

UNITED STATES PATENT AND TRADEMARK OFFICE

---

BEFORE THE PATENT TRIAL AND APPEAL BOARD

---

TREND MICRO, INC.  
Petitioner

v.

OPEN TEXT INC.  
Patent Owner

---

Case IPR2022-00692  
Patent No. 10,284,591

---

**DECLARATION OF PROFESSOR NENAD MEDVIDOVIC, Ph.D.  
IN SUPPORT OF PATENT OWNER'S RESPONSE TO PETITION**

Mail Stop PATENT BOARD  
Patent Trial and Appeal Board  
U.S. Patent & Trademark Office  
P.O. Box 1450  
Alexandria, VA 22313-1450

IPR2023-00692  
Open Text Inc. EX-2001

**TABLE OF CONTENTS**

|      |                                                                                                                       |    |
|------|-----------------------------------------------------------------------------------------------------------------------|----|
| I.   | QUALIFICATIONS .....                                                                                                  | 2  |
| II.  | UNDERSTANDING OF RELEVANT LEGAL PRINCIPLES .....                                                                      | 7  |
| III. | TECHNICAL BACKGROUND .....                                                                                            | 9  |
| A.   | COMPUTER SOFTWARE FUNDAMENTALS .....                                                                                  | 9  |
| 1.   | PROGRAMMING LANGUAGES.....                                                                                            | 9  |
| 2.   | FUNCTION CALLS.....                                                                                                   | 10 |
| 3.   | CALL STACKS .....                                                                                                     | 11 |
| 4.   | USER MODE V. KERNEL MODE.....                                                                                         | 15 |
| B.   | STATE OF THE PRIOR ART .....                                                                                          | 16 |
| IV.  | THE '591 PATENT.....                                                                                                  | 19 |
| A.   | SUMMARY OF THE '591 PATENT .....                                                                                      | 19 |
| B.   | CLAIM CONSTRUCTION .....                                                                                              | 22 |
| C.   | LEVEL OF ORDINARY SKILL IN THE ART.....                                                                               | 23 |
| V.   | OVERVIEW OF THE PRIOR ART .....                                                                                       | 25 |
| A.   | OVERVIEW OF DALCHER.....                                                                                              | 25 |
| B.   | OVERVIEW OF PAPPAS.....                                                                                               | 33 |
| VI.  | GROUND 1: THE COMBINATION OF DALCHER AND PAPPAS<br>DOES NOT DISCLOSE OR RENDER OBVIOUS THE<br>CHALLENGED CLAIMS ..... | 37 |
| A.   | THERE IS NO MOTIVATION TO COMBINE DALCHER<br>WITH PAPPAS.....                                                         | 37 |

|       |                                                                                                         |    |
|-------|---------------------------------------------------------------------------------------------------------|----|
| 1.    | A COMBINATION OF DALCHER AND PAPPAS<br>WOULD NOT ENHANCE TRACKING OF CALLS,<br>JUMPS, AND RETURNS ..... | 37 |
| 2.    | PAPPAS WOULD NOT REDUCE FALSE POSITIVES<br>IN DALCHER.....                                              | 38 |
| 3.    | DALCHER AND PAPPAS DISCLOSE<br>FUNDAMENTALLY DIFFERENT TEACHINGS .....                                  | 40 |
| 4.    | DALCHER AND PAPPAS WOULD PRODUCE<br>CONFLICTING RESULTS .....                                           | 44 |
| B.    | DALCHER DOES NOT DISCLOSE EVERY LIMITATION<br>OF INDEPENDENT CLAIMS 1, 9, AND 17 .....                  | 46 |
| 1.    | LIMITATION 1.E, 9.F, 17.E .....                                                                         | 46 |
| 2.    | LIMITATIONS 1.F, 9.G, 17.F .....                                                                        | 49 |
| VII.  | SECONDARY OBJECTIVE INDICIA OF NONOBVIOUSNESS .....                                                     | 50 |
| VIII. | CONCLUSION.....                                                                                         | 51 |

**EXHIBIT LIST**

| <b>Exhibit</b> | <b>Description</b>                                                                                                                                                                                                                                                            |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2001           | Declaration of Professor Nenad Medvidovic, Ph.D in Support of Patent Owner’s Preliminary Response To Petition                                                                                                                                                                 |
| 2002           | <i>Curriculum Vitae</i> of Professor Nenad Medvidovic, Ph.D.                                                                                                                                                                                                                  |
| 2003           | <i>Webroot, Inc. and Open Text Inc., v. Trend Micro Inc.</i> , Case no. 6:22-cv-00239, Complaint for Patent Infringement, Dkt. 1 (W.D. Tex., March 4, 2022)                                                                                                                   |
| 2004           | <i>Sonrai Memory Ltd. v. Kingston Tech. Co. and Kingston Tech. Corp.</i> , Case No. 6:21-cv-1284-ADA, Order Denying Kingston Tech. Co. and Kingston Tech. Corp.’s Opposed Motion to Stay Pending Resolution of <i>Inter Partes</i> Review, Dkt. 94 (W.D. Tex., Oct. 18, 2022) |
| 2005           | <i>mCom IP, LLC v. Cisco Systems, Inc.</i> , Case No. 6:22-cv-00261-ADA, Order Denying Defendant’s Motion to Stay Pending <i>Inter Partes</i> Review, Dkt. 42 (W.D. Tex., Oct. 20, 2022)                                                                                      |
| 2006           | <i>Intellectual Ventures I LLC et al. v. Hewlett Packard Enterprise Co.</i> , No. 6:21-cv-00596-ADA, Order Denying Hewlett Packard Enterprise Company’s Opposed Motion to Stay Pending Resolution of <i>Inter Partes</i> Review, Dkt. 104 (W.D. Tex., Dec. 22, 2022)          |
| 2007           | <i>Webroot, Inc. and Open Text Inc., v. AO Kaspersky Lab. et al.</i> , Case No. 6:22-cv-243-ADA-DTG, Dec. 9, 2022 Transcript of Motions Hearing, Dkt. 133 (W.D. Tex. Dec. 15, 2022) (excerpted)                                                                               |
| 2008           | Dani Kass, “Catching Up On Patent Litigation With Judge Al-bright” (Law360 Mar. 14, 2023);<br><a href="https://www.law360.com/pulse/articles/1582438/print?section=pulse/courts">https://www.law360.com/pulse/articles/1582438/print?section=pulse/courts</a>                 |

| Exhibit | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2009    | <i>Webroot, Inc. and Open Text Inc., v. AO Kaspersky Lab. et al.</i> , Case No. 6:22-cv-243-ADA-DTG, Order Granting Fourth Amended Scheduling Order, Dkt. 160 (W.D. Tex. Jan. 22, 2023)                                                                                                                                                                                                                                                                          |
| 2010    | U.S. District Courts—Federal Court Management Statistics—Profiles—During the 12-Month Periods Ending March 31, 2018 Through 2023, “Table N/A—U.S. District Courts—Combined Civil and Criminal Federal Court Management Statistics (March 31, 2023)”, available <a href="https://www.uscourts.gov/sites/default/files/data_tables/fcms_na_distprofile0331.2023.pdf">https://www.uscourts.gov/sites/default/files/data_tables/fcms_na_distprofile0331.2023.pdf</a> |
| 2011    | <i>Webroot, Inc. and Open Text Inc., v. Trend Micro, Inc.</i> , Case No. 6:22-cv-00239-ADA-DTG, Defendant’s Preliminary Invalidity Contentions (W.D. Tex., Sept. 13, 2022)                                                                                                                                                                                                                                                                                       |
| 2012    | <i>Webroot, Inc. and Open Text Inc., v. Sophos Ltd.</i> , Case No. 6:22-cv-240-ADA-DTG, Defendant’s Preliminary Invalidity Contentions (W.D. Tex., Sept. 13, 2022)                                                                                                                                                                                                                                                                                               |
| 2013    | <i>Webroot, Inc. and Open Text Inc., v. CrowdStrike, Inc. and CrowdStrike Holdings, Inc.</i> , Case No. 6:22-cv-00241-ADA-DTG, Second Supplemental Preliminary Invalidity Contentions (W.D. Tex., Jul. 14, 2023)                                                                                                                                                                                                                                                 |
| 2014    | U.S. Pat. Appl. Pub. No. 2009/0049550 A1 to S. Shevchenko, filed Jun. 6, 2008 and issued Feb. 19, 2009 (“Shevchenko”)                                                                                                                                                                                                                                                                                                                                            |
| 2015    | U.S. Pat. Appl. Pub. No. 2005/0108562 A1 to R. Khazan et al., filed Jun. 18, 2003 and issued May 19, 2005 (“Khazan”)                                                                                                                                                                                                                                                                                                                                             |
| 2016    | U.S. Patent 8,510,596 B1 to S. Gupta and P. Shenoy, filed Jul. 6, 2007 and issued Aug. 13, 2013 (“Gupta”)                                                                                                                                                                                                                                                                                                                                                        |
| 2017    | <i>Webroot, Inc. and Open Text Inc., v. AO Kaspersky Lab. et al.</i> , Case No. 6:22-cv-243-ADA-DTG, Claim Construction Order, Dkt. 236 (W.D. Tex. Mar. 16, 2023)                                                                                                                                                                                                                                                                                                |

| <b>Exhibit</b> | <b>Description</b>                                                                                                                                                                                                                                    |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2018           | Erickson, J., Hacking: The Art of Exploitation, 2nd Ed. (2008)                                                                                                                                                                                        |
| 2019           | Kleymenov, A. and Thabet, A., Mastering Malware Analysis, Packt Pub. (2019)                                                                                                                                                                           |
| 2020           | LWN.net, An Introduction to Last Branch Records (Mar. 23, 2016) (available at <a href="https://web.archive.org/web/20160413055733/https://lwn.net/Articles/680985/">https://web.archive.org/web/20160413055733/https://lwn.net/Articles/680985/</a> ) |
| 2021           | U.S. Patent 9,251,373 to K. AlHarbi and X. Lin, filed Mar. 14, 2013 and issued Feb. 2, 2016                                                                                                                                                           |
| 2022           | U.S. Patent 8,291,381 to T. Lai et al., filed Sep. 27, 2007 and issued Oct. 16, 2012                                                                                                                                                                  |
| 2023           | GeeksforGeeks, Stack Unwinding in C++                                                                                                                                                                                                                 |
| 2024           | IBM Documentation, Stack unwinding (C++ only)                                                                                                                                                                                                         |
| 2025           | Microsoft Learn, Exceptions and Stack Unwinding in C++                                                                                                                                                                                                |

I, Professor Nenad Medvidovic, declare as follows:

1. I have been engaged by King & Spalding LLP on behalf of OpenText Inc. (“Patent Owner”), to provide this Declaration concerning the technical subject matter relevant to the petition for Inter Partes Review (“Petition”) of U.S. Patent 10,284,591 (“the ’591 patent”).

2. I am over 18 years of age. I have personal knowledge of the facts stated in this Declaration and could testify competently to them if asked to do so.

3. I have reviewed and am familiar with the specification and the claims of the ’591 patent. In general, I will cite to the specification of a United States patent using the following formats: (Exhibit No. Col:Line Number(s)) or (Exhibit No. Paragraph Number(s)). For example, the citation (EX-1001 1:1–10) points to the ’591 Patent specification at column 1, lines 1–10. Also, for convenience, I use italics to denote limitations from the challenged claims.

4. All the opinions in this Declaration are based on the documents I reviewed and my knowledge and professional judgment. In forming the opinions expressed in this Declaration, I reviewed the Petition for *Inter Partes* Review of U.S. Patent No. 10,284,591 and supporting exhibits, as well as those Exhibits attached to the Patent Owner Preliminary Response.

5. I have been asked to provide my technical opinions regarding how a person of ordinary skill in the art would have understood the claims of the ’591

patent at the time of the alleged invention. I have also been asked to provide my technical opinions on how concepts in the '591 patent specification relate to claim limitations of the '591 patent. In reaching the opinions provided herein, I have considered the '591 patent, its prosecution history, and the references cited above and have drawn as appropriate on my own education, training, research, knowledge, and personal and professional experience.

## **I. QUALIFICATIONS**

6. In formulating my opinions, I have relied on my knowledge, training, and experience in the relevant field, which I will summarize briefly. A more detailed summary of my background, experience, and publications is set forth in my curriculum vitae (CV), which is submitted as Exhibit 2002.

7. I received a Bachelor of Science degree, Summa Cum Laude, from Arizona State University's Computer Science and Engineering department.

8. I received a Master of Science degree from the University of California at Irvine's Information and Computer Science department.

9. I received a Doctor of Philosophy degree from the University of California at Irvine's Information and Computer Science department. My dissertation was entitled "Architecture-Based Specification-Time Software Evolution."

10. For nearly 30 years, my research has focused on the building blocks that comprise software systems: software components (computational and data elements) and connectors (interaction-facilitating elements), and mechanisms for their implementation, primarily middleware facilities and software frameworks. In particular, my Ph.D. dissertation—whose first research results were published in 1995 while the dissertation was completed in 1999—yielded a new design style for the development and evolution of highly distributed software systems built out of independent components that exchange different types of information encoded in messages in order to accomplish specific goals. This research focused on the design principles behind such systems, the appropriate support for modeling the systems (both at the level of individual agents as well as their integrations into subsystems and entire systems), and the infrastructure for their implementation, deployment, and optimization. This work was applied across a wide range of domains, from traditional distributed systems to mobile devices and autonomous robots.

11. I am employed by the University of Southern California (“USC”) as a faculty member in the Computer Science Department and have been since January 1999. I currently hold the title of Professor with tenure. Since July 2022, I have served as my department’s Chairman. Between January 2009 and January 2013, I served as the Director of the Center for Systems and Software Engineering at USC. Between July 2011 and July 2015, I served as my Department’s Associate Chair for

Ph.D. Affairs. Between July 2021 and June 2022, I served as my Department's Vice-Chair for Tenure Affairs.

12. I teach graduate and undergraduate courses in Software Architecture, Software Engineering, and Embedded Systems, and I advise Ph.D. students. I have graduated 19 Ph.D. students and advise 10 students currently pursuing a Ph.D.

13. I served as Program Co-Chair for the flagship conference in my field—International Conference on Software Engineering (“ICSE”)—held in May 2011. I have served as Chair or Co-Chair for various other conferences in the Software Engineering field, including: the Fifth Working Institute of Electrical and Electronics Engineers (“IEEE”)/IFIP Conference on Software Architecture, the Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems, the Fifteenth International Association for Computing Machinery’s (“ACM”) Special Interest Group on Software Engineering (“SIGSOFT”) Symposium on Component Based Software Engineering, the IEEE/CSSE/ISE Workshop on Software Architecture Challenges for the 21st Century, and the Doctoral Symposia at the Sixteenth ACM SIGSOFT International Symposium on the Foundations of Software Engineering and at the 44th International Conference on Software Engineering. I am currently serving as General Co-Chair of the 21st International Conference on Software Architecture (ICSA 2024).

14. I serve or have served as an editor of several peer-reviewed journals, including: “IEEE Transactions on Software Engineering,” “ACM Transactions on Software Engineering and Methodology,” “Journal of Software Engineering for Robotics,” “Elsevier Information and Software Technology Journal,” “Journal of Systems and Software,” “Journal of Software Engineering Research and Development,” and “Springer Computing Journal.” Additionally, I have served as a guest editor of several special issues for different journals. Since January 2018, I have served as Editor-in-Chief of “IEEE Transactions on Software Engineering.”

15. Between September 2013 and September 2015, I served as Chair of the ICSE Steering Committee. Both prior to that (2008-2013) and subsequent to that (2015-present) I have been a member of the ICSE Steering committee. I am currently a member of the Steering Committee of the joint European Conference on Software Engineering and the Symposium on the Foundations of Software Engineering (“ESEC/FSE”). From July 2015 to September 2017, I chaired the Steering Committee of FSE, while it was a standalone symposium. I previously served as a member of the Steering Committees of the Working IEEE/IFIP Conference on Software Architecture.

16. Between July 2015 and June 2018, I served as Chair of the ACM SIGSOFT, the largest professional organization in my field of work. During the past

year, I have chaired the Committee in charge of ACM's Software System Award, one of the most important technical awards given in my field.

17. I co-authored "Software Architecture: Foundations, Theory, and Practice," a widely used textbook in the field of Software Systems' Architecture.

18. I have served as editor of various books in the Software Engineering field including: "Proceedings of the 3rd International Conference on Self-Adaptive and Self-Organizing Systems," "Proceedings of the Warm-Up Workshop for the 32nd International Conference on Software Engineering," "Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture," and "40 Editions of ICSE – The Ruby Anniversary Celebration."

19. I have authored or co-authored nearly 300 papers in the Software Engineering field. My most cited paper has been cited well over 3,000 times. A paper I co-authored in the 1998 International Conference on Software Engineering, my field's flagship conference, was given ten years later, in 2008, that conference's Most Influential Paper Award. Another one of my publications received a Most Influential Paper Award in 2020. Several of my other publications have received "best paper" or "most cited" awards."

20. Several of my publications have focused on different aspects of computer security. A paper describing a security-vulnerability detection tool for Android devices, built in my lab, won the Best Tool Paper award at the 2017

Automated Software Engineering Conference. And in May 2023, I was inducted into the University of California Irvine Donald Bren School of Information and Computer Science Alumni Hall of Fame. I have co-authored three patents in the area of cybersecurity.

21. I have been named a Distinguished Scientist of the ACM. I have been elected a Fellow of the IEEE, IEEE's highest grade that is granted to less than 0.1% of its membership annually.

## **II. UNDERSTANDING OF RELEVANT LEGAL PRINCIPLES**

22. I am not a lawyer, and I will not provide any legal opinions. Although I am not a lawyer, I have been advised certain legal standards are to be applied by technical experts in forming opinions regarding the meaning and validity of patent claims.

23. I understand that a patent claim is invalid if it is anticipated or obvious in view of the prior art, and that a claim can be unpatentable even if all of the requirements of the claim cannot be found in a single prior-art reference. I further understand that invalidity of a claim requires that the claim be anticipated or obvious from the perspective of a person of ordinary skill in the art at the time the invention was made.

24. I have been informed that a patent claim is invalid if it would have been obvious to a person of ordinary skill in the art. In analyzing the obviousness of a

claim, I understand the following factors may be taken into account: (1) the scope and content of the prior art; (2) the differences between the prior art and the claims; (3) the level of ordinary skill in the art; and (4) any so called “secondary considerations” of nonobviousness if they are present.

25. I have been informed that a claim can be obvious in light of a single prior art reference or multiple prior art references. I understand that to prove that prior art or a combination of prior art renders a patent obvious, it is necessary to:

- 1) Identify the particular references that, singly or in combination, make the patent obvious;
- 2) Specifically identify which elements of the patent claim appear in each of the asserted references; and
- 3) Explain why a person of ordinary skill in the art would have combined the references, and how they would have done so, to create the inventions claimed in the patent. I further understand that exemplary rationales that may support a conclusion of obviousness include:
  - Combining prior art elements according to known methods to yield predictable results;
  - Simple substitution of known element for another to obtain predictable results;
  - Use of known technique(s) to improve similar devices (methods or products) in the same way;
  - Applying a known technique to a known device (method or product) ready for improvement to yield predictable results;

- “Obvious to try” – choosing from a finite number of identified, predictable solutions with a reasonable expectation of success;
- Known work in one field of endeavor may prompt variations of the work for use in either the same field or a different field based on design incentives or other market forces if the variations are predicable to a person of ordinary skill in the art; and
- Some teaching, suggestion, or motivation in the prior art that would have led a person of ordinary skill in the art to modify the prior art reference or to combine the prior art reference teachings to arrive at the claimed invention.

26. I have been informed that, in considering obviousness, hindsight reasoning derived from the patent-at-issue may not be used.

### **III. TECHNICAL BACKGROUND**

#### **A. Computer Software Fundamentals**

27. In order to assist in understanding the state of anti-malware prior art and the '591 patent, I have included some discussion of the underlying computer software concepts.

##### **1. Programming Languages**

28. “A program is nothing more than a series of statements written in a specific language.” (EX-2018 6.)

But a computer doesn't natively understand English; it only understands machine language. To instruct a computer to do something, the instructions must be written in its language. However, *machine*

*language* is arcane and difficult to work with—it consists of raw bits and bytes, and it differs from architecture to architecture.

(*Id.* 7.) Assembly language is less cryptic than machine language but is still based on low-level details of the processor architecture. Most programmers use a high-level language (*e.g.*, C, C++, or Fortran), because they are more intuitive and can be compiled into different types of machine language. (*Id.* 7; *see also id.*, 36-37)

29. “A low level operation or instruction provides minimal abstraction from a computer’s instruction set architecture, for example either machine code or assembly language. The word ‘low’ refers to the small or nonexistent amount of abstraction between the programming language and machine language.” (EX-1001 3:47-52; *see also, e.g.*, EX-2018 7 (describing machine language, assembly language, high-level languages such as C, C++, and Fortran, and compilers that convert high-level languages into machine language).)

## **2. Function Calls**

30. When a programmer knows that a set of program instructions will need to be used several times, she may group these instructions into a smaller sub-program or subroutine, called a “function.” (EX-2018 16.) For example, the function of turning a car actually consists of many smaller instructions (*e.g.*, turn on the appropriate blinker, slow down, check for oncoming traffic, turn the steering wheel, etc.). (*Id.*) But because listing every instruction would be tedious, programmers

group these instructions into a function. The function may even include sub-functions or subsets of instructions which account for certain variables (e.g., instructions that apply only when there is oncoming traffic or instructions that apply only at a certain speed). (*Id.*)

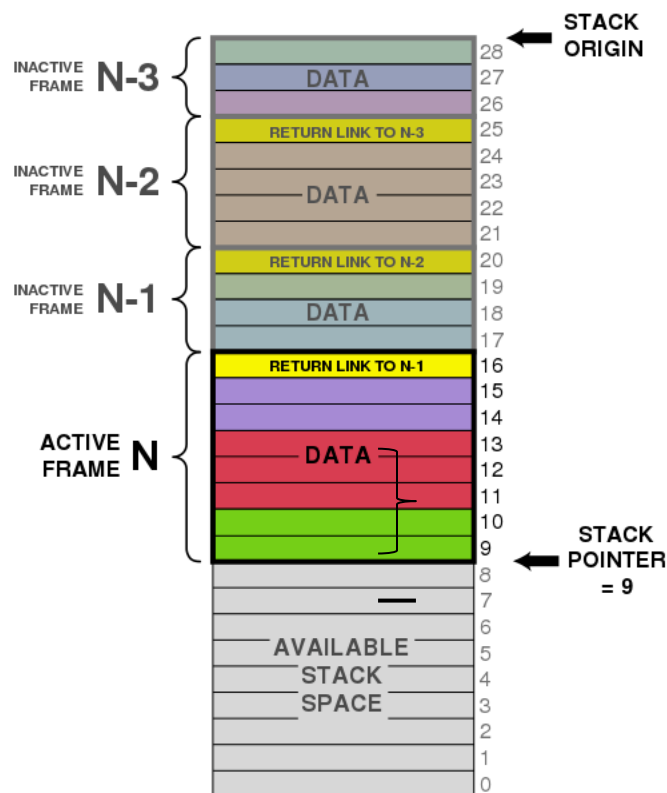
31. Once this function is established, a program can just call this function instead of repeating all the instructions. (EX-2018 16.) After calling the function, the instructions within the function may be executed (with any arguments or variables passed to it), after which, execution returns to where the program was before it called the function. (*Id.*)

### **3. Call Stacks**

32. A compiled program's memory (also referred to as a process memory) is divided into five segments, each of which has a certain purpose. (EX-2018 69.) The "text" segment—also known as the "code" segment—is the memory where the assembled machine language instructions of the program are actually located. (*Id.* 69, 43.) Because these instructions are not executed in a linear manner, the instructions will include "branch, jump, and call instructions" to move the instruction pointer (which points to the current instruction by its memory address) to a different address in memory. (*Id.*)

33. The "stack" segment of the process memory is used as a temporary scratch pad to store local function variables and context during function calls. (EX-

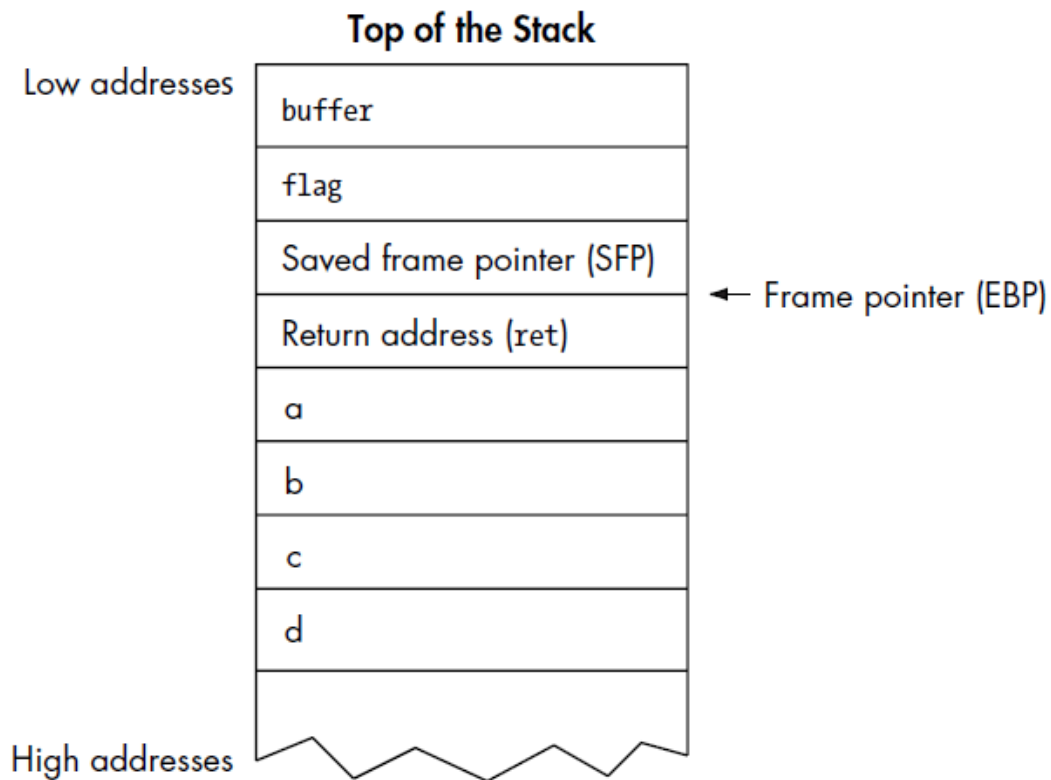
2018 70.) Because functions can contain variables, as well as further functions within the function (as discussed above), “the stack is used to remember all of the passed variables,” the local variables used by that function, and the address at which execution should begin again after the function is finished. (*Id.*) That is, a “stack” acts as an abstract data structure for tracking the functions as they are called. (*Id.*) Individual functions may be contained within their own stack frames, and a stack may contain many stack frames. (*Id.*) Below is an illustration that accurately represents an exemplary call stack with call frames N through N-3:



34. “When a function is called, several things are pushed [*i.e.*, added] to the stack together in a stack frame.” (EX-2018 70.) “Each stack frame contains the

parameters to the function, its local variables, and two pointers” which assist in returning the program to the correct addresses prior to the function being called. (*Id.*) For example, in the illustration above, Function N-3 is first pushed to the call stack in its own stack frame and then Function N-2 (and its associated information) is pushed to the call stack in its own stack frame. The stack follows a last-in, first-out ordering, which means that the last item that is put into a stack is the first item to come out of it. (*See id.*)

35. Some call stacks, such as the one illustrated above, grow downward towards lower-numbered memory addresses as stack frames are added. (EX-2018 70; EX-2021 1:44-45.) However, it is important to understand that call stacks may be oriented such that they grow upwards or downwards. (EX-2022 6:28-35.) For example, Exhibit 2025 illustrates a call stack which grows upwards—that is, the most recent call frames are added to the top of the stack, as the stack grows towards lower addresses:



(EX-2018 73.) And further still, “[i]t will be appreciated that...other call stacks may be configured to grow ‘downward,’ such that subsequent call stack frames are stored in higher-numbered memory addresses than preceding call stack frames.” (EX-2022 6:28-35; *see also id.* 18:44-48 (distinguishing between an “upward-growing call stack” and a “downward-growing call stack”).) This is because a “stack” merely serves as an abstract data structure for tracking functions as they are called. (EX-2018 70.)

36. Thus, any discussion of moving “up” or “down” a stack or whether a stack is growing towards lower or higher addresses must be considered within the context of a particular call stack.

37. Stack unwinding is generally understood in the art to refer to the removal of function entries from a call stack during run time. (EX-2033; EX-2034; EX-2035.) As execution of each function is completed, the stack frame for that function is “popped off” (*i.e.*, removed from) the stack and execution is returned to the previous, calling function. (EX-2018 75 (“As each function ends, its stack frame is popped off of the stack so execution can be returned to the previous function.”); EX-2028 70 (“When an item is placed into a stack, it’s known as *pushing*, and when an item is removed from a stack, it’s called *popping*.”).)

#### **4. User Mode v. Kernel Mode**

38. Most code running on a processing device will execute in user mode, where executing code has no ability to directly access hardware or reference memory. (EX-1001 3:63-4:1.) Conversely, in kernel mode, executing code has complete and unrestricted access to the underlying hardware—*i.e.*, it can execute any CPU instruction and reference any memory address. (*Id.* 3:58-61.) Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. (*Id.* 3:61-63.)

39. Thus, where a process requires access to hardware or memory, it will typically involve calling a system process through an application programming interface (“API”). (EX-1001 3:43-47.) An instruction from user mode will involve system call instructions, such as triggering an OS-dependent software interrupt or a

control transfer instruction (*e.g.*, `sysenter` assembly instruction). (*Id.* 4:1-4.) As demonstrated by the example below, a `sysenter` instruction is executed in user mode, allowing the process to enter kernel mode, where kernel functions may be executed:

## The execution path from user mode to kernel mode

Let's take a look at the life cycle of one API that requires kernel mode (in this example, it will be `FindFirstFileA`). We will dissect each step so that we can understand the role that each part of the system plays in handling process requests:

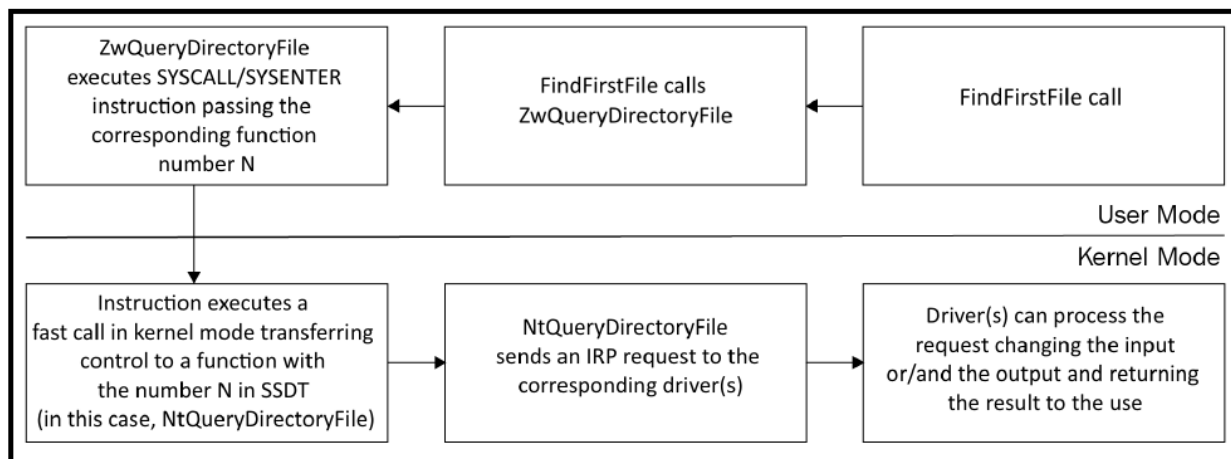


Figure 4: The API call life cycle

(EX-2019 217; *see* EX-1001 4:1-4.)

### B. State of the Prior Art

40. As demonstrated by the fundamentals above:

[A] program is made up of a complex set of rules following a certain execution flow that ultimately tells the computer what to do. Exploiting a program is simply a clever way of getting the computer to do what you want it to do, even if the currently running program was designed to prevent that action.

(EX-2018 115.) An “exploit” refers to a piece of code, software, data, or a sequence of commands that takes advantage of a vulnerability in order to cause unintended, potentially harmful behavior to occur on a computer. (EX-1001 1:12-16.) The intent of an exploit is commonly malicious. For example, exploit code might be created that unexpectedly launches a new process or manipulates an existing process to perform malicious execution. (*Id.* 1:21-23.) In order for this exploit code to be executed (and thereby cause the potentially harmful behavior), the program’s normal code flow execution might be hijacked and directed towards a zone of memory where the exploit code is currently stored. (*Id.* 1:17-21.)

41. In order for the exploit to accomplish its goal of starting a new process, the exploit must execute a complete execution stack, starting from the shellcode and running down through to a lowest level user function (*e.g.*, a function that includes machine or assembly instruction). (EX-1001 4:24-28.) An exploit will commonly inject shellcode into a target process before or at the same time as it exploits a vulnerability. (*Id.* 4:30-32.) Shellcode includes a malicious payload, which is capable of any one of a number of malicious actions—*e.g.*, executing another process, loading a module into the process memory, adding an admin account, or automatically removing lines from log files. (*Id.* 4:32-35, 3:34-37, 7:5-10, 7:61-62; EX-2018 281.)

42. Preventing software exploits from executing code is a paramount security concern for system users and administrators. (EX-1001 1:23-25.) Early anti-exploit methods and security platforms included detecting signatures (*i.e.*, unique identifiers) (*e.g.*, EX-2014 [0050] (describing signature scans generally)), known static code patterns (*e.g.*, (EX-2015 [0008])), or corrupted memory (*e.g.*, EX-2016 2:11-16). But these techniques were often resource and time-intensive, as well as costly to prepare and implement. (EX-2015 [0009]; EX-2016 34:65-35:4.)

43. Despite substantial investments in resources and time, the conventional systems were still unable to identify and prevent emerging (“zero-day”) threats from new or unknown malicious code. (*See, e.g.*, EX-2016 2:58-63 (explaining that signature-based methods were not sufficient for detecting “zero-day exploits”).) By early-to-mid 2000s, new threats escalated as network connectivity became widespread, and programs that mutate slightly with each new copy (polymorphic programs) appeared. These events, and others, rendered the traditional signature-based virus scan systems ineffective for these modern environments.

44. Many prior exploit prevention technologies relied on the ability to prevent an exploit from ever starting its own shellcode, such that it could never execute the shellcode’s malicious payload. (EX-1001 6:24-30, 7:56-62.) However, like other conventional anti-exploit methods discussed above, this technology would

be rendered ineffective if the exploit succeeded in starting its own shellcode. Accordingly, an improved anti-exploit technology was needed.

#### **IV. THE '591 PATENT**

##### **A. Summary of the '591 Patent**

45. Patent Owner's patented technology enabled the automatic and real-time analysis, identification, and neutralization of previously unknown malicious code. The '591 patent is entitled "Detecting and Preventing Execution of Software Exploits" and describes and claims an "anti-exploit" technique to prevent undesirable software and/or other computer exploits from executing a malicious payload. (EX-1001 1:13-28, 1:32-33.)

46. Unlike prior art anti-exploit systems, which sought to prevent an exploit from ever even starting its own shellcode, the '591 patent describes an anti-exploit component that is capable of preventing the shellcode from performing an ultimate execution of a malicious payload. (EX-1001 6:24-30.) That is, Patent Owner's patented technology included new methods of exploit analysis and new forensic techniques that enable fast, efficient, and accurate analysis of malicious attacks.

47. The '591 patent describes and claims novel anti-exploit techniques that involves the loading of a component (*e.g.*, a custom module) into the memory space of a monitored process. (EX-1001 4:45-47.) The '591 patent advises monitoring this memory space for functions "compris[ing] processes that attempt execution of

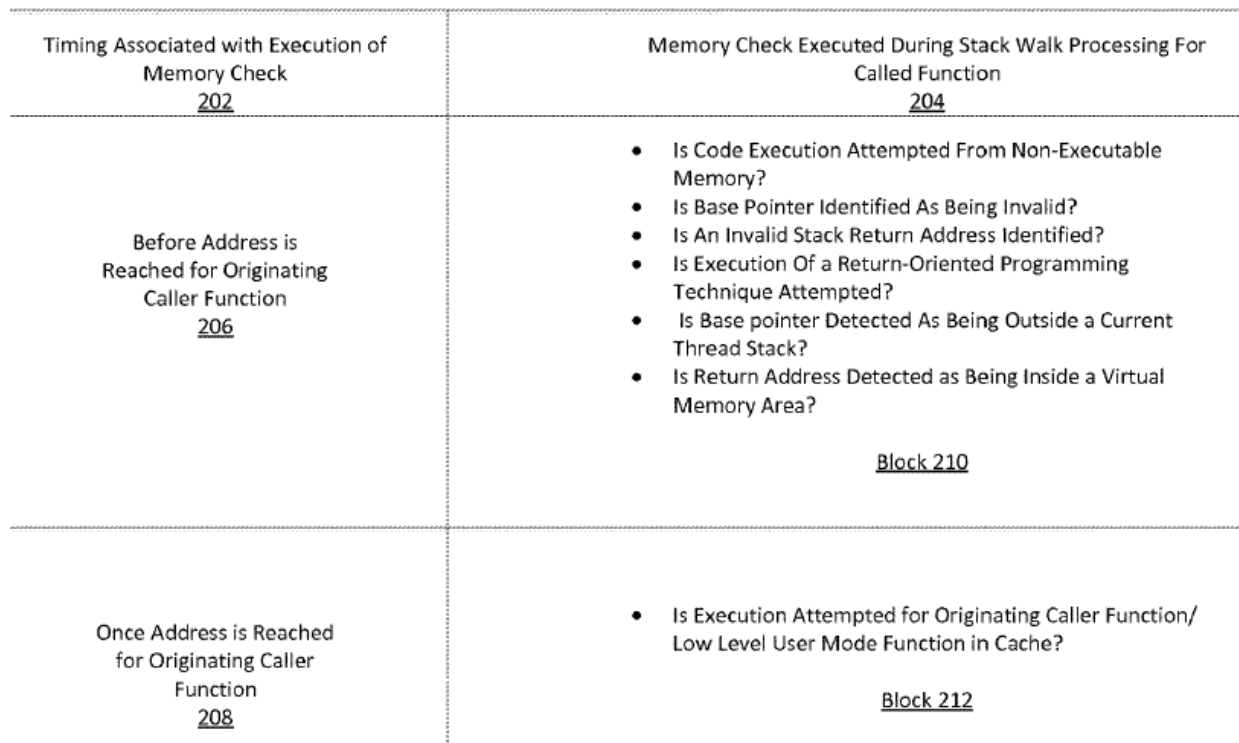
another process or loading of a module into a process memory.” (*Id.* 4:58-64.) Some exemplary functions include “NtCreateProcess, NtCreateProcessEX, NtCreateUserProcess, LdrLoadDll.” (*Id.*)

48. When one of the monitored functions is invoked, as a result of receiving a call from an API, the method executes “stack walk processing.” (EX-1001 1:33-39.) Stack walk processing walks “a call stack for the process (function) until an address of the first function of a called function is reached.” (*Id.* 5:26-29.) For example, one embodiment of the anti-exploit technology may walk the stack frames starting from the lowest level user mode function to an address where the call sequence started. (*Id.* 5:29-39.) In the stack walk processing, a base pointer address and stack pointer address are identified for a stack frame associated with the called function. (*Id.* 5:43-46.)

49. During the executing of stack walk processing, the anti-exploit method performs a memory check against a plurality of stack entries identified during the stack walk processing and triggers an alert of suspicious behavior if the check detects any one of a list of suspicious behaviors. (EX-1001 1:39-48, 2:34-38, cl. 1.) For example, claim 1 of the ’591 patent lists six possible detections that might trigger an alert. (*Id.* cl. 1.) In one embodiment, these memory checks are conducted before the address of the originating caller function is reached. (*Id.* cl. 1, 7:63-8:9.)

**FIG. 2**

**200**



(EX-1001, Fig. 2.) When an alert of suspicious behavior is triggered, the anti-exploit method prevents execution of a payload for the invoked function. (*Id.* 2:1-3, 6:24-30.)

50. The innovations of the '591 patent achieved a number of advantages over the prior art anti-exploit methods, including “enhanced security protection, improved detection of potential security exploits, reduction in error rate in identification and marking of suspicious behavior (*e.g.*, false positives), and improved usability and interaction with users who are not required to continuously monitor for security exploits, among other examples.” (EX-1001 2:44-51.)

**B. Claim Construction**

51. I understand that Petitioner applies the plain and ordinary meaning of all claim terms. (Petition 9-10.)

52. I understand that the District Court for the Western District of Texas issued a claim construction order on March 16, 2023, finding that no terms of the '591 patent were indefinite and that all terms should be construed to have their plain and ordinary meaning. The Court also provided several clarifying remarks, reproduced below. Consistent with the Court's order, I apply the plain and ordinary meaning of all claim terms.

| <b>Claim Term</b>                                                                                      | <b>Court's Construction</b>                                       |
|--------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|
| "a component for evaluating the at least one monitored function in the memory space"<br>(cls. 1, 9 17) | Not subject to Section 112(f).<br><br>Plain and ordinary meaning. |
| "invoking one of the plurality of functions"<br>(cls. 1, 9, 17)                                        | Plain and ordinary meaning.                                       |
| "stack walk processing"<br>(cls. 1, 3, 5, 9, 11, 13, 17-18)                                            | Plain and ordinary meaning.                                       |
| "before an address of an originating caller function is reached"<br>(cls. 1, 9, 17)                    | Plain and ordinary meaning.                                       |

| <b>Claim Term</b>                           | <b>Court's Construction</b>                                                                                                                                   |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| “low level user mode function” (cls. 3, 11) | Plain and ordinary meaning,<br><br>wherein the plain and ordinary meaning is “user mode function that is low level with respect to its layer of abstraction.” |
| “at least one processor. . .” (cls. 9-16)   | Not subject to Section 112(f).<br><br>Plain and ordinary meaning.                                                                                             |
| Claim 10                                    | Not indefinite. Plain and ordinary meaning.                                                                                                                   |

(EX-2017 12-17.)

### **C. Level of Ordinary Skill in the Art**

53. I understand that certain issues relating to the validity of the '591 patent must be judged from the perspective of a person of ordinary skill in the relevant art, as I discuss below. I have been asked to define the level of a “person of ordinary skill in the art” or “POSITA” that someone would have had at the time the alleged invention as claimed was made. In deciding the level of ordinary skill, I have considered the following factors, which I have been informed are relevant in determining the level of ordinary skill in the art:

- The levels of education and experience of persons working in the field;

- The types of problems encountered in the art;
- The prior art solutions to these problems;
- The rapidity with which innovations are made; and
- The sophistication of the technology.

54. In my opinion, a POSITA at the time of the invention of the '591 patent (around Jan. 27, 2014) would have had a bachelor's degree in an accredited program of electrical engineering, computer engineering, computer science, or a similar discipline, and have 2-3 years of practical work or research experience in the general fields of electrical engineering, computer science, networking, communications, and/or device and network security. More advanced degrees and/or training in a related discipline can compensate for shorter work experience.

55. I understand that Petitioner has defined the level of ordinary skill in the art as:

[A] person having ordinary skill in the art ("PHOSITA") at the time of the earliest claimed priority date for the '591 patent would have had a bachelor's degree in computer science, or a related field, or an equivalent technical degree or equivalent work experience, and an additional one or two years of education experience in the field of computer security topics such as malware/exploit detection. More education can supplement practical experience and vice versa.

(Petition 9.)

56. Despite minor variations in these two positions, these variations would not alter my opinions expressed in this Declaration.

57. Based on either standard, I was at least a person of ordinary skill in the art at the time of the invention of the '591 patent.

## **V. OVERVIEW OF THE PRIOR ART**

### **A. Overview of Dalcher**

58. Dalcher discloses a system, method, and computer program product directed to monitoring the execution flow of a function. (EX-1004 at Abstract.) Dalcher explains that its teachings “may be used to assemble a picture of an execution flow” of a function. (EX-1004 0048.) The execution flow information is used to “determine if code involved in handling a monitored operation is trusted and expected to be involved in the handling of the operation.” (EX-1004 0002.) By deeply analyzing the execution flow of a single function, Dalcher is better able to detection malicious activity on the computer. (EX-1004 0026.)

59. Dalcher’s execution-flow monitoring technique relies on identifying and analyzing “data associated with a function ... within a call stack.” (EX-1004 0017.) As I explained previously in Section III.A.3, the call stack is a data structure that stores the data associated with the execution of one or more functions. The call stack is made up of entries—information that the executing functions use for their

operation—organized into call stack frames. Dalcher explains that some pertinent call stack entries include a return address (i.e., the address that identifies the function that pushed the entry onto the call stack), local variables, reserved space for variables, and register values. (EX-1004 0018.)

60. Dalcher also discloses the use of freed memory in the call stack. When a function is invoked, memory on the stack is allocated to that function. This is called a call stack frame. When the function completes execution, the stack frame that is allocated to it is “popped” from the call stack and is no longer active. Though the stack space that previously held the stack frame is no longer available via normal stack usage, the data that was held there remains until it is overwritten. (EX-1004 0038.) Thus, while a popped stack frame is no longer active, it may continue to hold data relevant to the execution of the function. (EX-1004 0021.) Dalcher refers to freed memory as “memory in the stack that previously held an active stack entry.” (EX-1004 0021.)

61. To assemble a full picture of an execution flow, Dalcher operates in three steps, which are laid out in Figure 3: (1) identifying the data associated with the monitored function in the call stack (step 302); (2) determining a call stack frame from freed memory in the call stack (step 304); (3) monitoring the execution flow of the function, utilizing the call stack frame from freed memory (step 306). (EX-1004 0004.) I explain each of these steps in more detail below.

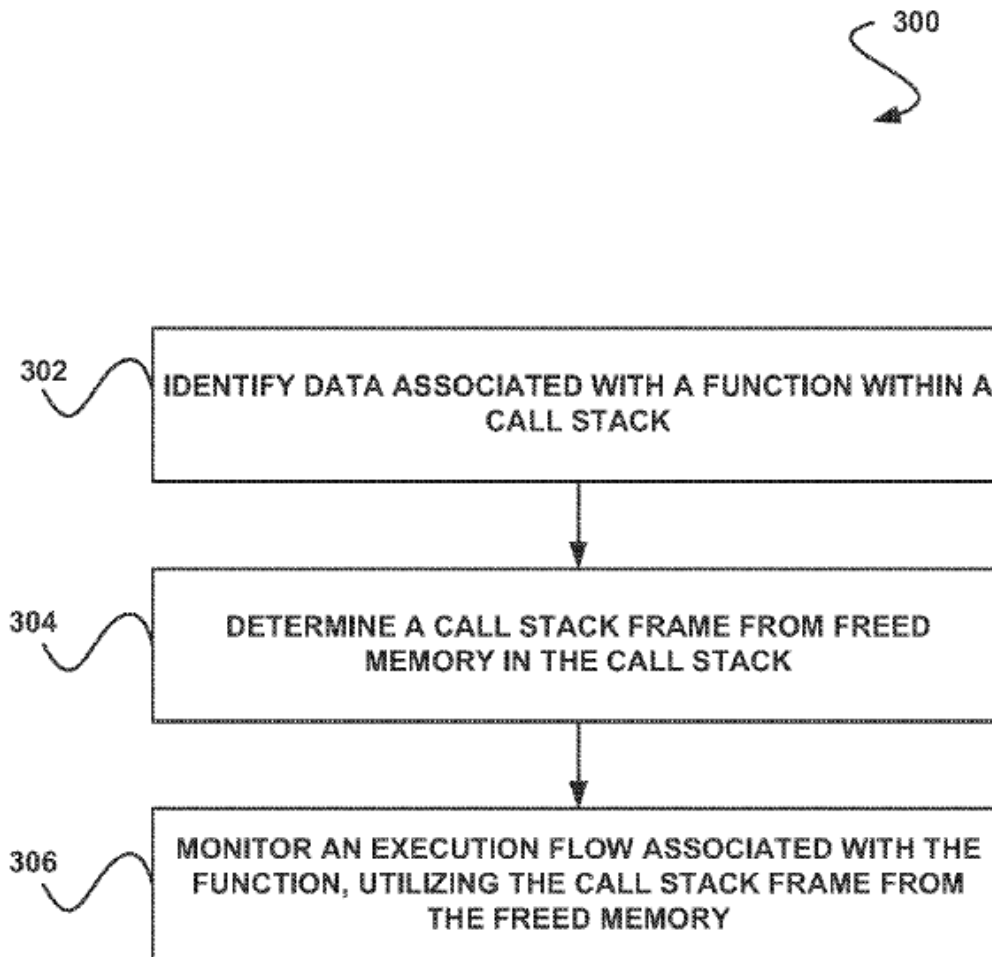


**FIGURE 3**

(EX-1004 at Figure 3.)

62. In the first step, Dalcher identifies “data associated with a function within a call stack.” (EX-1004 at Figure 3.) To identify said data, Dalcher discloses “monitoring the call stack.” (EX-1004 0020.) Dalcher discloses that this can be accomplished by hooking-based monitoring of the call stack. (EX-1004 0020.) Dalcher also teaches the use of a “test event (e.g., a trace event, etc.).” (EX-1004 0019.) A

test event is an event generated specifically for the purpose of determining how a function is handled. When performed in a controlled environment, a test event enables tracing the execution of the system “to determine one or more functions that are involved with a call stack associated with the test event.” (EX-1004 0019.)

63. In the second step, Dalcher determines “a call stack frame from freed memory in the call stack.” (EX-1004 0021.) This is a technique that is central to Dalcher’s claimed invention. Put shortly, Dalcher examines the call stack for “memory in the stack that previously held an active stack entry.” (EX-1004 0021.) The memory that held a frame that was “popped” from the call stack and is no longer active remains intact until it is overwritten. The inactive call stack frame can be overwritten by another function pushing a new call frame onto the stack. According to Dalcher, the process of examining freed memory is called “extended call stack analysis.” (EX-1004 0033, 0037.)

64. In the third step, Dalcher begins the operation of monitoring the execution flow associated with the function. (EX-1004 0023.) This step is outlined in Figure 4.

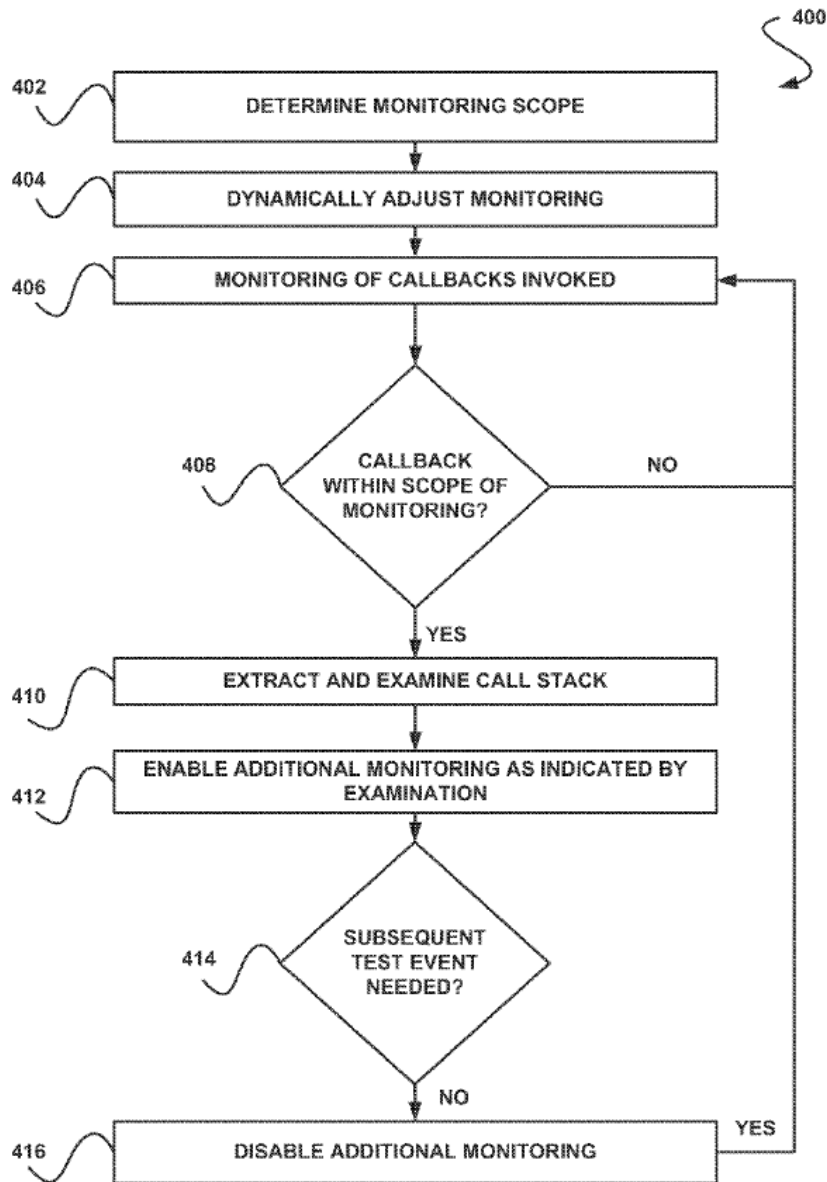


**FIGURE 4**

(EX-1004 at Figure 4.) Dalcher discloses determining a scope of monitoring, which could be limited to the test event generated earlier (i.e., tracing only the function executed as part of that test event). (EX-1004 0028.) Dalcher also discloses the use of “filter drivers (e.g., file system filters, etc.), event callbacks (e.g. registry

monitoring callbacks, etc.), etc” and “code hooking” when monitoring the execution flow based on the candidate stack frame. (EX-1004 0031.)

65. At step 410, Dalcher performs “stack analysis of the call stack.” (0036.) The analysis is outlined in Figure 5.

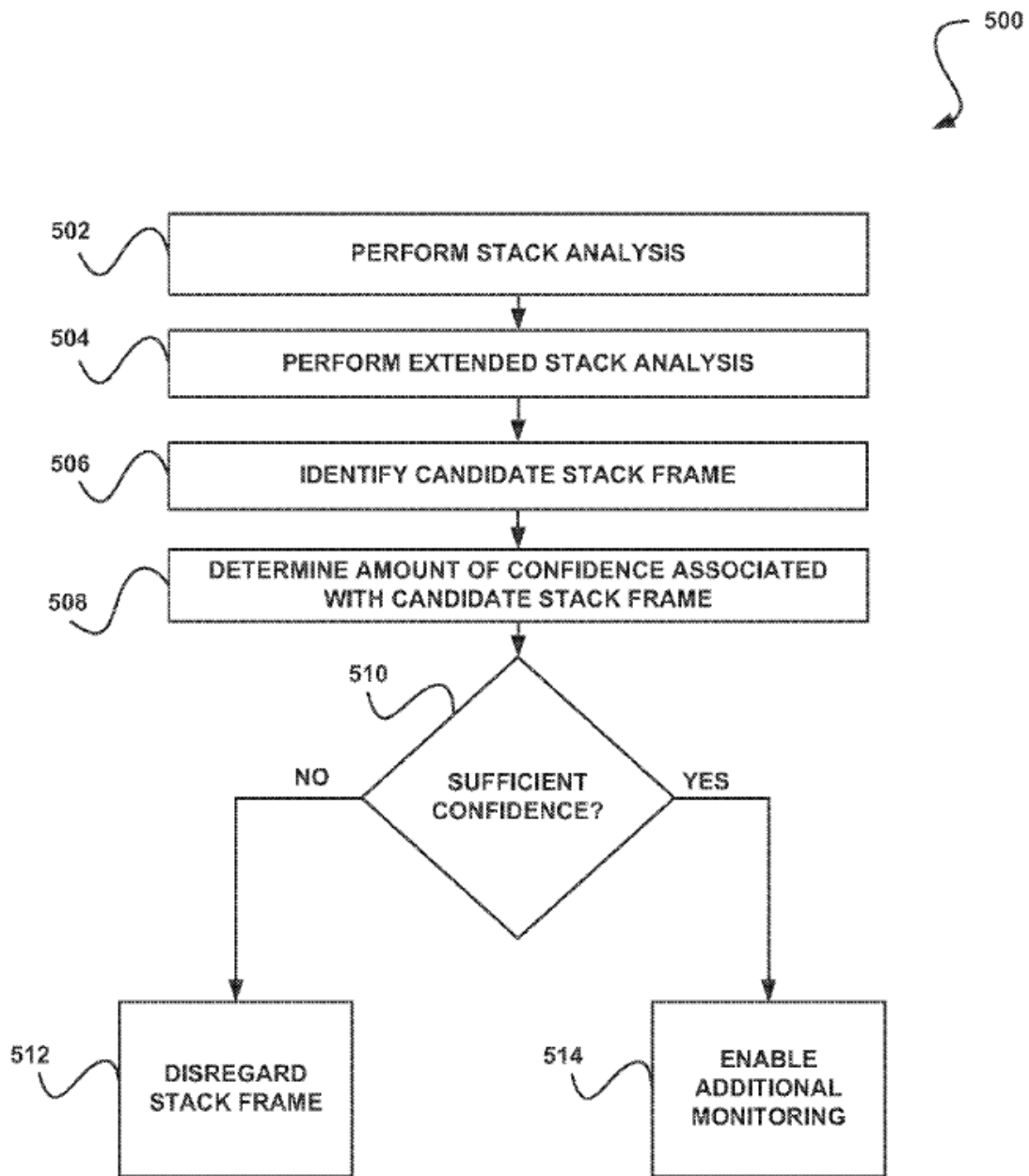


**FIGURE 5**

(EX-1004 at Figure 5.) Dalcher uses stack analysis and extended stack analysis to “determine an amount of confidence associated with the candidate stack frame.” (EX-1004 0039.) Confidence is a measure of whether a stack frame is “genuine.” (EX-1004 0041.) A genuine stack frame would be one that, for example, contains “a suspected call return address [that] points to code, and the code includes a call to the candidate stack frame” or, as another example, one that contains a link to a verifiably genuine stack frame. (EX-1004 0041.) Dalcher establishes a threshold value that it considers “sufficient confidence.” (EX-1004 0043.) A stack frame with insufficient confidence is disregarded, while one with sufficient confidence will become the target of additional monitoring. (EX-1004 0044.) In this way, Dalcher continues to follow the legitimate execution flow of the function by enabling further monitoring on those stack frames it considers sufficiently genuine. Additional monitoring includes, for example, monitoring “within code pointed to by the suspected call return address” of the candidate stack frame. (EX-1004 0044.)

66. Ultimately, Dalcher assembles “a picture of an execution flow” based on an analysis of the identified stack frames with sufficient confidence. (EX-1004 0048.) Because Dalcher disregards stack frames with insufficient confidence and does not indicate that they are used in any way, Dalcher does not generate any positive identifications of malware as Dr. Nielson suggests (EX-1002 ¶¶286-292.) Instead, Dalcher discloses that the system can flag if “other software was involved” in

handling the function or that “the system may be placed in quarantine.” (EX-1004 0026.) Neither of those examples is a positive identification of malware, but rather an indication that insufficient confidence was achieved in the monitoring process.

## **B. Overview of Pappas**

67. Pappas discloses “a practical runtime ROP exploit prevention technique for the protection of third-party applications.” (EX-1005 2:1.<sup>1</sup>) Return-oriented programming (ROP) exploits are techniques that utilize pieces of code from standard libraries in the computer’s memory known as “gadgets.” An attacker can chain together a series of gadgets by modifying the operation of a legitimate program. Through gadget chaining, attackers can essentially create what is a new, undesired program running on the computer that can be used to escalate privileges or perform malicious behavior.

68. To prevent ROP exploits, Pappas discloses a program called “kBouncer,” prototyped in the Windows 7 operating system. (EX-1005 9:2.) The kBouncer program is a module that operates in kernel space of the operating system and functions as an interface for inspecting the Last Branch Recording (“LBR”) stack. (EX-1005 Fig. 8.) The LBR stack is a series of model-specific registers used

---

<sup>1</sup> Citations to EX-1005 (Pappas) direct the reader to page and column, *e.g.*, EX-1005 2:1 directs the reader to page 2, column 1.

by certain types of Intel CPUs to “track[] the last N (16 for the CPU model [Pappas] used) most recent branches” of an executing program. (EX-1005 4:1.) Registers are hardware components that are part of the processor and are designed to store data that the processor uses. They are used because of their efficiency in storing and retrieving data. In the LBR stack, each branch record holds two values: the linear address of the branch instruction (where the code is located that initiated the jump) and the target instruction (where the process will jump to). (EX-1005 4:1.)

69. kBouncer operates by identifying two common characteristics of ROP techniques: ““i) illegal *ret* instructions that target locations not preceded by call sites, and ii) sequences of relatively short code fragments “chained” through any kind of indirect branches.” (EX-1005 6:2.) To identify the first characteristic, Pappas checks for whether the *ret* instruction points to a location in the code that is preceded by a function call (as a legitimate *ret* instruction would). (EX-1005:6:1-2.) A function call in the code is referred to as a “call site.” To accomplish call site detection, kBouncer “fetches a few bytes before each return target and attempts to decode any *call* instruction located right before the target instruction.” (EX-1005 10:2.) If no *call* instruction is found, the return is considered illegal. (EX-1005 7:1.) Pappas notes that for efficiency and simplicity, kBouncer “simply makes sure that *ret* instructions always target *any* among all valid call sites.” (EX-1005 7:2.) In other

words, Pappas does not verify that the call site identified actually corresponds to the executed function, instead Pappas only verifies that a call site exists.

70. To identify the second characteristic, Pappas checks for a sequence of gadgets exceeding eight gadgets in length. (EX-1005 9:2.) To identify gadgets, Pappas discloses an “offline gadget extraction and analysis toolkit,” which scans the executable segments of a protected application “in advance” and creates hash tables of the offsets of the gadgets. (EX-1005 9-10:1.) Pappas discloses that this scanning can be accomplished “on the fly at the time [modules] are loaded by a protected application.” (EX-1005 9-10:1.) Using the hash tables, “the number of consecutive targets that point to gadgets are counted” and if the threshold is exceeded, then Pappas considers the chain suspicious. (EX-1005 10:2.) Though Pappas discloses that a threshold of eight gadgets allows for “increased resilience to false positives” it is possible to use any threshold desired to accommodate either increased security or decreased false positives. (EX-1005 9:2.)

71. Pappas also discloses a checkpoint mechanism to ensure that malicious programs cannot bypass the kBouncer verification protocol. When a protected Windows API function is invocated, kBouncer “writes a checkpoint that denotes a legitimate invocation of that particular function.” (EX-1005 6:1.) Later, when the system call corresponding to the Windows API function is invoked, kBouncer “verifies that a proper checkpoint was previously set.” (EX-1005 6:1.) This checkpoint

mechanism ensures that code that has bypassed the kBouncer LBR stack verification is prevented from executing. (EX-1005 6:2.)

72. In operation, kBouncer is implemented as a two-tiered approach. First, kBouncer “selectively hooks the set of the protected Windows API functions.” (EX-1005 10:1.) By hooking the Windows API functions, kBouncer can detect when a protected function is executed and trigger the LBR stack verification to check for ROP exploit characteristics. When a protected function is called, kBouncer inspects the contents of the LBR stack for the characteristics described above. (EX-1005 10:1.) If “an attack attempt is detected after the analysis of the recorded branches, the process is terminated, and the user is informed with an alert message.” (EX-1005 10:2.)

73. In the second tier, after the LBR stack is checked, Pappas writes a checkpoint. (EX-1005 6:1.) When the system call corresponding to the Windows API function is invoked, kBouncer “verifies that a proper checkpoint was previously set.” (EX-1005 6:1.) The program is not allowed to continue unless the checkpoint was previously set. (EX-1005 6:2.)

74. Pappas is primarily a hardware-based solution to the ROP exploit technique. Pappas explains that “using hardware features of commodity processors” allows for “negligible runtime overhead” and “completely transparent operation without requiring any modifications to the protected applications.” (EX-1005 2:1.)

Relying mainly on hardware is what allows Pappas to achieve its claimed advantages. (EX-1005 3:1.)

**VI. GROUND 1: THE COMBINATION OF DALCHER AND PAPPAS DOES NOT DISCLOSE OR RENDER OBVIOUS THE CHALLENGED CLAIMS**

|                 |                |                                 |             |
|-----------------|----------------|---------------------------------|-------------|
| <b>Ground 1</b> | Section 103(a) | Obvious over Dalcher and Pappas | Claims 1-18 |
|-----------------|----------------|---------------------------------|-------------|

75. I understand that Petitioner and its expert allege that the combination of Dalcher and Pappas renders obvious every limitation of the independent claims 1, 9, and 17, and that this allegation underlies their challenge of the remaining dependent claims 3-8, 10-16, and 18. I disagree with the arguments in the Petition and opinions set forth by Dr. Nielson for at least the reasons that I discuss in further detail in this section.

**A. There Is No Motivation to Combine Dalcher with Pappas**

**1. A combination of Dalcher and Pappas would not enhance tracking of calls, jumps, and returns**

76. I disagree with Petitioner that a POSITA would have modified Dalcher in view of Pappas “in order to enhance the tracking of calls, jumps, and returns.” (Petition 13.) Dr. Nielson has not provided any reasonable explanation in support of the statement.

77. Pappas does not disclose a “tracking of calls, jumps, and returns” that would benefit Dalcher. Pappas utilizes a very limited record of branching history in the LBR stack. The LBR stack contains only the branch and target addresses of the last 16 jumps. (EX-1005 Fig. 6.) Dalcher makes complex call stack analysis determinations such as whether “a suspected call return address points to code, and the code includes a call to the candidate stack frame” or whether a “candidate stack frame contains a link to another stack frame, and the other stack frame can be verified.” (EX-1004 0041.) These determinations require the use of the call stack and would not be simplified, improved, or expedited by the use of the LBR stack.

## **2. Pappas would not reduce false positives in Dalcher**

78. I disagree with Petitioner and Dr. Nielson that “a POSITA would have been motivated to use Pappas for its low false positive rate” as a way to improve Dalcher, for several reasons. (EX-1002 ¶¶184, 321, 338, 350, 359, 374; Petition 13.) First, Dalcher does not generate false positives in a way that is ripe for improvement. Second, the method of controlling for false positives in Pappas is not applicable to (and thus would not benefit) Dalcher.

79. First, Dalcher does not acknowledge a risk of false positives nor would a POSITA be concerned of such a risk. As described above in Section V.A, Dalcher is directed to assembling “a picture of an execution flow” based on an analysis of the identified stack frames with sufficient confidence. (EX-1004 0048.) In Dalcher’s

method, “if it is determined in decision 510 that a sufficient amount of confidence is not associated with the candidate stack frame, then in operation 512 the candidate stack frame is disregarded.” (EX-1004 0043.) Though Dr. Nielson claims that Dalcher performs memory checks to identify suspicious behavior, he leaves out the important detail that stack frames with memory violations are *disregarded* and not relied upon. (EX-1002 ¶¶287-291.) Because candidate stack frames with insufficient confidence are disregarded—and Dalcher does not disclose taking any further action with disregarded stack frames—Dalcher does not disclose determining that malware is present. And because Dalcher never even explicitly makes a “positive” identification of malware, there is inherently no risk of false positives.

80. Second, Pappas’s technique for controlling for false positives is entirely inapplicable to Dalcher. As discussed above in Section V.B, Pappas achieves “increased resilience to false positives” by “set[ting] a robust detection threshold” of eight chained gadgets when checking for gadget chaining. (EX-1005 9: 2.) That is, Pappas’s solution to false positives is increasing the number of gadgets that must be chained before kBouncer detects that the rule has been violated and subsequently blocks the execution of the program. It is unclear how a POSITA would apply this teaching from Pappas to Dalcher and neither Petitioner nor Dr. Nielson make an effort to explain how to do so.

81. Dalcher would not benefit from an increased tolerance for gadget chains because Dalcher does not disclose monitoring for gadget chains. Dalcher's exemplary memory checks are "the call stack has not been overwritten," "other software was involved" in handling the test event, whether "a suspected call return address points to code, and the code includes a call to the candidate stack frame," and whether "the candidate stack frame contains a link to another stack frame" that can be verified. (EX-1004 0026, 0041.) Neither of these memory checks takes into account the number of instructions between a branch and its target, like Pappas's gadget chaining check. (EX-1005 9:1.) Because Dalcher has no equivalent check to Pappas's gadget chaining check, a POSITA would not be motivated to use Pappas to reduce false positives in Dalcher, since Pappas's technique would be inapplicable.

### **3. Dalcher and Pappas disclose fundamentally different teachings**

82. Petitioner and Dr. Nielson both claim that "[a] POSITA also would have seen that Pappas is compatible for combining with Dalcher" because "[b]oth systems are directed toward identifying malware that corrupts memory in one form or another and both systems make use of call stack analyses." (EX-1002 ¶186; Petition 13-14.) I disagree. At the outset, merely because both systems are directed "toward identifying malware that corrupts memory" is not an indication that they are compatible. Further, I disagree that "both systems make use of call stack analyses."

Finally, it is my opinion that the systems are not compatible because they rely on different parts of the computer system to operate (software approach vs. hardware approach and analysis of freed memory vs LBR stack).

83. Even taking Dr. Nielson's assumption that "both systems are directed toward identifying malware that corrupts memory" as true, a POSITA would not have relied on this as a motivation to combine Dalcher and Pappas. There are countless different approaches to detecting malware that corrupts memory, many of which would be incompatible in the sense that they would not be easily combined. For example, one approach would be to mark certain parts of memory as non-executable. (EX-1005 2:2.) Another approach would be to randomize address space (also known as address space layout randomization or "ASLR"). (EX-1005 2:2.) Pappas recognizes that the combination of these two techniques "is problematic." (EX-1005 2:2.) In the case of Pappas and Dalcher, combining the two reference is problematic because Pappas operates in real time to stop malware. In contrast, Dalcher must examine freed memory and thus must halt the execution flow of the program (even if temporarily) in order to ensure that freed memory is not overwritten. A POSITA would not find it obvious to combine these two techniques because it would require changing the mode in which each operates; this is a fundamental change that would undermine the effectiveness of each respective technique.

84. I disagree that both systems make use of “call stack analysis.” Pappas does not make use of “call stack analysis.” Though Pappas refers to the LBR registers as a “stack” they are not a true “call stack.” Instead, “[t]he registers act like a ring buffer that is continuously overwritten and provides only the most recent entries.” (EX-2020.) Moreover, the LBR registers are a hardware component that is part of the CPU. (EX-2020.) Additionally, the registers only store a specific type of information (the “from” and “to” addresses of each branch) and do not contain function-specific information. (EX-1005 4:1.) On the other hand, Dalcher uses the virtual memory stack in its analysis. This call stack operates like a true stack (hence the name), contains additional, function-specific information, and is implemented in software. (EX-1004 0017.)

85. Finally, Dalcher and Pappas are not compatible because they rely on different parts of the computer system to operate (software approach vs. hardware approach and analysis of freed memory vs LBR stack). Pappas “[relies] mainly on hardware” while Dalcher is a software-based implementation. (EX-1005 3:1; EX-1004 0014.) Pappas relies on hardware because it “allows for minimal runtime overhead and completely transparent operation, without requiring any modifications to the protected applications. (EX-1005 3:1.) On the other hand, Dalcher specifically discloses examining freed memory in the call stack. (EX-1004 0021.) This could not be achieved using hardware, because it requires analyzing virtual memory whose contents cannot be known ahead of time (until the software is executed). Put simply, virtual memory is not a physical component in the computer system that can be

analyzed through the use of hardware. Conversely, implementing Pappas using software would contradict Pappas's teachings because it would impact runtime, not be transparent to the running program, and would require modification of the underlying code. (EX-1005 4:2.)

86. Moreover, Dalcher operates as a post-hoc analysis by detecting issues in freed memory of the call stack, while Pappas monitors the LBR stack in real time. The two are not compatible, because each they each rely on an analysis of different computer components for different purposes. Dalcher discloses "stack analysis" as a means for determining "an amount of confidence associated with the candidate stack frame" and, as a result, whether that stack frame should be disregarded or whether Dalcher's monitoring method should enable additional monitoring to validate the candidate stack frame. (EX-1004 0039, 00042-0043, Fig. 5.) Pappas, however, analyzes the LBRs as a preliminary means to determine whether a checkpoint should be written, which will be used to authorize a subsequent invocation of a system call. (EX-1005-6 Fig. 3.) These are entirely distinct methods and goals, and the mere fact that both mention stack analysis does not mean that a POSITA would have been motivated to apply any teaching from one to the other. Indeed, Pappas's gadget chaining and illegal return checks could not be implemented by analyzing the call stack because of the risk that crucial information in the call stack was overwritten. For this reason, Pappas utilizes the LBR stack registers. Conversely, Dalcher could

not be implemented using the LBR stack registers because it relies upon an analysis of freed memory. There is no such concept of freed memory in the LBR stack.

**4. Dalcher and Pappas would produce conflicting results**

87. Besides all of the reasons discussed above, I also disagree that a POSITA would have been motivated to combine Dalcher and Pappas because a POSITA would have recognized that two systems could produce contradictory results. Faced with contradictory results, a POSITA would have his or her confidence undermined that either system has produced a correct result. Petitioner and Dr. Nielson both fail to account for how a POSITA would have disambiguated contradictory results between the implementations of Dalcher and Pappas.

88. There are several ways in which the Dalcher and Pappas systems could lead to conflicting results. For example, Dalcher determines whether a candidate call stack frame is genuine based on several criteria. One is whether “a suspected call return address points to code, and the code includes a call to the candidate stack frame.” (EX-1004 0041.) Thus, Dalcher identifies legitimate candidate call stack frames by analyzing whether their suspected return addresses point to a call site specific to the function associated with that candidate call stack frame. Meanwhile, as discussed above in Section V.B, Pappas checks for illegal returns only based on whether *any* call site is located at the branch address of the LBR stack register. (EX-1005-7:2.) This includes call sites that correspond to non-intended call instructions.

(EX-1005-7:2.) This is a simpler implementation intended to reduce processing overhead (allowing Pappas to be implemented in real time). (EX-1005 7:2.)

89. The discrepancy between Dalcher's requirement that a return address points to the specific call site that would invoke the execution of that call stack frame and Pappas's more relaxed requirement that a *ret* instruction target any among all valid call sites would lead to conflicting results. If a call stack frame is compromised such that the return address happens to point to a call site, but one belonging to a different function, Pappas would not flag that as suspicious, because it is a legitimate target "among all valid call sites." (EX-1005 7:2.) However, Dalcher would detect this as an ingenuine frame because the call site invocation does not match the candidate stack frame function. (EX-1004 0041.)

90. A POSITA faced with conflicting results between Dalcher and Pappas would have to disambiguate the two. Changing Dalcher to a more relaxed requirement like Pappas would undermine Dalcher's ability to recognize the legitimate execution flow of a function. Changing Pappas to a more stringent requirement like Dalcher would require increased processing overhead, contrary to Pappas's teachings to minimize processing time. As a result, a POSITA would not find it obvious to combine the two systems.

**B. Dalcher does not disclose every limitation of Independent Claims 1, 9, and 17**

**1. Limitation 1.e, 9.f, 17.e**

91. Dalcher does not disclose “performing, during the executing of the stack walk processing before an address of an originating caller function is reached, a memory check” as required by limitations 1.e, 9.f, and 17.e. As I explained above in section V.A, Dalcher begins its analysis by identifying “the data associated with the function” in the call stack. (EX-1004 0020, 0017, Fig. 3 (step 302).) Indeed, Dalcher explains that “the call stack frame may be associated with a first call stack entry of the function.” (EX-1004 0022.) From there, Dalcher broadens the scope of monitoring to search for candidate stack frames that can be verified. Indeed, because Dalcher moves through freed stack memory, Dalcher is necessarily monitoring previously executed call stack frames, which would require monitoring call frames likely on the opposite end of the stack of the “originating caller function.” And while Dalcher discloses “identifying a current top of the call stack” and discloses “moving through freed stack memory” to “identify one or more preserved call stack frames in the freed stack memory,” neither of these would suggest to a POSITA that Dalcher’s memory check takes place “before an address of an originating function is reached,” because neither implies where the stack walk begins or when the alleged memory check takes place.

92. As stated above, Dalcher explains that “freed memory” is “memory in the stack that previously held an active stack entry” that has been popped from the call stack. (Section V.A, *supra*; EX-1004 0021.) Freed memory is likely located at the opposite end of the call stack from the originating caller function (because it contains frames that have already been executed) and nothing in Dalcher indicates how Dalcher traverses a stack or whether the alleged memory check is performed before an address of an originating caller function is reached. Thus, because Dalcher begins its analysis at the call stack frame of the monitored function, it does not perform a memory check “before an address of an originating caller function is reached.”

93. Likewise, identifying a current top of the call stack simply means that the monitored function could be at the top of the call stack, because identifying a current top of the call stack does not imply that the stack was traversed to reach the top. That is, this disclosure still does not indicate where Dalcher’s stack walk begins, how it traverses the stack, where the originating caller function is located, or whether the alleged memory check is performed before an address of an originating caller function is reached.

94. Pappas, too, does not disclose performing a memory check “*during executing of stack walk processing before an address of an originating caller function is reached.*” Pappas only teaches that the Windows API function is hooked and does

not disclose an originating caller function. Because Pappas does not disclose an originating caller function, it cannot disclose that the memory checks are performed before the address of such originating caller function is reached.

95. Moreover, Dalcher does not disclose “wherein an alert of suspicious behavior is triggered when the performing of the memory check detects at least one of the following [elements]” as required by limitations 1.e, 9.f, and 17.e.

96. As I explained in Section VI.A.2, Dalcher does not disclose triggering “an alert of suspicious behavior.” Dalcher explicitly states that “the call stack frame may not be monitored if the call stack frame is determined to be compromised.” (EX-1004 0025.) That is, Dalcher explains that the criteria which Dr. Nielson contends are relevant to the analysis process are actually *disregarded* in the analysis process. Because these frames are disregarded, they are not used to generate an alert of suspicious behavior such as generating a flag. Instead, the behaviors that Dr. Nielson contends that “[a] POSITA would have recognized ... involves triggering an alert of suspicious behavior” are merely the result of Dalcher’s lack of confidence in the genuine execution flow of the function. (EX-1002 ¶291.) Dalcher does not purport to detect suspicious behavior, but rather only establishing confidence that “a handling of events is performed by expected and trusted software.” (EX-1004 0026.)

**2. Limitations 1.f, 9.g, 17.f**

97. Dalcher does not disclose “wherein when an alert of suspicious behavior is triggered, preventing execution of a payload for the invoked function from operating” as required by limitations 1.f, 9.g, and 17.f. Dr. Nielson contends that because Dalcher discloses analyzing whether “the call stack frame has been compromised” and whether “the call stack frame may include memory outside of that dictated by the call stack” that a POSITA would understand that Dalcher’s detection of such criteria “results in the quarantine process.” (EX-1002 ¶¶286-292.) I disagree.

98. The actions that Dr. Nielson identifies as disclosing “preventing execution of a payload ... from operating” are not accurate. (EX-1002 ¶291.) Dr. Nielson argues that “the system placed in quarantine blocks execution of many (or all) system operations, preventing the execution of the payload.” (EX-1002 ¶291.) Dalcher does not specify when the system is placed in quarantine and a POSITA would not understand that it occurs before the payload has been executed. Dalcher only explains that if there is insufficient confidence that the genuine execution flow of a function has been identified that remedial actions may be taken. As I previously stated, because Dalcher discloses disregarding compromised stack frames, Dalcher actually suggests that the remedial actions are not taken until analysis is complete—after the payload has been executed. If Dalcher inherently prevented the execution of a payload (which it does not), it would be redundant to then place the system in

quarantine; a POSITA would have understood that placing the system in quarantine is only necessary if execution of the payload cannot confidently be prevented.

99. Moreover, Dalcher discloses determining a call frame from freed memory in the call stack, i.e., a call frame that has already been executed and which is no longer “pending” execution. (EX-1004 0021.) More specifically, Dalcher monitors both frames that are pending execution and those that have already been executed. Dalcher does not disambiguate between call stack frames in active memory versus those in freed memory. Even more so, Dalcher does not clarify that its remedial actions are only as a result of identifying compromised frames in the active call stack. If, as Dr. Nielson suggests, Dalcher’s remedial actions are the result of a lack of confidence in particular call stack frames, there is no way to tell whether the lack of confidence stems from frames that are yet to execute or those that have already executed. This does not contradict Dalcher’s disclosure of remedial actions; it is still desirable to quarantine or further audit a machine that has been compromised, in order to analyze and prevent further spread of the exploit. Accordingly, Dalcher does not disclose that its remedial actions take place before the payload is executed.

## **VII. SECONDARY OBJECTIVE INDICIA OF NONOBVIOUSNESS**

100. I understand that in determining obviousness of the claimed invention, objective evidence of secondary considerations pertaining to nonobviousness must be considered, including long-felt but unsolved needs, failure of others, commercial

success, etc. *In re Huang*, 100 F.3d 135, 139 (Fed. Cir. 1996) (citing *Graham*, 383 U.S. at 17-18). I further understand that these indicia constitute independent evidence of nonobviousness that can be the most probative evidence in that regard “in avoiding the trap of hindsight when reviewing, what otherwise seems like, a combination of known elements.” *Leo Pharm. Prods., Ltd. v. Rea*, 726 F.3d 1346, 1358 (Fed. Cir. 2013).

101. As demonstrated by my summary of the prior art, above, there was a long-felt and persistent need to improve detection and neutralization of malicious exploits prior to the invention of the '591 patent. *See* Section III(B), *supra*. Prior attempts to develop reliable anti-exploit technology came with heavy costs associated with employee time and resources. *See* Section III(B), *supra*. Where others failed, the '591 patent succeeded. *See* Section IV(A), *supra*.

102. Thus, these secondary indicia only further support my opinion that the '591 patented method was non-obvious.

## **VIII. CONCLUSION**

103. For the foregoing reasons, it is my opinion that Petitioner's grounds do not anticipate or render obvious any of the challenged claims of the '591 patent.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true;

and further 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 1001 of Title 18 of the United States Code.

Executed this 11<sup>th</sup> day of August, 2023 in Los Angeles, California.

Respectfully submitted,

A handwritten signature in blue ink, appearing to read 'N Medvidovic', written over a horizontal line.

Nenad Medvidovic, Ph.D.