, where a ROP gadget may force API code to jump to another place of code in an order intended to exploit the API code. In a case where a ROP gadget is found inside a virtual memory management API, the address will be saved in the local cache and further evaluated using an additional memory check such as whether

execution of an originating caller function is attempted in the cache once the address for the originating caller function is reached in the stack walk processing.
(Ex. 1001 at 9:16-29).

281. For the reasons provided above, both Dalcher and Pappas disclose “performing, during the executing of the stack walk processing before an address of an originating caller function is reached, a memory check for a plurality of stack entries identified during the stack walk processing to detect suspicious behavior.” Additional explanations regarding memory checks (performed by Dalcher and Pappas) are provided below with reference to limitation 1[f], which are incorporated by reference. The memory checks described in reference to limitation 1[f] are performed before an address of the originating caller function is reached, because both Dalcher and Pappas disclose causing an alert once the memory check indicates that there is a problem (e.g., as it checks (during stack walk) a stack frame/entry in a stack (e.g., for suspicious behaviors)). A POSITA would have known that each analysis of a frame (stack entry) before reaching the original calling function represents a memory check before reaching the original calling function (originating caller function).

282. As I explained in Section IX.A., a POSITA would have been motivated to combine Dalcher and Pappas. A POSITA would have been motivated to use the additional insights and techniques from Pappas to better defend a computer system against malicious programs. In addition to these general motivations to combine, a POSITA would have been motivated to combine the “return-oriented” memory check of Pappas into Dalcher’s framework because, as Pappas explains, “[r]eturn-oriented programming (ROP) *has become the primary exploitation technique* for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2).

283. Thus, Dalcher discloses limitation 1[e]. Or, alternatively, Dalcher in view of Pappas discloses limitation 1[e].

6. Limitation 1[f]

284. Limitations 1[f] of the '591 patent reads:

wherein an alert of suspicious behavior is triggered when the performing of the memory check detects at least one of the following: code execution is attempted from non-executable memory, a base pointer is identified as being invalid, an invalid stack return address is identified, attempted execution of a return-oriented programming technique is detected, the base pointer is detected as being outside a current thread stack, and a return address is detected as being inside a virtual memory area,

285. Dalcher and Pappas each disclose limitation 1[f].

286. Dalcher discloses “wherein an alert of suspicious behavior is triggered **when the performing of the memory check detects at least one of the following:... a base pointer is identified as being invalid [or] an invalid stack return address is identified.**” As I have already explained, Dalcher discloses analyzing call stack frames (memory checks performed during stack walk) to determine if they had been overwritten. As I explained in Section V.E., a POSITA would have known that overwriting a stack frame for malicious exploitation generally involves overwriting the stack return address and the base pointer (which represents suspicious behavior). “Further, in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call

stack frame... include[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024]).

287. Dalcher also discloses analyzing the memory (memory check performed during stack walk) for other memory violations, which are another example of suspicious behavior. For example, “the call stack frame may include memory outside of that dictated by the call stack... For example, the call stack frame may include memory outside... a stack pointer associated with the call stack.” (Ex. 1004 at [0021]).

288. Dalcher further discloses “*wherein an alert of suspicious behavior is triggered* when the performing of the memory check detects at least one of the following: ... a base pointer is identified as being invalid, [or] an invalid stack return address is identified” (e.g., the above-discussed suspicious behaviors / conditions that are detected through memory checks performed during stack walk), for the reasons discussed below and those discussed for limitation 1[g]. I incorporate my analysis for limitation 1[g] by reference.

289. For example, Dalcher discloses blocking the execution of malware by a quarantine process, which is triggered by the above-discussed monitoring process.

Further, *the monitoring may allow for determining if a handling of events is performed by expected and trusted software*. For example, *a generated keystroke test event* may be expected to be handled only by a subset of OS-supplied software. If the monitoring indicates that that other software was involved in this handling, one or more actions may be taken. For example, *the monitoring could flag the involvement as outside allowed system operations, possibly indicating a presence of malicious software, etc. ... In yet another example, the system may be placed in a quarantine area* until the system is confirmed cleaned of malicious software. (Ex. 1004 at [0026],) emphasis added.

290. A POSITA would have understood that the monitoring process (referenced in the quote above) encompasses the above-discussed memory checks performed during stack walk (monitoring and analysis of a call stack, including extended call stack analysis), and thus would

have understood that the detection of above-discussed suspicious behaviors (detected through the memory checks) results in the quarantine process. (See, e.g., the quotes above and below). (Also, for example, as discussed above for limitations 1[b] and 1[c], keystroke test event (referenced in the quote above) relates to monitored functions.) Thus, from the quote above and those below (and for the reasons discussed above), a POSITA would have understood that the monitoring discussed in the quote above relates to the monitoring and analysis of a call stack and memory checks performed during stack walk.

In one embodiment, ***monitoring the execution flow of the function*** may include ***monitoring the call stack frame from the freed memory in the call stack. For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame.*** In another example, the monitoring point may identify code involved in the handling of the test event. In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. ***For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame.*** (Ex. 1004 at [0023], emphasis added).

In one embodiment, ***examining the call stack may include performing an extended call stack analysis on the call stack.***” (Ex. 1004 at [0033], emphasis added).

FIG. 5 shows a method 500 for performing ***extended call stack analysis***... As shown in operation 502, ***stack analysis*** of a call stack is performed. In one embodiment, ***performing the stack analysis*** may include identifying one or more call frames of the call stack ***by performing a stack walk***. For example, stack frames may be discovered by walking back in the stack when stack frame pointers are not present.” (Ex. 1004 at [0035]-[0036], emphasis added).

291. A POSITA would have recognized that putting a system in quarantine is or involves triggering an alert of suspicious behavior that prevents execution of a payload for the invoked function from operating and/or triggering an alert of suspicious behavior for the system so that the system can initiate the quarantine process. And, the system placed in quarantine blocks execution

of many (or all) system operations, preventing the execution of the payload (including that of the invoked function).

292. Thus, Dalcher discloses “wherein an alert of suspicious behavior is triggered when the performing of the memory check detects at least one of the following:... a base pointer is identified as being invalid [or] an invalid stack return address is identified.”

293. Pappas also discloses “wherein an alert of suspicious behavior is triggered ***when the performing of the memory check detects at least one of the following:... attempted execution of a return-oriented programming technique is detected***”. Pappas expands on overwritten stack return addresses and base pointers in disclosing detecting attempted execution of return-oriented programming (ROP). (As discussed above for limitation 1[e], Pappas performs memory checks during stack walk to detect suspicious behaviors such as attempted execution of ROP.) “In this paper we present a practical runtime ROP exploit prevention technique for the protection of third-party applications. Our approach is based on the ***detection of abnormal control transfers that take place during ROP code execution***.” (Ex. 1005 at p. 2, emphasis added). Pappas explains that the execution of ROP resulted in detectable abnormal instructions. “Our approach is based on monitoring the executed indirect branches at critical points during the lifetime of a process, and identifying abnormal control flow transfers that are inherently exhibited during the execution of ROP code.” (Ex. 1005 at p. 3).

294. Moreover, Pappas discloses analyzing the code at runtime (e.g., through memory checks performed during stack walk, as discussed above for limitation 1[e]) in order to catch the attempted execution of ROP code. Pappas’s “proposed approach uses runtime process monitoring to block the execution of code that exhibits return oriented behavior.” (Ex. 1005 at p. 3). Pappas also explains that “[t]he execution of ROP code is identified by looking for two prominent

attributes of its runtime behavior: i) illegal instructions that target locations not preceded by call sites, and ii) sequences of relatively short code fragments “chained” through any kind of indirect branches.” (Ex. 1005 at p. 6). With respect to the first attribute, Pappas goes on to say that “A ret instruction that transfers control to an instruction not preceded by a call is considered illegal, and the observation of an illegal ret is flagged by kBouncer as an indication of ROP code execution.” (Ex. 1005 at p. 7). Pappas describes the second attribute as being based on that “the observation of several short instruction sequences chained through indirect branches. This is a generic constraint that holds for both return-oriented and jump-oriented code (or potential combinations—in the rest of this section we refer to both techniques as ROP).” (Ex. 1005 at p. 8).

295.

296. Pappas further discloses “***wherein an alert of suspicious behavior is triggered*** when the performing of the memory check detects at least one of the following:... attempted execution of a return-oriented programming technique is detected.” For example, Pappas discloses both a user-visible alert and halting the execution of ROP-compromised code (e.g., when it is detected through memory checks performed during stack walk). “In case an attack attempt is detected after the analysis of the recorded branches, the process is terminated and the user is informed with an alert message.” (Ex. 1005 at p. 10). (Figure 9 showing the alert from Pappas is reproduced hereinbelow.) “[A]nalysis of the recorded branches” refers to Pappas’ stack trace (stack walk) and memory checks discussed above, which (as discussed above for limitations 1[e] and 1[f]) detect suspicious behaviors such as an attempted execution of a ROP technique. In addition, because the alert is triggered as a result of “an attack attempt” or “malicious activity” / “zero-day exploit” being detected (see quote above and figure below), a POSITA would have understood that the “an alert of suspicious behavior is triggered when the performing of the

memory check detects” certain suspicious behaviors (including attempted execution of a ROP technique).

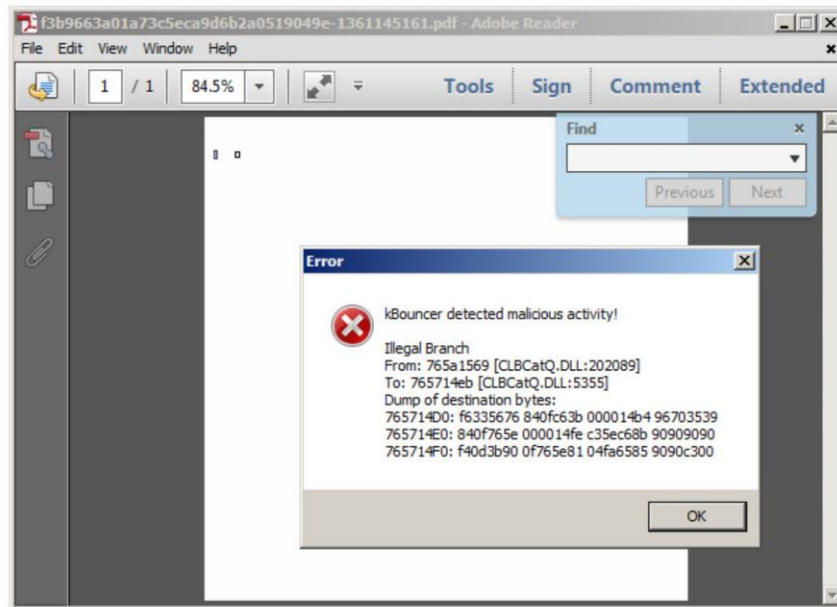


Figure 9: A screen capture of kBouncer in action, blocking a zero-day exploit against Adobe Reader XI [19].

297. Thus, Pappas discloses “wherein an alert of suspicious behavior is triggered when the performing of the memory check detects at least one of the following: ... attempted execution of a return-oriented programming technique is detected.”

298.

299. A POSITA would have recognized that the disclosures of Pappas are compatible with those of Dalcher.

300. Thus, Dalcher and Pappas each disclose limitation 1[f].

7. Limitation 1[g]

301. Limitation 1[g] of the '591 patent reads:

wherein when an alert of suspicious behavior is triggered, preventing execution of a payload for the invoked function from operating.

302. Dalcher and Pappas each disclose limitation 1[g].

303. Dalcher discloses “wherein when an alert of suspicious behavior is triggered, preventing execution of a payload for the invoked function from operating,” for the reasons disclosed above for limitation 1[f] (which are incorporated by reference). For example, Dalcher discloses blocking the execution of malware by a quarantine process.

Further, the monitoring may allow for determining if a handling of events is performed by expected and trusted software. For example, a generated keystroke test event may be expected to be handled only by a subset of OS-supplied software. If the monitoring indicates that that other software was involved in this handling, one or more actions may be taken. For example, the monitoring could flag the involvement as outside allowed system operations, possibly indicating a presence of malicious software, etc. ... In yet another example, the system may be placed in a quarantine area until the system is confirmed cleaned of malicious software. (Ex. 1004 at [0026]).

304. A POSITA would have recognized that putting a system in quarantine is or involves triggering an alert of suspicious behavior that prevents execution of a payload for the invoked function from operating and/or triggering an alert of suspicious behavior for the system so that the system can initiate the quarantine process. And, the system placed in quarantine blocks execution of many (or all) system operations preventing the execution of the payload (including that of the invoked function).

305. Pappas also discloses “wherein when an alert of suspicious behavior is triggered, preventing execution of a payload for the invoked function from operating.” For example, Pappas discloses both a user-visible alert and halting the execution of ROP-compromised code (e.g., when it is detected through memory checks performed during stack walk). “In case an attack attempt is detected after the analysis of the recorded branches, the process is terminated and the user is

informed with an alert message.” (Ex. 1005 at p. 10). (Figure 9 showing the alert from Pappas is reproduced hereinbelow.) “[A]nalysis of the recorded branches” refers to Pappas’ stack trace (stack walk) and memory checks discussed above, which (as discussed above for limitations 1[e] and 1[f]) detect suspicious behaviors such as an attempted execution of a ROP technique. In addition, because the alert is triggered as a result of “an attack attempt” or “malicious activity” / “zero-day exploit” being detected (see quote above and figure below), a POSITA would have understood that the “an alert of suspicious behavior is triggered when the performing of the memory check detects” certain suspicious behaviors (including attempted execution of a ROP technique).

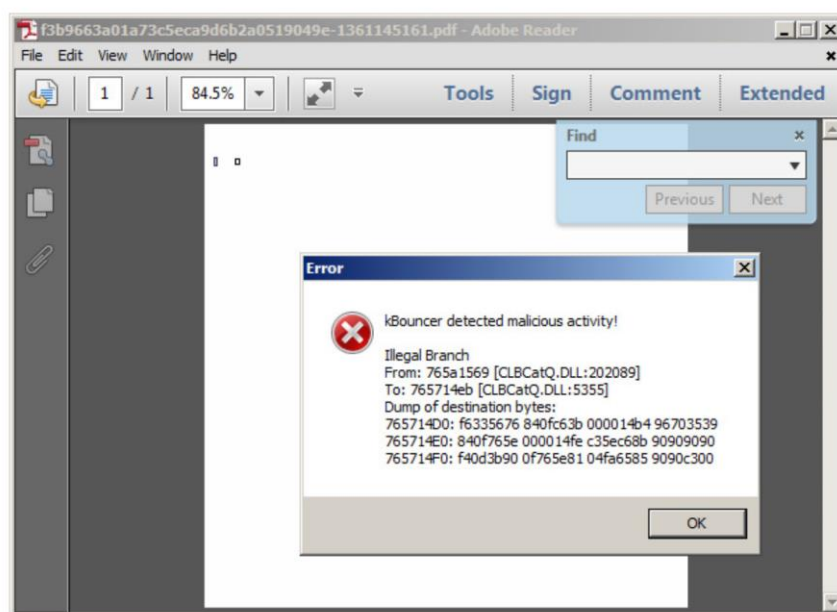


Figure 9: A screen capture of kBouncer in action, blocking a zero-day exploit against Adobe Reader XI [19].

306. Moreover, based on the above disclosure, “[i]n case an **attack attempt** is detected after the analysis of the recorded branches, the **process is terminated** and the user is informed with an alert message” (Ex. 1005 at p. 455, emphasis added) and “kBouncer in action, **blocking a zero-**

day exploit” (description of the figure above showing an alert of suspicious behavior, emphasis added), a POSITA would have understood that Pappas discloses “when an alert of suspicious behavior is triggered, preventing execution of a payload for the invoked function from operating.” (See also my analysis for limitation 1[f]).

307. A POSITA would have recognized that the disclosures of Pappas are compatible with those of Dalcher.

308. Thus, Dalcher and Pappas each disclose limitation 1[g].

309. Thus, Dalcher in view of Pappas discloses Claim 1.

C. Claim 2 is rendered obvious by Dalcher in view of Pappas

310. Each limitation of Claim 2 of the '591 patent, which depends from Claim 1, is disclosed by Dalcher or Dalcher in view of Pappas as described below.

1. Limitation 2[a]

311. Limitation 2[a] reads:

The method according to claim 1, further comprising, identifying an address associated with the invoked function; and

312. Dalcher in view of Pappas discloses limitation 2[a].

313. Dalcher in view of Pappas discloses this limitation for the reasons discussed above for Claim 1 (including but not limited to limitation 1[e]). For one example, Dalcher and Pappas' stack walk can identify addresses associated with functions associated with a call stack, including an address of the invoked function. My analyses for Claim 1 are incorporated by reference.

314. Moreover, Dalcher discloses placing monitoring points into call stacks of monitored functions. Analyzing a call stack is performed on an invoked function because only invoked functions have call stacks. Dalcher discusses including hooks or probes. As Dalcher explains:

In one embodiment, monitoring the execution flow of the function may include monitoring the call stack frame from the freed memory in the call stack. For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame. In another example, the monitoring point may identify code involved in the handling of the test event. In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame. (Ex. 1004 at [0023]).

315. A POSITA would have known that the “call stack frame” includes a series of addresses associated with a function call. Placing a monitoring point identifies an address of the function call. Thus, Dalcher discloses “identifying an address associated with the invoked function.”

316. Pappas also discloses “identifying an address associated with the invoked function.” Pappas places monitoring “hooks” into low-level user mode functions called Windows API calls which are analyzed when invoked. The hooks identify an API’s “entry point” for ensuring enforcement of the protection system wherein an entry point is an address of the API in memory. “After a clear LBR check at an API function’s entry point, kBouncer writes a checkpoint that denotes a legitimate invocation of that particular function. When the respective system call is later invoked, the system call handler verifies that a proper checkpoint was previously set by the expected API function...” (Ex. 1005 at p. 6). A POSITA would have known that the “entry point” refers to the starting address for the function call.

317. Although Dalcher discloses this limitation, a POSITA would have been motivated to combine Dalcher with Pappas. For example, Dalcher discusses monitoring (e.g., through hooking) for compromise: “Further, in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call

stack entry in the call stack frame... include[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024]).

318. A POSITA would have recognized that Pappas relies on a call-stack tracing (stack walk) implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6).

319. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher to better detect the addresses associated with the call stack, which include the invoked function. Dalcher teaches using hooks; a POSITA would have found it obvious to use these hooks to monitor specific execution paths in the manner of Pappas.

320. As stated previously, a POSITA would have been motivated to make this combination in order to achieve Dalcher’s goals of identifying “compromises.” The motivation is especially strong given that, as Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2).

321. Moreover, a POSITA would have been motivated to use Pappas for its low false positive rate, which was borne out of experimentation. “This is a generic constraint that holds for both return-oriented and jump-oriented code (or potential combinations—in the rest of this section we refer to both techniques as ROP)... For the current set of protected functions we use a threshold

of eight gadgets, *which allows for increased resilience to false positives.*” (Ex. 1005 at p. 8-9, emphasis added).

322. Thus, Dalcher in view of Pappas discloses limitation 2[a].

2. Limitation 2[b]

323. Limitation 2[b] reads:

*when the address of the invoked function address is found in a cache,
preventing execution of a payload for the invoked function.*

324. Dalcher in view of Pappas discloses limitation 2[b].

325. As I explained with reference to Claim 1 and limitation 2[a], Dalcher discloses analyzing call stacks that were found in freed memory. As explained below, a POSITA would have understood freed memory to be a type of cache, at least within the context of the '591 patent. In Section VI.A., I explained that the '591 patent describes a cache as a memory where the anti-exploit component keeps track of data about memory of the monitored process. And, as also mentioned in Section VI.A., in the provisional patent application of the '591 patent, a cache is described this way:

If the address is found in a cache entry, then the call is suspicious. Every entry present in the cache structure represents a part of memory that has been recently allocated with the flag EXECUTE, thus a potential piece of memory that could host malicious code.

(Ex. 1007 at p. 15).

326. A POSITA would have understood that freed memory stores information (e.g., stack frames with return addresses) used by Dalcher to look for exploits. The freed memory is also a space that should not be used and addresses in this cache are potentially memory hosting malicious code. (See Section V.A.).

327. Thus, when Dalcher analyzes the call stack in freed memory it is able to detect addresses of invoked functions found in the freed memory cache. As Dalcher explains:

In one embodiment, monitoring the execution flow of the function may include monitoring the call stack frame from the freed memory in the call stack. For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame. In another example, the monitoring point may identify code involved in the handling of the test event. In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame.
(Ex. 1004 at [0023]).

328. A POSITA would have known that the “freed memory” is a cache of recently executed stack frames. (See Section V.A., Ex. 1004 at [0022]). As Dalcher explains: “... as shown in operation 304, a call stack frame is determined from freed memory in the call stack... For example, the call stack frame may include memory in the stack that previously held an active stack entry.” (Ex. 1004 at [0021]).

329. To the extent that Dalcher does not disclose a cache (which is not conceded), Pappas explicitly discloses a cache: “Last Branch Recording” (LBR). As Pappas explains, “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that these “last N” branches identified call-stack information that can “...track... calls/jumps, returns, and so on.” (Ex. 1005 at p. 4). A POSITA would have known that calls and returns are the entrances and exits of call frames in a call stack. (See Section V.A.). Thus, the LBR is a cache of the call stack, and addresses from the call stack within the LBR are addresses within a cache.

330. Pappas also discloses setting “checkpoints”, which are data about Windows API calls, and storing them in memory:

Before the invocation of protected Windows API functions, the system inspects the LBR stack to identify whether the execution path that led to the call was part of ROP code, and writes a checkpoint. To account for ROP code that would bypass the check by jumping over kBouncer's function hook, the system then verifies the entry point of the API function at the time of the corresponding system call invocation. (Ex. 1005 at p. 6).

The code for setting a checkpoint can only run with kernel privileges, and the checkpoint itself is stored in kernel space so that i) the system call handler can later access it, and ii) any user-level code (and consequently the ROP code itself) cannot tamper with it. (*Id.*)

331. Pappas further discloses creating hash tables with gadgets (identified by address). Pappas identifies ROP gadgets by looking up addresses in these tables. When call stacks in the LBR are found to reference certain abnormal conditions (such as ROP gadgets), the process is terminated, which prevents the execution of the ROP payload for the invoked function. (*See also* my analyses for Claim 1.) Pappas creates the hash tables during an analysis phase: "The output of the analysis phase is two hash tables: one containing the [address] offsets of call-preceded gadgets, and another containing the rest of the found gadgets." (Ex. 1005 at p. 10). During execution, call return addresses, such as those identified by the assembly command "ret" (short for return), are checked against the tables. "Any ret targets are looked up in the call-preceded gadgets hash table, whereas call or jmp targets are looked up in both hash tables, call-preceded or not. The most recent branch target is not considered, as it does not point to a gadget, but to the protected API function." (Ex. 1005 at p. 10).

332. Finally, Pappas discloses blocking the execution of a process if the addresses were found in the LBR and hash table caches. "In case an attack attempt is detected after the analysis of the recorded branches [in the LBR and hash tables], the process is terminated and the user is informed with an alert message." (Ex. 1005 at p. 10).

333. A POSITA would have understood that Pappas' disclosures of data in memory used for analysis of function calls and stacks are consistent with the definition of "cache" in the '591 patent specification. (See Ex. 1001 at 6:31-57; *see also* my analyses in Section VI.A. and IX.A.).

334. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher describes providing a "call stack analysis" that traces one call to another. As I explained previously, Dalcher describes "in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... includ[ing] examining code associated with the call stack frame..." (Ex. 1004 at [0024]).

335. A POSITA would have recognized that Pappas relies on a call-stack tracing implementation called "Last Branch Recording" (LBR): "Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches." (Ex. 1005 at p. 4). Pappas explains that branches include the commands "call" and "ret", which are used to branch from one frame to another. (Ex. 1005 at p. 6).

336. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher, including the caching described in this section. As described above, Dalcher already describes analyzing data stored in memory; it would have been obvious to a POSITA to analyze call stack execution instructions cached in the Last Branch Recording set as suggested by Pappas.

337. As stated previously, a POSITA would have been motivated to make this combination in order to achieve Dalcher's goals of identifying "compromises." The motivation is

especially strong given that, as Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2).

338. Moreover, a POSITA would have been motivated to use Pappas for its low false positive rate, which was borne out of experimentation. “This is a generic constraint that holds for both return-oriented and jump-oriented code (or potential combinations—in the rest of this section we refer to both techniques as ROP)... For the current set of protected functions we use a threshold of eight gadgets, ***which allows for increased resilience to false positives.***” (Ex. 1005 at p. 8-9, emphasis added).

339. I also pointed out in Section IX.A., that the caches described by Dalcher (e.g., “freed memory”) and Pappas (e.g., LBR data, “checkpoints”, hash tables) are compatible with each other.

340. Also, as discussed above for Claim 1, Dalcher in view of Pappas discloses stack analysis such as performing memory checks during stack walk to detect suspicious behaviors and preventing execution of a payload for the invoked function when certain suspicious behaviors are detected (and an alert of suspicious behavior is triggered). (*See my analyses for Claim 1*). As discussed above (for this claim), a POSITA would have known that cache or freed memory is a space that should not be used and addresses in this cache are potentially memory hosting malicious code (suspicious behaviors). (*See also Section V.A.*). Also, as discussed above (for this claim), Dalcher and Pappas disclose checking cache as part of stack analysis (e.g., to detect malicious/suspicious behavior and compromise). Thus, a POSITA would have been motivated to implement the Dalcher-Pappas method discussed in Claim 1 (including its stack analysis (e.g., memory checks and stack walk)) to include checks to see if addresses associated with the call stack

(that is, walked by the stack walk), including the address of the invoked function, are contained in the above-discussed cache of Dalcher and/or Pappas (e.g., as an additional measure to detect a suspicious behavior or compromise), and if so, prevent execution of a payload for the invoked function in a manner discussed above for claim limitations 1[f] and 1[g], in order to further enhance security of the system.

341. Thus, Dalcher in view of Pappas discloses limitation 2[b].

D. Claim 3 is rendered obvious by Dalcher in view of Pappas

342. Claim 3 of the '591 patent is rendered obvious by Dalcher in view of Pappas. This claim, which depends from Claim 1, reads:

The method according to claim 1, wherein executing of the stack walk processing further comprises walking a stack frame associated with the invoked function from a low level user mode function to the originating address of the invoked function.

343. I have already explained with respect to limitations 1[d] and 1[e] that Dalcher discloses a stack walk that processes stack data up chains from one function call to previous (invoking) function calls (“executing of the stack walk processing [] comprises walking a stack frame associated with the invoked function from a . . . function to the originating address of the invoked function”). I incorporate my analysis of 1[d] and 1[e] into this section by reference.

344. Pappas further discloses that this chain could include a Windows API, which is a low-level user mode function for the Windows 7 Operating System.

User-level programs interact with the underlying system mainly through system calls. Unix-like systems provide to applications wrapper functions for the available system calls (often using the same name as the system call they invoke) as part of the standard library. In contrast, Windows does not expose the system call interface directly to user-level programs. Instead, programs interact with the OS through the Windows API [13], which is organized into several DLLs according to different kinds of functionality. In turn, those DLLs call functions from the undocumented Native API [59], implemented in ntdll.dll, to invoke kernel-level services.

(Ex. 1005 at p. 5).

345. Pappas further discloses, “Fortunately, as the Native API is not exposed to user-level programs, i.e., applications never invoke Native API calls directly; we can solve this issue by ensuring that system calls are always invoked solely through their respective Windows API functions.” (Ex. 1005 at p. 6).

346. A POSITA would have understood that Dalcher in view of Pappas would perform the stack walk from the low-level function to the address of the originating function, or a POSITA would have been motivated to configure the Dalcher-Pappas system in such a manner, because a POSITA would have known that Windows API functions are important functions to monitor and protect (e.g., from exploits), because Windows API functions are used to access important system calls / OS (which should be protected from exploits) (and thus a POSITA would have known that malicious code may try to compromise Windows API functions in order to gain unauthorized access to systems calls and OS). (*See id.* (discussing “ensuring that system calls are always invoked solely through their respective Windows API functions”); *see also* Ex. 1005 at p. 5 (“Windows does not expose the system call interface directly to user-level programs. Instead, programs interact with the OS through the Windows API”)).

347. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher describes providing a “call stack analysis” that traces one call to another. As I explained previously, Dalcher describes “in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... includ[ing] examining code associated with the call stack frame...” (Ex.

1004 at [0024]). Thus, Dalcher provides a framework for identifying the chain of frames in a call stack.

348. A POSITA would have recognized that Pappas relies on a call-stack tracing (stack walk) implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6).

349. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher’s framework. As stated previously, a POSITA would have been motivated to make this combination in order to achieve Dalcher’s goals of identifying “compromises.” The motivation is especially strong given that, as Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2).

350. Moreover, a POSITA would have been motivated to use Pappas for its low false positive rate, which was borne out of experimentation. “This is a generic constraint that holds for both return-oriented and jump-oriented code (or potential combinations—in the rest of this section we refer to both techniques as ROP)... For the current set of protected functions we use a threshold of eight gadgets, ***which allows for increased resilience to false positives.***” (Ex. 1005 at p. 8-9, emphasis added).

351. Thus, Claim 3 of the ’591 patent is rendered obvious by Dalcher in view of Pappas.

E. Claim 4 is rendered obvious by Dalcher in view of Pappas

352. Claim 4 of the '591 patent is rendered obvious by Dalcher in view of Pappas. This claim, which depends from Claim 1, reads:

The method according to claim 1, wherein the monitoring further comprises monitoring functions associated with memory management in the memory space by a user mode function.

353. I have already explained with reference to multiple limitations of Claim 1 that Dalcher in view of Pappas discloses the claimed “the monitoring,” including monitoring low-level functions and the call stack from those functions back to their originating functions. Such a call stack discloses the claimed “memory space”. I incorporate by reference my analyses for Claim 1.

354. Pappas discloses monitoring sensitive Windows API functions, which are low-level user mode functions. (See my analysis of limitation 1[b]). “Our selection of sensitive Windows API functions was made empirically based on a large set of different shellcode and ROP payload implementations [5, 3, 56, 12, 27, 60]. A list of the 52 currently protected functions is provided in the appendix.” (Ex. 1005 at p. 13). These 52 functions include functions such as “CreateProcessA”, “VirtualAlloc”, and “LoadLibraryA.” A POSITA would have known that these functions create processes in memory, allocate virtual memory, and load libraries into memory or are otherwise functions for memory management in the memory space for user mode functions. Thus, Pappas teaches the claimed “wherein the monitoring further comprises monitoring functions associated with memory management in the memory space by a user mode function.”

355. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher discusses monitoring operating-system functions. “For example, a generated keystroke test event may be expected to be handled only by a subset of OS-supplied software.” (Ex. 1004 at [0026]).

It would have been obvious to a POSITA to monitor other sensitive functions, such as those disclosed by Pappas.

356. Moreover, Dalcher describes providing a “call stack analysis” that traces one call to another. As I explained previously, Dalcher describes, “in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... includ[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024]). Thus, Dalcher provides a framework for identifying the chain of frames in a call stack.

357. A POSITA would have recognized that Pappas relies on a call-stack tracing (stack walk) implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6).

358. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher’s framework. As stated previously, a POSITA would have been motivated to make this combination in order to achieve Dalcher’s goals of identifying “compromises.” The motivation is especially strong given that, as Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2).

359. Moreover, a POSITA would have been motivated to use Pappas for its low false positive rate, which was borne out of experimentation. “This is a generic constraint that holds for both return-oriented and jump-oriented code (or potential combinations—in the rest of this section we refer to both techniques as ROP)... For the current set of protected functions we use a threshold of eight gadgets, **which allows for increased resilience to false positives.**” (Ex. 1005 at p. 8-9, emphasis added).

360. Thus, Claim 4 of the '591 patent is rendered obvious by Dalcher in view of Pappas.

F. Claim 5 is rendered obvious by Dalcher in view of Pappas

361. Claim 5 of the '591 patent is rendered obvious by Dalcher in view of Pappas. This claim, which depends from Claim 1, reads:

The method according to claim 1, wherein the monitoring further comprises applying, upon detecting a function call of one of the plurality of functions in the monitored memory space, a hooking technique to hook a lowest level user mode function of the invoked function, wherein the executing of the stack walk processing executes a reverse stack walk from the lowest level user mode function of the invoked function.

362. I have already explained that Dalcher in view of Pappas discloses Claim 1. Also, Dalcher in view of Pappas discloses “wherein the monitoring further comprises applying, upon detecting a function call of one of the plurality of functions in the monitored memory space, a hooking technique to hook a [low] level user mode function of the invoked function, wherein the executing of the stack walk processing executes a reverse stack walk from the [low] level user mode function of the invoked function” for the reasons discussed in my analyses for Claim 1. (I explained for Claim 1 that Dalcher and Pappas’ stack walks proceed backwards or in reverse, so a POSITA would have understood that they disclose “a reverse stack walk from the [low] level user mode function of the invoked function.”) I incorporate by reference my analyses for Claim 1.

Moreover, Dalcher in view of Pappas further discloses the process of Claim 5, wherein “lowest level user mode function” is used (e.g., “. . . a hooking technique to hook a lowest level user mode function of the invoked function, wherein the executing of the stack walk processing executes a reverse stack walk from the lowest level user mode function of the invoked function.”).

363. For example, Dalcher discloses inserting additional monitoring (including hooking) based on what is detected in an initial monitoring. In other words, Dalcher hooks additional functions based on detecting activity in a stack trace of an already monitored function. Dalcher explains that, “For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame... the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame.” (Ex. 1004 at [0023]).

Dalcher provides the following details:

Additionally, as shown in operation 412, additional monitoring is enabled as indicated by the examination of the call stack. ***For example, enabling the additional monitoring may include placing one or more additional monitoring points (e.g., hooks, probes, etc.) in the call stack*** or the freed memory in the call stack. In one embodiment, the additional monitoring may include dynamically adjusted monitoring. In this way, a more comprehensive view of call stack frames not visible from the callbacks of the initial level of monitoring may be provided. (Ex. 1004 at [0033], emphasis added).

364. Dalcher also discloses monitoring operating system calls. A POSITA would have known that system calls are calls from user-space into kernel space. (See Section V.B.). Because the system call then calls kernel functions, a system call is an example of the “lowest-level user mode function”. “For example, a generated keystroke test event may be expected to be handled only by a subset of OS-supplied software. If the monitoring indicates that that other software was involved in this handling, one or more actions may be taken. For example, the monitoring could

flag the involvement as outside allowed system operations, possibly indicating a presence of malicious software, etc.” (Ex. 1004 at [0026]).

365. Pappas expands on these concepts and introduces “low level user functions” in addition to “lowest level user mode functions.” Pappas explains that in Windows 7, the system calls (which subsequently call into kernel functions and are, therefore, the “lowest level **user mode** functions”) are generally not available to other programs and software. Instead, Windows 7 provides an API that is available to programs and software, which then calls to the actual system calls. The Windows APIs are low-level user mode functions because they are the lowest functions normally available to other software, but they are not the lowest-level user mode functions that call into the kernel. As Pappas explains:

User-level programs interact with the underlying system mainly through system calls. Unix-like systems provide to applications wrapper functions for the available system calls (often using the same name as the system call they invoke) as part of the standard library. In contrast, Windows does not expose the system call interface directly to user-level programs. Instead, programs interact with the OS through the Windows API [13], which is organized into several DLLs according to different kinds of functionality. In turn, those DLLs call functions from the undocumented Native API [59], implemented in ntdll.dll, to invoke kernel-level services.
(Ex. 1005 at p. 5).

366. In Pappas, when an API is called, it eventually resolves to one or more system calls. As explained above, a system call is a “lowest level user mode function of the invoked function”.

367. It would have been obvious to a POSITA with Dalcher in view of Pappas to apply Pappas’ teachings about different low levels (including lowest level user mode function) to Dalcher’s teachings of adding new monitoring points after an initial analysis such that the Windows 7 syscalls are monitored (via hooking) after analysis of the Windows 7 APIs. This is an example of detecting the API, or low-level user mode function, and then, recognizing a system call will be executed, hooking the system call, or the lowest level user mode function (e.g., to

protect the kernel space from exploitation). Pappas discloses that the lowest level syscalls require monitoring in order to connect them with the analysis made of the APIs. As Pappas explains, “[a]fter a clear LBR check at an API function’s entry point, kBouncer writes a checkpoint that denotes a legitimate invocation of that particular function. When the respective system call is later invoked, the system call handler verifies that a proper checkpoint was previously set by the expected API function...” (Ex. 1005 at p. 6).

368. It would have been obvious to a POSITA that this disclosure from Pappas applies to Dalcher’s teachings about additional monitoring, as described above. Pappas explains that this is necessary to identify a “legitimate invocation of that particular [API] function” for correctly tracing the access to the system call. (*Id.*). A POSITA would have been motivated to add this functionality into Dalcher’s stack walk disclosures.

369. As explained elsewhere (e.g., limitations of Claim 1), Dalcher discloses monitoring via stack walk from the most recently called function back to the original calling function, which discloses the claimed “reverse stack walk from the lowest level user mode function” back to the original invoked function (see my previous comments about Dalcher’s use of top and bottom to refer to memory and not the stack). “Additionally, as shown in operation 504, extended call stack analysis is performed. In one embodiment, performing extended call stack analysis may include identifying a current top of the call stack. For example, the call stack may grow from top to bottom as functions are called, and may shrink from bottom to top as functions return.” (Ex. 1004 at [0037]). Dalcher also explained that “the code pointed to by the suspected call return address may be... examined for a preceding call that may explain the invocation of the candidate stack frame. For instance, the preceding call may be a call to the candidate stack frame, another stack frame, etc.” (Ex. 1004 at [0039]).

370. Pappas also discloses a reverse call-stack tracing (stack walk) implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6). To analyze gadget chaining, Pappas performs a reverse stack walk from the called API to up to 20 previous calls, which is another example of executing a reverse stack walk from the “lowest level user mode function of the invoked function”. (See Ex. 1005 at p. 10). Pappas explains: “Gadget chaining patterns are identified as follows: starting from the most recent branch in the LBR stack, the number of consecutive targets that point to gadgets are counted. Any ret targets are looked up in the call-preceded gadgets hash table, whereas call or jmp targets are looked up in both hash tables, call-preceded or not.” (Ex. 1005 at p. 10).

371. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher describes providing a “call stack analysis” that traces one call to another. As I explained previously, Dalcher describes “in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... includ[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024]). Thus, Dalcher provides a framework for identifying the chain of frames in a call stack.

372. A POSITA would have recognized that Pappas relies on a call-stack tracing implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU

tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6).

373. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher’s framework. As stated previously, a POSITA would have been motivated to make this combination in order to achieve Dalcher’s goals of identifying “compromises.” The motivation is especially strong given that, as Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2).

374. Moreover, a POSITA would have been motivated to use Pappas for its low false positive rate, which was borne out of experimentation. “This is a generic constraint that holds for both return-oriented and jump-oriented code (or potential combinations—in the rest of this section we refer to both techniques as ROP)... For the current set of protected functions we use a threshold of eight gadgets, **which allows for increased resilience to false positives.**” (Ex. 1005 at p. 8-9, emphasis added).

375. Thus, Claim 5 of the ’591 patent is rendered obvious by Dalcher in view of Pappas.

G. Claim 6 is rendered obvious by Dalcher in view of Pappas

376. Claim 6 of the ’591 patent is rendered obvious by Dalcher in view of Pappas. This claim, which depends from Claim 1, reads:

The method according to claim 1, wherein the plurality of functions comprises an NtAllocateVirtualMemory function, an NtProtectVirtualMemory function, an NtCreateProcess function, an NtCreateProcessEX function, an NtCreateUserIProcess function, and an LdrLoadDll function.

377. I have already explained with respect to Claim 1 (including but not limited to limitations 1[b] and 1[c]) that Dalcher and Pappas both disclose monitoring OS-specific functions/APIs. I incorporate my analysis of Claim 1 into this section by reference.

378. Pappas further discloses specifically protecting specific Windows APIs that are analogous to those recited by Claim 6. In particular, Pappas discloses “CreateProcessA” and “CreateProcessW” (each of which “an NtCreateProcess function,” “an NtCreateProcessEX function,” and “an NtCreateUserIProcess function”), “VirtualAlloc” (“an NtAllocateVirtualMemory function”), “VirtualProtect” (“an NtProtectVirtualMemory function”), and “LoadLibraryA” and “LoadLibraryW” (each of which “an LdrLoadDll function”) (See, e.g., Ex. 1005 at p. 17).

379. To the extent that Pappas does not disclose all of “an NtAllocateVirtualMemory function, an NtProtectVirtualMemory function, an NtCreateProcess function, an NtCreateProcessEX function, an NtCreateUserIProcess function, and an LdrLoadDll function,” it would have been obvious to a POSITA to do so for the reason above (e.g., that Pappas discloses specifically protecting specific Windows APIs that are analogous to those recited by Claim 6) and for the reasons provided below. Pappas explains:

Our selection of sensitive Windows API functions was made empirically based on a large set of different shellcode and ROP payload implementations [5, 3, 56, 12, 27, 60]. A list of the 52 currently protected functions is provided in the appendix. Although current ROP exploits rely mainly on only a handful of API functions (see Sec. 5.2), we have included many others that have been used in the past in legacy shellcode, as some exploits might implement their whole functionality using purely ROP code (as demonstrated recently by an exploit against the latest version of Adobe Reader XI [19]). ***The set of protected functions can be easily extended with any additional potentially sensitive functions that we might have left out.*** (Ex. 1005 at p. 13, emphasis added).

380. It would have been obvious to a POSITA to extend Pappas to any specific function call that provides similar low-level features (i.e., virtual memory allocation, process creation, and library loading), all of which are specifically protected by Pappas. Moreover, these functions are standard Windows APIs that would have been well known to a POSITA many years prior to the priority date of the alleged invention. And, the importance of monitoring and protecting such Windows APIs was well known to a POSITA.

381. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher discusses monitoring operating-system functions. “For example, a generated keystroke test event may be expected to be handled only by a subset of OS-supplied software.” (Ex. 1004 at [0026]). It would have been obvious to a POSITA to monitor other sensitive functions, such as those disclosed by Pappas.

382. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher describes providing a “call stack analysis” that traces one call to another. As I explained previously, Dalcher describes “in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... includ[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024]). Thus, Dalcher provides a framework for identifying the chain of frames in a call stack.

383. A POSITA would have recognized that Pappas relies on a call-stack tracing implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas

explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6).

384. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher. As stated previously, a POSITA would have been motivated to make this combination in order to achieve Dalcher’s goals of identifying “compromises.” The motivation is especially strong given that, as Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2).

385. Thus, Claim 6 of the ’591 patent is rendered obvious by Dalcher in view of Pappas.

H. Claim 7 is rendered obvious by Dalcher in view of Pappas

386. Claim 7 of the ’591 patent is rendered obvious by Dalcher in view of Pappas. This claim, which depends from Claim 5, reads:

The method according to claim 5, wherein performing the memory check comprises evaluating state information associated with a cache entry for the hooked function.

387. Dalcher in view of Pappas discloses or renders obvious elements of this claim (e.g., performing the memory check comprises evaluating cache or cache entry for the hooked function) for the reasons similar to those provided for Claim 2. My analyses for Claim 2 are incorporated by reference. Moreover, Dalcher in view of Pappas also discloses or renders obvious “...evaluating state information associated with a cache entry...” and this claim for the reasons provided below.

388. Dalcher discloses placing monitoring points into call stacks of (additionally) monitored functions including through the use of hooks or probes. These functions become “hooked functions.” As Dalcher explains:

In one embodiment, monitoring the execution flow of the function may include monitoring the call stack frame from the freed memory in the call stack. ***For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame.*** In another example, the monitoring point may identify code involved in the handling of the test event. In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame. (Ex. 1004 at [0023], emphasis added).

389. A POSITA would have known that the “call stack frame” is a series of addresses associated with a function call. (See Section V.A.). Placing a monitoring point identifies an address of the function call. (See Section V.D.). Thus, Dalcher discloses identifying an address associated with the invoked function.

390. Dalcher discloses analyzing call stacks and state associated with stacks that were found in freed memory. This means that Dalcher’s memory checks in the stack walk evaluate function data within the freed memory cache. A POSITA would have known that call stacks include runtime state information for the function, including memory allocations, variable modifications, and so forth. (See Section V.A.). Returning to my previous citation above, Dalcher explains with respect to the monitoring and hooking process:

In one embodiment, monitoring the execution flow of the function may include ***monitoring the call stack frame*** from the freed memory in the call stack. For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame. In another example, ***the monitoring point may identify code involved in the handling of the test event.*** In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include ***enabling additional monitoring***, based on an analysis of the call stack frame. (Ex. 1004 at [0023]).

391. A POSITA would have known that the “freed memory” is a cache of recently executed stack frames. (See Section V.A., Ex. 1004 at [0022]). As Dalcher explains: “as shown

in operation 304, a call stack frame is determined from freed memory in the call stack... For example, the call stack frame may include memory in the stack that previously held an active stack entry.” (Ex. 1004 at [0021]).

392. Dalcher also provides details about the analysis of state for a given stack frame:

In yet another embodiment, the data may include any information within the call stack that is associated with the function. For example, the data may include an entry (e.g., a frame, etc.) in the call stack. In one embodiment, the entry may include a return address, and the return address may identify the function that pushed the entry onto the call stack. In another embodiment, ***the entry may include a snapshot of a state associated with the function. For example, the entry may include one or more preserved local variables, one or more spaces reserved in the call stack for variables, one or more registers, etc.***

(Ex. 1004 at [0018], emphasis added).

393. Pappas also discloses a cache: “Last Branch Recording” (LBR). As Pappas explains, “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that these “last N” branches relate to call-stack information that can “...track... calls/jumps, returns, and so on.” (Ex. 1005 at p. 4). A POSITA would have known that calls and returns are the entrances and exits of call frames in a call stack. (See Section V.A.). Thus, the LBR is a cache of the call stack and instruction addresses within the LBR are addresses within a cache.

394. Pappas further discloses creating hash tables with gadgets (identified by address). Pappas identifies ROP gadgets by looking up addresses in these tables. When call stacks in the LBR are found to reference certain abnormal conditions (such as ROP gadgets), the process is terminated, preventing execution of the ROP payload. Pappas creates the hash tables during an analysis phase: “The output of the analysis phase is two hash tables: one containing the [address] offsets of call-preceded gadgets, and another containing the rest of the found gadgets.” (Ex. 1005

at p. 10). During execution, call return addresses, such as those identified by the assembly command “ret” (short for return), are checked against the tables. “Any ret targets are looked up in the call-preceded gadgets hash table, whereas call or jmp targets are looked up in both hash tables, call-preceded or not. The most recent branch target is not considered, as it does not point to a gadget, but to the protected API function.” (Ex. 1005 at p. 10).

395. Pappas also discloses evaluating state information associated with the cache information. For example, Pappas explains evaluating state information about the LBR (cache) call stack of the invocation of the function, as shown in Figure 6 of Pappas reproduced and annotated hereinbelow:

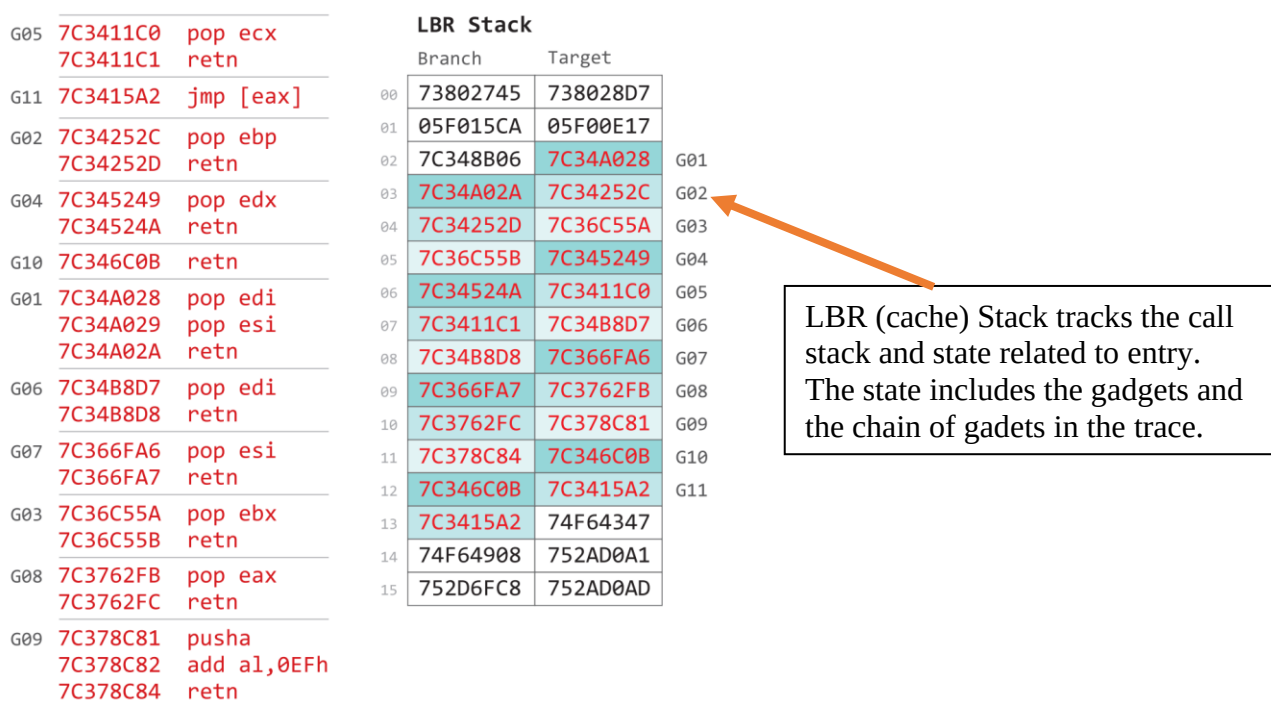


Figure 6: The state of the LBR stack at the time kBouncer blocks an exploit against Adobe Flash [2]. Diagonal pairs of addresses with the same shade correspond to the first and last instruction of each gadget.

(Ex. 1005 at Fig. 6 (annotated)).

396. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher describes providing a “call stack analysis” that traces one call to another. As I explained previously, Dalcher describes “in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... includ[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024]). Thus, Dalcher provides a framework for identifying the chain of frames in a call stack.

397. A POSITA would have recognized that Pappas relies on a call-stack tracing (stack walk) implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6). As described above, Dalcher already describes analyzing data stored in memory; it would have been obvious to a POSITA to analyze call stack execution instructions cached in the Last Branch Recording set as suggested by Pappas.

398. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher. As stated previously, a POSITA would have been motivated to make this combination in order to achieve Dalcher’s goals of identifying “compromises.” The motivation is especially strong given that, as Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2).

399. Also, as discussed above for Claim 1, Dalcher in view of Pappas discloses stack analysis such as performing memory checks during stack walk to detect suspicious behaviors/conditions. (See my analyses for Claim 1). As discussed above (for Claim 2), a POSITA would have known that cache or freed memory is a space that should not be used and entries such as addresses in this cache are potentially memory hosting malicious code (suspicious behaviors). In other words, a POSITA would have known that cache is a potential target for exploitation (e.g., by malicious code), and thus should be checked and protected. Also, as discussed above (for this claim and for Claim 2), Dalcher and Pappas disclose checking state information associated with memory and cache, and also checking cache (including state information associated with cache) as part of stack analysis (e.g., to detect malicious/suspicious behavior and compromise). Thus, a POSITA would have been motivated to implement the Dalcher-Pappas method discussed in Claim 1 (including its stack analysis (e.g., memory checks and stack walk)) to include checks of cache, including checking and evaluating state information associated with cache / cache entry for the functions associated with the call stack (that is walked by the stack walk), including the hooked functions (e.g., initially invoked function and/or subsequently invoked function that are hooked through Dalcher-Pappas' techniques discussed in Claim 1), in order to better protect the system.

400. Thus, Claim 7 of the '591 patent is rendered obvious by Dalcher in view of Pappas.

I. Claim 8 is rendered obvious by Dalcher in view of Pappas

401. Claim 8 of the '591 patent is rendered obvious by Dalcher in view of Pappas. This claim, which depends from Claim 7, reads:

The method according to claim 7, wherein when an address of the invoked function is found in a cache, preventing execution of a payload for the invoked function.

402. Dalcher in view of Pappas renders obvious Claim 8.

403. Dalcher in view of Pappas discloses or renders obvious this claim for the reasons provided for Claim 2 above. My analyses for Claim 2 are incorporated by reference.

404. Moreover, as I explained with reference to limitation 2[b], Dalcher discloses analyzing call stacks that were found in freed memory which, as I have already explained (*see* my analysis of Claims 2 and 7), includes “an addresses of the invoked function [that] is found in a cache”. As Dalcher explains:

In one embodiment, monitoring the execution flow of the function may include monitoring the call stack frame from the freed memory in the call stack. For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame. ***In another example, the monitoring point may identify code involved in the handling of the test event.*** In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame. (Ex. 1004 at [0023], emphasis added).

405. A POSITA would have known that the “freed memory” is a cache of recently executed stack frames. (*See* Section V.A., Ex. 1004 at [0022]). As Dalcher explains: “as shown in operation 304, a call stack frame is determined from freed memory in the call stack... For example, the call stack frame may include memory in the stack that previously held an active stack entry.” (Ex. 1004 at [0021]).

406. Pappas also discloses a cache: “Last Branch Recording” (LBR). As Pappas explains, “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that these “last N” branches relate to call-stack information that can “...track... calls/jumps, returns, and so on.” (Ex. 1005 at p. 4). A POSITA would have known that calls and returns are the entrances and exits of call frames in a

call stack. (See Section V.A.). Thus, the LBR is a cache of the call stack which, as I have previously explained means that addresses of instructions within the LBR are addresses in a cache.

407. Pappas further discloses creating hash tables with gadgets (identified by address). Pappas identifies ROP gadgets by looking up addresses in these tables. When call stacks in the LBR are found to reference certain abnormal conditions (such as ROP gadgets), the process is terminated, preventing execution of the ROP payload. (See *also* analyses for Claim 1.) Pappas creates the hash tables during an analysis phase: “The output of the analysis phase is two hash tables: one containing the [address] offsets of call-preceded gadgets, and another containing the rest of the found gadgets.” (Ex. 1005 at p. 10). During execution, call return addresses, such as those identified by the assembly command “ret” (short for return), are checked against the tables. “Any ret targets are looked up in the call-preceded gadgets hash table, whereas call or jmp targets are looked up in both hash tables, call-preceded or not. The most recent branch target is not considered, as it does not point to a gadget, but to the protected API function.” (Ex. 1005 at p. 10).

408. Finally, Pappas discloses blocking the execution of a process if the addresses were found in the LBR and hash table caches. “In case an attack attempt is detected after the analysis of the recorded branches [in the LBR and hash tables], the process is terminated and the user is informed with an alert message.” (Ex. 1005 at p. 10). Thus, Pappas teaches the claimed “when an address of the invoked function is found in a cache, preventing execution of a payload for the invoked function.”

409. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher describes providing a “call stack analysis” that traces one call to another. As I explained previously, Dalcher describes, “in another embodiment, it may be determined whether the call

stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... includ[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024]). Thus, Dalcher provides a framework for identifying the chain of frames in a call stack.

410. A POSITA would have recognized that Pappas relies on a call-stack tracing (stack walk) implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6).

411. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher. As stated previously, a POSITA would have been motivated to make this combination in order to achieve Dalcher’s goals of identifying “compromises.” The motivation is especially strong given that, as Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2).

412. Thus, Claim 8 of the ’591 patent is rendered obvious by Dalcher in view of Pappas.

J.Claims 9-16 are rendered obvious by Dalcher in view of Pappas

413. Claims 9-16 are almost identical to claims 1-8 and shown below in the following table.

1[a]. A computer- implemented method comprising:	9[a]. A system comprising:
	9[b]. at least one memory; and
	9[c]. at least one processor connected with the memory configured to perform operation of:
1[b]. monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions, wherein monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space;	9[d]. monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions, wherein monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space;
1[c]. invoking one of the plurality of functions as a result of receiving a call from an application programming instance;	9[e]. invoking one of the plurality of functions as a result of receiving a call from an application programming instance;
1[d]. executing stack walk processing upon the invocation of one of the plurality of functions in the monitored memory space; and	9[f]. executing stack walk processing upon the invocation of one of the plurality of functions in the monitored memory space; and

1[e]. performing, during the executing of the stack walk processing before an address of an originating caller function is reached, a memory check for a plurality of stack entries identified during the stack walk processing to detect suspicious behavior,	9[g]. performing, during the executing of the stack walk processing before an address of an originating caller function is reached, a memory check for a plurality of stack entries identified during the stack walk processing to detect suspicious behavior,
1[f]. wherein an alert of suspicious behavior is triggered when the performing of the memory check detects at least one of the following: code execution is attempted from non-executable memory, a base pointer is identified as being invalid, an invalid stack return address is identified, attempted execution of a return-oriented programming technique is detected, the base pointer is detected as being outside a current thread stack, and a return address is detected as being inside a virtual memory area,	9[h]. wherein an alert of suspicious behavior is triggered when the performing of the memory check detects at least one of the following: code execution is attempted from non-executable memory, a base pointer is identified as being invalid, an invalid stack return address is identified, attempted execution of a return-oriented programming technique is detected, the base pointer is detected as being outside a current thread stack, and a return address is detected as being inside a virtual memory area,
1[g].	9[i].

wherein when an alert of suspicious behavior is triggered, preventing execution of a payload for the invoked function from operating.	wherein when an alert of suspicious behavior is triggered, <i>the method further comprising</i> preventing execution of a payload for the invoked function.
2. The method according to claim 1, further comprising: identifying an address associated with the invoked function; and when the address of the invoked function address is found in a cache, preventing execution of a payload for the invoked function	10. <i>The system according to claim 9</i>, further comprising identifying an address associated with the invoked function; and when the address of the invoked function is found in a cache, <i>and wherein the processor further executing an operation</i> of preventing execution of a payload for the invoked function.
3. The method according to claim 1, wherein executing of the stack walk processing further comprises walking a stack frame associated with the invoked function from a low level user mode function to the originating address of the invoked function.	11. <i>The system according to claim 9</i>, wherein <i>the</i> executing of the stack walk processing further comprises walking a stack frame associated with the invoked function from a low level user mode function to the originating address of the invoked function.
4.	12.

The method according to claim 1, wherein the monitoring further comprises monitoring functions associated with memory management in the memory space by a user mode function.	<i>The system according to claim 9</i>, wherein the monitoring further comprises monitoring functions associated with memory management in the memory space by a user mode function.
5. The method according to claim 1, wherein the monitoring further comprises applying, upon detecting a function call of one of the plurality of functions in the monitored memory space, a hooking technique to hook a lowest level user mode function of the invoked function, wherein the executing of the stack walk processing executes a reverse stack walk from the lowest level user mode function of the invoked function	13. <i>The system according to claim 9</i>, wherein the monitoring further comprises applying, upon detecting a function call of one of the plurality of functions in the monitored memory space, a hooking technique to hook a lowest level user mode function of the invoked function, <i>and</i> wherein the executing of the stack walk processing executes a reverse stack walk from the lowest level user mode function of the invoked function.
6. The method according to claim 1, wherein the plurality of functions comprises an NtAllocateVirtualMemory function, an NtProtectVirtualMemory function, an NtCreateProcess function, an	14. <i>The system according to claim 9</i>, wherein the plurality of functions comprises an NtAllocateVirtualMemory function, an NtProtectVirtualMemory function, an NtCreateProcess function, an

NtCreateProcessEX function, an NtCreateUserIProcess function, and an LdrLoadDll function.	NtCreateProcessEX function, an NtCreateUserIProcess function, and an LdrLoadDll function.
7. The method according to claim 5, wherein performing the memory check comprises evaluating state information associated with a cache entry for the hooked function.	15. <i>The system according to claim 13,</i> wherein performing of the memory check comprises evaluating state information associated with a cache entry for the hooked function.
8. The method according to claim 7, wherein when an address of the invoked function is found in a cache, preventing execution of a payload for the invoked function.	16. <i>The system according to claim 15,</i> wherein when an address of the invoked function is found in a cache, <i>and wherein the processor further executing an operation of</i> preventing execution of a payload for the invoked function.

414. Thus, as shown above, the only substantive changes between claims 1-8 and 9-16 are that Claims 9-16 recite a system and require a memory and a processor connected with the memory configured to perform operation of the recited steps.

415. To the extent the preamble is considered limiting, both Dalcher and Pappas disclose a “system”. For example, Dalcher discloses, “**A system**, method, and computer program product are provided for monitoring an execution flow of a function.” (Ex. 1004 at [0004], emphasis

added). Pappas discloses, “In contrast to the above approaches, **our system monitors** the executed indirect branch instructions using Last Branch Recording (LBR) [39, Sec. 17.4], a recent feature of Intel processors introduced in the Nehalem architecture.” (Ex. 1005 at p. 4, emphasis added).

416. Both Dalcher and Dalcher in view of Pappas disclose a memory and a processor connected with the memory, as recited in limitations 9[b] and 9[c]. Dalcher, for example, explains that “a typical hardware configuration of a workstation in accordance with one embodiment [has] a central processing unit 210, such as a microprocessor... The workstation shown in FIG. 2 includes a Random Access Memory (RAM) 214...” (Ex. 1004 at [0012]-[0013]; see also Claim 19). I have also reproduced Dalcher Figure 2 (annotated), which illustrates “Random Access Memory (RAM) 214” and “central processing unit 210”, hereinbelow:

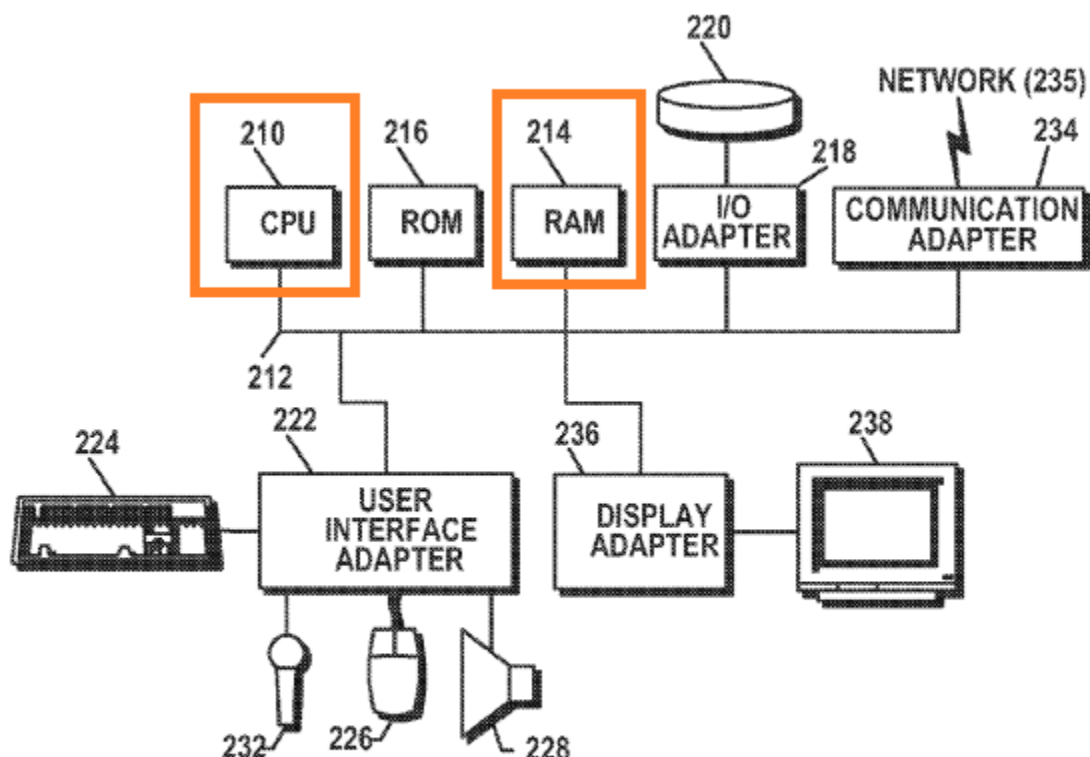


FIGURE 2

417. Dalcher discloses that its processes can be implemented as instructions for a device such as the above-discussed workstation with a processor connected with memory. Dalcher explains, “The workstation may have resident thereon any desired operating system. It will be appreciated that an embodiment may also be implemented on platforms and operating systems other than those mentioned. One embodiment may be written using JAVA, C, and/or C++ language, or other programming languages, along with an object oriented programming methodology. Object oriented programming (OOP) has become increasingly used to develop complex applications.” (Ex. 1004 at [0014]). A POSITA would have understood that running a program (or processes or instructions) on a computer with a processor involves having the program instructions executed by the processor to perform the processes (e.g., of Dalcher-Pappas such as those that disclose the processes recited by Claim 9 and its dependent claims). (*Id.*; *see also id.* at Claim 19). Moreover, Dalcher explains, “Of course, the various embodiments set forth herein may be implemented utilizing hardware, software, or any desired combination thereof. For that matter, any type of logic may be utilized which is capable of implementing the various functionality set forth herein.” (*Id.* at [0015]). So, a POSITA would have been motivated to configure the processor to perform the processes of Dalcher-Pappas such as those that disclose the processes recited by Claim 9 and its dependent claims.

418. Thus, Dalcher discloses or render obvious limitations 9[b] and 9[c].

419. Pappas also discloses a processor. “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Thus, Pappas discloses or renders obvious 9[b]. To the extent that Pappas does not disclose a memory (which is not conceded), Dalcher in view of Pappas does.

420. Thus, Dalcher and/or Dalcher in view of Pappas disclose or render obvious limitations 9[a] and 9[b].

421. Dalcher in view of Pappas discloses 9[d] – 9[i] based on the corresponding analysis for limitations 1[b]-1[g]. I incorporate my analysis of limitations 1[b]-1[g] into this section by reference. I do note for completeness that limitation 9[i] references “the method”, i.e., “the method further comprising”. However, Claim 9 is not a method claim. And, to the extent Claim 9 (e.g., limitation 9[i]) requires a method, Dalcher in view of Pappas discloses a method, as I discussed above for Claim 1. (See my analysis for Claim 1).

422. Thus, Dalcher in view of Pappas renders obvious Claim 9.

423. Dalcher in view of Pappas renders obvious Claims 10-16 based on the corresponding analysis for Claims 2-8. I incorporate my analysis of Claims 2-8 into this section by reference. Moreover, a POSITA would have understood that the processors disclosed by Dalcher/Pappas (discussed above) are configured to perform the operations of Claims 10-16 for the same reason discussed above as to why the processors are configured to perform the operations of Claim 9.

424. Thus, Dalcher in view of Pappas renders obvious Claims 10-16.

K. Claim 17 is rendered obvious by Dalcher in view of Pappas

425. Claim 17 is similar to Claim 9. I have illustrated the differences (e.g., in **bold** and ~~strikethrough~~) in the table below.

9[a]. A system comprising:	17[a]. A computer-readable storage device containing instructions
9[b]. at least one memory; and	

9[c]. at least one processor connected with the memory configured to perform operation of:	17[b]. that when executed on at least one processor, causing the processor to execute a process comprising:
9[d]. monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions, wherein monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space;	17[c]. monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions, wherein monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space;
9[e]. invoking one of the plurality of functions as a result of receiving a call from an application programming instance;	17[d]. invoking one of the plurality of functions as a result of receiving a call from an application programming instance;
9[f]. executing stack walk processing upon the invocation of one of the plurality of functions in the monitored memory space; and	17[e]. executing stack walk processing upon the invocation of one of the plurality of functions in the monitored memory space; and
9[g]. performing, during the executing of the stack walk processing before an address of	17[f]. performing, during the executing of the stack walk processing before an address of

an originating caller function is reached, a memory check for a plurality of stack entries identified during the stack walk processing to detect suspicious behavior,	an originating caller function is reached, a memory check for a plurality of stack entries identified during the stack walk processing to detect suspicious behavior,
9[h]. wherein an alert of suspicious behavior is triggered when the performing of the memory check detects at least one of the following: code execution is attempted from non-executable memory, a base pointer is identified as being invalid, an invalid stack return address is identified, attempted execution of a return-oriented programming technique is detected, the base pointer is detected as being outside a current thread stack, and a return address is detected as being inside a virtual memory area,	17[g](i). wherein an alert of suspicious behavior is triggered when the performing of the memory check detects at least one of the following: code execution is attempted from non-executable memory, a base pointer is identified as being invalid, an invalid stack return address is identified, attempted execution of a return-oriented programming technique is detected, the base pointer is detected as being outside a current thread stack, a return address is detected as being inside a virtual memory area, and 17[g](ii). a low level user mode function is attempted to be executed in a cache,

	wherein the cache is evaluated for an address of the invoked function, and
9[i]. wherein when an alert of suspicious behavior is triggered, the method further comprising preventing execution of a payload for the invoked function.	17[h]. wherein when an alert of suspicious behavior is triggered, ~~the method further comprising~~ preventing execution of a payload for the invoked function.

426. As visualized in the above table, the only substantive difference is the addition of limitation 17[g](ii), “a low level user mode function is attempted to be executed in a cache, wherein the cache is evaluated for an address of the invoked function,” which appears to have no analog in Claim 9.

1. Limitation 17[a]

427. To the extent that the preamble is considered limiting, Dalcher in view of Pappas discloses the claimed “A computer-readable storage device containing instructions...”. For example, Dalcher discloses, “A system, method, **and computer program product** are provided for monitoring an execution flow of a function.” (Ex. 1004 at [0004], emphasis added). Furthermore, Dalcher discloses a storage device connected to a processor: “a workstation in accordance with one embodiment having a central processing unit 210, such as a microprocessor, and a number of other units interconnected via a system bus 212. The workstation shown in FIG. 2 includes a Random. Access Memory (RAM) 214, Read Only Memory (ROM) 216, an I/O adapter 218 for connecting peripheral devices such as **disk storage units 220** to the bus 212, ...” (Ex. 1004 at [0012]-[0013]). A POSITA would have understood that the “computer program product” claimed

by Dalcher would have been stored on a “computer-readable storage device”, such as “disk storage units 220”. (See Section V.A.: “Running a program typically involves “loading” a program (or components thereof) from some kind of long-term storage, like a hard disk, into the computer’s memory. A recognized rendezvous point in the collection of instructions serves as a starting point and the processor begin executing the start-up instructions.”). Moreover, a POSITA would have understood that running a program (or processes or instructions) on a computer with a processor involves having the program instructions executed by the processor to perform the processes (e.g., of Dalcher-Pappas such as those that disclose the processes recited by Claim 17 and its dependent claim). (See *also* analyses for Claim 9.) Thus, Dalcher in view of Pappas discloses or renders obvious the preamble of Claim 17.

2. Limitation 17[b]-17[g](i) and 17[h]

428. Dalcher in view of Pappas further discloses or renders obvious limitations 17[b]-17[g](i) and 17[h] based on the analysis of the corresponding limitations in Claim 9 (e.g., limitations 9[b]-9[i]). I incorporate my analysis of those limitations herein by reference.

429. Limitation 17[g] contains one addition, limitation 17[g](ii), that is not present in the corresponding limitation of Claim 9 (e.g., 9[h]). Because limitation 17[g](ii) is part of a group of which only one element is required, it is not necessary to satisfy limitation 17[g](ii) in order for Dalcher in view of Pappas to render obvious Claim 17, so long as at least one of the elements of limitation 17[g](i) is satisfied. And, for the reasons discussed above for Claims 1 and 9, Dalcher in view of Pappas discloses or renders obvious at least one of the elements of limitation 17[g](i). (See analyses for Claims 1 and 9).

430. Moreover, Dalcher in view of Pappas does disclose limitation 17[g](ii), as explained below.

2. Limitation 17[g](ii)

431. Limitation 17[g](ii) of the '591 patent reads:

a low level user mode function is attempted to be executed in a cache, wherein the cache is evaluated for an address of the invoked function, and

432. Dalcher in view of Pappas discloses limitation 17[g](ii).

433. Dalcher in view of Pappas discloses “wherein an alert of suspicious behavior is triggered when the performing of the memory check detects... a low level user mode function is attempted to be executed in a cache, wherein the cache is evaluated for an address of the invoked function.”

434. Dalcher in view of Pappas discloses “wherein an alert of suspicious behavior is triggered when the performing of the memory check detects ...” for the reasons discussed for limitation 1[f], which are incorporated by reference.

435. As discussed with reference to limitation 1[d] (which has similar language to 17[e]; I will refer to 1[d]/17[e] in this section), Dalcher discloses stack walks associated with monitored functions, including functions that are monitored after the initial (or “original”) invoked function is triggered. As Dalcher explains,

As shown in operation 502, stack analysis of a call stack is performed. In one embodiment, performing the stack analysis may include identifying one or more call frames of the call stack by performing a stack walk. For example, stack frames may be discovered by walking back in the stack when stack frame pointers are not present... In another embodiment, performing the stack analysis may include extracting a call return address from an identified call frame. Further, in yet another embodiment, performing the stack analysis may include identifying an origin of code containing the return address.
(Ex. 1004 at [0036]).

436. As explained previously, Dalcher discloses that the stack walk identifies stack frames for analysis.

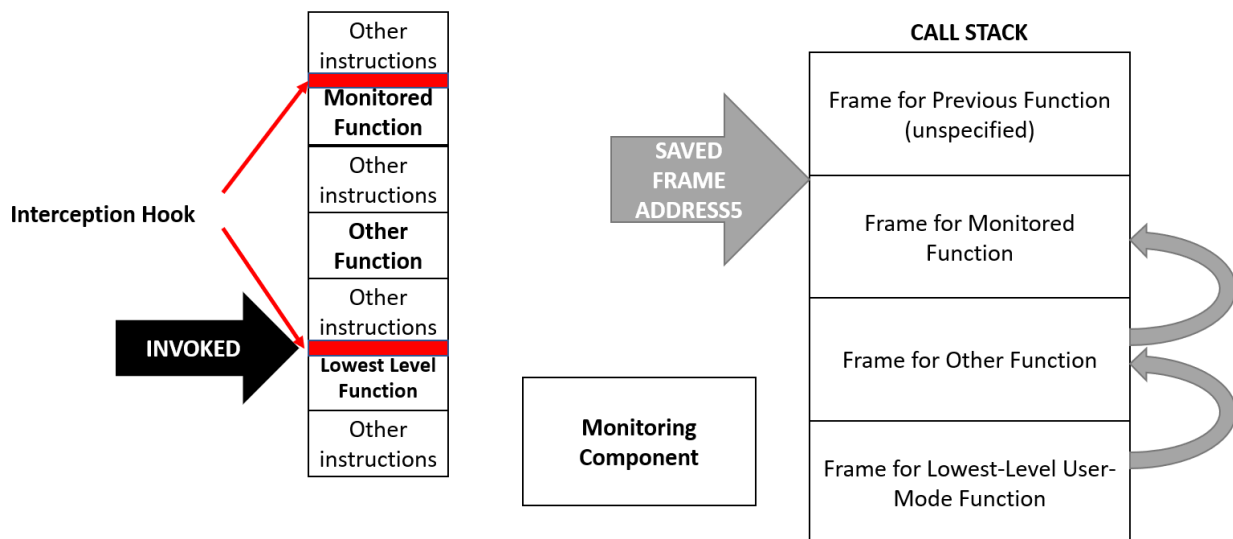
437. Dalcher discloses that, as part of the stack walk, analyzing stack frames prior to some kind of “top” frame (where top refers to the memory addresses of the stack as explained above). In more detail, Dalcher discussed identifying a “top” part of a call stack chain and analyzing the stack frames that come before it. A POSITA would have known that the “stack” refers to the chain of function/subroutine calls, the top of the stack is the most recent call in the chain, and the bottom of the stack is the call that originated the stack walk. (See Section V.A.). However as explained in the technical background, stacks are generally described as growing “downward” in memory (from the top of memory to the bottom, or from higher memory addresses toward lower memory addresses). As explained by Dalcher, “Additionally, as shown in operation 504, extended call stack analysis is performed. In one embodiment, performing extended call stack analysis may include identifying a current top of the call stack. For example, the call stack may grow from top to bottom as functions are called, and may shrink from bottom to top as functions return.” (Ex. 1004 at [0037]). Either way, Dalcher describes being able to trace the stack frames of a function call chain from the most recent function call back to any previous function call, including the initial monitored function.

438. As I explained with respect to limitation 1[d]/17[e], a POSITA would have understood executing the stack walk processing to include designating the *initial* invoked/monitored function as the “original” caller function (original calling function). When performing stack walk processing, stack frames are walked backwards from the lowest level user mode function back to the initial invoked/monitored function (invoked/originating caller function). The specification defines “stack walk processing” as “a process that walks a stack frame for a

hooked functioned [sic] from a lowest level user mode function to an address of the invoked function/originating caller function.” (Ex. 1001 at 10:35-39).¹⁰

439. Each stack frame has its own address so when walking backwards, each frame is processed until the address of the frame of the original call is reached. For clarity, I include my previous illustration for this process.

Stack Walk Processing Step 3



440. As explained with respect to limitation 1[d]/17[e], a monitored function can trigger additional monitoring and is also included in the stack processing. Dalcher explains:

Additionally, as shown in operation 412, ***additional monitoring is enabled as indicated by the examination of the call stack***. For example, enabling the ***additional monitoring may include placing one or more additional monitoring points (e.g., hooks, probes, etc.) in the call stack or the freed memory in the call stack***. In one embodiment, the additional monitoring may include dynamically adjusted monitoring. ***In this way, a more***

¹⁰ I have been informed that in a parallel district court litigation, the owner of the '591 patent is arguing that the '591 patent claims should be afforded their plain and ordinary meaning. I have also been requested to adopt patent owner's position on claim construction for the purposes of rendering my opinions set forth in this declaration. (See Section IV). Thus, prior art can disclose "stack walk processing" under patent owner's plain and ordinary meaning even if its disclosures do not completely match this definition at 10:35-39 of the '591 patent.

comprehensive view of call stack frames not visible from the callbacks of the initial level of monitoring may be provided.
(Ex. 1004 at [0033], emphasis added).

441. As I explained with respect to limitation 1[d]/17[e], Dalcher teaches that call stack processing is triggered by a monitored function by hooking other, related functions for monitoring. A POSITA would have understood that Dalcher's stack walk and other stack analysis teachings are applied to these additional monitoring points in order to obtain the "more comprehensive view of call stack frames." A POSITA also would have found it obvious in view of Pappas to include subsequent calls that are invoked in response to the original call, to be monitored, such as monitoring a system call after the Windows API is invoked.

442. A POSITA would have understood the initially or originally monitored function to be an "originating caller function." Thus, it would have been obvious to a POSITA to perform Dalcher's stack walk processing from a low-level function back to the originally monitored function. In terms of the claim language, the originally monitored function is the "invoked" function.

443. Dalcher also explained that each frame in the call stack could be analyzed to determine its location in the chain, to further determine if the original frame, such as the original call (the originally monitored function) in the code, had been reached. "Additionally, in another embodiment, it may be determined whether the suspected call return address points to code. Further, in yet another embodiment, the code pointed to by the suspected call return address may be analyzed. For example, the code may be examined for a preceding call that may explain the invocation of the candidate stack frame. For instance, the preceding call may be a call to the candidate stack frame, another stack frame, etc." (Ex. 1004 at [0039]). As I explained in Section

V.A., stacks are organized with the originating function at the bottom of the stack, but at the top of memory. Stack walking moves backwards toward the originating calling function.

444. Dalcher further explains that determining the validity or genuineness of one frame may depend on examining previous ones. “[I]n another example, if the candidate stack frame contains a link to another stack frame, and the other stack frame can be verified, then a high level of confidence may be associated with the candidate stack frame.” (Ex. 1004 at [0041]). A POSITA would have understood first, that stack frames within the same call stack are linked implicitly as discussed in Section V.A. Second, a POSITA would have understood that Dalcher’s other disclosures relating to analyzing a stack frame, as discussed in Section VIII.A. to be applicable to determining if the “other” stack frame can be verified.

445. One approach taught by Dalcher for determining if a frame was genuine was to check a return address associated with the frame and if it pointed to valid code. “Further, in yet another embodiment, the code pointed to by the suspected **call return address** may be analyzed. For example, the code may be examined for a preceding call that may explain the invocation of the candidate stack frame. For instance, the preceding call may be a call to the candidate stack frame, another stack frame, etc.” (Ex. 1004 at [0039], emphasis added). As discussed above, and as I discussed in Section V.A., an originating call’s stack frame is linked to the subsequent call’s stack frame by being in a call stack. Thus, a POSITA would have understood that Dalcher discloses determining if a stack frame of a subsequent call is valid (or invalid) based on determining if the stack frame of an originating call is valid (or invalid).

446. Dalcher in view of Pappas discloses a cache.

447. As explained below, a POSITA would have understood freed memory to be a type of cache, at least within the context of the ’591 patent. In Section VI.A., I explain that the ’591

patent describes a cache as a memory where the anti-exploit component keeps track of data about memory of the monitored process. And, as also mentioned in Section VI.A., in the provisional patent application of the '591 patent, a cache is described this way:

If the address is found in a cache entry, then the call is suspicious. Every entry present in the cache structure represents a part of memory that has been recently allocated with the flag EXECUTE, thus a potential piece of memory that could host malicious code.
(Ex. 1007 at p. 15).

448. A POSITA would have understood that freed memory stores information (e.g., stack frames with return addresses) used by Dalcher to look for exploits. The freed memory is also a space that should not be used and addresses in this cache are potentially memory hosting malicious code. (See Section V.A.).

449. Thus, when Dalcher analyzes the call stack in freed memory it is able to detect addresses of invoked functions found in the freed memory cache. As Dalcher explains:

In one embodiment, monitoring the execution flow of the function may include monitoring the call stack frame from the freed memory in the call stack. For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame. In another example, the monitoring point may identify code involved in the handling of the test event. In yet another embodiment, the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling additional monitoring, based on an analysis of the call stack frame.
(Ex. 1004 at [0023]).

450. A POSITA would have known that the “freed memory” is a cache of recently executed stack frames. (See Section V.A., Ex. 1004 at [0022]). As Dalcher explains: “... as shown in operation 304, a call stack frame is determined from freed memory in the call stack... For example, the call stack frame may include memory in the stack that previously held an active stack entry.” (Ex. 1004 at [0021]).

451. To the extent that Dalcher does not disclose a cache (which is not conceded), Pappas explicitly discloses a cache: “Last Branch Recording” (LBR). As Pappas explains, “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that these “last N” branches identified call-stack information that can “...track... calls/jumps, returns, and so on.” (Ex. 1005 at p. 4). A POSITA would have known that calls and returns are the entrances and exits of call frames in a call stack. (See Section V.A.). Thus, the LBR is a cache of the call stack, and addresses from the call stack within the LBR are addresses within a cache.

452. Pappas also discloses setting “checkpoints”, which are data about Windows API calls, and storing them in memory:

Before the invocation of protected Windows API functions, the system inspects the LBR stack to identify whether the execution path that led to the call was part of ROP code, and writes a checkpoint. To account for ROP code that would bypass the check by jumping over kBouncer’s function hook, the system then verifies the entry point of the API function at the time of the corresponding system call invocation.
(Ex. 1005 at p. 6).

The code for setting a checkpoint can only run with kernel privileges, and the checkpoint itself is stored in kernel space so that i) the system call handler can later access it, and ii) any user-level code (and consequently the ROP code itself) cannot tamper with it.
(*Id.*)

453. Pappas further discloses creating hash tables with gadgets (identified by address). Pappas identifies ROP gadgets by looking up addresses in these tables. When call stacks in the LBR are found to reference certain abnormal conditions (such as ROP gadgets), the process is terminated, which prevents the execution of the ROP payload for the invoked function. Pappas creates the hash tables during an analysis phase: “The output of the analysis phase is two hash

tables: one containing the [address] offsets of call-preceded gadgets, and another containing the rest of the found gadgets.” (Ex. 1005 at p. 10). During execution, call return addresses, such as those identified by the assembly command “ret” (short for return), are checked against the tables. “Any ret targets are looked up in the call-preceded gadgets hash table, whereas call or jmp targets are looked up in both hash tables, call-preceded or not. The most recent branch target is not considered, as it does not point to a gadget, but to the protected API function.” (Ex. 1005 at p. 10).

454. Finally, Pappas discloses blocking the execution of a process if the addresses were found in the LBR and hash table caches. “In case an attack attempt is detected after the analysis of the recorded branches [in the LBR and hash tables], the process is terminated and the user is informed with an alert message.” (Ex. 1005 at p. 10).

455. A POSITA would have understood that Pappas’ disclosures of data in memory used for analysis of function calls and stacks are consistent with the definition of “cache” in the ’591 patent specification. (See Ex. 1001 at 6:31-57; *see also* my analyses in Section VI.A. and IX.A.).

456. Dalcher in view of Pappas discloses “a low level user mode function is attempted to be executed in a cache.”

457. The “low level” function is, again, the subsequent call while the originating function is the originating invoking function. As discussed above, Dalcher discloses determining if a stack frame for the subsequent call is valid by evaluating if another stack frame (e.g., such as the originating function’s stack frame) is valid. As I explained in Section VI.A., because “the cache” as required by the claim language is the memory for storing data about monitored memory of process, “executing in” the cache refers to having tracked memory (e.g., memory addresses) about the executing function in the cache.

458. A POSITA would have understood that when the stack frame of the subsequent call is linked to a stack frame already known to be valid or invalid, and is, itself, now known to be valid or invalid, it is now being monitored in the cache. For example, in the cache disclosed by Dalcher in view of Pappas, data structures such as hash tables, data storage, or other memory tracking stack frames and whether or not they are valid. Thus, a POSITA would have known that Dalcher in view of Pappas discloses determining when the subsequent call (“low level user mode function”) is having its execution monitored in the cache (“attempted to be executed in the cache”).

459. Dalcher also discloses that in evaluating the function, “wherein the cache is evaluated for an address of the invoked function.” As I explained above, with respect to the invoked function, which is the originally monitored function, Dalcher discloses examining a call return address in order to determine if it is valid.

460. Dalcher in view of Pappas also discloses raising an alert.

461. Pappas discloses blocking the execution of a process if the addresses were found in the LBR and hash table caches. “In case an attack attempt is detected after the analysis of the recorded branches [in the LBR and hash tables], the process is terminated and the user is informed with an alert message.” (Ex. 1005 at p. 10).

462. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher describes providing a “call stack analysis” that traces one call to another. As I explained previously, Dalcher describes “in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... includ[ing] examining code associated with the call stack frame...” (Ex.

1004 at [0024]). Thus, Dalcher provides a framework for identifying the chain of frames in a call stack.

463. A POSITA would have recognized that Pappas relies on a call-stack tracing implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6). As described above, Dalcher already describes analyzing data stored in memory; it would have been obvious to a POSITA to analyze call stack execution instructions cached in the Last Branch Recording set as suggested by Pappas.

464. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher. As stated previously (*see* Section IX.A.), a POSITA would have been motivated to make this combination in order to achieve Dalcher’s goals of identifying “compromises.” The motivation is especially strong given that, as Pappas explains, “[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections.” (Ex. 1005 at p. 2).

465. Thus, Dalcher in view of Pappas discloses Limitation 17[g](ii).

466. Thus, Dalcher in view of Pappas renders obvious Claim 17.

L. Claim 18 renders obvious by Dalcher in view of Pappas

467. Each limitation of Claim 18 of the ’591 patent, which depends from Claim 17, is rendered obvious by Dalcher in view of Pappas as described below.

1. Limitation 18[a]

468. Limitation 18[a] reads:

The computer-readable storage device according to claim 17, wherein performing of the memory check to determine if code execution is attempted from non-executable memory, the base pointer is identified as being invalid, an invalid stack return address is identified, execution of a return-oriented programming technique is attempted, the base pointer is detected as being outside a current thread stack, and the return address is detected as being inside a virtual memory area, occurs before an address of the invoked function is reached, and

469. Dalcher in view of Pappas discloses limitation 18[a].

470. Dalcher in view of Pappas discloses limitation 18[a] for the reasons discussed above for limitations 1[e] and 1[f], which are incorporated by reference.

471. Limitation 18[a] recites that the memory check for certain types of suspicious behavior “occurs before an address of the invoked function is reached.” A POSITA would have known that this involves stack walk that begins after additional functions have been invoked, such as those described in Claim 1. That is, stack walk begins after an originating function has called subsequent functions, one or more of which subsequent functions have been hooked. (See, e.g., my analysis of limitation 1[d]).

472. Dalcher discloses inserting additional monitoring (including hooking) based on what is detected in an initial monitoring. In other words, Dalcher hooks additional functions based on detecting activity in a stack trace of an already monitored function. Dalcher explains that, “For example, a monitoring point (e.g., a hook, etc.) may be placed on the call stack frame... the monitoring of the execution flow of the function may be dynamically adjusted, based on the call stack frame. For example, monitoring the execution flow of the function may include enabling

additional monitoring, based on an analysis of the call stack frame.” (Ex. 1004 at [0023]). Dalcher provides the following details:

Additionally, as shown in operation 412, additional monitoring is enabled as indicated by the examination of the call stack. ***For example, enabling the additional monitoring may include placing one or more additional monitoring points (e.g., hooks, probes, etc.) in the call stack*** or the freed memory in the call stack. In one embodiment, the additional monitoring may include dynamically adjusted monitoring. In this way, a more comprehensive view of call stack frames not visible from the callbacks of the initial level of monitoring may be provided. (Ex. 1004 at [0033], emphasis added).

473. A POSITA would have understood that these additional monitoring calls can be used to trace subsequent functions called by the originally invoked function. (See, e.g., my analysis of limitation 1[d]).

474. Furthermore, the behaviors recited in limitation 18[a] (i.e., behaviors to be detected by the memory check) mirror those recited in limitation 17[g](i). I have already discussed with reference to limitations 1[f] and 17[g](i) that Dalcher in view of Pappas discloses detecting these behaviors. I incorporate my analysis of those limitations here by reference. Limitation 18[a] does not recite the behavior in limitation 17[g](ii): “a low level user mode function is attempted to be executed in a cache”. Instead, that behavior is recited separately in limitation 18[b], which I will analyze hereafter.

475. A POSITA would have been motivated to combine Dalcher and Pappas. Dalcher described adding additional monitoring points. It would have been obvious to use syscalls as additional monitoring points for Windows API calls as disclosed by Pappas. The stack walking techniques of both Dalcher and Pappas would then apply to the stack walk back to the invoked function. I have already explained in section IX.A. why a POSITA would have been motivated to combine Dalcher and Pappas, and I incorporate this analysis into this section by reference.

476. As I have already explained with reference to limitations 1[e], 1[f], and 2[b], Dalcher in view of Pappas performs memory checks for detecting invalid base pointers, invalid return addresses, and return-oriented programming techniques. I incorporate my analysis of limitations 1[e], 1[f], and 2[b] into this section by reference. As explained above, Dalcher explains that these checks are performed on each stack frame so that the checks are completed before an address of the invoked function is reached.

477. Thus, Dalcher in view of Pappas discloses limitation 18[a].

2. Limitation 18[b]

478. Limitation 18[b] reads:

wherein the performing of the memory check to determine whether the low level user mode function is attempted to be executed in the cache occurs upon reaching the address of the invoked function in the execution of the stack walk processing.

479. Dalcher in view of Pappas discloses limitation 18[b].

480. I have already explained with respect to limitation 17[g](ii) that Dalcher in view of Pappas discloses “wherein the performing of the memory check to determine whether the low level user mode function is attempted to be executed in the cache occurs... in the execution of the stack walk processing.” As I explained with respect to limitation 17[g](ii), Dalcher in view of Pappas discloses determining when the execution of the subsequent function (the “low level user mode function”) is valid or invalid (“attempted to be executed in the cache”).

481. Furthermore, a POSITA would have been motivated to configure Dalcher-Pappas’ process such that “the performing of the memory check to determine whether the low level user mode function is attempted to be executed in the cache occurs upon reaching the address of the

invoked function in the execution of the stack walk processing,” because as discussed above, invoked function is the bottom part of the stack (e.g., end of stack walk), and a POSITA would have been motivated to conduct this important check (“whether the low level user mode function is attempted to be executed in the cache”) as part of the final check of the stack walk. For example, as discussed above for Claims 1, 2, 5, 7, and 8, a POSITA would have understood the importance of monitoring and protecting low level user mode functions and cache (e.g., because they are potential targets for exploitation attempts by malicious code). Thus, a POSITA would have been motivated to conduct this check at the end of the stack walk to make sure there are no signs of exploitation attempts targeting low level user mode functions and cache.

482. A POSITA would have been motivated to combine Dalcher with Pappas. Dalcher describes providing a “call stack analysis” that traces one call to another. As I explained previously, Dalcher describes “in another embodiment, it may be determined whether the call stack frame has been compromised. For example, it may be determined whether the contents of the call stack frame have been overwritten by a second call stack entry subsequent to a first call stack entry in the call stack frame... includ[ing] examining code associated with the call stack frame...” (Ex. 1004 at [0024]). Thus, Dalcher provides a framework for identifying the chain of frames in a call stack.

483. A POSITA would have recognized that Pappas relies on a call-stack tracing implementation called “Last Branch Recording” (LBR): “Last Branch Recording (LBR)... [is] a recent feature of Intel processors introduced in the Nehalem architecture... [in which] the CPU tracks the last N (16 for the CPU model we used) most recent branches.” (Ex. 1005 at p. 4). Pappas explains that branches include the commands “call” and “ret”, which are used to branch from one frame to another. (Ex. 1005 at p. 6). As described above, Dalcher already describes analyzing

data stored in memory; it would have been obvious to a POSITA to analyze call stack execution instructions cached in the Last Branch Recording set as suggested by Pappas.

484. Thus, a POSITA would have recognized that Pappas provides a detailed implementation of call-stack tracing and compromise analysis that integrates with Dalcher. As stated previously, a POSITA would have been motivated to make this combination in order to achieve Dalcher's goals of identifying "compromises." The motivation is especially strong given that, as Pappas explains, "[r]eturn-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections." (Ex. 1005 at p. 2).

485. Thus, Dalcher in view of Pappas renders obvious Claim 18.

X. THERE ARE NO OBJECTIVE INDICIA OF NONOBVIOUSNESS

486. I am not aware of any objective indicia of non-obviousness related specifically to the '591 patent. Indeed, I am not aware of any evidence of commercial success, long felt but unsolved need, failure of others, industry praise, or unexpected results specifically related to the '591 patent.

XI. CONCLUSION

487. I declare that all statements made herein are true to the best of my knowledge and that all statements made on information and belief are believed to be true, and that this Declaration is made with knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under section 1001 of Title 18 of the United States Code.

Executed on March 7, 2023

By: 
Seth James Nielson