UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

AO KASPERSKY LAB
Petitioner

v.

WEBROOT, INC.
Patent Owner

IPR2023-_____
U.S. Patent No. 10,284,591

DECLARATION OF VENKATRAMANAN SIVA SUBRAHMANIAN, UNDER 37 C.F.R. § 1.68 IN SUPPORT OF PETITION FOR *INTER* PARTES REVIEW

TABLE OF CONTENTS

I. INTRODUCTION 4

II. QUALIFICATIONS AND PROFESSIONAL EXPERIENCE 5

III. LEVEL OF ORDINARY SKILL IN THE ART 13

IV. RELEVANT LEGAL STANDARDS 14

A. Claim Construction 14

B. Anticipation Under 35 U.S.C. § 102 16

C. Obviousness Under 35 U.S.C. § 103 16

V. OVERVIEW OF THE '591 PATENT 19

A. Summary of the '591 Patent 20

B. Overview of Prosecution History 24

VI. Technical Background Known to a Person of Ordinary Skill in the Art 28

A. GitHub 28

B. Processes and Memory Address Space 28

C. Call Stacks 29

D. Stack Walking 34

E. Code Hooking 36

F. Exploits and Vulnerabilities 37

G. Return-Oriented Programming 38

H. Monitoring Memory Spaces 39

VII. OVERVIEW OF THE PRIOR ART 41

A. OVERVIEW OF "RUNTIME PREVENTION OF RETURN-ORIENTED PROGRAMMING ATTACKS" TO FRATRIC ("FRATRIC") 41

B. OVERVIEW OF U.S. PATENT PUBLICATION NO. 2012/0255018 TO SALLAM ("SALLAM") 45

VIII. IDENTIFICATION OF HOW THE CLAIMS ARE UNPATENTABLE 47

A. Grounds 1A/1B: Claims 1, 3, 4, 6, 9, 11, 12, 14, and 17 are anticipated under 35 U.S.C. § 102 or obvious under 35 U.S.C. § 103 by Fratric 47

1. Claim 1 47
2. Claim 3 69
3. Claim 4 71
4. Claim 6 73
5. Claim 9 76
6. Claims 11, 12, 14 81
7. Claim 17 82
B. Ground 2: Claims 2, 5, 7, 8, 10, 13, 15, and 16 are obvious under 35 U.S.C. § 103 over Fratric in view of Sallam 86
8. Reasons to Combine Fratric and Sallam 86
9. Claim 2 90
10. Claim 5 96
11. Claim 7 103
12. Claim 8 107
13. Claim 10 107
14. Claims 13, 15, 16 108
IX. NO OBJECTIVE INDICIA OF NONOBVIOUSNESS 109
X. CONCLUSION 109

I, Venkatramanan Siva Subrahmanian, do hereby state and declare as follows:

I. INTRODUCTION

1. I have been retained by DeWitt LLP, counsel for Petitioner, AO Kaspersky Lab in the matter of the *Inter Partes* Review of U.S. Patent No. 10,284,591 ("the '591 patent") to Giuliani et al. My compensation is not contingent on the outcome of this matter or the specifics of my testimony.

2. I am also being reimbursed for reasonable and customary expenses associated with my work and testimony in this investigation. My compensation is not contingent on the outcome of this matter or the specifics of my testimony, and I have no other interest in this case or the parties thereto.

3. I have been asked to provide my opinions regarding whether claims 1-17 (the "Challenged Claims") of the '591 patent are unpatentable as they would have been anticipated or obvious to a person of ordinary skill in the art ("POSITA") at the time of the alleged invention, in light of prior art. It is my opinion that all limitations for the Challenged Claims would have been anticipated or obvious to a POSITA in light of the prior art. My opinions are based on my experience, knowledge and skill in computer security and cyber-security technology.

4. In preparation for this declaration, I have reviewed '591 patent, its prosecution history, and all of the exhibits cited in this declaration and Petition. I may rely upon these materials, my experience and knowledge, and/or additional materials to rebut arguments raised by the Patent Owner. Further, I may also consider additional documents and information in forming any necessary opinions, including documents that may not yet have been provided to me.

5. I am over the age of 18 and make this Declaration based upon personal knowledge, experience, and investigation that I have conducted. I will provide testimony concerning my statements, opinions and conclusions set forth below and may amplify or add thereto through such testimony, demonstratives, or in further written declarations.

II. QUALIFICATIONS AND PROFESSIONAL EXPERIENCE

6. I am the Walter P. Murphy Professor of Computer Science with tenure in the McCormick School of Engineering, Northwestern University (September 2021-present) and Faculty Fellow in the Buffett Institute of Global Affairs. Prior to this appointment, I was the Dartmouth College Distinguished Professor in Cybersecurity, Technology, and Society at Dartmouth College with tenure in the Computer Science Department (August 2017 – August 2021) and Director of the Institute for Security, Technology and Society (April 2018-August 2021). Prior to

that appointment, I was on the faculty of the University of Maryland's Computer Science Department from 1989 to 2017, as an Assistant Professor from 1989 to 1995, as an Associate Professor with tenure from 1995 to 2000 and as a full Professor from 2000 to 2017. I also served as Director of the University of Maryland's Institute for Advanced Computer Studies.

7. I graduated from Syracuse University with a Master of Science degree in Computer Science (1987) and Doctor of Philosophy degree in Computer Science (1989).

8. I have particular expertise in cyber-security having developed some of the first secure query answering algorithms, flexible authentication frameworks, unexplained behavior detection algorithms, techniques for the scalable detection of known threats, identifying human behaviors linked with cyber-vulnerabilities, and combating intellectual property theft. I led the development of the first ever Global Cyber Vulnerability Report consisting of the analysis of billions of anti-virus telemetry reports to quantify the vulnerability of 44 major nations. I developed some of the first methods to predict which vulnerabilities will be exploited, when, and how severe they will be. In addition, I have developed methods for malware identification, deploying deception within networks in order to deceive attackers, and separating false security alarms from real ones.

9. As evidence of my expertise and leadership in computer security and cyber-security, I have written numerous papers, edited several important books, and been invited to speak at many international conferences on the subject matter. See my attached CV for a full listing of my writings and presentations, which is attached to this report as Exhibit 1007. A sample of my recent books and keynotes include:

A. Author of *The Global Cyber-Vulnerability Report* (Springer 2013) with M. Ovelgonne, T. Dumitras, and B. A. Prakash.

B. S. Jajodia, G. Cybenko, V.S. Subrahmanian, V. Swarup, C. Wang, and M. Wellman. (eds). Adaptive, Autonomous, Secure, Cyber Systems, Springer, 2020.

C. S. Jajodia, V.S. Subrahmanian, V. Swarup, C. Wang (eds). Cyber-Deception: Building the Scientific Foundation, Springer, 2016.

D. S. Jajodia, P. Shakarian, V.S. Subrahmanian, V. Swarup, C. Wang (eds). Cyber-Warfare: Building the Scientific Foundation, Springer 2015. *Also released as Chinese student edition in 2016.*

E. Moving Target Defense II (eds. S. Jajodia, A.K. Ghosh, Subrahmanian, V.S., Swarup, V., Wang, C., Wang, X.S.), Springer, September 2012. *Also released in 2014 as a special Chinese edition.*

F. Keynote Address on AI & Cybersecurity, Wells Fargo *Confluence* conference, Dec 2021. (via Zoom)

G. Keynote Speaker, 18th International Conference on Security and Cryptography (SECRYPT 2021), July 2021. (Via Zoom)

H. Keynote Speaker on Ransomware, United Health Group Healthcare Technology Advisory Council, June 30, 2021. (Via Zoom)

I. Keynote Speaker, 1st International Conference on Autonomous Intelligent Cyber-defence Agents, Paris France (but via Zoom), March 2021.

10. Further details of my qualifications are presented in my curriculum vitae, which is attached to this report as Exhibit 1003. By my education, training, and experience, as evidenced by my curriculum vitae, I am qualified as one skilled in the art of computer security and cyber-security technology.

11. I have considered information from various sources in forming my opinions. Besides drawing from my 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:

A. Ex. 1001: U.S. Patent No. 10,284,591 (issued May 7, 2019)

B. Ex. 1002: Declaration of V.S. Subrahmanian, Ph.D. under 37 C.F.R. §1.68

C. Ex. 1003: Curriculum Vitae of V.S. Subrahmanian, Ph.D.

D. Ex. 1006: File History of U.S. Patent No. 10,284,591

E. Ex. 1007: Ivan Fratric, *Runtime Prevention of Return-Oriented Programming Attacks to Ivan Fratric*, https://github.com/ivanfratric/ropguard/blob/master/doc/ropguard.pdf ("Fratric")

F. Ex. 1008: U.S. Patent Publication No. 2012/0255018 (issued Oct. 4, 2012) ("Sallam")

G. Ex. 1009: *Hello World*, GITHUB DOCS, https://docs.github.com/en/get-started/quickstart/hello-world (last visited Apr. 26, 2023)

H. Ex. 1010: *Committing and Reviewing Changes to your Project*, GITHUB DOCS, https://docs.github.com/en/desktop/contributing-and-collaborating-using-github-desktop/making-changes-in-a-branch/committing-and-reviewing-changes-to-your-project (last visited Apr. 26, 2023)

I. Ex. 1011: *Setting Repository Visibility*, GITHUB DOCS, https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/managing-repository-settings/setting-repository-visibility (last visited Apr. 26, 2023)

J. Ex. 1012: *GitHub Glossary*, GITHUB DOCS, https://docs.github.com/en/get-started/quickstart/github-glossary#public-repository (last visited Apr. 26, 2023)

K. Ex. 1013: Scott Chacon, *GitHub Code Search*, THE GITHUB BLOG (November 3, 2008), https://github.blog/2008-11-03-github-code-search/

L. Ex. 1014: *About Searching on GitHub*, GITHUB DOCS, https://docs.github.com/en/search-github/getting-started-with-searching-on-github/about-searching-on-github (last visited Apr. 26, 2023)

M. Ex. 1015: *Finding Ways to Contribute to Open Source on GitHub*, GITHUB DOCS, https://docs.github.com/en/get-started/exploring-projects-on-github/finding-ways-to-contribute-to-open-source-on-github (last visited Apr. 26, 2023)

N. Ex. 1016: Claim Construction Order, *Webroot, Inc. and Open Text, Inc. v. AO Kaspersky Lab*, Case No. 6:22-CV-00243-ADA-DTG, (Mar. 16, 2023)

O. Ex. 1017: MICHAEL HUTTERMANN, DEVOPS FOR DEVELOPERS (Apress, 2012)

P. Ex. 1018: Nicolas Despres, *Automatic performance monitoring tool*, LABORATOIRE DE RECHERCHE ET DEVELOPPEMENT DE L'EPITA TECH REPORT No 0601, 915 (2006)

Q. Ex. 1019: Bryan D. Payne, Martin D.P. de A. Carbone, Wenke Lee, 2007, December. *Secure and Flexible Monitoring of Virtual Machines*, 23RD ANN. COMP. SEC. APPLICATIONS CONF., 385, 385-397 (2007)

R. Ex. 1020: Margaret Rouse, *Dynamic Library*, TECHOPEDIA (Mar. 2, 2012)

S. Ex. 1021: Krishnan, S. and Srihari, K., *A Knowledge-Based Object Oriented DFM Advisor for Surface Mount PCB Assembly*. 10 LI INT'L. J. ADVANCED MFG. TECH, 317, 317-329 (1995)

T. Ex. 1022: Henry Hanping Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenkee Lee, and Barton P. Miller, *Formalizing Sensitivity in Static Analysis for Intrusion Detection*, IEEE SYMP. SEC. PRIV., 194, 194-208

U. Ex. 1023: Wei Li, Lap-chung Lam, and Tzi-cker Chiueh, *How to Automatically and Accurately Sandbox Microsoft IIS*, 22ND ANN. COMP. SEC. APPLICATIONS CONF., 213, 213-222 (2006)

V. Ex. 1024: Kevin A. Roundy and Barton P. Miller, Hybrid Analysis and Control of Malware, RECENT ADVANCES IN INTRUSION DETECTION: 13TH INT'L SYMP., 317, 317-338 (2010)

W. Ex. 1025: Toshiyuki Maeda, 2002, *Safe Execution of User Programs in Kernel Mode Using Typed Assembly Language,* (Master's Thesis, University of Tokyo, 2002)

X. Ex. 1026: Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimirov, Moshe Bach, *Dynamic Program Analysis of Microsoft Windows Applications*, IEEE INT'L SYMP. PERFORMANCE ANALYSIS SYS. SOFTWARE, 2, 2-12 (2010)

Y. Ex. 1027: Tarjei Mandt, T., *Locking Down the Windows Kernel: Mitigating Null Pointer Exploitation*, NORMAN THREAT RESEARCH

Z. Ex. 1028: Xiangyu Dong, Cong Xu, Yuan Xie, Norman P. Jouppi, N.P., *NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory*, 31 IEEE TRANSACTIONS COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS SYS., 994, 994-1007 (2012)

AA. Ex. 1029: Father, H., *Hooking Windows API-Technics of hooking API functions on Windows*, 2 ASSEMBLY PROGRAMMING J., 2, 2-30 (2004)

BB. Ex. 1030: J. Berdajs and Z. Bosnić, *Extending Applications Using an Advanced Approach to DLL Injection and API Hooking*, 40 SOFTWARE: PRACTICE EXPERIENCE, 567, 567-584 (2010)

CC. Ex. 1031: Ulrich Bayer, Christopher Kruegel, and Engin Kirda, *TTAnalyze: A Tool for Analyzing Malware*, IKARUS SOFTWARE TECH. UNIV. VIENNA, 180, 180-192 (2006)

III. LEVEL OF ORDINARY SKILL IN THE ART

12. I understand the level of a person of ordinary skill in the art at the time of the claimed invention is determined through consideration of several factors. I understand that those factors include: (1) the educational level of the inventor(s); (2) type of problems encountered in the art; (3) prior art solutions to those problems; (4) rapidity with which innovations are made; (5) sophistication of the technology; and (6) educational level of active workers in the field.

13. After considering the factors mentioned above, and based on the disclosure of the ‘591 Patent, it is my opinion that a person of ordinary skill in the art at the relevant time for these patents would hold a Bachelor’s degree in computer science or a similar field, have about 2 years of experience as a software programmer or similar title, and have about 2 years of experience specifically in the computer security field. A person having more education, such as a Master’s degree, would require less experience, and a person not having at least a Bachelor's degree would need more experience.

IV. RELEVANT LEGAL STANDARDS

14. In forming my opinions and considering the subject matter of the '591 patent and its claims in light of the prior art, I am relying on certain legal principles that counsel in this case explained to me. My understanding of these concepts is summarized below.

15. I understand that the claims define the invention. I also understand that an unpatentability analysis is a two-step process. First, the claims of the patent are construed to determine their meaning and scope. Second, after the claims are construed, the content of the prior art is compared to the construed claims.

16. I understand that a claimed invention is only patentable when it is new, useful, and non-obvious in light of the "prior art." That is, the invention, as defined by the claims of the patent, must not be anticipated by or rendered obvious by the prior art.

A. Claim Construction

17. I understand that patent claims are generally interpreted according to their ordinary and customary meaning to a person of ordinary skill in the art taking into consideration the so-called "intrinsic evidence" of the patent consisting of (1) the claim language; (2) the specification and drawings; and (3) the prosecution history. I understand that a Court has discretion to take into consideration so-called "extrinsic

evidence" including references (prior art and non-prior art) as well as definitions from dictionaries and treatises to the extent they do not contradict intrinsic evidence.

18. I understand that claim terms may be explicitly defined in the patent specification or they may be implicitly defined through consistent usage in the specification. I also understand that the scope of claim terms may be limited by statements in the specification or prosecution history where the application clearly disavows or disclaims subject matter in a clear and unmistakable manner.

19. I understand that a patent is required to provide a written description of the invention, and the manner and process of making and using it, in such full, clear, concise, and exact terms as to enable any person skilled in the art to which it pertains, to make and use the claimed subject matter.

20. I understand that a patent claim is indefinite if, when read in light of the specification and the prosecution history, the claim fails to inform, with reasonable certainty, those skilled in the art about the scope of the invention.

21. It is my understanding that, although the terms of a claim may appear to be definite, inconsistency with the specification disclosure or prior art teachings may make an otherwise definite claim take on an unreasonable degree of uncertainty. It is also my understanding that indefiniteness may result from inconsistent prosecution history statements where the claim language and specification on their own leave an uncertainty that, if unresolved, would produce indefiniteness.

22. It is also my understanding that a fundamental rule of claim construction is that terms in a patent are construed with the meaning with which they are presented in the patent, and that if the specification reveals a special definition given to a claim term by the patentee that differs from the meaning it would otherwise possess, then the patentee's definition governs.

B. Anticipation Under 35 U.S.C. § 102

23. I understand that a patent claim is invalid as anticipated if a single piece of prior art teaches every element of the claims, viewed from the perspective of a person of ordinary skill in the art. I also understand that an anticipatory reference does not have to recite word for word what is in the anticipated claims. Anticipation can also occur when a claimed limitation is inherent in the relevant reference. I have been advised that if the prior art necessarily functions in accordance with, or includes, the claims limitations, it can anticipate even though the limitation is not expressly disclosed.

C. Obviousness Under 35 U.S.C. § 103

24. I understand that a patent claim is invalid as obvious if the claimed invention would have been obvious to a person of ordinary skill in the art at the time the claimed invention was made. This means that even if all of the elements of the claim

cannot be found in a single prior art reference that would anticipate the claim, a person of ordinary skill in the field who knew about all the prior art would have come up with the claimed invention. I understand that in an obviousness determination, the person of ordinary skill in the art is presumed to have knowledge of all material prior art. I understand that whether a claim is obvious is based upon the determination of several factual issues.

25. In considering obviousness, I understand that one must determine the scope and content of the prior art. I understand that, in order to be considered as prior art to a patent being considered, a prior art reference must be reasonably related to the claimed invention of that patent. A reference is reasonably related if it is in the same field as the claimed invention or is from another field to which a person of ordinary skill in the art would look to solve a known problem.

26. I understand that one must determine what differences, if any, existed between the claimed invention and the prior art.

27. I understand that a patent claim composed of several elements is not proved obvious merely by demonstrating that each of its elements was independently known in the prior art. In evaluating whether such a claim would have been obvious, one may consider whether a reason has been identified that would have prompted a person of ordinary skill in the art to combine the elements or concepts from the prior art in the same way as in the claimed invention. There is no single way to define the

line between true inventiveness on the one hand (which is patentable) and the application of common sense and ordinary skill to solve a problem on the other hand (which is not patentable). For example, market forces or other design incentives may be what produced a change, rather than true inventiveness.

28. I understand that one may consider whether (1) the change was merely the predictable result of using prior art elements according to their known functions, or whether it was the result of true inventiveness; (2) there is some teaching or suggestion in the prior art to make the modification or combination of elements claimed in the patent; (3) the innovation applies a known technique that had been used to improve a similar device or method in a similar way; (4) the claimed invention would have been obvious to try, meaning that the claimed innovation was one of a relatively small number of possible approaches to the problem with a reasonable expectation of success by those skilled in the art; (5) the invention merely substituted one known element for another known element in order to obtain predictable results; (6) the invention merely applies a known technique to a known device, method, or product to yield predictable results, or (7) known work in the field may have prompted variations of use of the same inventions in the same or different fields due to market forces or design incentives that would have been predictable to a person of ordinary skill in the art.

29. I understand that one must be careful not to determine obviousness using the benefit of hindsight; many true inventions might seem obvious after the fact. I understand that the determination should be based on the position of a person of ordinary skill in the field at the time the claimed invention was made and should not consider what is known today or what is learned from the teaching of the patent.

30. 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.

V. OVERVIEW OF THE '591 PATENT

31. The '591 patent is entitled "Detecting and Preventing Execution of Software Exploits." Ex. 1001, Cover. I understand that the '591 patent was filed on January 27, 2015, as application number 14/606,604 and claims priority to provisional application number 61/931,772, filed on January 27, 2014. For the purpose of my analysis, I have assumed the effective filing date is the provisional filing date of January 27, 2014.

A. Summary of the ‘591 Patent

32. The ‘591 patent, as described in further detail below, is for preventing the execution of software exploits using anti-exploit techniques. The exploits described that the ‘591 patent is concerned with preventing would be considered by one of skill in the art to be return-oriented programming exploits.

33. The ‘591 patent details a system, method, and computer-readable medium for monitoring a memory space for execution of functions, executing stack walking processing on invoked functions, and performing memory checks during the stack walking processing to detect suspicious behavior. Ex. 1001, Abstract.

34. The ‘591 patent provides anti-exploit examples “configured to detect and prevent software exploits that attempt to execute another process or load a module into process memory” on a computer system. Ex. 1001, 3:39-41.

35. The ‘591 patent describes an exploit as “a piece of code, software, data, or a sequence of commands that takes advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software, hardware, or any other electronic component.” Ex. 1001, 1:13-17, see also 3:29-32.

36. The ‘591 patent further explains that “[i]n order to reach a goal of starting a new process, an exploit has to execute a complete execution stack starting from the shellcode and running down through to a lowest level user function (e.g., instruction

that includes machine or assembly instruction). Shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. An exploit will commonly inject shellcode into the target process before or at the same time as it exploits a vulnerability to gain control over a program counter. The program counter is adjusted to point to the shellcode, after which the shellcode (including the malicious payload) is executed." Ex. 1001, 4:24-35.

37. The '591 patent attempts to prevent exploits by, "[u]nlike other exploit prevention technologies that try to prevent that an exploit starts its own shellcode, the anti-exploit examples implemented by the anti-exploit component 102 prevent that the shellcode performs an ultimate execution of a malicious payload . . . ." Ex. 1001, 6:24-29.

38. To detect attempted exploits and prevent the execution of a malicious pay load, the anti-exploit examples disclosed in the '591 patent include (1) "monitor a memory space of a process for execution of functions," Ex. 1001, 1:34-35, and (2) "performing at least one check to evaluate how memory is being allocated before setting/marking an area of memory space as suspicious." Ex. 1001, 2:36-38. "Functions monitored comprise processes that attempt execution of another process or loading of a module into a process memory . . . ." Ex. 1001, 4:58-60.

39. The '591 patent describes monitoring a memory space as including loading "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, 4:45-49.

40. The '591 patent further describes using a "function hooking technique" for monitored functions "to hook to the lowest level user mode function and intercept system routines of a called API." Ex. 1001, 4:36-38. "A hooked function . . . is monitored/evaluated through execution of stack walk processing . . . ," Ex. 1001, 5:23-26 that " . . . walks up a call stack for the process (function) until an address of the first function of a called function is reached." Ex. 1001, 5:26-29. "During the stack walk processing of the stack frame, a first call address of a call sequence is one that is present in the last valid stack entry. A list of conditions may be evaluated against each stack entry found during the stack walk." Ex. 1001, 5:55-59. "In a case where conditions, such as the above identified conditions, are satisfied, the stack walk processing may proceed with evaluating a memory region associated with a called function." Ex. 1001, 6:4-7.

41. The '591 patent describes that "[t]he stack walk processing executed by the anti-exploit component 102 performs a memory check or checks on a frame of call stack pertaining to the called process (including one or more functions) attempting to detect and identify suspicious behavior that may pertain to a software exploit." Ex. 1001, 6:17-22. "By performing the memory checks during the stack walk

processing (e.g., back-walk of the stack frame), shellcode is able to be prevented from performing execution of a malicious payload." Ex. 1001, 7:59-62.

42. The '591 patent also describes that "[t]he anti-exploit component 102 interfaces with the cache component 104 to evaluate the tracked memory information for a monitored process, for example while a memory check is being performed during the stack walk processing. The tracked memory information in the cache component 104 is evaluated before the anti-exploit technology allows code of the process to access memory space (e.g., system memory) of the memory component 106. Memory checks performed pertaining to tracked data in the cache may trigger an alert of suspicious behavior . . . ." Ex. 1001, 6:64-7:7.

43. The '591 patent further details that "[w]hen a memory check performed during the stack walk processing does not trigger an alert of suspicious behavior, the anti-exploit component 102 enables a complete called function to access the a [sic] system kernel via the memory component 106." Ex. 1001, 7:41-45. "In a case where a memory check performed during the stack walk processing triggers an alert of suspicious behavior (e.g., a potential security exploit), the anti-exploit component 102 prevents the execution of the payload of the called function, thus preventing access to the memory component 106." Ex. 1001, 7:46-52. "If decision operation 306 determines that an alert of suspicious behavior is triggered, the flow proceeds to

operation 310 where execution of code associated with an invoked function is prevented." Ex. 1001, 10:58-61.

44. The '591 patent further describes that "[o]nce the address of the originating caller function is reached in the stack walk processing (as identified in row 208), an additional check is performed to evaluate the address of the invoked/called function found in a cache entry is executable as shown in block 212." Ex. 1001, 9:55-59. "If an address of an invoked function is found in a cache entry, then the call is suspicious and an alert is triggered. Each entry present in the cache structure represents a part of memory that has been recently allocated with as executable (e.g., by a flag such as or similar to 'EXECUTE'), and is thus a potential piece of memory that could host malicious code." Ex. 1001, 9:61-67.

B. Overview of Prosecution History

45. It is my understanding that the '591 patent is a U.S. Application filed January 27, 2015, claiming priority to Provisional Application No. 61/931,772, filed January 27, 2014. A determination of whether the '591 patent is entitled to the earliest alleged priority date is unnecessary as the prior art relied upon pre-dates the earliest alleged priority date.

46. It is my understanding that during the prosecution of the '591 patent a first non-final office action was issued December 16, 2016, with rejections under 35

U.S.C. § 102 and 35 U.S.C. § 103. A response to the first non-final office action was filed June 16, 2017, including claim amendments and arguments. A first final office action was issued August 31, 2017, with rejections under 35 U.S.C. § 103. A request for continued examination and response to the first final office action were filed on February 28, 2018, including claim amendments and arguments. A second non-final office action was issued on March 30, 2018, with rejections under 35 U.S.C. § 103. A response to the second non-final office action was filed on October 1, 2018, including claim amendments and arguments. A notice of allowance was issued on December 20, 2018, and did not include an examiner's amendment or reasons for allowance. On March 20, 2019, the issue fee was paid and the patent issued on May 7, 2019. *See* Ex. 1006, pp. 1-238.

47. It is my understanding that over the several rounds of prosecution of the '591 patent, a number of amendments were made to the claims to overcome the rejections and prior art cited by the examiner, which included rejections over U.S. Patent Publication 2012/0255018 to Salaam and U.S. Patent Publication 2007/0016914 to Yeap.

48. Notably, in the June 16, 2017, response to non-final office action, the independent claims were amended to include that monitoring a memory space requires "loading a component for evaluating the at least one monitored function in the memory space." The applicant argued that Sallam's security agent was

configured to "execute on the electronic device at a level below all of the operating systems of the electronic device accessing the memory" and that "there is no teaching that any of Sallam's agents are loaded into the monitored memory space of a process." Ex. 1006, p. 121-131.

49. Additionally, in the February 28, 2018, response to office action the independent claims were amended to require the step of "invoking one of the plurality of functions as a result of receiving a call from an application programming instance." The applicant argued that because "Sallam generally relates to a security agent configured to execute on an electronic device at a level below all of the operating systems of the electronic device accessing the memory or a storage of the electronic device," Sallam does not disclose a stack walk that "occurs as a result of a call from a programming application" as required by the claims of the '591 patent. Applicant further argued that Yeap failed to disclose a security evaluation occurring at the kernel level because the API interception application of Yeap is shown as being in the user mode, creating the interrupt "prior to the call entering the kernel mode" and therefore no system actions are performed as a result of receiving a call as required by the amended claims which require the execution of "a stack walk upon an invocation of a system call from an API." Ex. 1006, pp. 89-101.

50. Further, in the October 1, 2018, response to office action, the claims were amended to require that <u>a</u> memory check <u>for the plurality of stack entries identified</u>

during the stack walk processing to detect suspicious behavior" is performed. The claims were further amended to require that the memory check be performed "during the executing of the stack walking processing before an address of an originating caller function is reached." The applicant argued that the claims require a stack walking process to be executed that "walks up a call stack for the process (function) until an address of the first function of a called function is reached." The applicant further argued that rather than preventing an exploit from starting its own shellcode, "the anti-exploit examples implemented by the anti-exploit component prevent that the shellcode performs an ultimate execution of a malicious payload." The applicant further argued that Sallam does not disclose this because Sallam's accessing of the stack does not occur "as a result of a call from a programming application." The applicant additionally argued that while Sallam discloses "'walking through the Execution Based Pointer through the stack' which may identify 'previous function calls' to provide 'context for the operation at hand', there is no disclosure within Sallam directed to performing the claimed memory check 'for a plurality of stack entries during the stack walk processing to detect suspicious behavior' 'during the executing of the stack walk processing before an address of an originating caller function is reached.'" It is my understanding this means that the pointer and virtual memory conditions in Sallam are performed during the stack walk processing as required by the claims. Ex. 1006, pp. 39-51.

VI. Technical Background Known to a Person of Ordinary Skill in the Art

A. GitHub

51. Launched in 2008 (https://en.wikipedia.org/wiki/GitHub), GitHub is now a widely used website to collaboratively create, share, and distribute data and software. As of 2012 or earlier, GitHub was known in the art for open-source repositories and collaborative programming projects.

52. Because software is often developed over time by several people in different locations, there is a need to keep track of many versions of the software and the changes to the code base that led to these versions. "Git" repositories on GitHub are software repositories that have additional software support to query, manage, update, and roll back changes as needed to such repositories. Many systems exist to manage a set of "Git" repositories. Even by 2012, researchers were describing GitHub in terms such as ". . . GitHub is our sole source of truth. This means that whatever the state of the 'master' branch in our repository, that is the state of the machines (give or take 15 minutes)." Ex. 1017, p. 153.

B. Processes and Memory Address Space

53. When a user of a computer invokes a program (e.g., Google maps on their phone), a "process" is launched by the operating system on which the program is

running. For instance, if the user is running Google Maps and Facebook on their phone, then two processes would be launched, one for each of the invocations of each program. The software code associated with each invocation of an application is executed in the process. When a process is launched, the operating system allocates a section of memory (e.g., addresses X through Y) to the process. The process uses this allocated memory to perform its computations. This is called the "address space" of the process.

C. Call Stacks

54. A call stack is a stack data structure that maintains information about active subroutines/functions of an application, including a return address for when the subroutine finishes. Typically, there is a single call stack for each running process.

55. In general, a stack is a linear data structure that stores data with all accesses, insertions and removals made at the top of the stack such that data is accessed on a last in first out basis. Data is "pushed" onto the top of the stack and "popped" off of the top of the stack. The stack has an indicator (pointer) that always points to the next available top slot on the stack.

56. As an example, see Figure 1 below. When a new stack called S is created by a program, the operating system reserves a space in memory in which to store the

stack. See Figure 1a where the blue box shows the full memory space and the grey box shows the contiguous section of memory allocated to the stack.

57. A pointer, denoted $\uparrow S$, contains the starting memory address of the stack. See Figure 1b. When an item A is loaded into the stack (or "pushed" onto the stack), it is placed in the location pointed to by $\uparrow S$ and is updated to point to the next location in the reserved memory. See Figure 1c. This process continues while more items are loaded in the stack. $\uparrow S$ points to the next available location in memory. See figures 1d and 1e showing the stack pointer after items B and C are also loaded into the stack. At all times, $\uparrow S$ points to the next available memory slot in the stack. When an item is removed (or "popped") from the stack (e.g., read into a new variable and removed), the address in the stack pointer $\uparrow S$ is decremented and the value is read into the new variable. The item removed from the stack is always the address location just before $\uparrow S$. *See* Figure 1f.

58. Stack overflows occur when the stack grows so large that the stack pointer $\uparrow S$ points to a location in memory that was not allocated to it. The stack would be full if $\uparrow S$ were to contain the address to the first location after the last memory location allocated to the stack by the operating system. If an attempt to insert more items into the stack were attempted at this stage, a *stack overflow* (or *buffer overflow)* error should occur. But sometimes the application program may not contain a check for such overflows. In this case, $\uparrow S$ may point to an area of memory not allocated to

the stack. This can lead to *stack overflow* or *buffer overflow* errors which I will describe further.

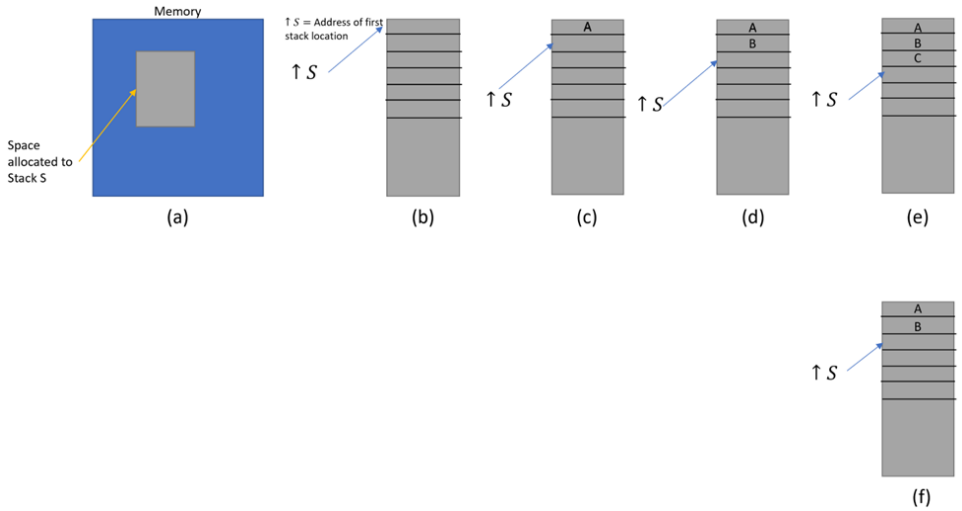


Figure 1. Illustration of Stacks

59. As I describe and show above, call stacks are a particular type of stack that contain information pertaining to executing subroutines/functions of a program. Applications are often implemented via a set of functions which are subroutines within the application that are executed. Referring further below to Figure 2, when a program is being executed, a function or subroutine F1 in the program's code may be invoked (with some arguments). When a function/subroutine of the application is invoked, memory is allocated in the process memory space for F1 to perform its computations. The call stack for the program is updated to include the return address for when F1 finishes. The call stack includes an indicator (pointer) that starts at the bottom of the call stack. For each additional subroutine that is called before the

current subroutine finishes the return address for that subroutine is added ("pushed") on the top of the call stack and the pointer is incremented to point to the new top of the call stack. When the active subroutine finishes, the return address for the subroutine is "popped" of the stack and the pointer is decremented to point to the new top of the call stack.

60. For example, Figure 2a below shows the memory (in blue) and the memory allocated by the operating system to the function F1 (a gray box at the top left). This allocated memory is called a "stack frame" - stack frame (i) in this case. (Note that at this stage, the two other gray boxes shown in Figure 2a have not yet been allocated). The call stack CS for the program is updated with the address of the stack frame for F1 (return address) and the call stack pointer $\uparrow CS$ points to the address of the stack frame for F1 at the bottom of the stack as shown in Figure 2c. But in order to successfully perform its computations, F1 may need to call another function F2.

61. At this stage, the operating system allocates memory for F2's computation – this memory is shown via the gray box toward the middle right of Figure 2a. A stack frame for F2 is created and the address for the start address of the memory allocated to F2 is "pushed" on to the top of the stack as shown in Figure 2d.

62. But suppose now that F2 needs to call yet another function F3 to finish its calculation. In this case, F3 is allocated a stack frame in memory (the gray box towards the middle of the bottom of memory in Figure 2a) and the start address of

this stack frame is then placed on top of the stack as shown in Figure 2e. Note that in all of these cases, the call stack pointer $\uparrow CS$ points to the next available address in which a new item can be added to the stack. Suppose now that F3 does not need to invoke any other functions – it performs its calculations and returns an answer. As soon as it returns the answer, the address of F3's stack frame is "popped" off the top of the stack and the returned value is used by F2's code to perform its calculation. This is shown in Figure 2f. When F2's computations are completed, it in turn returns a value and its address is popped off the top of the stack as shown in Figure 2g. The value returned by F2 is then used by F1 (whose address is now the only remaining stack entry) which then finishes its computation and returns its answer.

63. It is important to note that at any point in time, only the stack frame whose address is at the top of the stack is performing any computations – and when that computation is finished and a value is returned, it is popped off the top of the stack. The stack frame now at the top of stack then becomes active and performs its computation using the values returned previously.

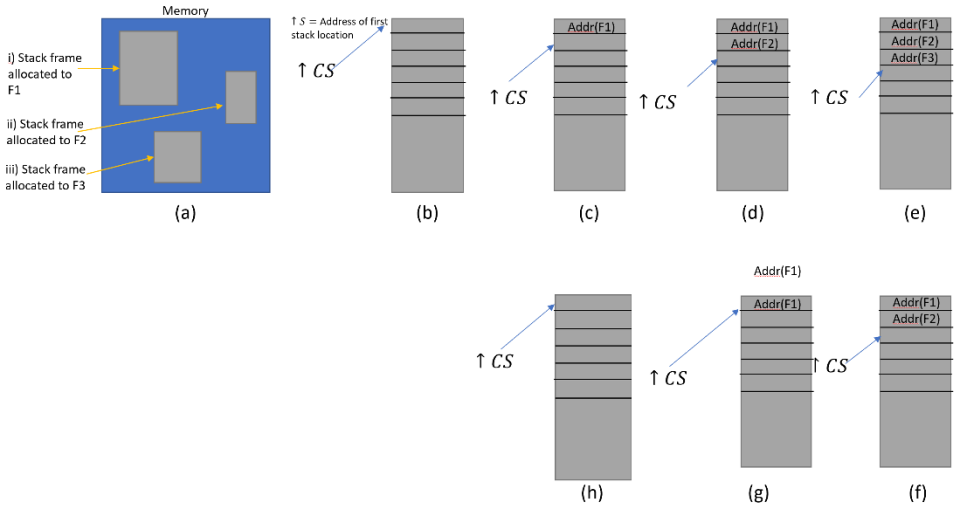


Figure 2. Pictorial Depiction of Call Stack Operation

D. Stack Walking

64. Stack walking refers to the process of unwinding the call stack and stack frames associated with the addresses on the call stack generally from the top of the call stack to the bottom of the call stack or until a designated point of the call stack. This process does not change the position of the call stack pointer or directly manipulate the data on the call stack, but rather is a process for accessing and examining the data on the call stack and the data of the stack frames associated with the addresses on the call stack. There are a number of reasons for performing the process of stack walking including identifying errors encountered while a process is being executed, debugging the code, or detecting malicious behavior. Stack walking may be implemented using code hooking techniques such that when a function is hooked, the stack walking process is performed.

65. For example, suppose a call stack has been built using the process described above, and the current stack consists of F1 (at the bottom), F2 (in the middle) and F3 (at the top of the stack) as shown in Figure 2e above. Suppose F3 is being processed, but rather than successfully returning a value, it throws an error. Such an error may occur for many reasons. For instance, it may be an error in the code associated with F3. Alternatively, the function F3 may have been invoked by the function F2 with an argument that F3 was not designed to handle. Such as, suppose F3 expects a userid with purely alphanumeric characters as input (i.e., a string composed of symbols from the alphabet or the digits 0 through 9). But suppose F2 invokes it instead with the string "john_26". This string has a format that is inconsistent with the expectations of F3 because of the presence of the underscore "_" symbol. Stack walking involves "reverse walking" the stack. When F3 throws an error, we walk back down the stack checking to identify data in the stack frames and which functions invoked the current function. When stack walking for this example, we realize that F2 called F3 and that the error may be in F2. But F2 was called by F1, so it is possible that the source of the problem was in F1 which invoked F2 with the incorrect string "john_26". F2 in turn used this input from F1 to generate the incorrect argument "john_26" when invoking F3.

E. Code Hooking

66. Code hooking includes techniques used to alter or augment the behavior of software components, operating systems, or applications by intercepting API function calls and/or messages between the components. Code hooking can be used for many purposes including intercepting operating system calls to monitor the behavior of applications. In particular, code hooking is a software technique that intercepts calls to such API functions as shown in Figure 3b in order to better study the behavior of the programs that are making such API calls.

67. At the lowest software levels, operating systems offer access to application program interfaces or APIs that are capable of performing low-level actions such as allocating specific memory addresses, writing or reading from protected memory, and more. Access to such APIs is typically reserved for highly trusted and privileged programs and is pictorially shown in Figure 3a. As indicated, Figure 3b is a pictorial explanation of code hooking.

68. For example, if we think of the calling program as a human being A and the API call as a human being B, an ordinary unhooked call would be a request made verbally by A to B. But a hooked version of the call would involve a person C who stands between A and B and intercepts the request made by A to B (before B ever sees the request). The hooking code C can then decide what to do with the request.

For example, C may decide to simply delete the request or it may decide to pass the request to B, while analyzing both the request and B's response, in order to see if A is exhibiting suspicious behavior.

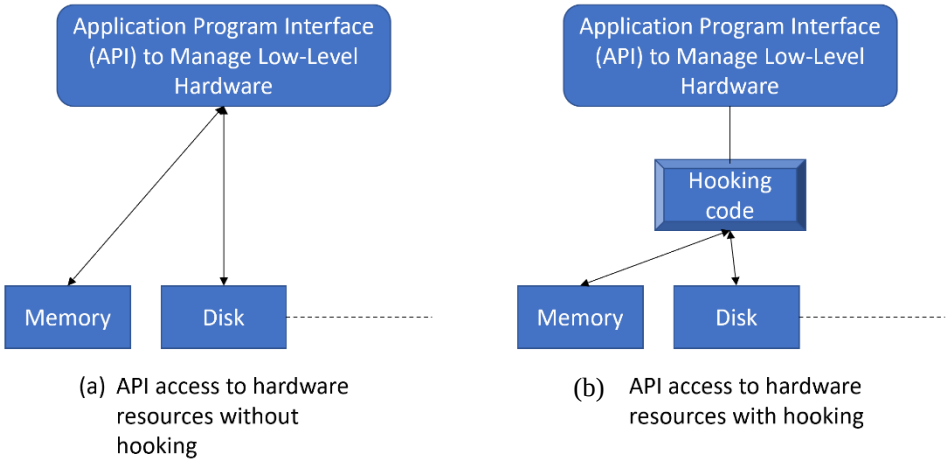


Figure 3. API Access to low-level hardware resources (a) without hooking and (b) with hooking

F. Exploits and Vulnerabilities

69. Most software and hardware components have vulnerabilities. For instance, a program that does not properly handle a malformed input (such as the "john_26" userid mentioned above) may be vulnerable if an attacker knows how to leverage the behavior the program exhibits when fed such a malformed input. Such vulnerabilities may occur because a programmer intent upon delivering a given capability to a customer did not think about the consequences of such malformed inputs.

70. An exploit is a piece of code written by a hacker that uses one or more vulnerabilities of the hardware or software to carry out some malicious functions. Such malicious behavior can include encrypting the user's files, stealing the user's data, stealing money from the user's bank account, causing fake charges to be incurred on a user's account with a third-party vendor, activating the user's camera and/or microphone and covertly recording the user, and more.

G. Return-Oriented Programming

71. Return-oriented programming, often called ROP for short, is a hacking exploit that leverages vulnerabilities in call stacks. Since the call stack includes the addresses in memory where the code to be executed for functions/subroutines reside, the return-oriented programming exploit centers around a hacker gaining control of the call stack and manipulating the call stack to exploit a vulnerability in the program.

72. For example, if we consider the call stack discussed in Figure 2 above, the stack frame for function F3 in Figure 2e specifies the start address in memory where the code for F3 resides. When this stack frame is at the top of the stack, this address can be read off the top of the stack and control passes to the start address in memory of the code for F3 which can then start executing and performing the task it is supposed to perform. However, in a return-oriented programming exploit a

malicious adversary, knowing the vulnerabilities of the software or hardware gains access to the call stack and replaces the stack frame for F3 in the call stack with the start address of a different function (let's call this function BAD) instead of the start address for F3. Then the function BAD will be executed instead of F3, leading to unintended results which could, in principle, be highly malicious.

73. ROP consists of a set of methods to achieve such behavior by manipulating one or more addresses in the call stack so that a malicious piece of code is executed rather than the piece of code intended for execution by the programmer of the original software. A typical ROP attack may split a complex attack into multiple smaller pieces (called "gadgets") so that the address where the gadget code is stored replaces a legitimate address in the call stack. Even if a single gadget in an ROP attack does not do anything malicious, chaining together a sequence of seemingly innocuous gadgets, each placed in the call stack, may lead to malicious activity. ROP attacks use this idea to manipulate the call stack and get what appears to be the normal execution of code to unintentionally end up generating malicious acts.

H. Monitoring Memory Spaces

74. As far back as January 2006, tools were known to monitor memory spaces of processes. In one example, Despres discloses that "[t]here are several tools that allow

us to measure the memory space used between two points of execution of a program." Ex. 1018, p. 11.

75. Further, prior to 2012, there were a number of tools available to monitor the memory space of a process for execution of a function. For instance, in 2007, Payne et al. proposed a monitoring system called XenAccess and stated that "[o]ur performance testing shows that our address translation, memory copying, and disk I/O monitoring functions have small overheads, making these capabilities effective for a variety of monitoring applications," such that methods such as XenAccess were already capable of monitoring a variety of functions. Ex. 1019, p. 396.

76. Payne further states that ". . . we show several example applications to demonstrate our monitoring capabilities. While reading through these examples, keep in mind that we are deliberately showing simple use cases as an introduction to the library. However, XenAccess provides complete read/write access to a VM's memory space and has complete access to the disk I/O. With this level of power, the potential applications are only limited by the user's imagination." Ex. 1019, p. 394. Accordingly, monitoring memory spaces was well within the knowledge of one of ordinary skill in the art.

VII. OVERVIEW OF THE PRIOR ART

A. OVERVIEW OF "RUNTIME PREVENTION OF RETURN-ORIENTED PROGRAMMING ATTACKS" TO FRATRIC ("FRATRIC")

77. Fratric is a whitepaper entitled "Runtime Prevention of Return-Oriented Programming Attacks" that describes a system called ROPGuard that can detect and prevent return-oriented programming (ROP) attacks. Ex. 1007, p. 1. The GitHub repository in which Fratric is stored is a public repository because of its "Public" repository visibility and can be viewed by anyone, including people who aren't GitHub users. The Fratric reference was publicly available as of the last commit date of August 26, 2012, because GitHub assigns a unique commit ID and date as of the date the document was committed.

78. Fratric explains that "return-oriented programming is a popular memory vulnerability exploitation technique that can be used to bypass the existing mitigation techniques such as the data execution prevention" and teaches a "system called ROPGuard that can detect and prevent the currently used forms of ROP attacks." *Id.*

79. Fratric explains that an "attacker must use ROP code to leverage the attack and in this process interact with other processes or the OS kernel" by "creating other processes, opening and writing to files etc." Ex. 1007, p. 4. As explained by Fratric, such interaction is done using a critical function: "a function by executing which the

attacker can leverage the attack, either by making modifications to the memory of the current process such that using ROP is no longer necessary or by 'escaping' the current process." *Id.* As explained by Fratric, "[I]n order to exploit the vulnerability, the attacker will need to call at least one critical function *from the ROP code*. ROPGuard uses this observation to perform the checks only when one of the critical functions gets called." *Id.*, emphasis in original.

80. Fratric describes at least six checks which can be performed at each critical function call to determine if the critical function was called from the ROP code. Ex. 1007, p. 4.

81. Fratric describes a first check as checking the stack pointer. As explained by Fratric, in "most forms of ROP attack, the attacker will need to have control over the stack." *Id.* As explained by Fratric, "at each critical function call, ROPGuard checks if the stack pointer is inside the boundaries of the current thread's stack. If it is not, this is a strong indication of the ongoing ROP attack." *Id.*

82. Fratric describes a second check as looking for the address of a critical function on the stack. As explained by Fratric, "[i]f a critical function was entered via RETN to its address instead of via a CALL (or a JUMP) instruction, then, at the moment of entering the critical function, the address of the critical function must be on the stack just above the current stack pointer." *Id.* at 5. As explained by Fratric, "ROPGuard saves a certain number of DWORDS on the stack from being modified

and later checks if the address of the current critical function can be found there. If it can, then it is possible that the critical function got entered via RETN, which could indicate that a ROP attack is taking place." *Id.*

83. Fratric describes a third check as checking the return address of the critical function. Each return address must be executable and the instruction at the return address must be preceded with a CALL instruction. *Id.* As explained by Fratric, "ROPGuard can also verify that the target of the CALL instruction preceding the return address is the same as the address of the current critical function." *Id.*

84. Fratric describes a fourth check as checking the stack frames. As explained by Fratric, "If the program is compiled in such a way that it uses EBP register as a stack frame pointer, this information can be used to check the consistency of the information on the stack and to check the return addresses of the functions the current critical function was called from if they also use the frame pointer." *Id.*

85. Fratric presents a stack walk as described in pseudocode in Listing 2.

```
If the current frame pointer does not point on the stack
    |   If the RequireFramePointers option is true
    |         Alert the user about the ROP attempt;
    |   Else
    |         Return;
If the frame pointer points above the stack pointer
      Alert the user about the ROP attempt;
For the number of frames specified in the configuration file
    |   Get the return address of the current frame; //below the frame ptr.
    |   If the return address is null
    |         Return;           //reached the bottom of the stack
    |   If the return address is not executable or not preceded by call
    |         Alert the user about the ROP attempt;
    |   Acquire the new frame pointer //pointed to by the current frame ptr.
    |   If the new frame pointer is not below the previous frame pointer
    |   or does not point on the stack
    |       |   If the RequireFramePointers option is true
    |       |         Alert the user about the ROP attempt;
    |       |   Else
    |               Return;
```

Listing 2: Pseudocode of the function that performs the stack frames checking

Id. at 5-6.

86. Fratric describes a fifth check as simulating the execution flow of a small number of instructions after the return from the critical function. *Id.* at 6. "ROPGuard simulates the instructions that change the stack pointer (PUSH, POP, ADD ESP, #, SUB ESP, #, RETN) and tracks the changes that these instructions will make to the value of the stack pointer. The remaining instructions are simply stepped over. Once RETN instruction is encountered, ROPGuard will check if the return address is executable and if the instruction at the return address is preceded by the call instruction." *Id.*

87. Fratric describes a sixth check as a function-specific check related to the arguments of the critical function called. "ROPGuard can prevent VirtaulProtect [sic] calls in which the program attempts to change the memory protection options

of the stack." *Id.* at 7. "ROPGuard can prevent LoadLibrary (and similar) calls in which a program attempts to load a library over SMB." *Id.*

88. Fratric can detect a ROP attack based on these checks and subsequently prevent the ROP attack. *Id.* at 10.

B. OVERVIEW OF U.S. PATENT PUBLICATION NO. 2012/0255018 TO SALLAM ("SALLAM")

89. Sallam describes computer security and malware protection by securing memory and storage of an electronic device with a below-operating system security agent. Ex. 1008, ¶0001.

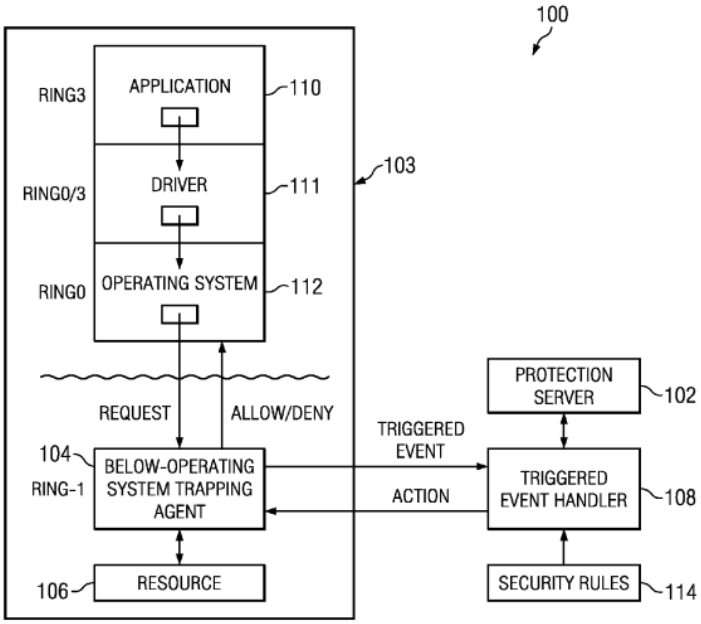


FIG. 1

Ex. 1008, FIG. 1.

90. Sallam describes using "security rules to determine a criteria by which an attempted access involving a transfer of content between memory and storage of an electronic device will be trapped." Ex. 1008, ¶0007. As I described above, the trapping described in Sallam is an example of code hooking.

91. If the criteria is met, Sallam traps the attempted access of data between the memory and the storage of the electronic device "at a level below all of the operating systems of the electronic device accessing a memory or a storage of an electronic device." *Id.*

92. Sallam describes identification of a particular resource of a processor that is to be trapped, including cache resources. As described in Sallam, "[a]s yet further examples, identifier 1007 may identify that the resource is one of other resources like a cache such as a translation lookaside buffer; a counter such as a time stamp counter; a logical core such as processor0, processor1 . . . processorN of the system." Ex. 1008, ¶0211. As I described above, cache is an example of memory address space of a process invoked in Sallam.

93. Sallam describes determining if an attempted access of data was indicative of malware. As described in Sallam, "SVMM security agent 217 may be configured to set the operation of SVMM 216 in terms of what actions will cause a trap or trigger. In one embodiment, SVMM 216 may be configured to communicate the detection of trapped actions to SVMM security agent 217. SVMM security agent 217 may be

configured to consult security rules 222 to determine whether the trapped actions indicate malware or malicious activities, and based upon security rules 222 may provide indications to SVMM 216 about what subsequent action to take. Such subsequent action may include allowing the attempted action, disallowing the attempted action, or taking other corrective steps." Ex. 1008, ¶0054.

VIII. IDENTIFICATION OF HOW THE CLAIMS ARE UNPATENTABLE

A. Grounds 1A/1B: Claims 1, 3, 4, 6, 9, 11, 12, 14, and 17 are anticipated under 35 U.S.C. § 102 or obvious under 35 U.S.C. § 103 by Fratric

1. Claim 1

[1.0] A computer-implemented method comprising:

94. I understand that a preamble typically does not limit a claim. However, to the extent that the preamble is limiting, it is my opinion that it is disclosed by Fratric. One of ordinary skill in the art would understand "computer-implemented" to mean using a computer program on a computer.

95. Fratric is directed to "[a] system called ROPGuard that can detect and prevent the currently used forms of ROP attacks." Ex. 1007, p. 1. From this statement alone, one of ordinary skill in the art would understand that the methods of detection and prevention of ROPGuard are implemented on a computer, as in the preamble of claim 1.

96. Fratric further describes implementation of the ROPGuard computer program on a computer. In particular, "ROPGuard consists of three files: an executable ('ropguard.exe'), a dll ('ropguarddll.dll') and a configuration file ('ropsettings.txt')." Ex. 1007, p. 7.

97. Fratric provides a number of examples of the computing benefits of the ROPGuard computer program, such as very low computing and memory overhead. *Id.* In one example, Fratric describes testing ROPGuard's computing overhead using a series of benchmarks using "a computer with an i7 720QM processor and 4GB RAM running Windows 7 64-bit." Ex. 1007, p. 10. Therefore, one of ordinary skill in the art would understand that Fratric describes a computer-implemented method.

[1.1.a] monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions,

98. It is my opinion that Fratric discloses each part of this limitation.

99. Limitation [1.1.a] first requires "execution of at least one monitored function of a plurality of functions."

100. Fratric describes the concept of a critical function as "a function by executing which the attacker can leverage the attack, either by making modifications to the memory of the current process such that using ROP is no longer necessary or by 'escaping' the current process." Ex. 1007, p. 4. "[A]n attacker will need to call at

least one critical function from the ROP code." *Id.* One of ordinary skill in the art would have understood the call of at least one critical function from the ROP code is the execution of the at least one monitored function that the ROPGuard computer program operates on.

101. Fratric describes several examples of critical functions including CreateProcess, VirtualProtect, VirtualAlloc, LoadLibrary, OpenFile, and WriteFile. *Id.* at 4. These examples of critical functions are the "plurality of functions."

102. The critical functions described in Fratric interact with other processes or the OS kernel, which makes them useful to attackers in ROP attacks. Examples of such interaction include creating other processes, opening files, and writing to files.

103. Limitation [1.1.a] next requires "monitoring a memory space of a process."

104. As I explained above with respect to processes and address spaces, one of ordinary skill in the art would understand that "memory space" includes the storage space in a computer where data is to be processed and instructions required for processing are stored. As I also explained above, monitoring a memory space was well known as of the priority date of the '591 patent. *See* Ex. 1018; Ex. 1019. Fratric further utilizes these concepts.

105. Fratric describes that ROPGuard "can be applied at runtime to any process." Ex. 1007, p. 1. One of ordinary skill in the art would have understood that ROPGuard

is applied to the memory space of the critical function during the critical function execution via the described runtime application to a process.

106. In particular, Fratric describes monitoring a memory space with six examples "which can be performed at each critical function call." Ex. 1007, p. 4.

107. Fratric describes checking a stack pointer. As I explained above, when a new stack is created by a program, the operating system reserves a space in memory in which to store the stack. As I also explained, ROP attacks are a hacking paradigm that leverages call stacks. Accordingly, in order to detect attacker control over the call stack, "ROPGuard checks if the stack pointer is inside the boundaries of the current thread's stack." *Id.* In looking for the address of critical function on the stack, "ROPGuard saves a certain number (specified in the configuration file) of DWORDS on the stack from being modified and later checks if the address of the current critical function can be found there." *Id.* One of ordinary skill in the art would understand that checking the stack pointer is monitoring a memory space of a process of the executed critical function.

108. Fratric also describes checking a return address. As I explained above with respect to the call stack structure, and as Fratric explains further, "if a critical function was entered via RETN to its address instead of via a CALL (or a JUMP) instruction, then, at the moment of entering the critical function, the address of the critical function must be on the stack just above the current stack pointer." *Id.* at 5.

If the address is on the stack just above the current stack pointer, it is an indication of a ROP attack.

109. In order to detect a likely ROP attack via an inappropriate entering of a critical function, ROPGuard can evaluate the return address. "Each return address must be executable and the instruction at the return address must be preceded with a CALL instruction." *Id.* One of ordinary skill in the art would understand that checking a return address on a stack is monitoring a memory space of a process of the executed critical function.

110. Fratric also describes checking stack frames. As I explained above with respect to stacks and stack pointers, and as Fratric explains further, "[i]f the program is compiled in such a way that it uses EBP register as a stack frame pointer, this information can be used to check the consistency of the information on the stack and to check the return addresses of the functions the current critical function was called from if they also use the frame pointer." *Id.*

111. In order to detect a likely ROP attack when EBP is the stack pointer, "at any frame EBP pointing to the location outside the stack is encountered, this is a strong indication that the stack is corrupted and that the attack is in progress." *Id.* One of ordinary skill in the art would understand that checking stack frames on a stack is monitoring a memory space of a process of the executed critical function.

112. Fratric also describes simulating execution flow. Fratric explains that the instructions in ROP gadgets can be evaluated. *Id.* at 5-6. "ROPGuard simulates the instructions that change the stack pointer (PUSH, POP, ADD ESP, #, SUB ESP, #, RETN) and tracks the changes that these instructions will make to the value of the stack pointer. The remaining instructions are simply stepped over. Once RETN instruction is encountered, ROPGuard will check if the return address is executable and if the instruction at the return address is preceded by the call instruction." *Id.* at 6. One of ordinary skill in the art would understand that simulating execution flow for instructions that change the stack pointer is monitoring a memory space of a process of the executed critical function.

113. Fratric also describes function-specific checks. "ROPGuard can prevent VirtaulProtect [sic] calls in which the program attempts to change the memory protection options of the stack." *Id.* at 7. "ROPGuard can prevent LoadLibrary (and similar) calls in which a program attempts to load a library over SMB." *Id.* One of ordinary skill in the art would understand that function-specific checks including changes to the memory protection options of the stack and load library calls are monitoring a memory space of a process of the executed critical function.

[1.1.b] wherein monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space;

114. It is my opinion that Fratric discloses each part of this limitation.

115. Fratric discloses "[w]hen ROPGuard is invoked to protect an existing process it will inject its dll into the address space of the target process. This dll will then perform all of the work related to the protection of the process and the main ROPGuard application (ropguard.exe) will exit. ROPGuard uses CreateRemoteThread method for injecting its dll into the target process." *Id.* at 8.

116. As I explained above, when processes are launched, the operating system would allocate a section of memory to the process (its "address space"). One of ordinary skill in the art would understand that the injection of ROPGuard dll into the address space of the target process is loading a component for evaluating the critical function in the memory space.

117. In particular, a dynamic linked library is "a programming concept in which shared libraries with special functionalities are launched only during program execution, which minimizes overall program size and facilitates improved application performance for reduced memory consumption." Ex. 1020; *See* also Ex. 1021, p. 326: "A dll is typically a C program with a number of functions in it. However, instead of compiling this program into an executable form, Microsoft Windows allows the compilation of the program into a dll. A unique feature of a dll

is that it allows the linking of the program (or the library) into any other program at run time instead of at compile time."

118. It is important to note that the functions within a DLL are not restricted. Thus, in the case of the DLL injected by ROPGuard, the functions in the DLL can analyze the behavior of the target process in any number of ways and also protect the target function in diverse ways.

119. Fratric further describes an inline-patch example for critical functions defined in the configuration file to perform appropriate ROP checks when called. For example, a header of the critical function is determined and saved, a patched function header is constructed in the executable memory region allocated previously by ROPGuard, and the function header is replaced by a jump instruction to the patched function header. Ex. 1007. One of ordinary skill in the art would further understand the inline-patch example to be a further example of loading a component for evaluating the at least one monitored function in the memory space.

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

120. It is my opinion that Fratric discloses each part of this limitation.

121. Fratric discloses that "the attacker will need to call at least one critical function from the ROP code. ROPGuard uses this observation to perform the checks only

when one of the critical functions gets called: when a critical function gets called, the appropriate checks will be performed to determine if the critical function was called from the ROP code or as a part of normal program execution." Ex. 1007, 4.

122. In programming, an instance is an occurrence of a class or object of a computer program. As I further explained above, when an algorithm from the computer program is being executed, a function in the algorithm's code may be invoked. Accordingly, one of ordinary skill in the art would understand that a plurality of functions are invoked as a result of receiving a call from the software program that is monitoring the memory space. In particular, the claimed application programming instance is the ROP code, and a critical function is itself invoked as a result of the call from the ROP code.

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

123. It is my opinion that Fratric discloses each part of this limitation.

124. The '591 patent 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, 10:35-39.

125. As I explained above, stack walking processing or "stack walking" refers to the process of unwinding the stack frames in the call stack to identify individual

stack frames. Stack walking was already a well-known concept as of the date of the '591 patent.

126. For example, in 2004, and referring to Exhibit 1022, Feng et al. explained stack walking, "[f]or example, the monitor can extract information about the stack activity of the program by walking the call stack. Our VPStatic model, a statically-constructed variant of the VtPath model, implements this technique." *See* Ex. 1022, p. 2.

127. In another example, in 2006, and referring to Exhibit 1023, Li et al. devoted a subsection (Sec. 3.4) to stackwalking, explaining, "[e]ventually, BASS chooses to maintain a shadow stack of return addresses. More specifically, it instruments each function call site to push the return address before entering the callee, and pop the return address after returning from the callee. Consequently, BASS's sandboxing engine can easily identify the return address sequence associated with an incoming system call. As a side effect, it also enhances the application security by detecting any stack smashing." Ex. 1023, p. 5.

128. In another example, in 2010, and referring to Exhibit 2024, Roundy and Miller explained, "[o]ur analysis factory uses instrumentation to construct annotated program CFG's and a stackwalk at the program's first socket communication" and "[f]or the malware samples that attempted to communicate with the network, our analysis factory walked their call-stacks to identify the code in the malicious

executable that triggered the network communication." Ex. 1024, pp. 15, 18. Roundy and Miller further state that "Figure 6 shows our traversal of Conficker's call stacks at its first call to the select routine. As seen in this stack trace, we are able to identify the stack frames of functions that lack symbol information, an important benefit of our analysis capabilities." Ex. 1024, p. 18 Figure 6 is reproduced below.

top	pc=0x7c901231 DbgBreakPoint_0x7c901230 in ntdll.dll	[Win DLL]
	pc=0x10003c83 DYNbreakPoint_0x10003c70 in dyn_RT.dll	[Instrument.]
	pc=0x100016f7 DYNstopThread_0x10001670 in dyn_RT.dll	[Instrument.]
	pc=0x71ab2dc0 select_0x71ab2dc0 in WS2_32.dll	[Win DLL]
base	pc=0x401f34 func=nosym1f058_0x41f058 in cf.exe	[Conficker]

Fig. 6: An SD-Dyninst stack walk taken when the Conficker A binary executes Winsock's `select` routine. The stack walk includes frames from our instrumentation, `select`, and Conficker.

129. With the understanding that stack walking was very well known, and with further reference to Fratric, ROPGuard walks the stack by (iteratively for the number or frames specified in the configuration file) retrieving the return address of the current frame, conducting its checks, and acquiring the new frame pointer for the next iteration. In this way, ROPGuard reverse walks the stack frame-by-frame to the bottom of the stack towards the originating function.

130. Fratric discloses stack walking in an "algorithm for performing stack frame checks" in Listing 2 as annotated below.

```
If the current frame pointer does not point on the stack
    | If the RequireFramePointers option is true
    |       Alert the user about the ROP attempt;
    | Else
    |       Return;
If the frame pointer points above the stack pointer
    Alert the user about the ROP attempt;
For the number of frames specified in the configuration file
    | Get the return address of the current frame; //below the frame ptr.
    | If the return address is null
    |       Return;          //reached the bottom of the stack
    | If the return address is not executable or not preceded by call
    |       Alert the user about the ROP attempt;
    | Acquire the new frame pointer //pointed to by the current frame ptr.
    | If the new frame pointer is not below the previous frame pointer
    | or does not point on the stack
    |   | If the RequireFramePointers option is true
    |   |       Alert the user about the ROP attempt;
    |   | Else
    |   |       Return;
```

Listing 2: Pseudocode of the function that performs the stack frames checking

Ex. 1007, pp. 5-6.

131. Listing 2 above shows stack walking in action applied to a "current frame". If the return address is not executable or not preceded by call, this means that the return address is likely to be a function that was never encountered when the call stack was created, suggesting an ROP attack is being attempted by diverting to a function that was never intended to be invoked. If that is not the case, then the current frame returns and a new frame pointer (corresponding to the function that invoked the function in the current frame) is now invoked. This pseudo code shows one iteration of the stack walking process.

132. Fratric also discloses executing the stack walk processing as part of the memory space monitoring discussed in [1.1.a], [1.1.b], and [1.2]. As I explained earlier, when a new stack is created by a program, the operating system reserves a

space in memory in which to store the stack. Fratric explains that "if the program is compiled in such a way that it uses EBP register as a stack frame pointer, this information can be used to check the consistency of the information on the stack and check not only the return address of the critical function (as described in the previous subsection), but also to check the return addresses of the functions the current critical function was called from if they also use the frame pointer." Ex. 1007, p. 5. One of ordinary skill in the art would understand from the program compiling and subsequent invocation of a function in the program to create the stack to walk that Fratric discloses executing stack walk processing upon the invocation of one of the monitored functions.

[1.4.a] 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,

133. It is my opinion that Fratric discloses each part of this limitation.

134. As I explained above, stack overflows can be leveraged by exploits. A POSITA would understand limitation [1.4.a] to refer to the fact that during the stack walk processing phase, memory checks must be made to ensure that suspicious

behavior is not occurring (e.g., stack overflows, unexpected functions being encountered in the stack even though they were not explicitly called).

135. Limitation [1.4.a] first requires executing of stack walk processing. As I discuss in [1.3], Fratric discloses executing of stack walk processing.

136. Limitation [1.4.a] also requires a memory check relative to an address of an originating caller function is reached. Fratric discloses "check not only the return address of the critical function" but also "to check the return addresses of the functions the current critical function was called from." *Id.* Fratric discloses evaluation of the return addresses in coordination with its stack walk processing, including before an address of an originating caller function is reached, as I will further explain.

137. Limitation [1.4.a] also requires a memory check for a plurality of stack entries identified during the stack walk. ROPGuard walks the stack using frame pointers ("Acquire the new frame pointer") to check a plurality of frames until the bottom of the stack is reached, as illustrated in the annotated Listing 2 of pseudocode below. Box 1 highlights operations for current frame; Box 2 highlights operations acquiring next frame and continuing the stack walk. *Id.*

Box 1 – Current

```
If the current frame pointer does not point on the stack
    If the RequireFramePointers option is true
          Alert the user about the ROP attempt;
    Else
          Return;
If the frame pointer points above the stack pointer
      Alert the user about the ROP attempt;
For the number of frames specified in the configuration file
    Get the return address of the current frame; //below the frame ptr.
    If the return address is null
          Return;           //reached the bottom of the stack
    If the return address is not executable or not preceded by call
          Alert the user about the ROP attempt;
    Acquire the new frame pointer //pointed to by the current frame ptr.
    If the new frame pointer is not below the previous frame pointer
    or does not point on the stack
        If the RequireFramePointers option is true
              Alert the user about the ROP attempt;
        Else
              Return;
```

Box 2 – Next Frame

Listing 2: Pseudocode of the function that performs the stack frames checking

138. One of the ordinary skill in the art would understand that a plurality of stack entries are identified in the stack walk. In the simplest example, a stack entry corresponding to the "current frame pointer" is evaluated according to the ROPGuard pseudocode. Then, a stack entry corresponding to the "new frame pointer" is evaluated according to the ROPGuard pseudocode.

139. Limitation [1.4.a] also requires a memory check on the identified frames in order to detect suspicious behavior. Fratric discloses several such memory checks. In the stack walk pseudocode, Fratric discloses a memory check of the current frame, "[i]f the current frame pointer does not point on the stack" and subsequently detecting suspicious behavior in response to the memory check by "[a]lert the user about the ROP attempt." This portion of pseudocode is copied below.

```
If the current frame pointer does not point on the stack
    |   If the RequireFramePointers option is true
    |         Alert the user about the ROP attempt;
```

Ex. 1007, Listing 2.

140. In the stack walk pseudocode, Fratric discloses another memory check of the current frame, "[i]f the return address is not executable or not preceded by call" and subsequently detecting suspicious behavior in response to the memory check by "[a]lert the user about the ROP attempt." This portion of pseudocode is copied below.

```
If the return address is not executable or not preceded by call
      Alert the user about the ROP attempt;
```

Ex. 1007, Listing 2.

141. In the stack walk pseudocode, Fratric discloses another memory check – for the new frame, "[i]f the new frame pointer is not below the previous frame pointer or does not point on the stack" and subsequently detecting suspicious behavior in response to the memory check by "[a]lert the user about the ROP attempt." This portion of pseudocode is copied below.

```
If the new frame pointer is not below the previous frame pointer
or does not point on the stack
    |   If the RequireFramePointers option is true
    |         Alert the user about the ROP attempt;
```

Ex. 1007, Listing 2.

142. One of ordinary skill in the art would have understood that Fratric's memory checks are done before reaching the address of an originating caller function (the bottom of the stack) by analyzing the stack frames that come before the bottom of the stack. As I explained above, when performing stack walk processing, stack frames are walked backwards from the lowest level function back to the initial invoked function. ROPGuard accordingly walks the stack frame-by-frame to the bottom of the stack towards the initial invoked function. One of ordinary skill in the art would have understood that Fratric's memory checks made during call stack walking as, for example, the "critical function" and "the functions the current critical function was called from" using the "current frame" and "new frame" are memory checks that are performed before the stack frame address of the originating caller function is reached in the stack walk. Ex. 1007, pp. 5-6.

[1.4.b] 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,

143. It is my opinion that Fratric discloses each part of this limitation.

144. Limitation [1.4.b] first requires an alert of suspicious behavior is triggered when performing of the memory check. As I discuss in [1.4.a], Fratric discloses executing of stack walk processing.

145. Limitation [1.4.b] also requires triggering of the alert of suspicious behavior based on at least one of several possible suspicions.

146. It is my understanding that when a claim covers several structures or compositions, either generically or as alternatives, the claim is deemed anticipated if any of the structures or compositions within the scope of the claim is known in the prior art. It is my opinion that Fratric discloses several of the alternatives recited, including 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, and the base pointer is detected as being outside a current thread stack.

147. Fratric discloses an alert based on code execution is attempted from non-executable memory. In particular, Fratric discloses "[a]t the entry point of each critical function, its return address can be easily acquired – it is on top of the stack. Each return address must satisfy the following conditions: - It must be executable." Ex. 1007, p. 5. One of ordinary skill in the art would have understood that the return address condition that the return address must be executable is an evaluation of code execution attempted from non-executable memory.

148. In another example of an alert based on code execution is attempted from non-executable memory, Fratric discloses, "ROPGuard can prevent VirtaulProtect [sic] calls in which the program attempts to change the memory protection options of the stack. Calling VirtualProtect to make stack executable is one of the common ways of leveraging ROP attacks." Ex. 1007, p. 7. One of ordinary skill in the art would have understood that detecting an attempt to change the stack from non-executable to executable is an evaluation of code execution attempted from non-executable memory.

149. Fratric discloses an alert based on a base pointer is identified as being invalid. In particular, Fratric discloses "if the program is compiled to use EBP only as the stack frame pointer, and at any frame EBP pointing to the location outside the stack

is encountered, this is a strong indication that the stack is corrupted and that the attack is in progress." Ex. 1007, p. 5. One of ordinary skill in the art would have understood that detecting a stack frame pointer pointing to a location outside the stack is identifying a base pointer as being invalid.

150. Fratric discloses an alert based on an invalid stack return address being identified. Fratric discloses "check not only the return address of the critical function," but also "check the return addresses of the functions the current critical function was called from." Ex. 1007, p. 5. More particularly, [t]he instruction at the return address must be preceded with a CALL instruction" and "the target of the CALL instruction preceding the return address is the same as the address of the current critical function." Ex. 1007, p. 5. In another example, Fratric also discloses a simulation memory check to "decode the instruction at the return address (NEG EBX), step over it and take the next instruction which is RETN. On encountering the RETN instruction, ROPGuard will check the return address. As, in this example, the return address is not executable, an attack attempt will be reported." Ex. 1007, p. 7.

151. One of ordinary skill in the art would have understood that checking return addresses as preceding with a CALL instruction, the target of the CALL instruction preceding the return address being the same as the address of the current critical

function, and the simulation RETN instruction check of the return address being executable are evaluations of whether the stack return address is valid.

152. Fratric discloses an alert based on a base pointer is detected as being outside a current thread stack. Fratric discloses "at each critical function call, ROPGuard checks if the stack pointer is inside the boundaries of the current thread's stack" (base pointer is detected as being outside a current thread stack). Ex. 1007, p. 4. One of ordinary skill in the art would have understood that checking if the stack pointer is inside the boundaries of the current thread's stack is an evaluation of a base pointer being detected as being outside a current thread stack.

153. Fratric discloses an alert based on an attempted execution of a return-oriented programming technique is detected. Fratric discloses "[r]eturn-oriented programming is a popular memory vulnerability exploitation technique that can be used to bypass the existing mitigation techniques such as the data execution prevention. A system called ROPGuard was developed that can detect and prevent the currently used forms of ROP attacks." Ex. 1007, p. 10. One of ordinary skill in the art would have understood that ROPGuard detecting and preventing forms of ROP attacks is alerting based on attempted execution of a return-oriented programming technique being detected.

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

154. It is my opinion that Fratric discloses each part of this limitation.

155. Limitation [1.5] first requires an alert of suspicious behavior is triggered. As I discuss in [1.4.b], Fratric discloses that an alert of suspicious behavior is triggered.

156. Limitation [1.5] next requires preventing execution of a payload for the invoked function from operating based on the alert. Fratric discloses "[a] system called ROPGuard was developed that can detect and prevent the currently used forms of ROP attacks." Ex. 1007, p. 10.

157. In an example, "[a] special input file that will cause the buffer overflow, execute the ROP code given in Listing 1 and after that execute the example payload ('calc.exe' will be started on successful exploitation) was crafted. When the application is protected with ROPGuard . . . when the crafted input (available as 'vulnapp-input-rop.txt') is given to the vulnerable application and the application is protected using ROPGuard, ROPGuard will detect the exploitation attempt successfully." Ex. 1007, p. 9. One of ordinary skill in the art would have understood that based on the disclosed purpose of ROPGuard to detect and prevent ROP attacks and the example of calc.exe exploit detection that Fratric prevents execution of a payload for the invoked function from operating when an alert of suspicious behavior is triggered.

158. Thus, claim 1 is anticipated or rendered obvious over Fratric.

2. Claim 3

[3.0] The method according to claim 1,

159. See [1.0]-[1.5].

[3.1] 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.

160. It is my opinion that Fratric discloses the additional limitations of dependent claim 3.

161. As I explained above with respect to [1.3] and [1.4.a], Fratric's memory checks are done before reaching the address of an originating caller function (the bottom of the stack) by analyzing the stack frames that come before the bottom of the stack. As I further explained, when performing stack walk processing, stack frames are walked backwards from the lowest level function back to the initial invoked function. ROPGuard accordingly walks the stack frame-by-frame to the bottom of the stack towards the initial invoked function. See Ex. 1007, Listing 2. My analysis of [1.3] and [1.4.a] is also incorporated herein.

162. More particularly, as I also explained above with respect to my example in Section VI, a function F1 that is invoked by an application may invoke a lower-level function F2 which in turn may invoke another lower-level function F3, and so forth. Such a stack walk may be performed if, for example, F3 attempted to perform a suspicious operation, and a malware analyst or analysis algorithm wanted to trace the function that originally invoked F3. In this case, stack walk processing involves walking back the stack from F3 to F2 to F1 to uncover the fact that F1 is the function that originally called F3.

163. Fratric also explicitly explains that ROPGuard "protects only the functions in the user mode." Ex. 1007, p. 4. And further, "critical functions are defined in ROPGuard's configuration file. In this way, critical functions can be added at any time to improve security and even process-specific critical functions can be added." *Id.* Fratric's disclosed protection of user mode functions via critical functions further emphasizes the relationship of such functions, such as on a stack. In particular, the "low level user mode function" is a subsequent call (which can itself call a critical function), while the originating function is a different invoked function.

164. One of ordinary skill in the art would have understood, as I have explained with respect to how the stack is organized and traversed, that Fratric's stack walk works towards the bottom of the stack and the originating function or walking *from* a low level user mode function to the originating address of the invoked function.

Therefore, Fratric discloses executing of the stack walk processing by walking a stack frame associated with the invoked function from a low level user mode function to the originating address of the invoked function.

165. As I have explained, one of ordinary skill in the art would have expected success in walking a stack frame associated with the invoked function from a low level user mode function to the originating address of the invoked function, given Fratric's disclosure, because Fratric's disclosed stack walking is the use of known components in accordance with their conventional uses in a predictable art.

166. Thus, claim 3 is anticipated or rendered obvious over Fratric.

3. Claim 4

[4.0] The method according to claim 1,

167. See [1.0]-[1.5].

[4.1] wherein the monitoring further comprises monitoring functions associated with memory management in the memory space by a user mode function.

168. It is my opinion that Fratric discloses the additional limitations of dependent claim 4.

169. One of ordinary skill in the art would understand the difference between user-mode functions and kernel-mode functions. For example, as was known in the art in

2002, "[i]n traditional operating systems, a kernel is protected from user programs by a privilege level facility and an MMU of a CPU. First, user programs and a kernel are separated in different privilege levels of a CPU: the user programs are in the user mode, which is the least privileged level, and the kernel is in the kernel mode, which is the most privileged level. Then, the kernel is mapped in memory as a privileged region which only programs in the kernel mode can access, by a page table of an MMU in the CPU." Ex. 1025, p. 1. Thus, functions in user-mode operate at a lower level of privilege than kernel-model functions. A POSITA would understand this limitation to reflect monitoring of functions of the user programs.

170. As I have explained for a number of limitations including for claims 1 and 3, Fratric discloses monitoring including monitoring user mode functions back to originating functions with the call stack. For example, Fratric discloses that ROPGuard "protects only the functions in the user mode." Ex. 1007, p. 4. My analysis for [1.0]-[1.5] and [3.1] is incorporated herein. One of ordinary skill in the art would understand that the call stack monitoring of Fratric is monitoring functions associated with memory management in the memory space.

171. In another example, Fratric discloses a simulation algorithm for evaluating memory management in a memory space. Fratric discloses "[a]s an example of this simulation algorithm and its usefulness, let's assume that the critical function got called from the following ROP gadget CALL [EAX]; NEG EBX; RETN; and that

stack is aligned so that RETN in this gadget will direct program flow to a non-executable area of memory (for example, the current critical function is VirtualProtect and the attacker's intention is to use it to make some memory page executable and then return into this memory page." Ex. 1007, p. 7. One of ordinary skill in the art would understand that an attempt to change a non-executable area of memory to executable is associated with memory management in the memory space by a protected user mode function.

172. As I have explained, one of ordinary skill in the art would have expected success in monitoring functions associated with memory management in the memory space by a user mode function, given Fratric's disclosure, because Fratric's disclosed monitoring is the use of known components in accordance with their conventional uses in a predictable art.

173. Thus, claim 4 is anticipated or rendered obvious over Fratric.

4. Claim 6

[6.0] The method according to claim 1,

174. See [1.0]-[1.5].

[6.1] 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.

175. It is my opinion that Fratric discloses the additional limitations of dependent claim 6.

176. This limitation lists a set of functions within the Windows kernel API. For instance, the NtAllocateVirtualMemory function is used to commit memory to a process, the NtProtectVirtualMemory function changes the memory permissions of a page, the NtCreateProcess. NtCreateProcessEx and NtCreateUserIProcess are variants of each other that seek to create a new user process. LdrLoadDll loads a dynamic linked library into a user process. One of ordinary skill in the art would understand the purpose of monitoring these functions. *See* Ex. 1026; s*ee also* Ex. 1027.

177. Fratric discloses "examples of critical functions include: - CreateProcess – by calling which the attacker can create another process which can then be used to perform some malicious action and compromise the user's system - VirtualProtect, VirtualAlloc, LoadLibrary – using which the attacker can make arbitrary code executable and no longer need to use ROP code in later stages of the exploit. -

OpenFile – by calling which the attacker can open and possibly write to arbitrary file (using WriteFile function).” Ex. 1007, p. 4.

178. One of ordinary skill in the art would understand that the Fratric disclosure of VirtualAlloc is the same or equivalent to the claimed NtAllocateVirtualMemory function because both functions can be used to allocate new blocks of committed memory.

179. One of ordinary skill in the art would understand that the Fratric disclosure of VirtualProtect is the same or equivalent to the claimed NtProtectVirtualMemory function because both functions change the permissions on a page in memory.

180. One of ordinary skill in the art would understand that the Fratric disclosure of CreateProcess is the same or equivalent to the claimed NtCreateProcess function because both functions create a new process.

181. One of ordinary skill in the art would understand that the Fratric disclosure of CreateProcess is the same or equivalent to the claimed NtCreateProcessEX function because both functions create a new process.

182. One of ordinary skill in the art would understand that the Fratric disclosure of CreateProcess is the same or equivalent to the claimed NtCreateUserIProcess function because both functions create a new process.

183. One of ordinary skill in the art would understand that the Fratric disclosure of LoadLibrary is the same or equivalent to the claimed LdrLoadDll function because

both functions can be used to load a library module into the address space of the process.

184. As I have explained, one of ordinary skill in the art would have expected success in monitoring the plurality of functions including an NtAllocateVirtualMemory function, an NtProtectVirtualMemory function, an NtCreateProcess function, an NtCreateProcessEX function, an NtCreateUserIProcess function, and an LdrLoadDll function, given Fratric's disclosure, because Fratric's disclosed examples of critical functions and their corresponding monitoring are the use of known components in accordance with their conventional uses in a predictable art.

185. Thus, claim 6 is anticipated or rendered obvious over Fratric.

5. Claim 9

[9.0] A system comprising:

186. I understand that a preamble typically does not limit a claim. However, to the extent that the preamble is limiting, it is my opinion that it is disclosed by Fratric.

187. Fratric discloses "[a] system called ROPGuard that can detect and prevent the currently used forms of ROP attacks. The system can be applied at runtime to any process and has a very low computing and memory overhead." Ex. 1007, Abstract.

From this statement alone, one of ordinary skill in the art would understand that Fratric discloses a system.

[9.1] at least one memory; and

188. It is my opinion that Fratric discloses each part of this limitation.

189. One of ordinary skill in the art would understand that as of the date of the '591 patent, different forms of memory with varying speeds were widely available (e.g., normal memory, higher speed caches). For instance, among non-volatile memory alone, Dong et al. references many types of memory such as ". . . spin-torque-transfer memory (STT-RAM, or MRAM), phase-change random-access memory (PCRAM), and resistive random-access memory (ReRAM)." Ex. 1028, p. 994.

190. Fratric discloses implementation using memory. Fratric provides several examples of ROPGuard benefits, including for memory usage. Ex. 1007, p. 1. In one example, Fratric describes testing ROPGuard's computing overhead using a series of benchmarks using a computer with an i7 720QM processor and 4GB RAM running Windows 7 64-bit." Ex. 1007, p. 9. In another example, "[a]dditional memory used by ROPGuard, such as the memory used for patched function headers, configuration and the executable module cache is very small (4 kB for patched function headers and approximately 5 kB for the configuration and cache." Ex. 1007, p. 10.

191. Therefore, one of ordinary skill in the art would understand that the disclosure in Fratric of 4GB RAM and additional memory such as the memory used for patched function headers, configuration and the executable module cache describes at least one memory.

[9.2] at least one processor connected with the memory configured to perform operation of:

192. It is my opinion that Fratric discloses each part of this limitation.

193. One of ordinary skill in the art would understand that standard systems including a processor with memory configured to perform a very wide range of computing tasks were well known to a POSITA many years prior to the priority date of the '591 patent.

194. Fratric discloses several examples of ROPGuard implementation on a processor and its connected memory. For example, "[b]enchmarks were performed on a computer with an i7 720QM processor and 4GB RAM running Windows 7 64-bit." Ex. 1007, p. 9. ("[d]ue to the fact that i7 processor used in the experiments can change its clock frequency at runtime, the results may vary for different runs. This is especially visible during the CPU-heavy tests like in the SuperPI benchmark where the best run with protection was faster than the best run without protection.")

195. Therefore, one of ordinary skill in the art would have understood that Fratric's ROPGuard system includes at least one processor, such as the i7 720QM processor connected to memory, such as the described 4GB RAM, to perform the ROPGuard operations.

[9.3.a] monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions,

196. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.1.a].

[9.3.b] wherein monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space;

197. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.1.b].

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

198. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.2].

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

199. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.3].

[9.6.a] 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,

200. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.4.a].

[9.6.b] 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,

201. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.4.b].

[9.7] wherein when an alert of suspicious behavior is triggered, the method further comprising preventing execution of a payload for the invoked function.

202. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.5].

203. Thus, claim 9 is anticipated or rendered obvious over Fratric.

6. Claims 11, 12, 14

204. I have reviewed claims 11, 12, and 14, which depend from claim 9, and it is my determination that these claims are substantially similar to claims 3, 4, and 6, respectively. It is my opinion that Fratric discloses each part of each limitation in

these claims for the same reasons I expressed above with respect to claim limitations [3.0]-[3.1] for claim 11, [4.0]-[4.1] for claim 12, and [6.0]-[6.1] for claim 14. Thus, claims 11, 12, and 14 are anticipated or rendered obvious over Fratric.

7. Claim 17

[17.0] A computer-readable storage device containing instructions, that when executed on at least one processor, causing the processor to execute a process comprising:

205. I understand that a preamble typically does not limit a claim. However, to the extent that the preamble is limiting, it is my opinion that it is disclosed by Fratric.

206. Fratric specifically teaches instructions on a computer-readable storage device, "ROPGuard consists of three files: an executable ('ropguard.exe'), a dll ('ropguarddll.dll') and a configuration file ('ropsettings.txt'). All three files must be in the same folder." One of ordinary skill in the art would have understood that the executable, dll, and/or configuration file are the instructions for execution by the processor, and the storage in the same folder is on a computer-readable storage device. Ex. 1007, p. 7.

207. Fratric further discloses instructions execution by at least one processor, causing the processor to execute a process. For example, Fratric discloses the

ROPGuard executes to measure "[b]enchmarks [] performed on a computer with an i7 720QM processor and 4GB RAM running Windows 7 64-bit." Ex. 1007, p. 9.

208. One of ordinary skill in the art would have understood that Fratric's ROPGuard system includes at least one processor, such as the i7 720QM processor connected to memory, such as the described 4GB RAM, where the instructions such as the executable, dll, and/or configuration file, would be contained for execution by the processor, such as in the same folder as computer-readable storage device, to execute the ROPGuard process.

[17.1.a] monitoring a memory space of a process for execution of at least one monitored function of a plurality of functions,

209. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.1.a].

[17.1.b] wherein monitoring the memory space comprises loading a component for evaluating the at least one monitored function in the memory space;

210. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.1.b].

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

211. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.2].

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

212. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.3].

[17.4.a] 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,

213. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.4.a].

[17.4.b] 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 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

214. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.4.b].

215. It is my opinion that limitation [17.4.b] contains one addition that is not present in the corresponding limitation of claim 9, 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. It is my understanding that when a claim covers several structures or compositions, either generically or as alternatives, the claim is deemed anticipated if any of the structures or compositions within the scope of the claim is known in the prior art. It is my opinion that Fratric already discloses several of the alternatives recited in [1.4.b], and it therefore is not necessary to satisfy the additional limitation in order for Fratric to anticipate or render obvious claim 17.

[17.5] wherein when an alert of suspicious behavior is triggered, preventing execution of a payload for the invoked function.

216. It is my opinion that Fratric discloses each part of this limitation for the same reasons I expressed above with respect to claim limitation [1.5].

217. Thus, claim 17 is anticipated or rendered obvious over Fratric.

B. Ground 2: Claims 2, 5, 7, 8, 10, 13, 15, and 16 are obvious under 35 U.S.C. § 103 over Fratric in view of Sallam

8. Reasons to Combine Fratric and Sallam

218. In my opinion, one of ordinary skill in the art would have been motivated to combine the teachings of Fratric and Sallam for multiple reasons. The obvious, beneficial, and predictable results of the combination include detecting and preventing different forms of ROP attacks. Doing so would have been within the POSITA's skillset and would produce beneficial, predictable results.

219. When considering the teachings of Fratric, a POSITA would also have considered the teachings of Sallam. Both Fratric and Sallam are analogous prior art in the same field of endeavor as the '591 patent – detecting and preventing memory vulnerability exploitation techniques. Ex. 1007, p. 1; Ex. 1008, Abstract, ¶¶0007-0009.

220. In my opinion, the combination of Fratric and Sallam would have simply combined prior art elements (utilizing an address associated with an invoked function for memory space evaluation in Fratric; cache memory space evaluation taught in Sallam) according to known devices (systems of memory space evaluation) to yield predictable results (detect and prevent different forms of ROP) as Fratric intended.

221. It is my opinion that a POSITA would have been specifically motivated to incorporate Sallam's cache checks to expand Fratric's ability to identify return-oriented programming in additional memory spaces. Fratric discloses memory checks primarily focused on evaluations of the stack, simulation, and function-specific checks. *See* Ex. 1007, pp. 4-7. Sallam discloses other memory checks, including for cache. *See* Ex. 1008, ¶0211. Including other areas of memory vulnerable to ROP attacks would make ROPGuard more robust.

222. More particularly, a POSITA would have been specifically motivated to incorporate Sallam's known techniques for detection of ROP in cache, including by hooking, because doing so would address a known inefficiency in Fratric.

223. As explained in Fratric, "[a]lso note that the RETN instruction is only one way to transfer the execution flow from one gadget to the other. Indirect jump (for example, JMP [EAX]) and indirect call instructions can be used for this purpose as well [3]. This versatility makes detecting ROP exploitation attempts difficult." Ex.

1007, p. 2. Further, Fratric explains, "[i]n the future work, this approach could be extended to simulate other instructions as well (including indirect CALL and JMP instructions)." *Id.* at 6. Therefore, Fratric concedes a problem of detecting an attack using data fetched into cache from an indirect jump.

224. In light of this shortcoming, a POSITA would be motivated to augment Fratric with a known technique for detection of ROP in cache. Sallam provides that technique. In particular, Sallam explicitly discloses detecting an attack using an indirect jump into a resource like cache. Sallam discloses, "[a]s other examples, identifier 1004 may identify that the resource is an instruction such as 'JMP,' 'JZ' (jump if condition is equal to zero), 'JNZ' (jump if condition is not equal to zero), 'MOV' (move a value), or 'SysEnter' (a fast call to a Ring 0 procedure). As yet further examples, identifier 1004 may identify that the resource is one of other resources like a cache." Ex. 1008, ¶0211.

225. Utilizing Sallam's techniques in Fratric's system would have not only been advantageous, but a POSITA would have had a reasonable expectation of success in producing the systems and methods of the claims through this combination. In particular, Sallam is explicit that its techniques are applicable between different types and portions of memory, demonstrating that one skilled in the art would have reasonably expected to successfully produce the systems and methods recited through such combination. Ex. 1008, ¶0284. Moreover, as Sallam and I have

explained, function hooking was a very well-known technique well before the priority date of the '591 patent. *See* Ex. 1029; Ex. 1030; and Ex. 1031.

226. A person of ordinary skill in the art would have seen that the system of Fratric was compatible for combining with Sallam. The system of Fratric is explained with an explicit comment that future work would benefit ROPGuard. Fratric explains, "[i]n the future work, the execution flow simulation, which is found to be one of the most promising directions of this work could be extended to simulate the full instruction set instead of just tracking the stack pointer changes and RETN instructions." Ex. 1007, p. 10. Fratric's simulation aims to "examine what will happen after the critical function gets called without relying on the frame pointers." Ex. 1007, p. 6. Fratric explains that execution flow simulation is one of the most promising areas to improve Fratric.

227. As I explained, hooking involves intercepting operating system calls to monitor the behavior of applications. The hooking of Sallam can be utilized to examine what will happen after the critical function gets called without relying on the frame pointers, as desired by Fratric.

228. Accordingly, the expectation of success is evident because the combination of Fratric and Sallam involved the use of known techniques and components in accordance with their conventional uses in a predictable art. A POSITA would have

expected success using Sallam's technique according to the respective established functions (ROP detection in cache, including for example, by hooking).

9. Claim 2

[2.0] The method according to claim 1, further comprising:

229. See [1.0]-[1.5].

[2.1] identifying an address associated with the invoked function; and

230. It is my opinion that Fratric alone or combined with Sallam discloses each part of this limitation.

231. As I explained above, when an algorithm from a computer program is being executed, a function in the algorithm's code may be invoked. Accordingly, one of ordinary skill in the art would understand a critical function is itself invoked as a result of the call from the ROP code.

232. Fratric discloses that a return address is identified for the critical function. "At the entry point of each critical function, its return address can be easily acquired – it is on top of the stack." Ex. 1007, p. 5. "Each return address . . . must be executable" *Id.* One of ordinary skill in the art would understand that the executable critical function is therefore necessarily invoked. Further, the return address acquired on the top of the stack is an address associated with the invoked function.

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

233. It is my opinion that Fratric alone or combined with Sallam discloses each part of this limitation.

234. Limitation [2.2] first requires preventing execution of a payload for the invoked function. As I discussed with respect to my analysis in [1.5], Fratric discloses preventing execution of a payload for the invoked function. My analysis of that limitation is incorporated herein.

235. Limitation [2.2] next requires preventing execution when the address of the invoked function address is found in a cache.

236. The ’591 patent explains that “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. The cache component 104 maintains a cache for a monitored process. A cache is a specific memory region allocated in each monitored process where information about monitored memory pages is tracked.” Ex. 1001, 6:31-40.

237. Fratric discloses many examples of evaluation of the address of the invoked function address in “hardware or software that stores data related to a memory region

for a monitored process" as defined by the '591 patent. Ex. 1007, pp. 4-7. As I explained above, when a new stack is created by a program, the operating system reserves a space in memory in which to store the stack. Accordingly, memory is allocated for the process to perform its computations, and the stack can be considered hardware or software that stores data related to a memory region for a monitored process.

238. As I have further explained, checks of Fratric including checking the stack pointer, looking for the address of the critical function on the stack, checking the return address, and checking the stack frames all operate with the stack and are examples of the claimed monitoring of a memory space when the address of the invoked function address is found in data related to a memory region for a monitored process.

239. To the extent Fratric does not explicitly disclose when the address of the invoked function address is found in a *cache*, Sallam explicitly discloses preventing execution of a payload for the invoked function when the address of the invoked function address is found in a cache.

240. As an initial matter, Sallam discloses preventing execution of a payload for an invoked function. Sallam discloses "[t]he security agents of system 400 may be configured to work together to prevent malware and its malicious operations" and "[r]esource identifiers 1007 may include an identification of a particular resource of

the processor that is to be trapped." Ex. 1008, ¶¶0109, 0211. One of ordinary skill in the art would understand that trapping particular resources to prevent malware and its malicious operations is preventing execution of a payload for a function invoked from the malware.

241. Sallam explicitly discloses that "[i]dentifier 1004 may identify that the resource is one of other resources like a cache." Ex. 1008, ¶0211. "In another example, a cache or pointer may be similarly triggered when it is written, written with a particular value, or read." Ex. 1008, ¶0214. One of ordinary skill in the art would understand that a suspicion of malware would be a write to cache with the invoked function address so that it may execute in cache.

242. The '591 patent further explains that "[m]emory used for a cache is allocated by the anti-exploit technology, for example by the anti-exploit component 102. The cache component 104 maintains a cache for a monitored process. A cache is a specific memory region allocated in each monitored process where information about monitored memory pages is tracked." Ex. 1001, 6:37-57. Sallam also discloses that "[b]elow-O/S trapping agent 920 may include a processor resource control structure ("PRCS") 926. PRCS 926 may be implemented in a record, data structure, table, or any other suitable structure. PRCS 926 may contain information specifying which instructions, information, or attempted access of processor resources 924 are to be trapped." Ex. 1008, ¶0208.

243. Accordingly, one of ordinary skill in the art would understand that Sallam's disclosure of trapping agent 920 allocating memory for its data control structure to store information about resource identifiers 1007 is a specific memory region in each monitored process where information about monitored memory pages is tracked, as explained in the '591 patent.

244. A person of ordinary skill in the art would have found it obvious to combine Fratric and Sallam to identify an address of the invoked function in cache and prevent execution of the payload for the invoked function. As I previously explained, Fratric already discloses identifying an address associated with the invoked function and preventing execution of a payload for the invoked function. I incorporate by reference my analysis in [2.1] and [1.4.b] and [1.5].

245. Fratric also already considers cache concepts like storing in cache for purposes of memory space evaluation. For example, "ROPGuard saves the state of all general-purpose registers at the point of entry of each critical function." Ex. 1007, p. 5. Accordingly, ROPGuard already stores items in cache. It would have been obvious to a person of ordinary skill in the art to retrieve information from cache stored by other operations (like address information) as another suspicion evaluation in order to further ROPGuard's abilities in another associated memory space.

246. As I have described above, Fratric describes techniques of ROP attack detection. Ex. 1007, p. 1. Sallam describes security rules to determine a criteria by

which an attempted access involving a transfer of content between memory and storage of an electronic device will be trapped. Ex. 1008, ¶0007.

247. Combining Fratric with Sallam is nothing more than applying Fratric's known techniques of utilizing an address associated with the invoked function for memory space evaluation of ROP attacks to Sallam's known system that utilizes a different memory space evaluation to yield the predictable results of a system that detects and prevents different forms of ROP attacks as Fratric intended.

248. One of ordinary skill in the art would have been motivated to make this combination in order to achieve Fratric's goal of detecting ROP attacks using the "key observation that, if we disregard the attacks that can be done fully within the context of the exploited process (such as, for example, XSS attacks), the attacker must use ROP code to leverage the attack and in this process interact with other processes or the OS kernel." Ex. 1007, p. 4. Sallam specifically utilizes this concept in its suspicion of malware based on an address of an invoked function found in cache. Specifically, the address of the invoked function found in cache is an attempt to interact with other processes or the OS kernel.

249. Thus, claim 2 is rendered obvious over Fratric and Sallam.

10. Claim 5

[5.0] The method according to claim 1,

250. See [1.0]-[1.5].

[5.1] wherein the monitoring further comprises applying, upon detecting a function call of one of the plurality of functions in the monitored memory space,

251. It is my opinion that Fratric alone or combined with Sallam discloses each part of this limitation.

252. Limitation [5.1] first requires detecting a function call of one of the plurality of functions in the monitored memory space. Fratric discloses that "the attacker will need to call at least one critical function" Ex. 1007, p. 4. As I have explained above, one of ordinary skill in the art would understand that an attacker calling at least one critical function is a function call of one of the plurality of functions in the monitored memory space.

253. Limitation [5.1] next requires operation (applying) upon detecting the function call. Fratric discloses that, "ROPGuard… perform[s] the checks only when one of the critical functions gets called: when a critical function gets called, the appropriate checks will be performed to determine if the critical function was called from the ROP code or as a part of normal program execution." Ex. 1007, p. 4. One

of ordinary skill in the art would understand that appropriate checks being be performed "when a critical function gets called" is the claimed "upon detecting a function call."

254. As I have explained above, and as exemplified by Fratric's ROPGuard, a number of techniques existed to monitor the memory space used by an invoked function. *See* Ex. 1031 ("[t]hus, it is reasonable to monitor the system services that a process requests in order to analyze its behavior").

[5.2] a hooking technique to hook a lowest level user mode function of the invoked function,

255. It is my opinion that Fratric alone or combined with Sallam discloses each part of this limitation.

256. As I explained above, at the lowest software levels, operating systems offer access to application program interfaces or APIs that are capable of performing low-level actions such as allocating specific memory addresses, writing or reading from protected memory, and more. Access to such APIs is typically reserved for highly trusted and privileged programs. Code hooking is a software technique that intercepts calls to such API functions in order to better study the behavior of the programs that are making such API calls.

257. To the extent Fratric does not explicitly disclose a hooking technique, Sallam explicitly discloses this well-known technique, and moreover, explicitly describes hooking a low-level user mode function as in [5.2]. For example, Sallam discloses, "some anti-malware solutions may hook various portions of an operating system to trap and evaluate low-level operations of the applications." Ex. 1008, ¶0090. In its implementation, Sallam discloses "in-O/S security agent 219 may hook or filter NDIS.SYS." Ex. 1008, ¶0049. Like Fratric, Sallam applies its operation in response to detecting a function call. For example, "in-O/S security agent 219 may hook . . . and monitor for file writes to a particular file." *Id.* Therefore, one of ordinary skill in the art would understand that Sallam's disclosure of hooking various portions of an operating system to trap and evaluate low-level operations of the applications is a hooking technique to hook a lowest level user mode function of the invoked function.

[5.3] wherein the executing of the stack walk processing executes a reverse stack walk from the lowest level user mode function of the invoked function.

258. It is my opinion that Fratric alone or combined with Sallam discloses each part of this limitation.

259. As I have explained with respect to several limitations above, Fratric discloses a reverse stack walk. ROPGuard walks the stack using frame pointers ("Acquire the

new frame pointer") to check a plurality of frames until the bottom of the stack is reached. In this way, ROPGuard reverse walks the stack frame-by-frame to the bottom of the stack towards the originating function, as illustrated by the annotated pseudocode below:

```
If the current frame pointer does not point on the stack
    | If the RequireFramePointers option is true
    |       Alert the user about the ROP attempt;
    | Else
    |       Return;
If the frame pointer points above the stack pointer
      Alert the user about the ROP attempt;
For the number of frames specified in the configuration file
    | Get the return address of the current frame; //below the frame ptr.
    | If the return address is null
    |       Return;          //reached the bottom of the stack
    | If the return address is not executable or not preceded by call
    |       Alert the user about the ROP attempt;
    | Acquire the new frame pointer //pointed to by the current frame ptr.
    | If the new frame pointer is not below the previous frame pointer
    | or does not point on the stack
    |   | If the RequireFramePointers option is true
    |   |       Alert the user about the ROP attempt;
    |   | Else
    |   |       Return;
```

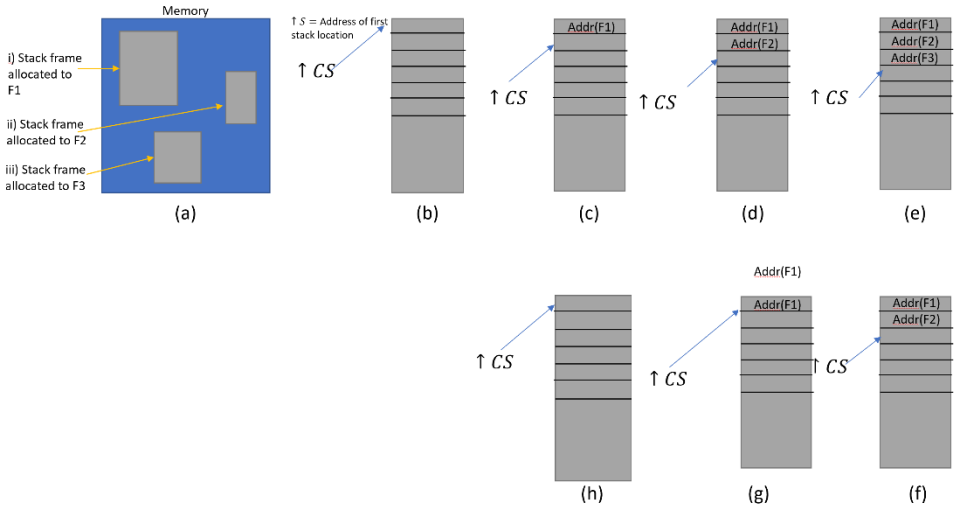
Listing 2: Pseudocode of the function that performs the stack frames checking

Ex. 1007, Listing 2. Further incorporated by reference is my analysis with respect to [1.3].

260. The '591 patent describes 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). Ex. 1001, 5: 34-39.

261. Referring again to the example I described above, if a current stack consists of F1 (at the bottom), F2 (in the middle) and F3 (at the top of the stack), as shown

in (e) below, F3 is the lowest level user mode function of the invoked function, as claimed.



262. Fratric discloses that "by using stack frames, we can find the stack frames of other possibly critical functions if they call the defined critical function" (like F3). Ex. 1007, p. 9. One of ordinary skill in the art would understand that the "possibly other critical functions called" are a lowest level user mode function of the invoked function because of how the stack is organized as illustrated above.

263. As illustrated in (e) above, the current stack pointer points to Addr(F3). Fratric stack walk pseudocode utilizes frame pointers relative to the current stack pointer; for example, "[i]f the frame pointer points above the stack pointer." Ex. 1007, Listing 2. Fratric thus begins its stack walk at the lowest level user mode function in order to evaluate all of the frames on the stack.

264. As I have explained earlier, it would have been obvious to one of ordinary skill in the art that the stack walk is executed as a reverse stack walk because of the

way the stack is organized and traversed (from the top of the stack towards the bottom of the stack) until the bottom of the stack is reached. Unwinding the stack frames in the call stack to identify the stack frames that potentially led to an attack involves checking all of the stack frames, starting from the top and walking to the bottom.

265. It would have been obvious to one of ordinary skill in the art to combine Fratric and Sallam to use a hooking technique to hook a lowest level user mode function and execute a reverse stack walk from the lowest level user mode function upon detecting a function call of one of the plurality of functions.

266. Fratric already discloses monitoring a memory space for a function call and conducting a reverse stack walk based on the call. See [1.3]. It would have been obvious to one of ordinary skill in the art to hook the function as described by Sallam as part of another function call evaluation.

267. As I have explained, code hooking is a software technique that intercepts calls to API functions in order to better study the behavior of the programs that are making such API calls. Sallam discloses hooking for an example purpose of "monitor[ing] for file writes to a particular file. SVMM security agent 216 may monitor input and output commands. If SVMM security agent 216 determined more writes than should have been seen based on the list of function calls seen by in-O/S

security agent 219, then malware may be clandestinely writing to disk outside of the functions provided by operating system 212." Ex. 1008, ¶0049.

268. Accordingly, hooking the lowest level user mode function as described by Sallam as part of another function call evaluation in order to study the behavior of the program making such calls prior would have provided another opportunity for the Fratric's ROPGuard to further evaluate the memory space.

269. Combining Fratric with Sallam is nothing more than applying Fratric's known techniques of monitoring a memory space after a function call to Sallam's known function hooking techniques for memory space monitoring to yield the predictable results of a system that detects and prevents different forms of ROP attacks as Fratric intended, and thus would have been obvious to one or ordinary skill in the art.

270. One of ordinary skill in the art would have been motivated to make this combination in order to achieve Fratric's goal of detecting ROP attacks using its key concept of performing checks when one of the critical functions gets called. Ex. 1007, p. 3. Performing checks only on critical function calls can create a very efficient protection mechanism in terms of performance overhead. Ex. 1007, p. 4. Sallam specifically utilizes this concept in its "monitor[ing] for file writes." Ex. 1008, ¶0049. As Fratric explains, a file write function is a critical function. Ex. 1007, p. 3.

271. Thus, claim 5 is rendered obvious over Fratric and Sallam.

11. Claim 7

[7.0] The method according to claim 5,

272. See [5.0]-[5.3].

[7.1] wherein performing the memory check comprises evaluating state information

273. It is my opinion that Fratric alone or combined with Sallam discloses each part of this limitation.

274. To the extent Fratric does not explicitly disclose evaluating state information, Sallam explicitly discloses performing a memory check by evaluating state information.

275. The '591 patent describes state information as "information about the allocation of memory pages for a monitored process and the initial permissions and type of the memory pages as well as any later changes of permissions for each memory page." Ex. 1001, 6:46-50. Thus, state information includes information about the allocation of memory pages, initial permissions, and type of the memory pages.

276. Sallam explicitly discloses the same state information such as "permissions of a code section for a memory page of virtual memory 1204 or the address of physical

memory 1203." Ex. 1008, ¶0262. One of ordinary skill in the art would understand that the "permissions of a code section for a memory page" in Sallam is the same as the "information about the allocation of memory pages for a monitored process and the initial permissions and type of the memory pages" as claimed.

277. Sallam further discloses "[t]he memory page of virtual memory 1204 or the address of physical memory 1203 may be secured by below-O/S security agent 1220, and upon a trapped attempt to write a new value to the permissions in such a location, below-O/S security agent 1220 may determine whether the requestor of the attempt is allowed to make such changes. For example, if the request to change the permissions of a code section of a process arose from a different process, the attempted change in permissions may be denied." *Id.* One of ordinary skill in the art would understand that a determination of whether a requestor of the attempt is allowed to make such changes based on the permissions of the code section is evaluating state information.

[7.2] associated with a cache entry for the hooked function.

278. It is my opinion that Fratric alone or combined with Sallam discloses each part of this limitation.

279. To the extent Fratric does not explicitly disclose evaluating state information associated with a cache entry for the hooked function, Sallam explicitly discloses evaluating state information associated with a cache entry for the hooked function.

280. Sallam discloses that "[a]n attempt to change the values of the memory table, for example, to change the permissions of a code section of a process from 'read' to 'write,' may itself be trapped." *Id.* As I have explained above, trapping an attempt to change values of a memory table is a hooking technique because the trap intercepts the call to change the permission in order to better study the behavior of the attempt.

281. Sallam further discloses, "upon a trapped attempt to write a new value to the permissions in such a location, below-O/S security agent 1220 may determine whether the requestor of the attempt is allowed to make such changes." *Id.* One of ordinary skill in the art would understand that the "permissions in such a location" refers to cache. Again, referring to the definition of the '591 patent, "cache is a specific memory region allocated in each monitored process where information about monitored memory pages is tracked." Ex. 1001, 6:38-40. Thus, the "permissions in such a location" tracked by below-O/S security agent 1220 is the claimed cache.

282. It would have been obvious to one of ordinary skill in the art to combine Fratric and Sallam to evaluate state information associated with a cache entry for a hooked function.

283. As I have explained, Fratric already discloses monitoring a memory space of a monitored function. See [1.1.a]. As I have explained, Fratric also already considers cache concepts like storing in cache for purposes of memory space evaluation. For example, "ROPGuard saves the state of all general-purpose registers at the point of entry of each critical function." *See* Ex. 1007, p. 5. Accordingly, ROPGuard already stores items in cache. It would have been obvious to one of ordinary skill in the art to retrieve information from cache stored by other operations (like a hooked function) as part of another suspicion evaluation.

284. Combining Fratric with Sallam is nothing more than applying Fratric's known techniques of monitoring a memory space of a monitored function to Sallam's known cache evaluation to yield the predictable results of a system that detects and prevents different forms of ROP attacks as Fratric intended, and thus would have been obvious to one of ordinary skill in the art.

285. One of ordinary skill in the art would have been motivated to make this combination in order to achieve Fratric's goal of detecting ROP attacks using the "key observation that, if we disregard the attacks that can be done fully within the context of the exploited process (such as, for example, XSS attacks), the attacker

must use ROP code to leverage the attack and in this process interact with other processes or the OS kernel." Ex. 1007, p. 4. Sallam specifically utilizes this concept in its evaluation of state information in cache. Specifically, this evaluation detects an attempt to interact with other processes or the OS kernel by detecting unauthorized changes to the cache.

286. Thus, claim 7 is rendered obvious over Fratric and Sallam.

12. Claim 8

[8.0] The method according to claim 7,

287. See [7.0]-[7.2].

[8.1] wherein when an address of the invoked function is found in a cache, preventing execution of a payload for the invoked function.

288. It is my opinion that Fratric and Sallam disclose each part of this limitation for the same reasons I expressed above with respect to claim limitation [2.2].

13. Claim 10

[10.0] The system according to claim 9, further comprising

289. See [9.0]-[9.7].

[10.1] identifying an address associated with the invoked function; and

290. It is my opinion that Fratric and Sallam disclose each part of this limitation for the same reasons I expressed above with respect to claim limitation [2.1].

[10.2] when the address of the invoked function is found in a cache, and wherein the processor further executing an operation of preventing execution of a payload for the invoked function.

291. It is my opinion that Fratric and Sallam disclose each part of this limitation for the same reasons I expressed above with respect to claim limitation [2.2].

14. Claims 13, 15, 16

292. I have reviewed claims 13, 15, and 16. Claim 13 depends from claim 9 and is substantially similar to claim 5. It is my opinion that Fratric and Sallam disclose each part of each limitation in claim 13 for the same reasons I expressed above with respect to claim limitations [5.0]-[5.3] for claim 5.

293. Claim 15 depends from claim 13 and is substantially similar to claim 7. It is my opinion that Fratric and Sallam disclose each part of each limitation in claim 15 for the same reasons I expressed above with respect to claim limitations [7.0]-[7.2] for claim 7.

294. Claim 16 depends from claim 15 and is substantially similar to claim 8. It is my opinion that Fratric and Sallam disclose each part of each limitation in claim 15 for the same reasons I expressed above with respect to claim limitations [8.0]-[8.1] for claim 8.

IX. NO OBJECTIVE INDICIA OF NONOBVIOUSNESS

295. 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.

X. CONCLUSION

296. For the reasons set forth above, it is my opinion that claims 1-17 of the ’591 patent are invalid in view of the prior art. In signing this declaration, I understand that the declaration will be filed as evidence in a contested case before the Patent Trial and Appeal Board of the United States Patent and Trademark Office. I acknowledge that I may be subject to cross-examination in this case and that cross-examination will take place within the United States. If cross-examination is required of me, I will appear for cross-examination within the United States during the time allotted for cross-examination.

297. I declare that all statements made herein of my knowledge are true, that all statements made on information and belief are believed to be true, and that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1007 of Title 18 of the United States Code.

Executed: April 28, 2023

By:

/Venkatramanan Siva Subrahmanian/

Venkatramanan Siva Subrahmanian