Application stress testing Achieving cyber security by testing cyber attacks

Conference Paper · November 2012

DOI: 10.1109/THS.2012.6459909

4 authors, including:

Al Underbrink
Sentar, Inc.
5 PUBLICATIONS **26** CITATIONS
SEE PROFILE

Andrew Potter
University of North Alabama
26 PUBLICATIONS **151** CITATIONS
SEE PROFILE

Holger Jaenisch
Raytheon Company
107 PUBLICATIONS **372** CITATIONS
SEE PROFILE

Application Stress Testing

Achieving Cyber Security by Testing Cyber Attacks

Al Underbrink, Andrew Potter, PhD, and Holger Jaenisch, PhD
Sentar, Inc.
315 Wynn Drive, Suite 1
Huntsville, Alabama 35805
al.underbrink@sentar.com, andrew.potter@sentar.com,
and holger.jaenisch@sentar.com

Donald J. Reifer
Reifer Consultants, LLC
14820 North Dragon's Breath Lane
Prescott, Arizona 86305
don@reifer.com

***Abstract*—Application stress testing applies the concept of computer network penetration testing to software applications. Since software applications may be attacked – from inside or outside a protected network boundary – they are threatened by actions and conditions which cause delays, disruptions, or failures. Stress testing exposes software systems to simulated cyber attacks, revealing potential weaknesses and vulnerabilities in their implementation. By using such testing, these internal weaknesses and vulnerabilities can be discovered earlier in the software development life cycle, corrected prior to deployment, and lead to improved software quality. Application stress testing is a process and software prototype for verifying the quality of software applications under severe operating conditions. Since stress testing is rarely – if at all – performed today, the possibility of deploying critical software systems that have been stress tested provides a much stronger indication of their ability to withstand cyber attacks. Many possible attack vectors against critical software can be verified as true threats and mitigated prior to deployment. This improves software quality and serves as a tremendous risk reduction for critical software systems used in government and commercial enterprises. The software prototype models and verifies failure conditions of a system under test (SUT). The SUT is first executed in a virtual environment and its normal operational modes are observed. A normal behavior model is generated in order to predict failure conditions based on attack models and external SUT interfaces. Using off-the-shelf software tools, the predictions are verified in the virtual environment by stressing the executing SUT with attacks against the SUT. Results are presented to testers and system developers for dispensation or mitigation.**

Keywords-penetration testing; application testing; attack; software assurance; softwaer quality

I. INTRODUCTION

Software applications are typically tested to validate their required functionality and runtime performance [10]. These tests assure that the software fulfills its intended purpose and performs as expected in its intended environment. The software testing approach applies controlled inputs and measures program outputs to verify that the application behaves and performs acceptably under normal operating conditions. But what about functionality and performance under unusual or anomalous operating conditions? If a program receives corrupt, excessive, or insufficient inputs, will it function as specified?

A cyber attack can be viewed as a relevant and important type of unusual or anomalous operating condition. During a cyber attack, an adversary (terrorist, subversive, or military) attempts to disrupt or disable the normal operating conditions of a system often by gaining access to an application via one or more interfaces. If a web application uses a database, that database can be corrupted or disabled to effectively disrupt the web application via such corruption. The web application is attacked via the interface to that database often via its most vulnerable point, the SQL server. This example illustrates the complexity and interdependency of modern software systems. In the context of systems of systems (SOS), therefore, the attacker need not attack entire SOS. Instead, as in this example, the overall mission of the SOS can be compromised by defeating just one of its component software systems.

A problem facing modern software system developers is how to assure the software continues to operate even when under threat. With virtually unlimited ways to attack a software system, the question often posed to security analysts is how can a system developer know if the software system can complete its "mission" without interruption, whether the mission is electrical power generation, banking, automotive manufacturing, or missile defense?

A common approach to assuring such continued operations of large, networked software systems is penetration testing. In "pen testing", a team of human operators probes the system at its extremes in an attempt to identify weaknesses or vulnerabilities [7]. These weaknesses and vulnerabilities often appear as poorly or wrongly configured network systems, operating systems, databases, and software applications. Even when protected, modern SOS are so complex, and change so rapidly, that it is virtually impossible to verify that all of their weaknesses and vulnerabilities have been eliminated. Pen testing provides an independent assessment team one proven way to achieve this goal.

The concept of application stress testing is analogous the world of network penetration testing [1, 2]. A software application is "attacked" by providing is corrupted data, excessive data, and insufficient data via a SQL injection attack. This attack floods the application with excessive data typically through a rapid succession of service requests. Such an attack results in anomalous operating behavior as the software

application is disrupted or disabled. As the system fails, so does the entire SOS because other systems may depend on the databases maintained within it. To combat such an attack, the software application is placed under stressful operating conditions and monitored for its response. We refer to this defensive posture as application stress testing.

II. THE STESS TESTING CONCEPT

A. *What Application Stress Testing is Not*

Application stress testing appears similar to automated testing on the surface. However, there is a fundamental difference that is important. Automated tests may be derived from a finite set of requirement specifications. That is, each requirement may be tested in a very specific and sometimes different way. The functional behavior and performance characteristics, typically captured as system requirements, dictate which tests can be automated and which have to be executed otherwise. For example, exception conditions must be generated singly, while repetitive tests like those captured in an end-to-end execution sequence can be automated via test scripts.

Application stress testing has a much less limited focus. Essentially, any external input to the software application that does *not* generate the expected behavior is fair game. The possibilities are virtually endless and only increase with the size and complexity of a software application. The universe of potential anomalous inputs for application stress testing is vastly large and therefore hard to control when compared to functional and performance testing.

B. *How to Stress Test*

The Sentar approach to application stress testing attempts to include legacy software systems about which analysts have some knowledge of its internal design, functional behavior, or performance is unavailable. Without such design details, the software application would be difficult to test. We prefer not to test exhaustively, but there may be little information available about the software application, its expected behavior, and its normal operating conditions. We just know that it works (and has perhaps been working for a long time). How might we stress test such a legacy application?

The first step in application stress testing is to develop some sort of characterization of the system under test (SUT). This step classifies the SUT according to its intended purposes. For example, many applications use a database management system storing and retrieving data. It is easily inferred that such a SUT may be vulnerable to attacks against the database or corruption of data from the database. Without sufficient design details about the SUT, stress testing can obviously test every possible data input, but this is inefficient. Worse, if a SUT does *not* use a database, it would be wasteful to, for example, launch a SQL injection attack against the SUT. We need a way to filter out those attacks that will obviously not affect the operation of the SUT.

From a clean slate, the first step in stress testing is to characterize the SUT under its normal operating conditions. One way to accomplish this is to look for attack entry points

This research and development was sponsored by the Missile Defense Agency (MDA) Small Business Innovation Research (SBIR) Program.

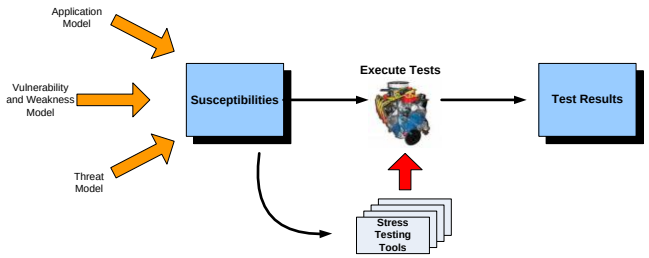

Figure 1. Application stress testing concept.

either through a dialog with the system developer or owner or a detailed examination of the design specification, if it still exists. Does it use a database? Is it a real time system (and therefore potentially subject to race conditions)? Does it communicate using TCP, UDP, or some other network protocol? What are the external applications with which it interacts? What types of data, how much data, and when are those data exchanged?

It is easy to surmise that a user dialog would be inefficient and ultimately ineffective. But, it is necessary when dealing with legacy systems when the design specifications are no longer available. Dozens or perhaps hundreds of questions need to be answered in order to fully characterize any SUT. In the spirit of automated testing, application stress testing that can automatically characterize a SUT by monitoring the SUT, in its normal operating environment, under it normal operating conditions, would enable a far more efficient process.

Once a SUT has been characterized, however, there are ways in which stress testing can become more efficient. One method under research is to use and apply common enumerations of relevant patterns. Standard classification schemes for vulnerabilities (CVE), weaknesses (CWE), attack patterns (CAPEC), technical security operations (SCAP)[1], and web applications (OWASP) [2] exist which allow manual mappings between likely weaknesses and vulnerabilities to attack patterns. These enumerations, however, are rather abstract and require specific attacks to be developed before they can become operative. Furthermore, specific attacks relate to their configuration on hardware and software platforms (CCE). However, when combined, these specifications can be used to narrow the universe of stress tests which may apply to a SUT.

A conceptual view of application stress testing, depicted in Figure 1, therefore brings together modeling details about 1) the type of application to be stressed, 2) vulnerabilities and weaknesses associated with that type of application, and 3) threats against the potential vulnerabilities and weaknesses of that type of SUT. From these frameworks, a model of applicable susceptibilities[3] can be derived in an automated way.

[1] See the National Institute for Standards and Technologies (NIST) at http://nvd.nist.gov/ for details on CVE, CWE, SCAP, CAPEC, CCE, and other useful enumerations related to this topic.

[2] See https://www.owasp.org/ for details on OWASP.

[3] We define a susceptibility as an unverified, but potential, weakness or vulnerability in a software application.

Figure 3. Operational flow of application stress testing.

A set of specific stress tests can then be selected and corresponding attack tools can be used to execute those stress tests against the SUT. Once the series of tests has been executed against the anticipated susceptibilities, results can be presented and reported for remediation and mitigation by the system developer or owner [3].

This concept of application stress testing enables a focus of stress tests against a SUT. The most likely vulnerabilities and weaknesses can be explicitly probed in a systematic and automated fashion, often by "fuzzing" inputs to the SUT over a network or other type of interface. For software system development or acquisition, application stress testing can provide tremendously useful information about the behavior of a software application under anomalous operating conditions or cyber attack. This is especially true when the attack points that impact SOS operations are sought.

III. Implementing the Concept

Part of the Sentar research and development is to create a prototype software system for application stress testing. The prototype follows a flow pattern as described above and depicted in Figure 3 to prepare for stress testing.

First, the SUT must be characterized according to its function and operational platform. The Sentar prototype draws from OWASP and CCE modeling concepts to then identify the most likely weaknesses and vulnerabilities (the susceptibilities) present in the SUT. Susceptibilities are distinguished from vulnerabilities and weaknesses based on likelihood of occurrence. Vulnerabilities and weaknesses are typically known as are attack points; susceptibilities are potential vulnerabilities and weaknesses and attack points are viewed as the means and methods used to gain access. From the identified susceptibilities, abstract attack patterns (CAPEC) can be used to map to specific attacks using specific attack tools. The combination of attacks and tools constitute the actual stress tests to launch against the SUT and are intended to expose likely vulnerabilities and weaknesses.

A. *Automated Stress Testing*

A more automated approach to characterizing the SUT is to monitor its execution under normal operating conditions. It is assumed that most SUTs are functionally tested under those normal operating conditions and nominal behaviors are validated. If this is the case, we also collect data on the SUTs interactions with external systems during functional testing along with behaviors. Data are collected for the network protocol(s), port usage, data types exchanged, frequency of data exchanges, and amount of data exchanged. In the prototype, this information is stored in a SQL database for later comparison with stressed operating conditions.

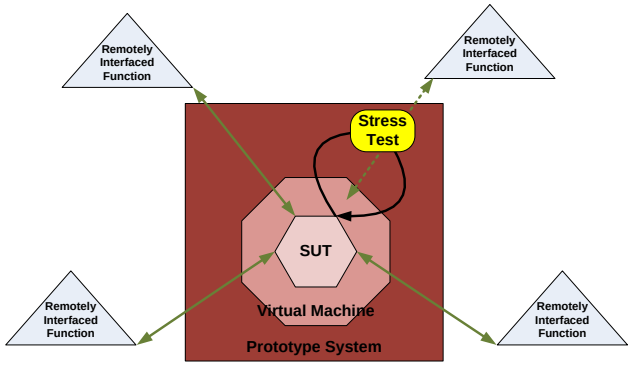

Figure 2. Application stress testing environment.

Next, mapping the susceptibilities to stress tests is performed. Currently, this is a limited mapping of the twenty-five most common weaknesses as ranked by MITRE Corporation[4]. The prototype focuses on these because the development team has manually performed the mappings and they cover the vast majority of common programming errors and design flaws (i.e., the 90% solution).

At this point, the prototype begins running a series of stress tests. To do this, the SUT operating environment must first be established. The prototype uses a VMWare® virtual machine (VM) to simulate that environment, which is the same as that used for functional testing. Besides establishing a run-time environment, the VM also enables the stress testing prototype to intercept data exchanges over each interface in isolation. This arrangement is shown in Figure 3.

The SUT, shown in the center, executes within a VM, which in turn executes on a host platform for the application stress testing prototype. The stress testing host is arranged within the other systems so that it can run and examine distributed applications which communicate across the VM boundaries. These, as mentioned previously, operate in the same manner as during functional testing. However, the VM environment enables control of the interfaces. Thus, the SUT has access to and can probe all systems that are part of the SOS.

For each external SUT interface over the network, one or more stress tests are executed. The stress testing infrastructure replaces data input to the SUT with corrupt, excessive, or insufficient data and captures the output responses from the SUT. In this way, the stress tests can be applied as a series of singular tests which can be measure, repeated, and duplicated. The tests, when run, identify abnormal conditions accomplished through abuse. Such conditions may not be easy to spot when probed using functional tests because their behaviors are not outside of the nominal performance envelopes used for validation.

Stress testing must be tightly controlled in order to scientifically measure the results. Before each test, the VM running the simulated environment, the SUT, the monitoring

[4] http://cwe.mitre.org/top25/

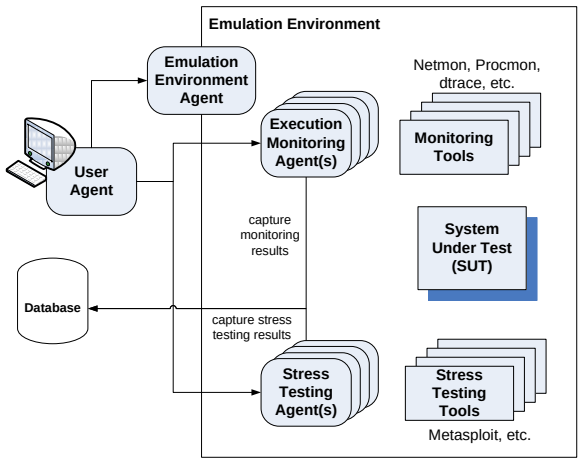

Figure 4. Prototype architecture.

tools, and the attack tools must be reset and restarted. This is to ensure that results for each test are independent of partial results leftover from prior tests. This is accomplished in our prototype using scripts written in Python. Stress testing results are once again stored in a database where the stressed SUT results can be compared with the normal functional results.

Lastly, the stress testing results for presented to end users/system developers/system owners for assessment. For system developers, the stress testing results can be used to redesign or refine the implementation of the SUT. For legacy systems, mitigation strategies can be developed to address what are now known vulnerabilities and weaknesses. Mitigation strategies include the use of security wrappers, software agents, or configuration changes that prevent the successful disruption or disablement of the SUT during cyber attack. During the acquisition or integration phases of software systems, stress testing feedback can lead to more robust and resilient SOS.

B. Prototype Architecture

The Sentar prototype architecture relies on software agents to provide all aspects of application stress testing. Software agents are effective in that a software architecture based on agents enables the rapid extension of prototype functionality. Communication between software agents is performed via message passing, further supporting extensibility of the prototype.

The architecture for the application stress testing prototype is shown in Figure 4. A User Agent manages interaction with the prototype end user by setting up stress testing, executing the stress tests, and reviewing the results. Similarly, the User Agent accesses and presents data stored in the prototype database by the other components of the architecture.

An Emulation Environment Agent controls the establishment of the simulation environment for the SUT as described previously. The simulated environment must be reset and all components restarted with each stress test. Functions include establishing the virtual environment, starting the monitoring tools and attack tools, running the SUT, and restarting the entire environment between stress tests. This component of the architecture is responsible for much of the automation benefits from application stress testing.

The Execution Monitoring Agent manages the execution of monitoring tools used in the prototype via its knowledge base. Since the specific monitoring tools used to collect data on the SUT behavior are dependent on the SUT, the agent-based prototype architecture and the Execution Monitoring Agent enable to use of the appropriate monitoring tools without large scale re-implementation of the prototype. For the current prototype, Sentar is using common tools such as the Netmon packet analyzer and the Procmon process monitor from Microsoft, plus the dtrace tool from Sun Microsystems for monitoring kernel activity. These, and other monitoring and attack tools, are under research and evaluation for use as we develop the application stress testing prototype.

Like the Execution Monitoring Agent, the specific stress tests to be launched against the SUT drive the use of specific stress testing tools. Stress Testing Agents each control the application and execution of the tools, including what parameters are used in a stress test and which data are collected during each stress test. Current stress testing tools in the prototype include the Metasploit framework for its extensive set of available attacks; a combination of Python scripts, Wapati, and SQLMap to launch SQL injection attacks; Python scripts, Wapati, and the Metasploit framework for customized remote file injection attacks; and SpikePROXY for probing an SUT, scanning directories, fuzzing parameters for input to the SUT, and XML variations to corrupt input to the SUT for web applications. Once again, the software agent architecture readily supports the use of such attack tools for specifically targeting a SUT for its identified susceptibilities.

The agent-based prototype architecture has been very effective at enabling the functionality necessary for running application stress tests. The architecture is highly extensible, allowing the addition of new monitoring and attack tools for different types of SUT and for different types of experiments.

IV. REVIEWING RESULTS

The next step in application stress testing is to present test results in a way that is useful to end users. Sentar is investigating visualization techniques that go beyond a long list of tabular results of each stress test. Part of this task is to define a set of metrics that can be visualized and used to demonstrate the value of stress testing as part of the prototype. To accomplish this, the project team has begun developing an information matrix that is uses the Goal-Question-Metric (GQM) paradigm popularized by Dr. Victor Basili of the University of Maryland to define measures of interest. The GQM paradigm follows a structured approach to identifying the most relevant measures of the stress testing results.

Initially, a set of base metrics were defined for application stress testing. These are the size of test data set, the number of overall tests run, the number of stress tests in a nominal test data set (i.e., the number of stress tests), the size of the test run, the number of tests for a test run (number of tests per test run), the number of stress tests for a test run (number of stress tests per test run), the percent of stress tests in which the SUT continued normal operations (i.e., the survivability), the

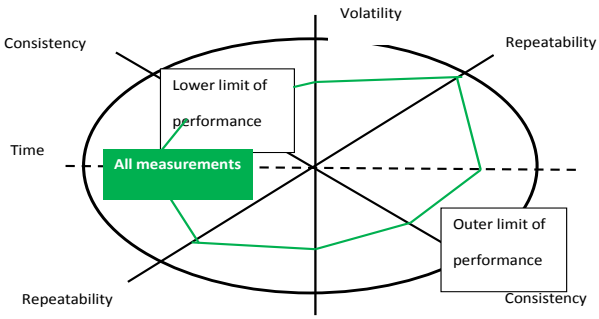

Figure 5. Polar chart representation of stress test results.

number of restarts during stress testing for a test run (restarts/number of stress tests), the amount of test time completed before a stress test failure and the total time required to run all tests, the number of stress testing failures per total number of failures, and the added value of stress testing. The added value of stress testing is an overall numeric value of the utility of performing the application stress testing. It may be defined, based on the base metrics, as the ratio of the dollar value of effort to mitigate discovered risks (weaknesses and vulnerabilities) versus the dollar value of risks left unmitigated. These values after could be computed in terms of labor hours expended/saved multiplied by the dollar value to mitigate the risks. This metric, of course, must be an accumulation for all test runs that are part of the test scenario. Other metrics will record measures of success (MOS) and measures of performance (MOP), as needed.

One potential visualization technique is the use of a polar chart comparing the measured performance of each measure with the lower and upper limits of performance. An example of a polar diagram is shown in Figure 5. In the example, the measured performance is shown as a green graph connecting measured values along each metric axis. The polar chart brings together multiple metrics into a single view that can be used to assess the overall results of application stress testing. Other visualizations are also being explored, such as fishbone diagrams and stop light charts, to augment the SUT stress testing results.

V. AUTOMATICALLY GENERATING SUSCEPTIBILITIES

As with most research and development projects, there are additional concepts and ideas that emerge. One of the most important benefits of application stress testing is the ability to identify SUT weaknesses and vulnerabilities prior to deployment. With such a large scope of possible exploits in most complex software systems, the ability to focus stress testing on those most likely to exist in a SUT is a significant advantage especially when SOS considerations must be examined. Thus, Sentar's subsequent research and development involves automatically generating the susceptibilities during the SUT characterization phase. To do this, the project team has begun investigating machine learning techniques.

Table 1. Heptor features for characterizing a SUT.

Heptor Features	
H1 (standard deviation)	H5 (M8, 8th order moment)
H2 (skewness)	H6 (DJ, entropic fractal dimension)
H3 (kurtosis)	H7 (DH, morphological fractal dimension)
H4 (M6, 6th order moment)	

Specifically, data modeling is being explored as a means to predict which susceptibilities will cause a SUT fault or failure [4, 5, 6]. Data models are an effective tool for novelty detection and ambiguity resolution between nominal and anomalous conditions even in information environments that are rich, diverse, and dynamic in information content and range. Data modeling has been used to address similarly difficult problems in text mining, fault detection, radar applications, worm detection, and other application areas. The application of Data models to text mining is of particular relevance, since the ability to perform classification of unconstrained text is a logical stepping-stone to malware classification.

Data models applicable to application stress testing are derived from training exemplars in the form of a series of controlled inputs selected as representative of what will be encountered in the runtime environment. In the case of the prototype, the data collected while monitoring the SUT running under normal operating conditions is used as inputs to the data model. The inputs are treated as raw attributes and are characterized with a set of descriptive parametric and non-parametric (fractal) features. These features are collectively called a *heptor*, as they consist of a vector of the seven features identified in Table 1. As shown, these statistical features are not uncommon, but when used for these purposes constitute very useful data models for many application domains. It is a form of to machine learning that is able to effectively work with unknown-unknowns (in this case, what unknown conditions will disrupt or disable the SUT?).

This can be achieved using ideas initiated by Shewhart [8, 9] in statistical process control chart theory. In essence, without a priori modeling and simulation of a complex assembly line, he demonstrated that the entire assembly process could be monitored using two simple control charts. One monitored the range of deviations is measurable parameters of a product produced by the assembly line and the other measured the average value of a measurement.

When result values occurred more than three standard deviations from the mean of the values, the condition was called a "tip-off" due to the fact that an assignable cause had put the assembly line outside of statistical process control. The tip-off did not indicate the source or nature of the anomaly, but only that one was present. Additional diagnostics had to be initiated to determine the exact cause.

We can derive a data model along the same lines by characterizing a set of observables (the normal operating behavior of the SUT) from the descriptive statistics captured in the heptor. By declaring a collection of them to be nominal and assigning the category value of one-half to them, simple regression modeling of function enables a projection filter to be derived. Projecting the training data (again, the normal operating behavior of the SUT) through this function sets the

upper and lower bound equal to the minimum and maximum of the projected data. New data are now characterized by the same descriptive statistics as the training data and also projected through the filter. If it comes out within the two upper and lower bands it is recognized as nominal. If it is outside the upper and lower bands, it is considered to be anomalous and generates a tip-off.

This yields a predictor for the stress testing. If a set of input values to the SUT fall within the nominal range, it indicates that no disruption or disablement will take place. If a set of input values to the SUT fall outside the minimum or maximum, it is a strong indication that a fault or failure will occur. The potential benefit of this approach is that these predictions can be directly tested, thereby reducing any dependence on exhaustively testing every possible combination of inputs to the SUT. As the application stress testing prototype development proceeds, this data modeling technique will be investigated and developed.

VI. SUMMARY AND CONCLUSIONS

Application stress testing is an approach to software assurance that is more inclusive of legacy systems. Any software application, including "black box" software, can be automatically probed and assessed for vulnerabilities and weaknesses. Once known, such vulnerabilities and weaknesses can be mitigated through further design or implementation changes or through the use of external software capabilities such as wrappers. Application stress testing is a viable and useful technology to use for improving robustness and resiliency in application software and systems of systems.

ACKNOWLEDGMENT

Sentar acknowledges our sponsor, the Missile Defense Agency Small Business Innovation Research Program. Specifically, we thank Mr. Aaron Corder, the MDA technical point of contact, for his guidance and contribution to our research. The authors thank Mr. David Giametta and Mr. Brendan Romanowski for the technical contributions to the application stress testing design and prototype.

REFERENCES

[1] Avgerinos, T., Cha, S. K., Hao, B.L.T., and Brumley, D.. AEG: Automatic Exploit Generation. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2011.

[2] Brumley, D., Poosankam, P., Song, D., and Jiang Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," *IEEE Symposium on Security and Privacy*, May 2008.

[3] Fraser, Timothy, Badger, Lee, and Feldman, Mark, "Hardening COTS Software with Generic Software Wrappers," in the *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 9-12, 1999.

[4] Jaenisch, Holger M., Handley, James W., Pooley, J. C., and Murray, S. R., "Data Modeling for Fault Detection," *Proceedings of 57th MFPT*, April 14-18, 2003. Virginia Beach, VA.

[5] Jaenisch, Holger M., Handley, James W., Faucheaux, Jeffrey P., and Harris, Brad, "Data Modeling of Network Dynamics," *Applications and Science of Neural Networks, Fuzzy Systems, and Evolutionary Computation VI*, edited by Bosacchi, Bruno; Fogel, David B.; Bezdek, James C. Proceedings of the SPIE, Volume 5200, pp. 113-124, 1999.

[6] Jaenisch, Holger M. "Spatial Voting with Data Modeling for Behavior Based Tracking and Discrimination of Human from Fauna from GMTI Radar Tracks," *Proc. of SPIE 8388, SPIE Defense and Security Symposium*, Baltimore, MD, April 2012.

[7] McClure, Stuart, *Hacking Exposed: Network Security Secrets and Solutions*, McGraw-Hill, 2009, ISBN 978-0-07-161374-3.

[8] Shewhart, W. A., *Economic Control of Quality of Manufactured Product*, 1931, ISBN 0-87389-076-0.

[9] Shewhart, W. A., *Statistical Method from the Viewpoint of Quality Control* , 1939, ISBN 0-486-65232-7.

[10] Whittaker, James, A., Arbon, Jason, and Carollo, Jeff, *How Google Tests Software*, ISBN 0321803027, 2012.