Kennesaw State University
DigitalCommons@Kennesaw State University

KSU Proceedings on Cybersecurity Education, Research and Practice

2016 KSU Conference on Cybersecurity Education, Research and Practice

Teaching Static Call Analysis to Detect Anomalous Software Behavior

Jordan Shropshire
University of South Alabama, jshropshire@southalabama.edu

Philip Menard
University of South Alabama, pmenard@southalabama.edu

Follow this and additional works at: https://digitalcommons.kennesaw.edu/ccerp

Part of the Curriculum and Instruction Commons, Information Security Commons, Management Information Systems Commons, and the Technology and Innovation Commons

Shropshire, Jordan and Menard, Philip, "Teaching Static Call Analysis to Detect Anomalous Software Behavior" (2016). *KSU Proceedings on Cybersecurity Education, Research and Practice*. 1.
https://digitalcommons.kennesaw.edu/ccerp/2016/Academic/1

This Event is brought to you for free and open access by the Conferences, Workshops, and Lectures at DigitalCommons@Kennesaw State University. It has been accepted for inclusion in KSU Proceedings on Cybersecurity Education, Research and Practice by an authorized administrator of DigitalCommons@Kennesaw State University. For more information, please contact digitalcommons@kennesaw.edu.

Abstract
Malicious code detection is a critical part of any cyber security operation. Typically, the behavior of normal applications is modeled so that deviations from normal behavior can be identified. There are multiple approach to modeling good behavior but the most common approach is to observe applications' system call activity. System calls are messages passed between user space applications and their underlying operating systems. The detection of irregular system call activity signals the presence of malicious software behavior. This method of malware-detection has been used successfully for almost two decades. Unfortunately, it can be difficult to cover this concept at the right level of detail for undergraduate information systems students. Some instructors provide only superfluous descriptions of malware, others delve into in-depth reviews of application code. This paper advocates an approach which teaches the fundamentals of code analysis to non-programmers. The approaches integrates visualization tools such as flame graphs to help students interpret software behavior. It has been found to be especially valuable for upper division information systems courses on cyber security.

Disciplines
Curriculum and Instruction | Information Security | Management Information Systems | Technology and Innovation

This event is available at DigitalCommons@Kennesaw State University: https://digitalcommons.kennesaw.edu/ccerp/2016/Academic/1

INTRODUCTION

A core function of cyber security is detecting malicious activity against host computer systems. Anomaly-based intrusion detection systems model the expected behavior of user-space applications (Hofmeyr, Forrest, & Somayaji, 1998). They observe events generated by the applications and note their impact on the operating system. Once a model is established, it is used to analyze subsequent activity and identify behavior which does not follow expected patterns (Bau & Mitchell, 2011). The assumption is that anomalous behavior represents a threat to host security. The main advantage of this approach is the ability to detect previously-unknown attacks.

Models of system behavior typically focus on system call activity (Warrender, Forrest, & Pearlmutter, 1999). A system call is request sent from a user-space application to an operating system (Figure 1). Applications are not allowed to perform restricted tasks or interact with hardware directly. Instead, they must ask the operating system to perform these tasks on their behalf. These communication sequences provide a ready channel for observing software behavior (Kuhn, Wallace, & Gallo, 2004). Applications issue the same sequences of system calls in order to execute the same basic processes. For instance, the same set of functions must be used in order to write to network. This allows for prediction of future behavior. System call analysis is a popular method for behavioral modeling because it is relatively low-cost in terms of performance, relatively accurate, and unobtrusive (Bowring, Rehg, & Harrold, 2004). Much of the information which is required can be gathered using built-in stack trace utilities.

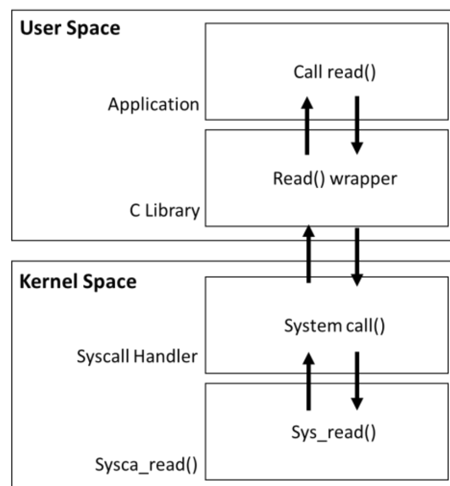

Figure 1: System Calls Within the Modern Operating System

Although the software for detecting host intrusion is highly reliable, it remains important for information systems professionals to understand the detection process (Mahoney & Chan, 2003). These individuals are expected to ensure system security and create comprehensive security strategies. They are also expected to make informed purchase and implementation decisions(Allen, Christie, Fithen, McHugh, & Pickel, 2000). Furthermore, intrusion detection systems occasionally misclassify application behavior (Engler, Chelf, Chou, & Hallem, 2000). It is occasionally necessary for IS professionals to overrule classification decisions so that valid software is allowed to remain while malicious code is removed(Kang, Fuller, & Honavar, 2005). Therefore, it is not only helpful, but in some cases essential that the information systems professional have a working knowledge of the system call process and the operation of host intrusion detection systems (von Solms & Niekerk, 2013).

Despite the importance of this skillset, relative few IS professionals are given adequate training. Intrusion detection has become highly automated and the levels of false positives has dropped significantly over the past two decades (Allen et al., 2000). The ability to statistically detect suspicious software has been taken for granted. Undergraduate information systems student tend to either receive cursory training or are required to take a full-length course which requires advanced programming skills (Bishop, 2003; von Solms & Niekerk, 2013). However, neither of these options are ideal. The former provides too little substance while the latter requires skills which IS majors may not possess (Cone, Irvine, Thompson, & Nguyen, 2007). Therefore, this manuscript proposes training which is appropriate for IS majors wishing to focus on cyber security. The training is called static system call analysis. It integrates a visualization tool called a flame graph (Gregg, 2014). Flame graphs show the most used code-paths through series of system calls. The hallmarks of modified software - changes in system call frequencies and changes in code paths are easily discerned through a comparison of computer-generated graphs (Mancuso, 2012). The graphs can be printed and included in lecture slides. The purpose is to provide a basis for learning how intrusion detection systems work. From this starting point, students will be able to branch out and learn how other types of intrusion detection systems operate.

The proposed method for static analysis of software behavior is useful for understanding the principles of intrusion detection. It assumes an understanding of system and software operations but does not require extended code walkthroughs. The remainder of this manuscript is organized as follows: the following section provides the background. It reviews related research and methods for teach malware detection concepts in undergraduate courses. After contextual information is provided, the proposed static analytical method is introduced. This section describes flame graphs and the proposed method of static analysis. Finally, the paper is summarized and implications are suggested.

BACKGROUND

Certain subject areas, especially in computing-based fields, lend themselves to visual representations (Cimons, 2012). Diagrams, graphs, and figures have traditionally been used in topics from computer science and information systems curricula (object-oriented programming, entity-relationship diagrams, Unified Modeling Language, etc.). Attempting to teach even basic programming or architecture concepts without the use of visual aids would be unnecessarily difficult for both the instructor and the students. Because of their prior exposure and reliance on visual tools for learning, students in computing-based fields are well-suited to understanding concepts through the use of visual materials (Eppler, 2006). Visualization has a rich history in computer science and other related fields (Baecker, 1998), but has been curiously lacking in computing classes based on cybersecurity principles.

Designing lessons with visual materials is especially relevant for computing concepts that are unfamiliar, abstract, or inherently difficult to understand. Students who major in a computing-based field can often choose a program focus from a variety of concentration areas; therefore, not all computing majors are focused primarily on programming or software engineering. For students outside of these focus areas, learning about basic information security topics, such as malware or intrusion detection, can be difficult when text-based examples are the primary teaching tools. Intrusion detection in particular, which is commonly demonstrated through the analysis of a Linux stack trace, is a topic students struggle to understand conceptually using traditional teaching methods.

PROPOSED METHOD

Overview

In response to a need to teach the essentials of malware detection to non-programmers, this section describes method of static analysis of function calls. The purpose is to identity normal software activity and then detect anomalous behavior. The proposed method centers on the use of visual aides to interpret low-level system activity and identify anomalies. Prior to introducing the method in class, the instructor should first conduct a primer on modern operating systems and malware. Next, the instructor should outline the various types of intrusion detection systems. Once this context is provided, the method should be introduced.

To begin, a stack trace of a user-layer application should be collected and displayed. The concept of system calls should be introduced if it have not already been described. The stack trace should then be piped through an algorithm which produces flame graphs. The instructor should describe how to interpret flame graphs and note any emerging patterns. Next, the stack trace from a corrupted or compromised version of the same software should be procured. After the corresponding flame graph is drafted, the two visualizations should be compared. A tool called flamegraphdiff may be used to assist in the comparison (Bezemer, 2014). This software provides a graphic depiction of the differences between any two graphs and gives precise differences in frequency statistics.

After presenting the two graphs, the instructor should draw implications from key differences in frequency and code path. The session should end with a discussion of the risks of false positives and negatives. For homework, students, should be given several flame graphs of the same software. One graph should be marked trusted with the others are unmarked. The students should be expected to analyze the outputs and identify suspicious behavior. Alternatively, students could be asked to create two flame graphs on their own – one for a trustworthy version of any application and another for a compromised or altered version of the same application.

Flame Graphs

The concept of the flame graph was born out of a need to detect and analyze CPU performance problems (Gregg, 2016). A common technique for analyzing system performance is sampling stack traces. Tools such as Linux perf_events and DTrace may be used to collect the stack trace data (McDougall, Mauro, & Gregg, 2006). The traces typically depict each function as a separate line and in reverse-calling order (e.g., child-to-parent). This provides a very comprehensive view of system operations. However, it can also be overwhelming (see Figure 2) (Fattori, Paleari, Martignori, & Monga, 2010). A few seconds of collection will result in tens of thousands of stacks and hundreds of thousands of lines of output. The data must be condensed in order to be of utility to analysts. There are various methods of achieving this. For instance, data can be condensed into a tree structure with counts or percentages for each code-path branch. Alternatively, the trace could be sampled at finite intervals. However, even the condensed output may be difficult to interpret. The flame graph was developed to overcome this type of problem.

```
dnx  3407/3515  [000] 3005552.151144:     1001001 cpu-clock:
                      8c665 memset (/lib/x86_64-linux-gnu/libc-2.19.so)
                     2a97fb _ZN3WKS7gc_heap19a_fit_segment_end_pEiPNS_12heap_segmentE
                     2aa060 _ZN3WKS7gc_heap14allocate_smallEimP13alloc_contexti (/hom
                     2ab886 _ZN3WKS7gc_heap23try_allocate_more_spaceEP13alloc_context
                     2cbc9d _ZN3WKS6GCHeap5AllocEP13alloc_contextmj (/home/ubuntu-vm/
                     166b83 _Z26FastAllocatePrimitiveArrayP11MethodTableji (/home/ubu
                     17bb1a _Z11JIT_NewArr1P21CORINFO_CLASS_STRUCT_l (/home/ubuntu-vm
               7ff7913483c0 instance void [mscorlib] System.Collections.Generic.Dicti
                       ...
                     10e73b _ZL31ManagedThreadBase_DispatchOuterP22ManagedThreadCallS
                     10ee79 _ZN17ManagedThreadBase10ThreadPoolE4ADIDPFvPvES1_ (/home/
                     2dd148 _ZN26ManagedPerAppDomainTPCount16DispatchWorkItemEPbS0_ (
                     12b41c _ZN13ThreadpoolMgr17WorkerThreadStartEPv (/home/ubuntu-vm
                     56f2da _ZN7CorUnix10CPalThread11ThreadEntryEPv (/home/ubuntu-vm/

dnx  3407/3520  [001] 3005552.151205:     1001001 cpu-clock:
                     50a0c2 _ZN9SigParser14SkipExactlyOneEv (/home/ubuntu-vm/.dnx/run
                     509d0b _ZN9SigParser14SkipExactlyOneEv (/home/ubuntu-vm/.dnx/run
                      f2e7a _ZN7MetaSig7NextArgEv (/home/ubuntu-vm/.dnx/runtimes/dnx-
                     232ef9 _ZN11ClassLoader26CanAccessSigForExtraChecksEP18AccessChe
                     232dc6 _ZN11ClassLoader29CanAccessMemberForExtraChecksEP18Access
                     2324f2 _ZN11ClassLoader9CanAccessEP18AccessCheckContextP11Method
                     175d4d _ZN10InvokeUtil11CheckAccessEP13RefSecContextP11MethodTab
                     175a63 _ZN10InvokeUtil15CanAccessMethodEP10MethodDescP11MethodTa
                     19eaed _Z26PerformSecurityCheckHelperP6ObjectP10MethodDescP11Met
                     19ed68 _ZN20ReflectionInvocation20PerformSecurityCheckEP6ObjectP
               7ff79029abdc void [mscorlib] System.RuntimeMethodHandle::PerformSecuri
                       ...
                     2f757d _Z32QueueUserWorkItemManagedCallbackPv (/home/ubuntu-vm/.
                     10e73b _ZL31ManagedThreadBase_DispatchOuterP22ManagedThreadCallS
                     10ee79 _ZN17ManagedThreadBase10ThreadPoolE4ADIDPFvPvES1_ (/home/
                     2dd148 _ZN26ManagedPerAppDomainTPCount16DispatchWorkItemEPbS0_ (
                     12b41c _ZN13ThreadpoolMgr17WorkerThreadStartEPv (/home/ubuntu-vm
                     56f2da _ZN7CorUnix10CPalThread11ThreadEntryEPv (/home/ubuntu-vm/

dnx  3407/3518  [002] 3005552.151250:     1001001 cpu-clock:
                     3c10b2 _ZN11ArrayNative9ArrayCopyEP9ArrayBaseiS1_iib (/home/ubun
               7ff79029cb70 instance void [mscorlib] System.Collections.Generic.List`
```

Figure 2: Raw Output From Linux Stack Trace

Flame graphs uses stack traces as inputs and create a visualization of common stack trace paths and their frequencies. Each stack trace is represented as a column of boxes, where each box represents a function. The stack depth is shown on the Y-axis while the X-axis spans the stack trace collection. The boxes on the top of the graph are the child functions while the boxes below are parent functions. Thus, a top-to-bottom review shows the ancestry for each stack. The width of each box is an indication of the frequency of the associated function. Larger boxes represent more common called functions within the stack trace. Although flame graphs depict boxes as different colors, the actual color selection is not significant. Finally, each box is labeled with the name (or abbreviation of the associated function). A sample is depicted below (Figure 3).

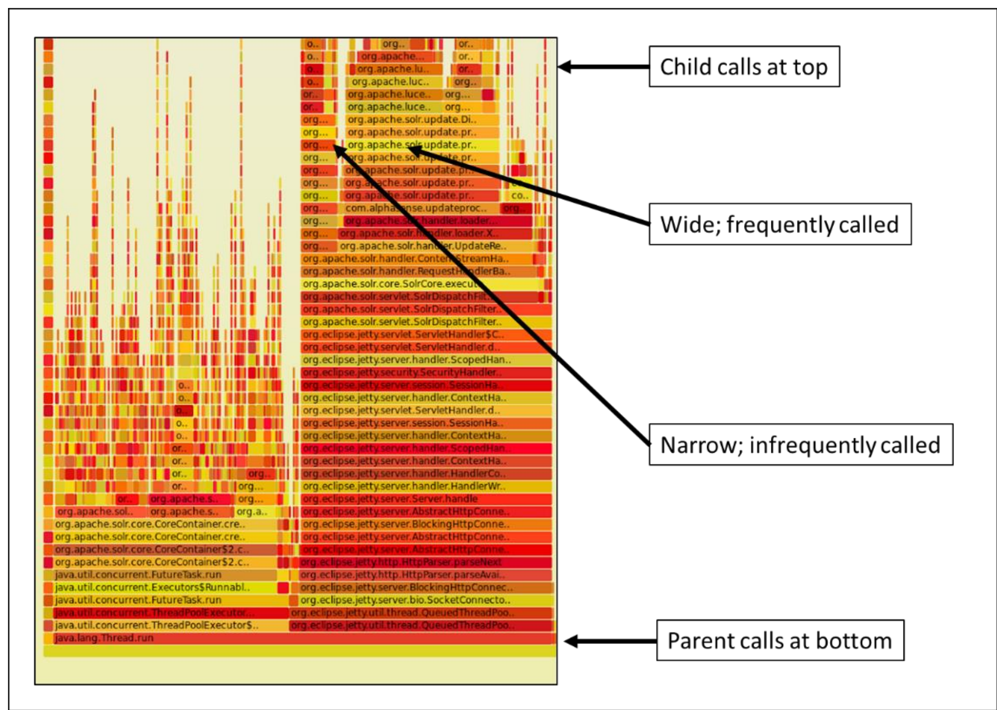

Figure 3: Example Flame Graph

Static Stack Analysis

The procedure for comparing flame graphs and detecting anomalies relies on controlled data collection. Less noise in the stack trace allows for more accurate analysis. To alleviate confusion, sampling should be limited to only those functions associated with one application. This can easily be done with most performance monitoring utilities. Further, the length of sampling time should also be controlled. This ensures that only the behaviors of interest are observed. It should be expected that changes, patches, or software updates will change the behavior of any given application. Each time a program is updated recalibration is necessary.

It is important to distinguish between stochastic and deterministic periods of software activity. Stochastic behavior varies according to time. It is not predictable. For instance, user interaction with software is a stochastic process because the user's input changes software behavior. Two periods of interaction with the same program could lead to vastly different flamegraphs. Therefore, it is important to sample the stack trace during periods of deterministic or predictable activity. For instance, application boot up is typically deterministic. The same procedure for initialization is followed on every load request. This predictability allows for behavioral patterning. As long as an application is not modified the stack traces of multiple load-ups should be similar.

Assuming that two or more controlled collections of stack data, it is possible to detect anomalous application behavior. At least one of the flame graphs should represent activity which was observed at a trust point in the application's lifecycle. This visualization represents a baseline of normal activity. Ideally, multiple collections should be taken in order to ensure that the behavior under observation repeats under the same conditions. It should be noted that due to noise and other uncontrollable variables, flame graphs may vary slightly even for static processes. For instance, Figure 4 (below) shows a comparison of flame graphs for an immutable application. Even though the software has not been modified or in any way corrupted, the resulting graphs differ slightly. A closer inspection reveals that no existing stacks were removed and no new call stacks were introduced in the application on restart. Thus, the real baseline is thought to approximate an average of the visualizations.

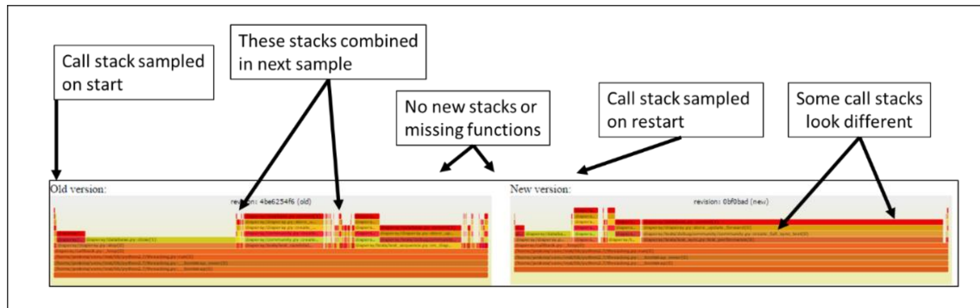

Figure 4: Analysis of Normal Behavior

Once a baseline is established, follow-up application behavior can be statically analyzed. The charts can be compared manually or using a compare_diff function to isolate differences. The main indications of abnormal activity include changes in call stack composition (such as the introduction of new functions within a stack), introduction of new stacks, removal of existing stacks, or changes in stack frequency. These changes represents a deviation in software behavior. Most have benign causes (e.g. system library is updated). However, some are indicative of deviant behavior. For instance, Figure 5 (below) depicts several modifications to the boot process of a common database management system. Several new stacks have been injected into the boot process. These stacks initiate a database query and then write several packets worth of content to the network. This is an example of subtle data exfiltration: each time the database is loaded, admin login credentials are sent to a fixed IP address. The key indications that this is suspicious behavior include the query of the mysql_users table and the addition of network write functions within a process in which none are normally made.

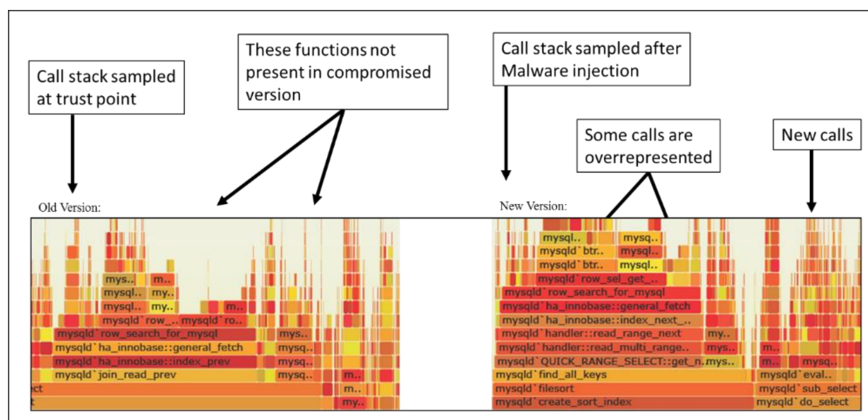

Figure 5: Detection of Anomalous Behavior

CONCLUSION

It should be assumed that IS professionals, especially those employed in a cyber security capacity, will need to understand the interworking of anti-malware programs. At some point in their careers, they will likely be expected to make anti-malware acquisition and implementation decisions, manually detect suspicious activity, and occasionally overrule automated detection systems. Thus, this manuscript proposes a method for teaching static system call analysis to detect malicious behavior. The proposed method of static call stack analysis has been used for two semesters to teach the concept of host based intrusion detection to undergraduate information systems majors. Over this period of time, a pool of flame graph sets has been developed. Each set focuses on a specific application and includes at least one baseline image. The pool is made available online to the students enrolled in the course. Unfortunately, there is not yet enough data to support statistical analysis from past semesters. However, future will empirically explore the efficacy of the proposed approach against existing methods. For those wishing to explore this method, some best practices herein conveyed:

- Don't assume that learners are familiar with system calls; provide a succinct primer
- Students should be coached into focusing on the degree of difference between graphs, not binary classification of behavior (e.g. benign or malicious)
- Encourage interpretation of call stacks as behaviors rather than a series of isolated functions
- Focus on qualitative differences rather than quantitative differences between graphs

The static method of system call analysis provides information systems undergraduates with an opportunity to identify anomalous activity. Beyond this outcome, it also facilitates a deeper understanding of the host-based intrusion detection decision making process. Even though IDS differ according to their decision making criteria, they tend to focus on the same basic behavioral traits. This lab is useful because it cuts through several layers of technical details and allows non-programmers to understand application behavior. Future research will compare learner performance using traditional teaching approaches versus the proposed methods in order to empirically assess the artifact. It is expected that the results will confirm the efficacy of the static analytical method for detecting malware.

REFERENCES

Allen, J., Christie, A., Fithen, W., McHugh, J., & Pickel, J. (2000). *State of the Practice of Intrusion Detection Technologies*. Retrieved from Pittsburg, PA:

Baecker, R. (1998). Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science. *Software Visualization: Programming as a Multimedia Experience*, 369-381, MIT Press.
Bau, J., & Mitchell, J. (2011). Security modeling and analysis. *IEEE Security & Privacy, 9*(3), 18 - 25.
Bezemer, C. (2014). flamegraphdiff. https://github.com/corpaul/flamegraphdiff: GitHub.
Bishop, M. (2003). What is computer security? *IEEE Security & Privacy, 1*(1), 67-69.
Bowring, J., Rehg, J., & Harrold, M. (2004). *Active learning for automatic classification of software behavior*. Paper presented at the ISSTA '04 Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, Boston, MA.
Cimons, M. (2012). Science of Spatial Learning. *U.S. News & World Report*
Cone, B., Irvine, C., Thompson, M., & Nguyen, T. (2007). A video game for cyber security training and awareness. *Computers & Security, 26*(1), 63-72.
Engler, D., Chelf, B., Chou, A., & Hallem, S. (2000). *Checking system rules using system-specific, programmer-written compiler extensions*. Paper presented at the Proceedings of the 4th conference on Symposium on Operating System Design & Implementation, Berkeley, CA.
Eppler, M. (2006). A comparison between concept maps, mind maps, conceptual diagrams, and visual metaphors as complementary tools for knowledge construction and sharing. *Information Visualization, 5*(2), 202-210.
Fattori, A., Paleari, R., Martignori, L., & Monga, M. (2010). *Dynamic and transparent analysis of commodity production systems*. Paper presented at the IEEE/ACM international conference on Automated software engineering, Singapore.
Gregg, B. (2014). *Flame Graphs on FreeBSD*. Paper presented at the FreeBSD Developer and Vendor Summit, Ottawa, Canada.
Gregg, B. (2016). The flame graph. *Communications of the ACM, 59*(6), 48-57.
Hofmeyr, S., Forrest, S., & Somayaji, A. (1998). Intrusion detection using sequences of system calls. *Journal of Computer Security, 6*(3), 151-180.
Kang, D., Fuller, D., & Honavar, V. (2005). *Learning classifiers for misuse and anomaly detection using a bag of system calls representation*. Paper presented at the Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop, West Point, NY.
Kuhn, D., Wallace, D., & Gallo, A. (2004). Browse Journals & Magazines > IEEE Transactions on Software *IEEE Transactions on Software Engineering, 30*(6), 418 - 421.
Mahoney, M., & Chan, P. (2003). *Learning rules for anomaly detection of hostile network traffic*. Retrieved from Melbourne, FL:
Mancuso, V. (2012). *idsNETS: An experimental platform to study situation awareness for intrusion detection analysts*. Paper presented at the IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support, New Orleans, LA.
McDougall, R., Mauro, J., & Gregg, B. (2006). *Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Upper Saddle River, NJ: Prentice Hall PTR.

von Solms, R., & Niekerk, J. (2013). From information security to cyber security. *Computers & Security, 38*(1), 97-102.

Warrender, C., Forrest, S., & Pearlmutter, B. (1999). *Detecting intrusions using system calls: alternative data models.* Paper presented at the Proceedings of the 1999 IEEE Symposium on Security and Privacy, Oakland, CA.