

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

**PETITION FOR *INTER PARTES* REVIEW OF
UNDER 35 U.S.C. § 312 AND 37 C.F.R. § 42.104**

TABLE OF CONTENTS

PETITIONER’S EXHIBIT LIST	iv
I. INTRODUCTION	1
II. MANDATORY NOTICES.....	1
A. Real Party-in-Interest.....	1
B. Related Matters.....	1
C. Counsel.....	2
D. Service Information	3
III. CERTIFICATION OF GROUNDS FOR STANDING	3
IV. TIME FOR FILING PETITION	4
V. SUMMARY OF THE ‘591 PATENT.....	6
VI. PROSECUTION HISTORY.....	10
VII. LEVEL OF ORDINARY SKILL IN THE ART	13
VIII. CLAIM CONSTRUCTION.....	13
IX. STATEMENT OF PRECISE RELIEF REQUESTED FOR EACH CLAIM CHALLENGED	15
X. PRIOR ART OVERVIEW	16
A. Fratric	16
B. Sallam.....	23
XI. IDENTIFICATION OF HOW THE CLAIMS ARE UNPATENTABLE	24
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	24
1. Claim 1	24
2. Claim 3.....	37
3. Claim 4.....	39
4. Claim 6.....	40
5. Claim 9.....	41

6. Claims 11, 12, 14.....	44
7. Claim 17	45
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	48
8. Reasons to Combine Fratric and Sallam	48
9. Claim 2	50
10. Claim 5	54
11. Claim 7	58
12. Claim 8	60
13. Claim 10	61
14. Claims 13, 15, 16	61
XII. DISCRETIONARY DENIAL WOULD BE INAPPROPRIATE	62
A. THE BOARD SHOULD NOT DENY INSTITUTION UNDER SECTION 314(a)	62
1. Factor 1: Whether the district court granted a stay or a stay may be granted if a proceeding is instituted	62
2. Factor 2: Proximity of the court’s trial date to the Board’s projected statutory deadline for a final written decision	63
3. Factor 3: Investment in the parallel proceeding by the court and the parties	64
4. Factor 4: Overlap between issues raised in the petition and in the parallel proceeding	65
5. Factor 5: Whether the petitioner and the defendant in the parallel proceeding are the same party	65
6. Factor 6: Other circumstances that impact the Board’s exercise of discretion, including the merits	66
B. THE BOARD SHOULD NOT DENY INSTITUTION UNDER <i>GENERAL PLASTIC</i>	67
1. Factor 1: Whether the Same Petitioner Previously Filed a Petition	67

2. Factor 2: Whether at the Time of Filing of the First Petition the Petitioner Knew of the Prior Art Asserted in the Second Petition or Should Have Known of It.....	69
3. Factor 3: Whether at the Time of Filing of the Second Petition the Petitioner Already Received the Patent Owner’s Preliminary Response to the First Petition	70
4. Factors 4 & 5: The Length of Time that Elapsed between the Time the Petitioner Learned of the Prior Art Asserted in the Second Petition and the Filing of the Second Petition, and Whether the Petitioner Provides Adequate Explanation.....	71
5. Factors 6 & 7: The Finite Resources of the Board, and Requirement to Issue a Final Determination within One Year from Institution	71
C. THE BOARD SHOULD NOT DENY INSTITUTION UNDER 35 U.S.C. § 325(d)	72
1. Advanced Bionics Step 1 (Becton Factors (a), (b), (d)) favors institution. 72	
2. <i>Advanced Bionics</i> Step 2 (<i>Becton</i> Factors (c), (e), (f)) favors institution. ..	73
XIII. CONCLUSION	76
CERTIFICATE OF COMPLIANCE UNDER 37 C.F.R. § 42.24(d).....	77
CERTIFICATE OF SERVICE.....	78

PETITIONER’S EXHIBIT LIST

Exhibit	Description
1001	U.S. Patent No. 10,284,591 (issued May 7, 2019)
1002	Declaration of V.S. Subrahmanian, Ph.D. under 37 C.F.R. §1.68
1003	Curriculum Vitae of V.S. Subrahmanian, Ph.D.
1004	Motion for Order Authorizing Alternative Service
1005	Waiver of Service of Summons
1006	File History of U.S. Patent No. 10,284,591
1007	Ivan Fratric, <i>Runtime Prevention of Return-Oriented Programming Attacks</i> to Ivan Fratric, https://github.com/ivanfratric/ropguard/blob/master/doc/ropguard.pdf (“Fratric”)
1008	U.S. Patent Publication No. 2012/0255018 (issued Oct. 4, 2012) (“Sallam”)
1009	<i>Hello World</i> , GITHUB DOCS, https://docs.github.com/en/get-started/quickstart/hello-world (last visited Apr. 26, 2023)
1010	<i>Committing and Reviewing Changes to your Project</i> , 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)
1011	<i>Setting Repository Visibility</i> , 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)
1012	<i>GitHub Glossary</i> , GITHUB DOCS, https://docs.github.com/en/get-started/quickstart/github-glossary#public-repository (last visited Apr. 26, 2023)
1013	Scott Chacon, <i>GitHub Code Search</i> , THE GITHUB BLOG (November 3, 2008), https://github.blog/2008-11-03-github-code-search/
1014	<i>About Searching on GitHub</i> , GITHUB DOCS, https://docs.github.com/en/search-github/getting-started-with-searching-on-github/about-searching-on-github (last visited Apr. 26, 2023)
1015	<i>Finding Ways to Contribute to Open Source on GitHub</i> , GITHUB DOCS, https://docs.github.com/en/get-started/exploring-projects-on-

Exhibit	Description
	github/finding-ways-to-contribute-to-open-source-on-github (last visited Apr. 26, 2023)
1016	Claim Construction Order, <i>Webroot, Inc. and Open Text, Inc. v. AO Kaspersky Lab</i> , Case No. 6:22-CV-00243-ADA-DTG, (Mar. 16, 2023)
1017	MICHAEL HUTTERMANN, <i>DEVOPS FOR DEVELOPERS</i> (Apress, 2012)
1018	Nicolas Despres, <i>Automatic performance monitoring tool</i> , LABORATOIRE DE RECHERCHE ET DEVELOPPEMENT DE L'EPITA TECH REPORT No 0601, 915 (2006)
1019	Bryan D. Payne, Martin D.P. de A. Carbone, Wenke Lee, 2007, December. <i>Secure and Flexible Monitoring of Virtual Machines</i> , 23 RD ANN. COMP. SEC. APPLICATIONS CONF., 385, 385-397 (2007)
1020	Margaret Rouse, <i>Dynamic Library</i> , TECHOPEDIA (Mar. 2, 2012)
1021	Krishnan, S. and Srihari, K., <i>A Knowledge-Based Object Oriented DFM Advisor for Surface Mount PCB Assembly</i> . 10 LI INT'L. J. ADVANCED MFG. TECH, 317, 317-329 (1995)
1022	Henry Hanping Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenkee Lee, and Barton P. Miller, <i>Formalizing Sensitivity in Static Analysis for Intrusion Detection</i> , IEEE SYMP. SEC. PRIV., 194, 194-208
1023	Wei Li, Lap-chung Lam, and Tzi-cker Chiueh, <i>How to Automatically and Accurately Sandbox Microsoft IIS</i> , 22 ND ANN. COMP. SEC. APPLICATIONS CONF., 213, 213-222 (2006)
1024	Kevin A. Roundy and Barton P. Miller, <i>Hybrid Analysis and Control of Malware</i> , RECENT ADVANCES IN INTRUSION DETECTION: 13 TH INT'L SYMP., 317, 317-338 (2010)
1025	Toshiyuki Maeda, 2002, <i>Safe Execution of User Programs in Kernel Mode Using Typed Assembly Language</i> , (Master's Thesis, University of Tokyo, 2002)
1026	Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimirov, Moshe Bach, <i>Dynamic Program Analysis of Microsoft Windows Applications</i> , IEEE INT'L SYMP. PERFORMANCE ANALYSIS SYS. SOFTWARE, 2, 2-12 (2010)
1027	Tarjei Mandt, T., <i>Locking Down the Windows Kernel: Mitigating Null Pointer Exploitation</i> , NORMAN THREAT RESEARCH
1028	Xiangyu Dong, Cong Xu, Yuan Xie, Norman P. Jouppi, N.P., <i>NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging</i>

Exhibit	Description
	<i>Nonvolatile Memory</i> , 31 IEEE TRANSACTIONS COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS SYS., 994, 994-1007 (2012)
1029	Father, H., <i>Hooking Windows API-Technics of hooking API functions on Windows</i> , 2 ASSEMBLY PROGRAMMING J., 2, 2-30 (2004)
1030	J. Berdajs and Z. Bosnić, <i>Extending Applications Using an Advanced Approach to DLL Injection and API Hooking</i> , 40 SOFTWARE: PRACTICE EXPERIENCE, 567, 567-584 (2010)
1031	Ulrich Bayer, Christopher Kruegel, and Engin Kirda, <i>TTAnalyze: A Tool for Analyzing Malware</i> , IKARUS SOFTWARE TECH. UNIV. VIENNA, 180, 180-192 (2006)

I. INTRODUCTION

Pursuant to 35 U.S.C. §§ 311, 314(a), and 37 C.F.R. § 42.100, AO Kaspersky Lab (“Petitioner”) respectfully requests that the Board review and cancel as unpatentable claims 1-17 (the “Challenged Claims”) of U.S. Patent No. 10,284,591 (the “’591 patent,” Ex. 1001). The ’591 patent describes systems and methods for detecting and preventing software exploits. Ex. 1001, Abstract. As shown below and confirmed in the Declaration of V.S. Subrahmanian, Ph.D. (See Ex. 1002, Ex. 1003), these concepts were well known, and the references presented in this Petition render unpatentable the Challenged Claims.

II. MANDATORY NOTICES

A. Real Party-in-Interest

The sole real party-in-interest for this Petition is AO Kaspersky Lab, Petitioner.

B. Related Matters

The Petitioner is the defendant in a case alleging infringement of the ‘591 patent styled *Webroot, Inc. and Open Text, Inc. v. AO Kaspersky Lab* (Case No.

6:22-CV-00243-ADA-DTG), pending in the Western District of Texas, Waco Division.

The Petitioner further understands that the ‘591 patent has been asserted in the following matters, which matters have been consolidated only for pre-trial purposes with the above matter: *Webroot, Inc. and Open Text, Inc. v. Trend Micro, Inc.* (Case No. 6:22-CV-00239-ADA-DTG); *Webroot, Inc. and Open Text, Inc. v. Sophos Ltd.* (Case No. 6:22-CV-00240-ADA-DTG); and *Webroot, Inc. and Open Text, Inc. v. CrowdStrike, Inc. and CrowdStrike Holdings, Inc.* (Case No. 6:22-CV-00241-ADA-DTG) (collectively the “Combined Parallel Litigation”). Each of the four cases will be tried separately.

Finally, the Petitioner understands that the ‘591 patent is the subject of the following IPR petitions: *CrowdStrike, Inc. v. Webroot, Inc.* in IPR2022-01522 (“CrowdStrike IPR”) and *Trend Micro Inc. v. Webroot, Inc.* in IPR2023-00692 (“Trend Micro IPR”).

C. Counsel

Petitioner designates Joseph Miotke (Reg. No. 47,798) as lead counsel and Eric Chadwick (Reg. No. 41,664), Christian J. Girtz (Reg. No. 68,613), and Erin Block (Reg. No. 78,154) as back-up counsel.

D. Service Information

Service of any documents may be made at the postal mailing address of Dewitt LLP, 901 Marquette Ave., 2100 AT&T Tower, Minneapolis, MN 55402, (Tel: (612) 305-1400 and Fax: (612) 305-1414).

Petitioner consents to electronic service by email to the email addresses:

jtm@dewittllp.com;

ehc@dewittllp.com;

cgirtz@dewittllp.com;

eblock@dewittllp.com; and

docketing@dewittllp.com.

III. CERTIFICATION OF GROUNDS FOR STANDING

Petitioner certifies that the '591 patent is eligible for IPR and that Petitioner is not barred or estopped from requesting IPR challenging the patent claims. 37 C.F.R. § 42.104(a).

IV. TIME FOR FILING PETITION

On March 4, 2023, Patent Owner filed a complaint asserting the ‘591 Patent in the Western District of Texas. However, Patent Owner was unable to effect service on Petitioner and, on March 24, 2023, filed a motion for an order authorizing alternative service. Ex. 1004. On May 24, 2022, at the request of counsel for Patent Owner, counsel for Petitioner executed a Waiver of Service Summons, and on May 25, 2022, the executed Waiver was filed with the Court. Ex. 1005.

Section 315(b) states that “[a]n inter partes review may not be instituted if the petition requesting the proceeding is filed more than 1 year after the date on which the petitioner, real party in interest, or privy of the petitioner is served with a complaint alleging infringement of the patent.” 35 U.S.C. § 315(b). The “plain meaning of the phrase ‘served with a complaint’ is ‘presented with a complaint’ or ‘delivered a complaint’ in a manner prescribed by law.” *Game and Tech. Co. v. Wargaming Group Ltd.*, 942 F.3d 1343, 1349 (Fed. Cir. 2019), quoting *Click-To-Call Techs., LP v. Ingenio, Inc.*, 899 F.3d 1321, 1330 (Fed. Cir. 2018), cert. granted in part sub nom; *Dex Media, Inc. v. Click-To-Call Techs., LP*, 139 S. Ct. 2742 (2019).

A patent owner who files a lawsuit in a federal court and a petitioner who chooses to waive service of a summons are required to comply with the Federal Rules of Civil Procedure in the infringement action. Rule 4(d)(4) of the Federal Rules of Civil Procedure provides that “[w]hen the plaintiff files a waiver, proof of service is not required and these rules apply *as if a summons and complaint had been served at the time of filing the waiver*” (emphasis in original). “Therefore, in the situation where the petitioner waives service of a summons, the one-year time period begins on the date on which such a waiver is filed.” *Motorola Mobility LLC v. Patent of Michael Arnouse*, IPR2013-00010, Paper 20, at 6 (PTAB Jan 30, 2013); *See also Macauto U.S.A. v. BOS GMBH & KG*, IPR2012-00004, Paper 18, at 16 (PTAB Jan. 24, 2013) (to establish a service date the waiver or proof of service must be filed where Petitioner signed and returned a waiver of service under Fed.R.Civ.P. 4(d)).

Having not been served a complaint alleging infringement of the ‘591 patent, as conceded by Patent Owner in its motion for alternative service, and ultimately acquiescing to waiver of service, Petitioner is filing this petition within the one-year period from the time the waiver of service was filed on May 25, 2022, in accordance with 35 U.S.C. § 315(b).

V. SUMMARY OF THE ‘591 PATENT

The ‘591 patent is directed to system and method of 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, 3:37-41. 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, col. 1:13-17, *see also* 3:29-32.

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. 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, whether the malicious payload is loading another module or executing another process.” Ex. 1001, 6:24-30.

To detect attempted exploits and prevent the execution of a malicious payload, the anti-exploit examples disclosed in the ‘591 patent (1) “monitor a memory space of a process for execution of functions,” Ex. 1001, 1:34-35, and (2) “perform[] at least one check to evaluate how memory is being allocated before setting/mark[ing] 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.

Monitoring a memory space includes 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.

The ‘591 patent further describes using a “function hooking technique” for monitored functions that “function hooking technique 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, 5:27-29. “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.

“The 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.

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. “When 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.

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 [sic] 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.

VI. PROSECUTION HISTORY

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.

Over several rounds of prosecution and several amendments, the Applicant was able to overcome the Examiner’s rejections.

First, in the June 16, 2017, response to non-final office action, Applicant amended the claims to include that monitoring a memory space requires “loading a

component for evaluating the at least one monitored function in the memory space” and argued that Sallam’s security agent is 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, pp. 122-126, 128.

Second, in the February 28, 2018, response to office action, Applicant amended the claims to require the step of “invoking one of the plurality of functions as a result of receiving a call from an application programming instance” and 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. Ex. 1006, pp. 90-94, 96. 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. 96-98.

Third, in the October 1, 2018, response to office action, Applicant amended the claims to require that “a memory check for the plurality of stack entries identified during the stack walk processing to detect suspicious behavior” is performed. Ex. 1006, pp. 40-45. Applicant further amended the claims 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,” and 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.” Ex. 1006, p. 47. 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.” *Id.* 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.” Ex. 1006, p. 48. Applicant further 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.”” *Id.* Accordingly, the pointer and virtual memory conditions in Sallam are performed during the stack walk processing, as required by the claims. *Id.*

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. Ex. 1006, p. 20-26, 12-19.

VII. LEVEL OF ORDINARY SKILL IN THE ART

A Person of Ordinary Skill in The Art (“POSITA”) in January 2014 (the earliest alleged priority date) 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. Ex. 1002, ¶¶12-13.

VIII. CLAIM CONSTRUCTION

In an *inter partes* review, claims “shall be construed using the same claim construction standard that would be used to construe the claim in a civil action under

35 U.S.C. § 282(b), including construing the claim in accordance with the ordinary and customary meaning of such claim as understood by one of ordinary skill in the art and the prosecution history pertaining to the patent.” 37 C.F.R. § 42.100(b). The Board only construes the claims to the extent necessary to resolve the underlying controversy. *Nidec Motor Corp. v. Zhongshan Broad Ocean Motor Co.*, 868 F.3d 1013, 1017 (Fed. Cir. 2017).

The Court in *Webroot, Inc. and Open Text, Inc. v. AO Kaspersky Lab* (Case No. 6:22-CV-00243-ADA-DTG) issued its claim construction order on March 16, 2023. Ex. 1016. All terms that the parties requested be construed were ascribed their plain and ordinary meaning without further elaboration except for “low level use mode function.” *Id.* For that term, the Court's construction was “plain-and-ordinary meaning” wherein the plain and ordinary meaning is “user mode function that is low level with respect to its layer of abstraction.” *Id.*

Petitioner submits that for the purposes of this proceeding, the terms of the Challenged Claims should be given a construction consistent with the Court’s order.¹

¹ Petitioner does not concede that any term in the Challenged Claims meets the statutory requirements of 35 U.S.C. § 112.

IX. STATEMENT OF PRECISE RELIEF REQUESTED FOR EACH CLAIM CHALLENGED

Petitioner requests review under 35 U.S.C. § 311 and cancelation of claims 1-17 in view of the following references and grounds:

Grounds	Claim(s)	Basis
1A/1B	1, 3, 4, 6, 9, 11, 12, 14, 17	§ 102 Fratric (1A) or § 103 Fratric (1B)
2	2, 5, 7, 8, 10, 13, 15, 16	§ 103 Fratric and Sallam

Non-patent literature to Fratric (“Fratric,” Ex. 1007) was published at least as early as August 26, 2012. Fratric is thus prior art under at least 35 U.S.C. § 102(a)(1).

U.S. Patent Application Publication No. 2012/0255018 to Sallam (“Sallam”, Ex. 1008) was published as an application for patent on October 4, 2012. Sallam is thus prior art under at least 35 U.S.C. § 102(a)(2).

Petitioner’s analysis also cites additional prior art to demonstrate the background knowledge of a POSITA and to provide contemporaneous context to support Petitioner’s assertions regarding what a POSITA would have understood from the prior art. *See Yeda Research v. Mylan Pharm. Inc.*, 906 F.3d 1031, 1041-

1042 (Fed. Cir. 2018); 37 C.F.R. § 42.104(b); *see also K/S HIMPP v. Hear-Wear Techs., LLC*, 751 F.3d 1362, 1365-66 (Fed. Cir. 2014).

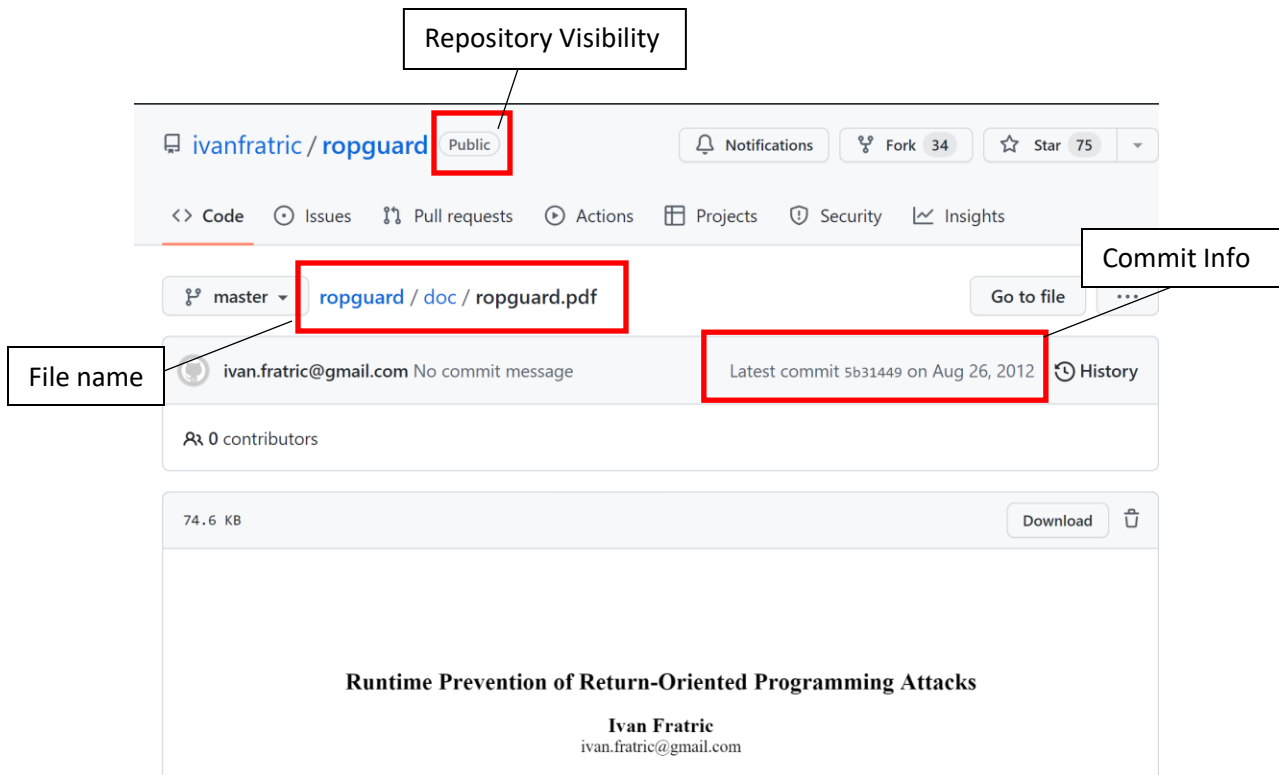
Each of these grounds is discussed in detail below.

X. PRIOR ART OVERVIEW

A. Fratric

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. Fratric was published at least by August 26, 2012, to GitHub, an Internet hosting service for software development and version control that is commonly used to host open-source and other public software projects. Ex. 1009; Ex. 1002, ¶¶51-52.

Fratric is located in the “ropguard.pdf” document in the GitHub repository at: <https://github.com/ivanfratric/ropguard/blob/master/doc/ropguard.pdf>, as depicted in the annotated screenshot below:



Fratric was published at least as of the GitHub “Latest commit” date of August 26, 2012, displayed on the publicly-accessible GitHub, Inc. webpage. Ex. 1002, ¶77. “[A] commit records changes to one or more files in your [GitHub] branch. Git assigns each commit a unique ID, called a SHA or hash, that identifies: [t]he specific changes[,] [w]hen the changes were made[,] [w]ho created the changes.” Ex. 1010. Ex. 1002, ¶77.

The repository in which Fratric is stored is publicly viewable. Ex. 1002, ¶77. GitHub repositories can be designated “public,” “private” (hidden from the public), or “internal” (visible only to those in an organization). Ex 1011. GitHub branches in public repositories can be viewed by anyone, including people who aren't GitHub

users. Ex. 1012; Ex. 1002, ¶77. The GitHub repository of Fratric is a public repository. Ex. 1002, ¶77.

Therefore, Fratric was sufficiently accessible to a POSITA at least as of the date of publication on GitHub. *See e.g., In re Cronyn*, 890 F.2d 1158 at 1161 (Fed. Cir. 1989).

GitHub is meaningfully searchable and has been since at least August 2012. *See* Ex. 1013 (November 3, 2008, Blog: “GitHub Codesearch can be found at github.com/codesearch and will let you type in anything you’re looking for in source code and get highlighted results of any files in our public repositories that match.”) For example, GitHub’s integrated search covers the many repositories, users, and lines of code on GitHub. One can search for the following information across all repositories you can access on GitHub.com: Repositories, Topics, Issues and pull requests, Discussions, Code, Commits, Users, Packages, Wikis. Ex. 1014. In another example:

[i]f there's a particular topic that interests you, visit github.com/topics/<topic>. For example, if you are interested in machine learning, you can find relevant projects and good first issues by visiting <https://github.com/topics/machine-learning>. You can browse popular topics by visiting Topics. You can also search for repositories that match a topic you're interested in.

Ex. 1015; *See also e.g., In re Lister*, 583 F.3d 1307 (Fed. Cir. 2009)

(discussing public accessibility of prior art maintained in commercial database).

Therefore, Fratric was sufficiently publicly accessible because a POSITA interested in preventing return-oriented programming would have been independently aware of GitHub and its open-source and public software projects repositories, one of which was Fratric. Ex. 1002, ¶¶51-52. More particularly, a POSITA interested in return-oriented programming would have known to look at GitHub for relevant information and then exercised reasonable diligence in executing index-based or keyword searches to find Fratric. Ex. 1002, ¶¶51-52, 77.

Like the ‘591 patent, Fratric relates to prevention of return-oriented programming attacks. 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.” Ex. 1007, p. 1.

Fratric recognizes 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. 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.* “[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.

At least six checks are described 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.

A first check is checking the stack pointer. In “most forms of ROP attack, the attacker will need to have control over the stack.” *Id.* “[A]t 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.*

A second check is looking for the address of critical function on the stack. *Id.* at 5. “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.*

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. 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.

A third check is checking the return address of the critical function. *Id.* Each return address must be executable and the instruction at the return address must be preceded with a CALL instruction. *Id.* “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.*

A fourth check is checking the stack frames. *Id.:*

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.

A stack walk is described in pseudocode in Listing 2 of Fratric.

```
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 4-5.

A fifth check is simulating the execution flow of a small number of instructions after the return from the critical function. *Id.* at 6-7.

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.

A sixth check is a function-specific check related to the arguments of the critical function called. *Id.* at 7. “ROPGuard can prevent VirtualProtect [sic] calls in

which the program attempts to change the memory protection options of the stack.”

Id. “ROPGuard can prevent LoadLibrary (and similar) calls in which a program attempts to load a library over SMB.” *Id.*

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

B. Sallam

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

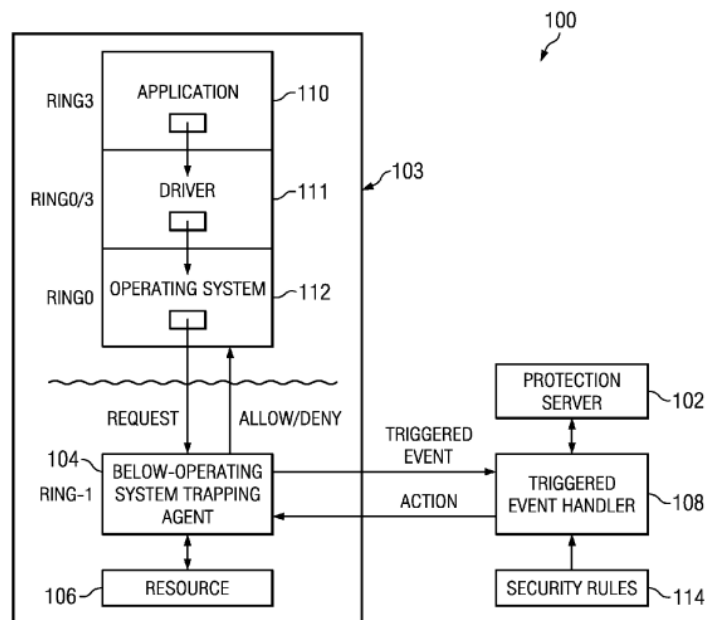


FIG. 1

Id. at FIG. 1.

In particular, Sallam discloses 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.” *Id.* at ¶0007. 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.* Sallam also determines if the attempted access was indicative of malware. *Id.* at ¶0054.

XI. 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

Fratric teaches every element of claims 1, 3, 4, 6, 9, 11, 12, 14, and 17 (Grounds 1A/1B).

1. Claim 1

[1.0] A computer-implemented method comprising:

To the extent the preamble is limiting, 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, p. 1; Ex. 1002, ¶ 94-96; *See also* Ex. 1007 at 9 (“[t]o test the ROPGuard’s **computing** overhead, a series of benchmarks was performed with and without the protection. The default configuration was used. Benchmarks were performed on a **computer** with an i7 720QM processor and 4GB RAM running Windows 7 64-bit.”). Since Fratric’s method of detection and prevention is implemented on a “*computer*,” Fratric discloses a *computer-implemented method*.² Ex. 1002, ¶97.

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

Fratric discloses this limitation. Ex. 1002, ¶98.

First, Fratric discloses execution of at least one monitored function of a plurality of functions. An “attacker will need to call at least one **critical function** from the ROP code.” Ex. 1007, p. 4. A critical function (*monitored function*) “is 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

² **Emphasis** in quoted material has been added. Claim terms are presented in *italics*.

necessary or by ‘escaping’ the current process.” *Id.*; Ex. 1002, ¶¶99-100. “[E]xamples of **critical functions include**” CreateProcess, VirtualProtect, VirtualAlloc, LoadLibrary, and OpenFile (using WriteFile function, which may also be consider[ed] critical). Ex. 1007, p. 3; Ex. 1002, ¶¶101-102.

Second, Fratric discloses monitoring a memory space of a process. “The system can be applied at runtime to any **process**.” *Id.* at 1. “[S]ix checks are defined which can be performed at each critical function call.” *Id.* at 4 (each describing “*monitoring a memory space*”); Ex. 1002, ¶¶53, 54-63, 74-76, 104-106, citing Exs. 1018, 1020.

In checking a stack pointer, “ROPGuard checks if the **stack pointer** is inside the **boundaries of the current thread’s stack**.” Ex. 1007, p. 4. 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.* at 5; Ex. 1002, ¶107.

In checking a return address, “[a]t 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 and the instruction at the return address must be preceded with a CALL instruction. *Id.*; Ex. 1002, ¶¶108-109.

In checking stack frames, “[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.” Ex. 1007, p. 5; Ex. 1002, ¶110.

In simulating execution flow:

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**.

Ex. 1007, p. 6; Ex. 1002, ¶112.

In function-specific checks, “ROPGuard can prevent VirtualProtect [sic] calls in which the program **attempts to change the memory protection options** of the stack.” Ex. 1007, p. 7. “ROPGuard can prevent LoadLibrary (and similar) calls in which a program attempts to **load a library over SMB**.” *Id.*; Ex. 1002, ¶113.

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

Fratric discloses this limitation. Ex. 1002, ¶114.

First, as discussed in [1.1.a], Fratric discloses *monitoring a memory space*.

Second, Fratric further discloses loading a component for evaluating the at least one monitored function in the memory space. “When ROPGuard is invoked to protect an existing process it **will inject its dll**” (*loading a component for evaluating*) **“into the address space of the target process**. Ex. 1007, p. 8. “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.*; Ex. 1002, ¶¶115-119, citing Ex. 1021.

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

Fratric discloses this limitation. Ex. 1002, ¶120.

[T]he attacker will need to **call at least one critical function**” (*invoking one of the plurality of functions*) **“from the ROP code**” (*application*

programming instance). 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, p. 4; Ex. 1002, ¶¶121-122.

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

Fratric discloses this limitation. Ex. 1002, ¶123. Fratric discloses, “[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.**” Ex. 1007, p. 5. An “algorithm for **performing the stack frame checks**” is provided in Listing 2. *Id.* Stack walk processing operations are annotated in Listing 2 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

Id. at 5-6; Ex. 1002, ¶¶54-65, 123-132, citing Exs. 1022-1024.

[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,

Fratric discloses this limitation. Ex. 1002, ¶133.

First, as discussed in [1.3], Fratric discloses *executing of stack walk processing*. Ex. 1002, ¶¶69-70, 134-135.

Second, in checking stack frames, 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” (*address of an originating caller function*). Ex. 1007, p. 5; Ex. 1002, ¶136.

Third, Fratric discloses *a memory check for a plurality of stack entries identified during the stack walk* because ROPGuard walks the stack using frame pointers (“get the return address of the current frame”) 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). Ex. 1007, pp. 5-6; Ex. 1002, ¶¶54-6

```
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 1 – Current Frame

Box 2 – Next Frame

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

Fourth, Fratric stack walk pseudocode discloses example stack frame checks:
“**[i]f the current frame pointer does not point on the stack”** (*memory check*),
“**[a]lert the user about the ROP attempt”** (*detect suspicious behavior*); “**[i]f the**

frame pointer points above the stack pointer” (*memory check*), “[a]llert the user about the ROP attempt; (*detect suspicious behavior*); “[i]f the return address is not executable or not preceded by call” (*memory check*), “[a]llert the user about the ROP attempt” (*detect suspicious behavior*); and “[i]f the new frame pointer is not below the previous frame pointer or does not point on the stack” (*memory check*), “[a]llert the user about the ROP attempt;” (*detect suspicious behavior*). Ex. 1007, pp. 5-6; Ex. 1002, ¶¶139-141.

The Fratric stack walk pseudocode discloses the above example memory checks during the frame-by-frame stack walk before reaching the address of an originating caller function because the memory checks are conducted during the stack walk, either on the current frame, or on the new frame prior to reaching the originating function.

Fratric’s surrounding disclosure would have confirmed to a POSITA that Fratric’s memory checks would have been done *before reaching the address of an originating caller function* because ROPGuard walks the stack frame-by-frame to the bottom of the stack (towards the originating function) or a POSITA would have found this obvious based on Fratric’s disclosure. Ex. 1002, ¶142.

[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,

Fratric discloses this limitation. Ex. 1002, ¶143.

First, as discussed in [1.4.a], Fratric discloses *an alert of suspicious behavior is triggered when performing of the memory check*. Ex. 1002, ¶144.

Second, 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. *Brown v. 3M*, 265 F.3d 1349, 1351, 60 USPQ2d 1375, 1376 (Fed. Cir. 2001). Fratric's memory checks detect at least one of the alternatives recited. Specifically, Fratric discloses 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, as explained in the subsequent paragraphs. Ex. 1002, ¶145-146.

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**” (*code execution is attempted from non-executable memory*). Ex. 1007, p. 5; Ex. 1002, ¶147. “ROPGuard can prevent VirtualProtect [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” (*code execution is attempted from non-executable memory*). Ex. 1007, p. 6; Ex. 1002, ¶148.

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” (*base pointer is identified as being invalid*). Ex. 1007, p. 5; Ex. 1002, ¶¶149-152.

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” (*invalid stack return address is identified*). Ex. 1007, p. 5; Ex.

1002, ¶150. 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” (*invalid stack return address is identified*). Ex. 1007, p. 7; Ex. 1002, ¶151. “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” (*invalid stack return address is identified*). Ex. 1007, p. 5; Ex. 1002, ¶151.

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**” (*attempted execution of a return-oriented programming technique is detected*). Ex. 1007, p. 10; Ex. 1002, ¶153.

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; Ex. 1002, ¶152.

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

Fratric discloses this limitation. Ex. 1002, ¶154.

First, as discussed in [1.4.b], Fratric discloses *an alert of suspicious behavior is triggered*. Ex. 1002, ¶155.

Second, Fratric discloses “[a] system called ROPGuard was developed that can detect and **prevent** the currently used forms of ROP attacks.” Ex. 1007, p. 10; Ex. 1002, ¶156.

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; Ex. 1002, ¶157.

Fratric accordingly anticipates claim 1 because it discloses each limitation as recited in the claim. To the extent Fratric does not anticipate claim 1, Fratric still renders claim 1 obvious as explained above. One skilled in the art would have a

reasonable expectation of successfully achieving the method of claim 1 because the disclosure of Fratric coupled with the knowledge of one skilled in the art involved nothing more than the use of known components in accordance with their conventional uses in a predictable art. Ex. 1002, ¶¶53-65, 71-76, 158.

2. Claim 3

[3.0] The method according to claim 1,

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.

Fratric discloses this limitation. Ex. 1002, ¶¶160-161.

First, Fratric’s ROPGuard “protects only the *functions in the user mode.*” Ex. 1007, p. 4; Ex. 1002, ¶163. 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.*

Second, Fratric discloses “check not only the **return address of the**

critical function” (*associated with the invoked function*) but also “check the return addresses of the functions the current critical function was called from” (*originating address of the invoked function*). Ex. 1007, p. 5; Ex. 1002, ¶161-162.

Third, Fratric’s stack walk “**us[es]the stack frames**” (*walking a stack frame*) to check the return addresses of the **functions that the critical function will return into**” (*originating address of the invoked function*). Ex. 1007, p. 5; Ex. 1002, ¶¶ 54-65, 164.

To the extent Fratric does not explicitly disclose walking *from* a low level user mode function to the originating address of the invoked function, which Petitioner does not concede, it would have been obvious to a POSITA because of the way the stack is organized and traversed to check functions that the critical function returns into (towards the bottom of the stack and the originating function). *See* [1.3]; Ex. 1002, ¶¶161-162.

One skilled in the art would have a reasonable expectation of successfully achieving the method of claim 3 because the disclosure of Fratric coupled with the knowledge of one skilled in the art involved nothing more than the use of known components in accordance with their conventional uses in a predictable art. Ex. 1002, ¶¶165-166.

3. Claim 4

[4.0] The method according to claim 1,

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.

Fratric discloses this limitation. Fratric's ROPGuard "protects only the *functions in the user mode.*" Ex. 1007, p. 4; Ex. 1002, ¶¶168-170, citing Ex. 1025. Fratric's simulation algorithm evaluates "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; Ex. 1002, ¶171.

With reference to Ground 1B, one skilled in the art would have a reasonable expectation of successfully achieving the method of claim 4 because the disclosure of Fratric coupled with the knowledge of one skilled in the art involved nothing more than the use of known components in accordance with their conventional uses in a predictable art. Ex. 1002, ¶¶172-173.

4. Claim 6

[6.0] The method according to claim 1,

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.

Fratric discloses this limitation. Fratric's:

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; Ex. 1002, ¶¶175-183, citing Exs. 1026-1027.

With reference to Ground 1B, one skilled in the art would have a reasonable expectation of successfully achieving the method of claim 6 because the disclosure of Fratric coupled with the knowledge of one skilled in the art involved nothing more

than the use of known components in accordance with their conventional uses in a predictable art. Ex. 1002 ¶¶184-185.

5. Claim 9

[9.0] A system comprising:

To the extent the preamble is limiting, 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, p. 1; Ex. 1002 ¶¶186-187.

[9.1] at least one memory; and

Fratric discloses this element because Fratric discloses implementation using memory. Fratric explains that “[t]he system can be applied at runtime to any process and has a very low computing and **memory** overhead.” Ex. 1007, p. 1; *see also id.* at 10 (“[t]o test the ROPGuard’s computing overhead, a series of benchmarks was performed with and without the protection. The default configuration was used. Benchmarks were performed on a computer with an i7 720QM processor and **4GB RAM** running Windows 7 64-bit.”); *see also id.* (“[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. 1002, ¶¶188-191, citing Ex. 1028.

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

Fratric discloses this element because Fratric discloses implementation on a processor and its connected memory. "Benchmarks were performed on a computer with an **i7 720QM processor and 4GB RAM** running Windows 7 64-bit." *Id.*; see also *id.*, ("[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.") Therefore, a POSITA 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. Ex. 1002, ¶¶192-195.

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

See [1.1.a]. Ex. 1002, ¶196.

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

See [1.1.b]. Ex. 1002, ¶197.

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

See [1.2]. Ex. 1002, ¶198.

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

See [1.3]. Ex. 1002, ¶199.

[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,

See [1.4.a]. Ex. 1002, ¶200.

[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,

See [1.4.b]. Ex. 1002, ¶201.

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

See [1.5]. Ex. 1002, ¶¶202-203.

6. Claims 11, 12, 14

Claims 11, 12, and 14 depend from claim 9 and are substantially similar to respective claims 3, 4, and 6. See analysis in [3.0]-[3.1], [4.0]-[4.1], and [6.0]-[6.1]. Ex. 1002, ¶204.

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:

To the extent the preamble is limiting, Fratric discloses memory (“a *computer-readable storage device*”) having instructions for execution by a processor. Specifically, Fratric discloses “[b]enchmarks were performed on a computer with an i7 720QM processor and 4GB RAM running Windows 7 64-bit.” Ex. 1007, p. 10. Fratric specifically teaches that “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.” A POSITA would understand that the executable, dll, and/or configuration file are the instructions for execution by the processor stored in the computer’s memory (in the same folder). Ex. 1007, p. 7; Ex. 1002, ¶205-207.

Therefore, a POSITA 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 to execute the ROPGuard process. Ex. 1002, ¶208.

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

See [1.1.a]. Ex. 1002, ¶209.

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

See [1.1.b]. Ex. 1002, ¶210.

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

See [1.2]. Ex. 1002, ¶211.

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

See [1.3]. Ex. 1002, ¶212.

[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,

See [1.4.a]. Ex. 1002, ¶213.

[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

See [1.4.b]. Ex. 1002, ¶¶214-215.

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

See [1.5]. Ex. 1002, ¶¶216-217.

**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

A POSITA would have been motivated to combine the teachings of Fratric and Sallam for multiple reasons, including to produce the obvious, beneficial, and predictable results of detecting and preventing different forms of ROP attacks. Doing so would have been within the POSITA's skillset and would produce beneficial, predictable results. Ex. 1002, ¶218.

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; Ex. 1002, ¶219.

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. Ex.1002, ¶220.

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. Ex. 1002, ¶221. In particular, 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. Ex. 1002, ¶¶222-223.

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. In light of this shortcoming (e.g., an attack using the data fetched into the cache from an indirect jump), a POSITA would be motivated to augment Fratric with a known technique for detection of ROP in cache. Ex. 1002, ¶224. Sallam provides that technique. *See also KSR Int'l Co. v. Teleflex Inc.*, 550 U.S. 398, 419-20 (2007) ("[A] patent's subject matter can be proved obvious is by noting that there existed at the time of invention a known problem for which there was an obvious solution encompassed by the patent's claims.")

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 recited through this combination. Ex. 1002, ¶¶225-226, citing Exs. 1029-1031. 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. Such expectation is evident because such combination 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 its established function (ROP detection in cache, including for example, by hooking). Ex. 1002, ¶¶66-68, 227-228; *see also* KSR, 550 U.S. at 417 (finding obvious "the predictable use of prior art elements according to their established functions").

9. Claim 2

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

See [1.0]-[1.5].

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

Fratric alone or combined with Sallam renders this limitation obvious. Ex. 1002, ¶230.

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” (*identifying address associated with the function*). Ex. 1007, p. 5. “Each return address must . . . be **executable**” (*invoked*). *Id.* Ex. 1002, ¶¶231-232.

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

Fratric alone or combined with Sallam renders this limitation obvious. Ex. 1002, ¶233.

First, as discussed in [1.5], Fratric discloses *preventing execution of a payload for the invoked function*. Ex. 1002, ¶234.

Second, 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. Ex. 1001, 6:31-35. The stack as evaluated in Fratric can be considered

hardware or software that stores data related to a memory region for a monitored process. Ex. 1002, ¶¶235-236.

Checks disclosed in 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 evaluation of the address of the invoked function address is found in data related to a memory region for a monitored process. Ex. 1007, pp. 4-5; Ex. 1002, ¶¶54-65, 236-238.

To the extent Fratric does not explicitly disclose *when the address of the invoked function address is found in a cache*, which Petitioner does not concede, Sallam renders this limitation obvious. Ex. 1002, ¶239.

First, 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 1004 may include an identification of a particular resource of the processor that is to be **trapped**” (*preventing execution of a payload*). Ex. 1008, ¶¶0109, 0211. Ex. 1002, ¶240.

Second, Sallam 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” (e.g., *invoked function address*). Ex. 1008, ¶0211; Ex. 1002, ¶241-243.

It would have been obvious to a POSITA 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. Fratric already discloses identifying an address associated with the invoked function and preventing execution of a payload for the invoked function. Ex. 1002, ¶244; *See also* [1.4.b] and [1.5].

Fratric also already considers cache concepts like storing in cache for purposes of memory space evaluation. See Ex. 1007, pp. 4-5. (“ROPGuard saves the state of all general-purpose registers at the point of entry of each critical function”). It would have been obvious to a POSITA to retrieve information from cache stored by other operations (like address information) as a suspicion evaluation for cache. Ex. 1002, ¶245-246.

A POSITA would have had a reasonable expectation of success in producing the method recited in claim 2 through the combination of Fratric with Sallam. In particular, the combination is nothing more than applying Fratric’s known techniques of utilizing an address associated with the invoked function for memory

space evaluation 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, and thus would have been obvious to a POSITA. Ex. 1002, ¶247-249.

10. Claim 5

[5.0] The method according to claim 1,

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,

Fratric alone or combined with Sallam renders this limitation obvious. Ex. 1002, ¶251.

Fratric discloses that “the attacker will need to **call at least one critical function**” (*function call of one of the plurality of functions in the monitored memory space*). Ex. 1007, p. 4. “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” (*upon detecting*). *Id.*; Ex. 1002, ¶252-254.

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

Fratric alone or combined with Sallam renders this limitation obvious. Ex. 1002, ¶255.

To the extent Fratric does not explicitly disclose *a hooking technique*, Sallam renders this limitation obvious. Ex. 1002, ¶¶66-68, 256-257.

Sallam discusses this well-known technique, “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 and monitor for **file writes** to a particular file.” Ex. 1008, ¶0049.

[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.

Fratric alone or combined with Sallam renders this limitation obvious. Ex. 1002, ¶258. Fratric discloses a *reverse stack walk from the lowest level user mode function of the invoked 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

It would have been obvious to a POSITA that the stack walk is done in reverse 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. Ex. 1002, ¶¶66-68, 259-264.

It would have been obvious to a POSITA 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. Ex. 1002, ¶265.

Fratric already discloses monitoring a memory space for a function call and conducting a reverse stack walk based on the call. See [1.3]; Ex. 1002, ¶¶266. It would have been obvious to a POSITA to hook the function as described by Sallam as part of another function call evaluation. Ex. 1002, ¶¶66-68, 267-268.

A POSITA would have had a reasonable expectation of success in producing the method recited in claim 5 through the combination of Fratric with Sallam. In particular, the combination 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 a POSITA. Ex. 1002, ¶269.

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. Sallam's

specifically uses this concept as part of its function hooking techniques. Ex. 1002, ¶270.

11. Claim 7

[7.0] The method according to claim 5,

See [5.0]-[5.3].

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

Fratric alone or combined with Sallam renders this limitation obvious. Ex. 1002, ¶273.

To the extent Fratric does not explicitly disclose *evaluating state information*, which Petitioner does not concede, Sallam discloses state information such as “**permissions of a code section**” (*state information*) for a **memory** page of virtual memory 1204 or the address of physical memory 1203” and “**determine whether the requestor** of the attempt [to write a new value] **is allowed to make such changes**” (*evaluating*). Ex. 1008, ¶0262; Ex. 1002, ¶274-277.

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

Fratric alone or combined with Sallam renders this limitation obvious. Ex. 1002, ¶278.

To the extent Fratric does not explicitly disclose evaluating state information *associated with a cache entry for the hooked function*, which Petitioner does not concede, Sallam discloses “[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**” (*hooked function*). Ex. 1008, ¶0262; Ex. 1002, ¶280. “[U]pon a trapped attempt to write a new value to the permissions **in such a location** (*cache*), below-O/S security agent 1220 may determine whether the requestor of the attempt is allowed to make such changes.” Ex. 1008, ¶0262; Ex. 1002, ¶281.

It would have been obvious to a POSITA to combine Fratric and Sallam to evaluate state information associated with a cache entry for a hooked function. Ex. 1002, ¶282.

Fratric already discloses monitoring a memory space of a monitored function. See [1.1.a]; Ex. 1002, ¶283. It would have been obvious to a POSITA to retrieve

information from cache stored by other operations (like a hooked function) as a suspicion evaluation for cache. Ex. 1002, ¶283.

A POSITA would have had a reasonable expectation of success in producing the method recited in claim 7 through the combination of Fratric with Sallam. In particular, the combination 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 a POSITA. Ex. 1002, ¶284-286.

12. Claim 8

[8.0] The method according to claim 7,

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.

See [2.2]. Ex. 1002, ¶288.

13. Claim 10

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

See [9.0]-[9.7].

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

See [2.1]. Ex. 1002, ¶290.

[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.

See [2.2]. Ex. 1002, ¶291.

14. Claims 13, 15, 16

Claim 13 depends from claim 9 and is substantially similar to claim 5. See analysis in [5.0]-[5.3].

Claim 15 depends from claim 13 and is substantially similar to claim 7. See analysis in [7.0]-[7.2].

Claim 16 depends from claim 15 and is substantially similar to claim 8. See analysis in [8.0]-[8.1].

XII. DISCRETIONARY DENIAL WOULD BE INAPPROPRIATE

A. THE BOARD SHOULD NOT DENY INSTITUTION UNDER SECTION 314(a)

The six factors considered for § 314 denial strongly favor institution. *See Apple Inc. v. Fintiv, Inc.*, IPR2020-00019, Paper 11 (Mar. 20, 2020) (precedential).

1. Factor 1: Whether the district court granted a stay or a stay may be granted if a proceeding is instituted

No motions to stay have been entered or are pending in the co-pending parallel litigation at this time. However, at least two of the three other defendants in the co-pending parallel litigation have filed IPRs and at least one of those defendants has indicated it intends to file a motion to stay should its IPR be instituted. Therefore, at minimum this factor is neutral because the Board should not presume how the district court would rule should a motion to stay be filed. *Sand Revolution II, LLC v. Continental Intermodal Group-Trucking LLC*, IPR2019-01393, Paper 24, 7 (PTAB Jun. 16, 2020); *see also Dish Network L.L.C. v. Broadband iTV, Inc.*, IPR2020-01359, Paper 15 at 11 (Feb. 12, 2021).

**2. Factor 2: Proximity of the court’s trial date to the Board’s
projected statutory deadline for a final written decision**

Trial in the co-pending parallel litigation is currently set for August 19, 2024, approximately 16 months from the filing date of this petition. The co-pending parallel litigation includes more than ten patents asserted by the Patent Owner, numerous counterclaim patents asserted by the Defendants, four additional defendants, and three scheduled *Markman* hearings, with the first Markman hearing having recently been completed. Therefore, the co-pending parallel litigation is at an early stage. The five lawsuits are consolidated only for pre-trial purposes and each defendant will receive its own trial. Thus, the current trial date likely applies only to the first defendant to go to trial. Of the five defendants, Petitioner was served fourth. Accordingly, if the cases are tried in the order in which the lawsuits were filed, Petitioner’s trial date will likely be weeks or months after the August 2024 date. Even if the trial date remains as currently scheduled, there is likely sufficient time to complete the IPR proceeding prior to the trial date. Accordingly, this factor weighs in favor of institution. See *Dish Network L.L.C. v. Broadband iTV, Inc.*, IPR2020-01359, Paper 15 at 16 (Feb. 12, 2021) (acknowledging that for a trial date scheduled nine months in the future much can change during that time). But even if the Board believes this factor weighs in favor of denial, the Board should balance all six factors

when determining whether to apply discretionary denial. *Dish Network L.L.C. v. Broadband iTV, Inc.*, IPR2020-01359, Paper 15 at 14-15 (Feb. 12, 2021).

3. Factor 3: Investment in the parallel proceeding by the court and the parties

As indicated above, the parallel proceeding is in its early stages. The Markman hearing for claim construction of the ‘591 patent was held March 14, 2023. However, fact discovery has just opened on April 18, 2023, and is not scheduled to close until January 18, 2024. The deadline to add parties is not until October 3, 2023. Opening expert reports are not due until January 30, 2024, and expert discovery does not close until March 12, 2024. Therefore, the parties have invested minimal time in the parallel proceeding. The lack of investment weighs in favor of institution or is at least neutral. *See Sand Revolution II, LLC v. Continental Intermodal Group-Trucking LLC*, IPR2019-01393, Paper 24, 10-11 (PTAB Jun. 16, 2020) (contrasting the *Fintiv* decision where the Markman hearing was completed and contention discovery had been completed where the Board found this factor was slightly in favor of denial from the facts in *Sand Revolution II* where the Markman Order had been entered).

4. Factor 4: Overlap between issues raised in the petition and in the parallel proceeding

Petitioner has raised Fratric and Sallam in preliminary invalidity contentions filed in the parallel litigation. However, final infringement and invalidity contentions are not due until July 30, 2023. Therefore, the extent of overlap is unknown at this time. Further, the Petitioner reserves the right to stipulate to elimination of the overlap between this proceeding and the parallel litigation should the IPR be instituted. This would mitigate the concerns over duplication. *Sotera Wireless, Inc. v. Masimo Corp.*, IPR2020-01019, Paper 12, at 18-19 (PTAB Dec. 1, 2020); *SandRevolution II, LLC v. Cont'l Intermodal Group-Trucking LLC*, IPR2019-01393, Paper 24, at 11-12, (PTAB June 16, 2020). Further, the request for institution of the IPR involves additional claims 5, 7, 8, 13, 15, and 16 that are not asserted in the parallel proceeding. Therefore, this factor weighs in favor of institution.

5. Factor 5: Whether the petitioner and the defendant in the parallel proceeding are the same party

Petitioner and Webroot are parties in the parallel litigation. The Patent Owner is the petitioner in the parallel proceeding and the Petitioner is one of four defendants in the consolidated co-pending litigation. This factor is therefore neutral regarding

institution. *See HP Inc. v. Slingshot Printing LLC*, IPR2020-01084, Paper 13 at 9 (Jan. 14, 2021); *see also Cisco Sys., Inc. v. Ramot at Tel Aviv Univ. Ltd.*, IPR2020-00122, Paper 15, at 11 (J. Crumbley dissenting) (PTAB May 15, 2020) (noting that disfavoring a defendant in parallel litigation is “contrary to the goal of providing district court litigants an alternative venue to resolve questions of validity.”).

6. Factor 6: Other circumstances that impact the Board’s exercise of discretion, including the merits

This factor includes consideration of the merits of the proceeding, which weigh in favor of institution. The main reference, Fratric in this matter was not considered in prosecution and the secondary reference Sallam was not considered in combination with Fratric during prosecution. Further the Patent Owner did not dispute or argue against the use of Sallam during prosecution for any of the claims in this petition for which Sallam is being asserted. Since the arguments being asserted in this petition are materially different from those presented in prosecution, the considerations as to art previously before the USPTO, are favorable to institution. *Hewlett Packard Enterprise Co., et al. v. Billjco LLC*, IPR2022-00420, Paper 17, 6 (PTAB July 12, 2022).

Further, the Patent Owner has asserted the '591 patent against multiple defendants. Therefore, there is strong public interest in the Board instituting the IPR and adjudicating the validity of the patent. As such, this factor weighs in favor of institution.

**B. THE BOARD SHOULD NOT DENY INSTITUTION UNDER
*GENERAL PLASTIC***

The '591 patent has been challenged in the CrowdStrike IPR and the Trend Micro IPR. Discretionary denial is not appropriate because *General Plastic* factors favor institution. See *Gen. Plastic Indus. Co. v. Canon Kabushiki Kaisha*, IPR2016-01357, Paper 19, 16 (Sept. 6, 2017) (Section II.B.4.i. precedential). The Board considers the *General Plastic* factors when multiple petitions are filed against the same patent.

**1. Factor 1: Whether the Same Petitioner Previously Filed a
Petition**

This factor weighs against denial because Petitioner Kaspersky is a *different* petitioner than the petitioner in the CrowdStrike IPR and Trend Micro IPR. Moreover, there is not a significant relationship between Petitioner and either

CrowdStrike or Trend Micro. Although the Board held in *Valve Corp. v. Elec. Scripting Prods., Inc.*, IPR2019-00062, Paper 11 (Apr. 2, 2019) (precedential) that the application of the first *Generic Plastic* factor is not limited to instances where multiple petitions are filed by the same petitioner, the facts here are distinguishable from *Valve*. First, in *Valve*, both the petitioner (Valve) and HTC (who filed the earlier IPR) were co-defendants in the same district court case and were accused of infringing the patent-at-issue based on the same product. *See Valve*, Paper 11, at 9. However, in the present case, Petitioner was sued separately, accused of infringement based on different products, and is entitled to its own trial relating to its own products.

The Board routinely finds that “*General Plastic* and *Valve* do not apply” to the circumstances here, where the petitioners were sued independently, were sued on different products, and have no significant relationship. *See, e.g., NetNut Ltd. V. Bright Data Ltd.*, IPR2021-00465, Paper 11, at 8-11 (Aug. 12, 2021) (“*NetNut*”) (“Under these circumstances, we decline to extend *General Plastic* and *Valve* to the situation here: where a follow-on petition is filed by a petitioner who is not the same as previous petitioners and does not have any relationship, much less a significant relationship, with previous petitioners.”); *Sony Mobile Commc’ns AB. V. Ancora Techs., Inc.*, IPR2021-00663, Paper 17, 7-11 (Jun. 10, 2021) (“*Sony Mobile*”);

Shenzhen AOTO Elecs. Co., Ltd. V. Ultravision Techs., LLC, IPR2021- 00190, Paper 7, 14-16 (Jun. 9, 2021).

Further, the attorneys preparing this IPR have not coordinated with CrowdStrike or Trend Micro in doing so. *See Twitter, Inc., v. Palo Alto Research Center Inc.* IPR2021-01458, Paper 11 at 33 (April 6, 2022) (declining to discretionarily deny the petition under *General Plastic* and finding that there was no evidence that Petitioner had coordinated with previous filers).

Accordingly, with no significant relationship between Petitioner and either CrowdStrike or Trend Micro, this factor heavily weighs against discretionary denial.

2. Factor 2: Whether at the Time of Filing of the First Petition the Petitioner Knew of the Prior Art Asserted in the Second Petition or Should Have Known of It

Because Petitioner is not a petitioner in the CrowdStrike IPR or the Trend Micro IPR, factor 2 “is of little probative value.” *NetNut*, 9; *Western Digital Corp. v. Spex Technologies, Inc.*, IPR2018-00084, Paper 14 at 17 (April 25, 2018) (“Because Petitioner has not previously filed a petition against the same patent, factors 2–5 bear little relevance in this case.”).

Regardless, Petitioner became aware of its references during the course of its own prior art search, not from CrowdStrike or Trend Micro. The prior art and combinations asserted in the instant Petition are different than the combinations asserted in the CrowdStrike IPR and the Trend Micro IPR. More particularly, there are no overlapping references in the CrowdStrike IPR or the Trend Micro IPR and the instant Petition. This factor weighs against discretionary denial.

**3. Factor 3: Whether at the Time of Filing of the Second
Petition the Petitioner Already Received the Patent Owner's
Preliminary Response to the First Petition**

Patent Owner has not filed a Patent Owner's Preliminary Response (POPR) in the Trend Micro IPR. Patent Owner has filed a POPR in the CrowdStrike IPR. However, Petitioner has not gained any unfair tactical advantage because the POPR addresses totally different prior art and prior art combinations than the instant Petition. This factor weighs against discretionary denial.

4. Factors 4 & 5: The Length of Time that Elapsed between the Time the Petitioner Learned of the Prior Art Asserted in the Second Petition and the Filing of the Second Petition, and Whether the Petitioner Provides Adequate Explanation

Because this is Petitioner's *first* petition challenging the '591 patent, the fourth and fifth factors are inapplicable. *See, e.g., Sony Mobile*, Paper 17, at 13-15. Accordingly, these factors weigh "strongly" against discretionary denial. *Id.* At 14, 15.

5. Factors 6 & 7: The Finite Resources of the Board, and Requirement to Issue a Final Determination within One Year from Institution

There is no relation between the one-year deadlines for final written decisions on the disparate grounds advanced in Petitioner's or CrowdStrike's or Trend Micro's respective petitions. Petitioner is not a party to any other petition challenging the '591 patent. There is no indication that the Board would be unduly burdened or unable to render a final written decision within the statutory deadline. These factors weigh against discretionary denial.

**C. THE BOARD SHOULD NOT DENY INSTITUTION UNDER 35
U.S.C. § 325(d)**

The Board should institute review and should **not** exercise its discretion to deny institution under § 325(d). *Advanced Bionics, LLC v. Med-El Elektromedizinische Geräte GMBH*, IPR2019-01469, Paper 6, at 6-11 (PTAB Feb. 13, 2020) (precedential).

**1. Advanced Bionics Step 1 (Becton Factors (a), (b), (d)) favors
institution.**

Becton, Dickinson factors (a), (b), and (d) relate to part one of the *Advanced Bionics* framework—whether the art or arguments presented in the Petition are the same or substantially the same as those previously presented to the Office. *Advanced Bionics*, Paper 6 at 10.

Fratric (Grounds 1A/1B) was not cited during prosecution. Petitioner acknowledges that Sallam was considered and used by the Examiner during prosecution. However, the combination of Fratric and Sallam (Ground 2) was not considered.

The references in the Petition are not cumulative of those presented during prosecution, nor do the arguments made during prosecution overlap with those in

the petition. During prosecution, Sallam was used first as a §102 reference and subsequently as a primary reference in a §103 rejection. Ex. 1006. Fratric is not substantially the same as Sallam. Further, Sallam is only cited in this Petition as a secondary reference to address dependent claims 2, 5, 7, 8, 10, 13, 15, and 16. Petitioner does not rely on Sallam to teach the limitations in independent claims 1, 9, and 17, (or dependent claims 3, 4, 6, 11, 12, and 14, for that matter). Applicant never disputed Sallam's teachings with respect to dependent claims 2, 5, 7, 8, 10, 13, 15, or 16 because the Applicant gained allowance solely by amending the independent claims, thereby never developing that portion of the record.

Factors (a), (b), and (d) favor institution.

2. *Advanced Bionics* Step 2 (*Becton* Factors (c), (e), (f)) favors institution.

Becton, Dickinson factors (c), (e), and (f) relate to part two of the *Advanced Bionics* framework—whether the Office committed a material error in its prior consideration of the art or arguments. *Advanced Bionics*, Paper 6 at 10.

Although Sallam was cited as part of a rejection, factor (c) favors institution because the Applicant never disputed its teachings with respect to dependent claims 2, 5, 7, 8, 10, 13, 15, or 16. Specifically, the Examiner rejected all

pending claims over Sallam and subsequently, Sallam and other various references. See Ex. 1006. For dependent claims 2, 5, 7, 8, 10, 13, 15, and 16, the Examiner pointed to specific portions of Sallam. Ex. 1006. With respect to dependent claims 2, 5, 7, 8, 10, 13, 15, and 16, Applicant simply stated that the “independent claims... were allowable over the cited Sallam. All other claims... depend from one of the allowable claims... and are thus also allowable.” Ex. 1006. Applicant did not address the portions of Sallam pointed to by the Examiner as disclosing claims 2, 5, 7, 8, 10, 13, 15, or 16, nor did Applicant specifically dispute that such portions failed to disclose the limitations recited in claims 2, 5, 7, 8, 10, 13, 15, or 16. Accordingly, Petitioner is not asking the Office to reconsider any arguments with respect to claims 2, 5, 7, 8, 10, 13, 15, and 16, but merely to make determinations consistent with the Examiner’s previous and uncontested finding that Sallam discloses the features of claims 2, 5, 7, 8, 10, 13, 15, and 16.

Advanced Bionics explains that “if the record of the Office’s previous consideration of the art is not well developed or silent, then a petitioner may show the Office erred by overlooking something persuasive under factors (e) and (f).” *Advanced Bionics* at 10. Here, the prosecution record is not well developed as to whether Sallam teaches dependent claims 2, 5, 7, 8, 10, 13, 15, and 16. Applicant failed to develop that portion of the record when it chose to advance prosecution

exclusively on amendments to the independent claims. Ex. 1006, pp. 121-131, 89-101, 39-51. Nothing in the file history suggests that there was a meaningful dispute as to the relevance of Sallam disclosure with respect to claims 2, 5, 7, 8, 10, 13, 15, and 16.

With respect to factor (f), the Examiner never considered Sallam in combination with the new prior art reference cited in this petition. *See Teradata Corp. v. SAP SE*, IPR2020-00943, Paper 12 at 20-24 (Nov. 25, 2020) (granting institution when a secondary reference was used as a primary reference during prosecution, but the examiner never considered the combination of references in the petition). Further, the Petition is supported by a new Ex. 1002 declaration explaining how a POSITA would have understood Sallam, including that it discloses the limitations of claim 2, 5, 7, 8, 10, 13, 15, and 16. This new evidence also weighs against discretionary denial. *See, e.g., Puma North Am., Inc. v. Nike, Inc.*, IPR2019-01058, Paper 10, 19 (PTAB Oct. 31, 2019). Factors (c), (e), and (f) favor institution.

XIII. CONCLUSION

Petitioner asks that the Board institute review and cancel each challenged claim.

Dated: April 28, 2023

Respectfully submitted,

By: /Joseph Miotke/

Joseph Miotke, Reg. No. 47,798
Dewitt LLP
901 Marquette Avenue
2100 AT&T Tower
Minneapolis, MN 55402
Tel: (612) 305-1400

CERTIFICATE OF COMPLIANCE UNDER 37 C.F.R. § 42.24(d)

Pursuant to 37 C.F.R. § 42.24(d), Petitioner hereby certifies, in accordance with and reliance on the word count provided by the word-processing system used to prepare this Petition, that the number of words in this paper is 13,648. Pursuant to 37 C.F.R. § 42.24(d), this word count excludes the table of contents, mandatory notices under § 42.8, certificate of service, certificate of word count, appendix of exhibits, and any claim listing.

Dated: April 28, 2023

Respectfully submitted,

By: /Joseph Miotke/

Joseph Miotke, Reg. No. 47,798
Dewitt LLP
901 Marquette Avenue
2100 AT&T Tower
Minneapolis, MN 55402
Tel: (612) 305-1400

CERTIFICATE OF SERVICE

The undersigned certifies that, in accordance with 37 C.F.R. § 42.6(e) and 37 C.F.R. § 42.105, service was made on Patent Owner as detailed below.

Date of service April 28, 2023

Manner of service FEDERAL EXPRESS

Documents served Petition for Inter Partes Review Under 35 U.S.C. § 312

and 37 C.F.R. § 42.104 of U.S. 10,284,591; Petitioner's
Exhibit List; Exhibits 1001-1031; and Petitioner's
power of attorney.

Persons served Sprinkle IP Law Group

1301 W. 25th Street, Suite 400

Austin, TX 78705

Dated: April 28, 2023

Respectfully submitted,

By: Joseph Miotke

Joseph Miotke, Reg. No. 47,798
Dewitt LLP
901 Marquette Avenue
2100 AT&T Tower
Minneapolis, MN 55402
Tel: (612) 305-1400