Cyberdiversity: Measures and Initial Results

Konstantinos Kravvaritis
Department of Informatics
Athens University of Economics and Business
Athens, Greece
kravaritisk@aueb.gr

Dimitris Mitropoulos
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
dimitro@aueb.gr

Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
dds@aueb.gr

Abstract— **Cyberdiversity is a concept borrowed from biology and refers to the introduction of diversity into the different levels of a computer. This kind of diversity is used to avert attacks that can threat a large number of systems that share common characteristics and as a result common vulnerabilities. Currently, there are many methods that introduce cyberdiversity into systems but there is no attempt to measure the existing cyberdiversity. In this paper we introduce a novel approach that measures the existing diversity in software. To accomplish that, we specify three different metrics. The concept of our approach is to collect specific information and then process it in order to find distinct similarities or differences within software. To test our approach, we implemented a system, based on the client-server architecture.[1]**

Keywords: Diversity, monoculture,metrics,computer security

I. INTRODUCTION

Cyberdiversity has its roots in biology and specifically in biodiversity. Nature has given humanity many lessons and some of them can be applied in computer security. Nature teaches us that the richest and most robust ecosystems are those that are the most diverse, i.e. those that consist of a large number of different species. When a disease infects a biological system, its genetic diversity would have as an effect the survival of a significant part of the infected population. Respectively, cyberdiverse computer systems could prove to be more resistant to potential attacks than systems that tend to monoculture, which is the exact opposite of diversity [1].

Monocultures can be seen as a population that consists of identical members that belong to the same organism. Even if monocultures are very rare in nature, in cyberspace they seem to flourish. A collection of identical computer platforms is easier, therefore cheaper to manage, because, for example, they will share the same configuration while maintaining minimum user training costs. In addition, interoperability and standardization is easier to be achieved and maintained in a monoculture [2]. However, these advantages can become at the same time disadvantages. In a monoculture, when a piece of malware manage to intrude in one member of the monoculture, in a similar way it can affect the rest of them because all share the same vulnerabilities [1, 3].

Currently, there is a great controversy whether the benefits of cyberdiversity could be overshadowed by its side-effects [1 - 8].

A. Problem Statement

Until now there are many ways proposed to introduce cyberdiversity into computer systems but there is no attempt to measure whether cyberdiversity exists in software or not. This paper aims to measure the diversity that exists in the software that is used today. Specifically we are planning to measure software diversity at the realization level (i.e. at the level of binary files). To accomplish that, we need to collect and process a variety of computer data.

In more detail, our contribution includes:

- Deciding which data to collect.
- An implementation that collects the data.
- Specify and apply novel metrics to extract results.

B. Organization

A brief survey of the major schemes for the introduction of cyberdiversity in computer systems is presented in Section 2. The system that collects the data is presented in Section 3. In Section 4, the metrics and the results of the measurements are presented. Finally we state our conclusions in Section 5.

II. RELATED WORK

Several position papers that assess the value of cyberdiversity have been lately published [1 - 8]. Also, there are numerous papers that present various methods for the

[1] In Costas Vassilakis and Nikolaos Tselikas, editors, *PCI 2010: Proceedings of 14th Panhelenic Conference on Informatics*, pages 135–140, Los Alamitos, CA, USA, September 2010. IEEE Computer Society. (doi:10.1109/PCI.2010.43).
Copyright © 2012 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

automated introduction of cyberdiversity into source or binary code. Such methods are known as synthetic diversity techniques. In this paper, these methods are categorized in those that modify the structure of a system and in those that modify the behavior of a system, mostly through the modification of the execution environment. Keromytis and Prevelakis have also adopted a similar categorization [9]. In addition, we present three software development architectures that produce cyberdiverse software.

A. *Modifying the structure*

The concept of biologically inspired diversity was reffered for first time in a paper by Forrest et al. [10]. This paper proposed several possible approaches to implement cyberdiversity through randomized transformations. To validate their approach, the authors modified the gcc compiler and focused on the elimination of buffer overflow vulnerabilities. Similar techniques include StackGuard [11], MemGuard [11] and PointGuard [12].

Collberg, Thomborson and Low were the first that focused on Java code obfuscation [13]. Wang et al. described transformation techniques of binary code by transforming the control and the data flow of a program [14]. Wroblewski generalized the approaches of Collberg and Wang, by presenting methods of controllable obfuscation [15]. Linn and Debray approached obfuscation by focusing on the initial disassembly phase and disrupting the static disassembly process [16].

Bhatkar et al. proposed address obfuscation [17]. Address obfuscation randomizes the absolute locations of all code and data, as well as the distances between different data items. The authors of the PaX project modified the Linux kernel to randomize the base address of each program region [18]. However, a study of Shacham et al. shows that the insertion of randomization at 32-bit address space is not effective [19]. Bhatkar et al. developed a new technique to avoid the drawbacks of the above techniques [20]. In this paper, the authors developed a new approach that supports randomization, where the absolute locations of all objects, as well as their relative distances are randomized. Bhatkar also proposed along with Sekar the concept of data space randomization, where randomization is inserted into the representation of data stored in program memory [21].

B. *Modifying the environment*

Techniques that randomly modify the environment of a system can hinder an attacker by increasing the complexity of the attack. Among the components of the environment that can be modified are the instruction set of a computer architecture and the various parts of the operating system or the network topology of a system.

Kc et al. proposed an instruction-set randomization (ISR) technique for countering code injection attacks [22]. Instruction-set randomization creates an execution environment that is unique to the running process, so that the attacker cannot "communicate" with the computer. This technique can be also applied to other contexts, such as SQL injection attacks [23]. Another similar technique is developed by Barrantes et al. [24]. Keromytis describes the limitations of the instruction-set randomization approach and proposes future directions and improvements [25]. However, [26] and [27] analyzed some weaknesses that are faced by the above approaches. Chew and Song have applied randomization techniques on system-call mappings, global library entry points, and stack-frames [28].

O'Donnell and Sethu study algorithms for the assignment of distinct software packages to individual systems in a network, in order to increase the available diversity of the system [29, 30]. In a similar study, Yang et al. consider diversity towards increasing security of sensor networks against worm attacks [31].

C. *Cyberdiverse software development architectures*

Apart from the single cyberdiversity introduction methods, there are architectures that combine some of these methods to produce complete frameworks that could be used for the production of software.

Cox et al. proposed their framework, based on N-variant systems [32]. Williams et al. developed a software toolchain for the application dynamic cyberdiversity techniques. This toolchain uses a virtual machine to apply diversity transformations to binary files [33]. Just and Cornwell developed and proposed another framework, described in [34].

III. DESIGN OF DATA COLLECTION SYSTEM

For the collection of the needed data, we developed a system based on the client-server model. A client–side application will run at users' computers and send specific data back to our server. This data is stored in a database that is used during the measurements.

A. *Type of the needed data*

Our primary goal is to measure the diversity of software. So we should find a way to compare different instances of the same files. By different instances of the same files, we mean different instances of a particular file (e.g. "foo.exe") that reside in different computers. For example, the file "foo.exe" that resides in computer A and the file "foo.exe" that resides in computer B are two different instances of the same file.

Cyberdiversity introduction techniques apply transformations to software that lead to the production of files, which have differences between them but share the same functionality. Thus, the files that should be collected are the binary files. Furthermore we should also collect the files that have external dependencies with the binaries. These files are the static and dynamic libraries. Specifically, for Windows our client-side application collects executable and library files (.exe and .dll), for Unix-like systems apart from the executable files, it collects the static libraries (.a files) and the dynamic libraries (.so files). Additionally, for the Mac OS X operating system we also collect the mac-oriented library files (.dylib libraries).

Here, a specific problem arises. It is impossible to collect and send back these files "as is". In this case we would need enormous storage space to store the files and excessive bandwidth to send them back to the server. The solution is to send back the MD5 hash of these files instead. Hence, no

useful information is lost because we want to examine if files are exactly the same and not the degree of their similarity. Given that the hash produced from the MD5 algorithm is unique for each input with extremely high possibility, we can make the desirable comparison.

B. Description of the system

Our client-side application runs only with the users permission and it has three main tasks to complete. The first is to collect information about the operating system of each computer that it is running at. This information refers to the version of the operating system. Secondly, it has to scan each computer in order to collect the data, which will be used for the extraction of our results. Its last task is to assure that they will run only one time at every computer. This is necessary because it ensures that the results will not be falsified. To accomplish that, our client-side application retrieves a signature from each computer. This signature is the machine SID for Windows and the MAC address for Unix-like systems.

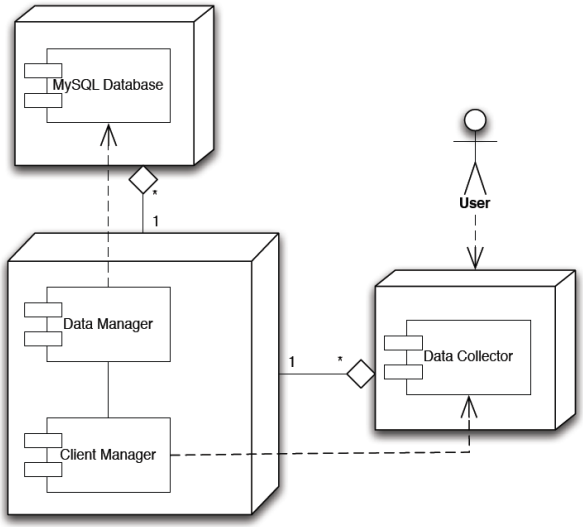

Figure 1. The architecture of the system

Our server-side scheme has also three tasks to accomplish. Its first task is to notify the client-side applications if they have already run at a computer in order not to run again. This is achieved by comparing the signatures that are saved in server's database with the corresponding signature of each computer the client-side applications are running at. Its second task is to store the data that is collected from the client-side applications to the server-side database. The final task of the scheme is to calculate the similarity percentage of every computer with all the computers that have already been visited by a client application.

C. Facebook application

To collect as many data as possible, a Facebook application was developed. This application consists of two parts: A web application running through the Facebook website and modified client-side applications that can connect with Facebook servers and consequently with the web application. Users can run the Facebook client-side application at their computers and then they can visit the web application in order to compare their results with their Facebook friends that have also run the client at their computers.

IV. METRICS AND MEASUREMENTS

In this section, we present the metrics that we used to measure our data and the results that were extracted from this process.

A. Cyberdiversity metrics

Before the metric definitions, we will define what we mean with the term "variant of a file". Cyberdiversity introduction techniques produce different variations of a given file but these variations share exactly the same functionality. For example, let's assume that the correspondent executable file for a program "foo" is "foo.exe". If "foo.exe" is produced with the use of cyberdiversity introduction techniques, different variations of this file will be produced. These variations will have internal differences between them but all of them will share the same functionality under the same name ("foo.exe"). We call these variations: "variants of a file". We also defined previously the term "instance of a file".

The first metric calculates the probability of a successful targeted attack, if the attack targets the most frequent variant of a file. Hence we calculate the percentage of the computers that are affected in the worst case, where the attack affects the most frequent variant of a file. The smaller the probability, the more cyberdiverse the file is and thus, the rate of the propagation of the attack will be slower. This probability is calculated as follows:

$$\rho = \iota / \tau. \quad (1)$$

Where ρ is the probability, ι stands for the number of instances of the most frequent variant of a given file and τ represents the total number of instances of that file.

The second metric is the ratio of the number of variants to the total number of instances of all the variants of a file. This ratio shows if a file has enough variants. This in turn, indicates that this file is diverse. The bigger the ratio is for a file, the more variants this file has and as a result more attacks are needed to compromise all the instances of this file. The smaller the ratio is for a file, the more instances are accumulated in every variant of a file, thus the bigger will be the number of instances that could be compromised by a single attack. This ratio is calculated as follows:

$$\text{ratio} = v / \tau. \quad (2)$$

Where v represents the number of the variants a given file has and τ represents the total number of instances of that file.

The third metric is the coefficient of variation (CV) of the variants of each file. CV is the ratio of the standard deviation to the mean. In our case, CV shows how the instances of a file are distributed amongst the variants of a file. If the CV is small enough, this means that the instances are distributed uniformly. In this case, the probability of a single attack that compromises a large number of instances of a file is significant.

B. Measurements

The sample collected for our study consists of data retrieved from 214 computers. 176 of these computers run Windows as their operating system, 27 run Linux, 10 run Mac OS X and one runs FreeBSD. From these computers we collected 205,221 files, which altogether have 1,309,834 instances. 111636 of these files have only one instance, hence they can't take part into the process of the results. For the extraction of the results, we take into consideration only the files that have 10 instances or more, in order to have more accurate results. We present our results in Fig 2, 3 and 4. The column of every diagram depicts the number of the processed files while every row depends on the corresponding metric.

The results based on (1) are shown in Fig. 2. The diagrams show that the probability for the biggest part (58% for Windows and 55.7% for Linux) of the files is over 50% and for

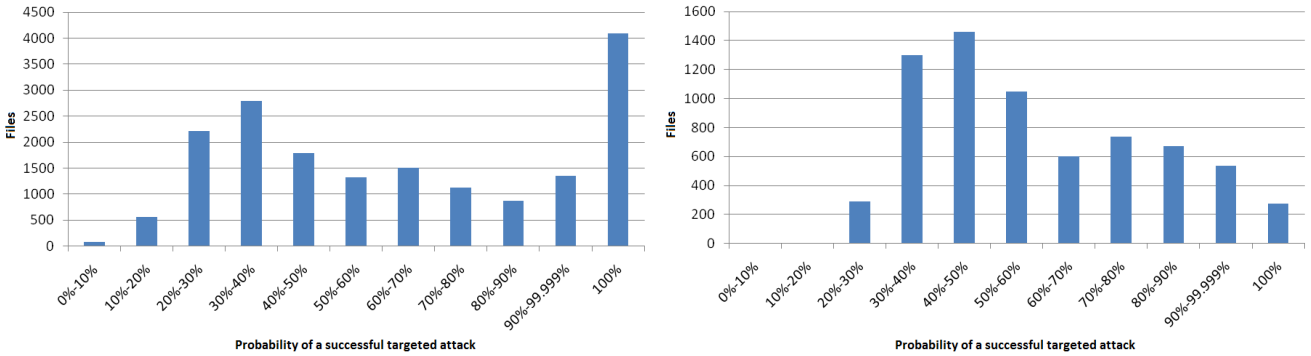

Figure 2. Results based on the probability of a successful targeted attack for Windows (left) and Linux (right)

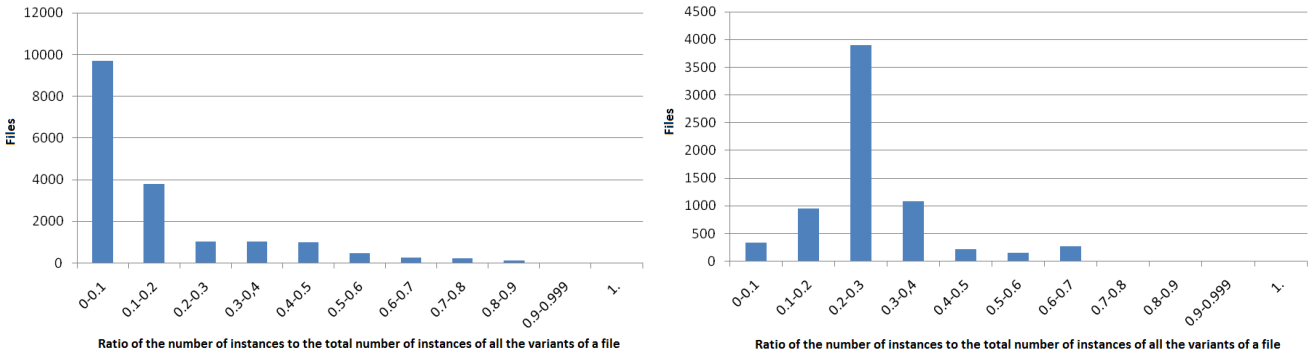

Figure 3. Results based on ratio of the number of instances to the total number of instances of all the variants of a file for Windows (left) and Linux (right)

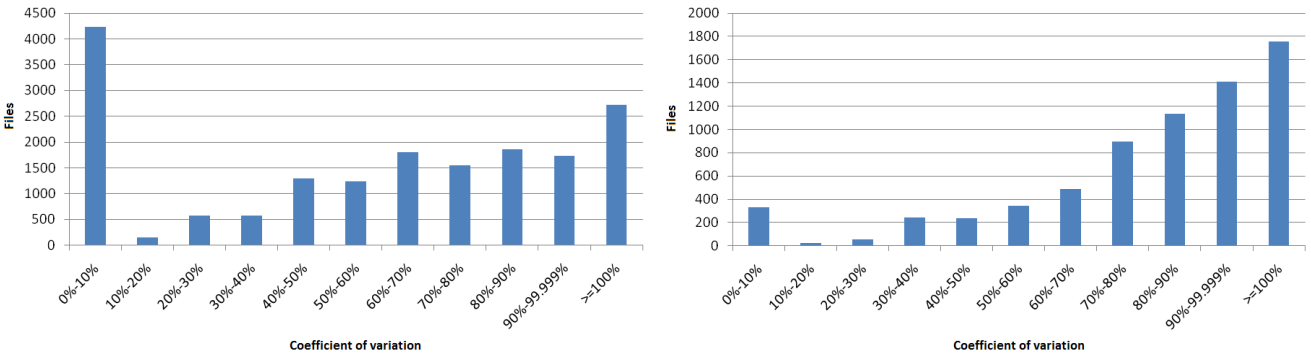

Figure 4. Results based on coefficient of variation for Windows (left) and Linux (right)

a minimal part (3.7% for Windows and 0% for Linux) of the files the possibility is under 20%. This shows that there is not enough cyberdiversity. If cyberdiversity was sufficient, the percentage of files with under 20% probability should be bigger and there should be a little amount of files with over 50% probability. Even if there are differences between the Windows-diagrams and the Linux-diagrams, the main conclusion for both is that the diversity is not sufficient. The most important difference is that there are fewer files with exactly 100% probability in Linux. These files are not cyberdiverse and thus there are more Windows files that have no diversity.

In Fig. 3 the results based on (2) are presented. The diagrams show that for the biggest part of the files the ratio is small (under 0.3). This indicates that with a single attack a malicious user could compromise a great number of instances of a file. Here there is a distinct difference between Windows and Linux. For Linux there is congestion in the range of 0.2 to 0.3 while for Windows there is congestion in the range of 0 to 0.1. This can be explained by the fact that there are fewer files that have no diversity. However, this difference is not so important and this does not change the general conclusion that there cyberdiversity is not sufficient.

The results based on CV are shown in Fig. 4. Given that the smaller the CV is, the more uniformly the instances are distributed. Here we observe that most of the files aren't distributed uniformly and most of them are concentrated in the biggest percentages. The concentration noticed in the first column is almost exclusively the result of the fact that the files that correspond in this column have no diversity and this column should not be considered in the extraction of the results because here we evaluate how uniform the diversity is and the files that correspond to the first column have no diversity. Finally, to emphasize the lack of diversion and its extend, we have also included the files that have no diversity at all.

V. Conclusion and Future Work

Cyberdiversity is and will continue to be an object of interest of the security community. In this paper we do not aim to answer whether cyberdiversity is useful or not. Our main research contribution is to find if cyberdiversity exists and to what extent.

The main result of this paper is that cyberdiversity exists but exists in minimal extent. Also, its qualitative characteristics are bad. We can also assume that this cyberdiversity does not stem from cyberdiversity introduction techniques but its extent may indicate that this is the result of other factors. For example it can be the result of variations in the version of applications that has as effect the existence of different files for the same applications. Our measurements showed also that the lack of cyberdiversity does not depend on the various operating systems.

Future work on this study includes the collection of a larger sample in order to have more reliable results. Furthermore, it is important the sample to have a greater dispersion between the operating systems. Additional, it will be very useful to define more metrics that would put cyberdiversity on the map.

References

[1] D. Geer. Monopoly considered harmful. IEEE Security & Privacy Magazine, 1(6):14–16, December 2003.

[2] Birman, K. P. and Schneider, F. B. 2009. The Monoculture Risk Put into Context. *IEEE Security and Privacy* 7, 1 (Jan. 2009), 14–17.

[3] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S.Quarterman, and B. Schneier. Cyberinsecurity: The cost of monopoly. Tech report, CCIA, 2003. http://www.ccianet.org/papers/cyberinsecurity.pdf.

[4] D. Aucsmith. Monocultures are hard to find in practice. IEEE Security & Privacy Magazine, 1(6):15–16, December 2003.

[5] J. A. Whittaker. Monocultures are hard to find in practice. IEEE Security & Privacy Magazine, 1(6):18–19, December 2003.

[6] M. Stamp. Risks of monoculture. *Commun. ACM*, 47(3):120, 2004.

[7] Parnas, D. L. 2007. Which is riskier: OS diversity or OS monopoly?. *Commun. ACM* 50, 8 (Aug. 2007), 112.

[8] G. Goth. Addressing the monoculture. IEEE Security & Privacy Magazine, 1(6):8–10, December 2003.

[9] A. D. Keromytis and V. Prevelakis. Dealing with system monocultures. In *NATO Information Systems Technology (IST) Panel Symposium on Adaptive Defense in Unclassified Networks*, Toulouse, France, April 2004.

[10] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 67–72. IEEE Computer Society, 1997.

[11] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, January 1998.

[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuardTM: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, Aug 2003.

[13] C. Collberg, C. Thomborson, and D. Low. "A Taxonomy of Obfuscating Transformations". Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[14] Chenxi Wang, "A Security Architecture for Survivability Mechanisms." PhD thesis, University of Virginia, October 2000.

[15] Gregory Wroblewski; "General Method of Program Code Obfuscation," 2002 International Conference on Software Engineering Research and Practice (SERP'02), June 24 - 27, 2002, Monte Carlo Resort, Las Vegas, Nevada, USA.

[16] Cullen Linn, Saumya Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," ACM Conference on Computer and Communications Security, Washington DC, October 27 - 31, 2003.

[17] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," 12th USENIX Security Symposium, August 2003.

[18] PaX Project. Address space layout randomization, Mar 2003. Accessed on August 7th, 2005: http://pax.grsecurity.net/docs/aslr.txt.

[19] H. Shacham et al., "On the Effectiveness of Address-Space Randomization," *Proc. 11th ACM Int'l Conf. Computer and Comm. Security*, ACM Press, 2004, pp. 298–307.

[20] Bhatkar, S., Sekar, R., and DuVarney, D. C. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Baltimore, MD, July 31 - August 05, 2005).

[21] Bhatkar, S. and Sekar, R. 2008. Data Space Randomization. In *Proceedings of the 5th international Conference on Detection of intrusions and Malware, and Vulnerability Assessment* (Paris, France, July 10 - 11, 2008). D. Zamboni, Ed. Lecture Notes In Computer Science, vol. 5137. Springer-Verlag, Berlin, Heidelberg, 1–22.

[22] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 272–280. ACM Press, 2003.

[23] S.W. Boyd and A.D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," *Proc. 2nd Int'l Conf. Applied Cryptography and Network Security*, Springer, 2004, pp. 292–302.

[24] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 281–289. ACM Press, 2003.

[25] Keromytis, A. D. 2009. Randomized Instruction Sets and Runtime Environments Past Research and Future Directions. *IEEE Security and Privacy* 7, 1 (Jan. 2009), 18–25.

[26] A. Sovarel, D. Evans, and N. Paul, "Where's the FEEB?: The Effectiveness of Instruction Set Randomization," *Proc. Usenix Security Symp*., Usenix Assoc., 2005, pp. 145–160.

[27] Y. Weiss and E.G. Barrantes, "Known/Chosen Key Attacks against Software Instruction Set Randomization," *Proc. Annual Computer Security Applications Conf.* (ACSAC), ACSA, 2006, pp. 349–360.

[28] M. Chew, D. Song. "Mitigating Buffer Overflows by Operating System Randomization," Technical Report CMUCS- 02-197.

[29] O'Donnell, A. J. and Sethu, H. 2004. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (Washington DC, USA, October 25 - 29, 2004). CCS '04. ACM, New York, NY, 121–131

[30] O'Donnell, A. J. and Sethu, H. 2005. Software Diversity as a Defense against Viral Propagation: Models and Simulations. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation* (June 01 - 03, 2005). Workshop on Parallel and Distributed Simulation. IEEE Computer Society, Washington, DC, 247–253.

[31] Yang, Y., Zhu, S., and Cao, G. 2008. Improving sensor network immunity under worm attacks: a software diversity approach. In *Proceedings of the 9th ACM international Symposium on Mobile Ad Hoc Networking and Computing* (Hong Kong, Hong Kong, China, May 26 - 30, 2008). MobiHoc '08. ACM, New York, NY, 149-158.

[32] Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. 2006. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (Vancouver, B.C., Canada, July 31 - August 04, 2006).

[33] Williams, D., Hu, W., Davidson, J. W., Hiser, J. D., Knight, J. C., and Nguyen-Tuong, A. 2009. Security through Diversity: Leveraging Virtual Machine Technology. *IEEE Security and Privacy* 7, 1 (Jan. 2009), 26–33.

[34] Just, J. E. and Cornwell, M. 2004. Review and analysis of synthetic diversity for breaking monocultures. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode* (Washington DC, USA, October 29 - 29, 2004). WORM '04. ACM, New York, NY, 23–32.