Amrita Vishwa Vidyapeetham

Amrita School of Computing, Chennai

Computer Science and Engineering-Cyber Security

20CYS302 Secure Coding

Enhancing Code Security with CodeSheriff: A Multilingual Vulnerability Scanner

Aravind.M , Kausal S D , Moulish G A

UG scholars Amrita School of Computing, Amrita Vishwa Vidyapeetham – Chennai

Abstract:

In an era of ever-evolving cybersecurity threats, safeguarding software against vulnerabilities is paramount. This research presents CodeSheriff, an innovative vulnerability scanning bot designed to bolster code security. Leveraging a fastText model trained on the Rosetta Code dataset, it accurately identifies uploaded files' programming language and tailors its analysis approach to each recognized language, ensuring comprehensive and language-specific vulnerability scans. Beyond mere detection, CodeSheriff equips developers with practical mitigation strategies. This paper introduces the CodeSheriff architecture, outlines its methodology, and presents a performance comparison with existing tools. CodeSheriff's dynamic adaptability to diverse programming languages showcases its potential to enhance code security across the development spectrum, making it a valuable addition to the realm of code security solutions.

Keywords: Vulnerability scanning, programming language identification, language-specific scanning, fastText model, Rosetta Code dataset, , software security.

1.Introduction:

In our ever-evolving digital landscape, where software permeates every aspect of contemporary life, the imperatives of software security stand as an unassailable cornerstone of the digital age. As software applications become increasingly integral to daily operations, from powering critical infrastructures to serving on personal devices, the vulnerability of software to malicious exploits remains a pressing concern. Consequently, the imperative to safeguard software projects against latent security threats lurking within their code has never been more acute.

This research endeavors to meet this pressing need with the introduction of "CodeSheriff," an innovative and adaptable Telegram bot meticulously crafted to reinforce the security posture of software projects. CodeSheriff represents a versatile and multifaceted vulnerability scanning tool engineered for accessibility and automation. It stands as a vigilant sentinel, poised to identify and address an expansive array of security vulnerabilities concealed within code snippets.

At the heart of CodeSheriff's efficacy lies its unique ability to dynamically discern the programming language embedded within uploaded code. This impressive capability is achieved through the implementation of a fastText model, diligently trained on the

extensive Rosetta Code dataset. Once the programming language is accurately identified, CodeSheriff seamlessly aligns itself with a language-specific approach, thereby optimizing the process of vulnerability

assessment for each specific language. In doing so, CodeSheriff provides developers with a tailored and precise tool for identifying and mitigating security risks within their code, ensuring a comprehensive and dynamic defense against potential threats.

2. Literature Review

This section delves into the significance of accurately identifying programming languages and provides an extensive examination of existing real-world vulnerability scanning tools along with their inherent limitations.

2.1 Significance of Identifying Programming Languages:

Accurate identification of the programming language within a codebase is akin to comprehending the language of a written document. It is foundational to grasping the code's syntax, semantics, and specific constructs. This identification process offers invaluable contextual information that enables subsequent code analysis tools and developers to apply language-specific analysis techniques effectively.

Programming languages exhibit distinct sets of vulnerabilities and coding patterns that are susceptible to exploitation by malicious entities. Language-specific vulnerabilities may encompass buffer overflows in C/C++, injection attacks in SQL and JavaScript, or type-related issues in Python. Identifying the language is the pivotal first step in targeting these vulnerabilities with precision.

Tailoring mitigation strategies to the programming language is essential for comprehensive code security. Developers rely on language-specific guidance to remediate vulnerabilities effectively. For instance, mitigating strategies in web applications may involve input validation, while lower-level languages may require memory management safeguards. Precise language identification ensures the appropriateness of mitigation measures.

2.2 Existing Vulnerability Scanning Tools and Their Limitations:

While vulnerability scanning tools have evolved significantly, it's crucial to acknowledge that they are not without constraints. Each tool possesses its unique strengths and weaknesses, which necessitate careful consideration when choosing the most suitable solution for specific scenarios. Recognizing these limitations holds paramount importance for cybersecurity professionals and organizations striving to fortify their digital security effectively. This section is dedicated to exploring the realm of established vulnerability scanning tools, providing insights into their capabilities and the challenges they encounter.

Nessus, despite its strengths, is not without limitations. Scalability is a potential concern as it may struggle to efficiently scan large-scale networks or complex infrastructures, which can limit its usability in enterprise-level environments. Additionally, Nessus heavily relies on vulnerability databases for signature updates, which can introduce delays in detecting newly emerging vulnerabilities until they are added to the database. This dependency on external data sources can impact its timeliness in providing protection against the latest threats.

OpenVAS, while known for its flexibility, has its own set of limitations. One notable limitation is the complexity of configuration and setup. Setting up OpenVAS can be challenging, especially for users with limited cybersecurity expertise. This complexity may deter smaller organizations with resource constraints from effectively utilizing the tool. Furthermore, like many vulnerability scanners, OpenVAS is susceptible to false positives, which can result in wasted time and effort investigating non-existent issues, potentially causing frustration among users.

Brakeman, as a Ruby-specific tool, exhibits certain limitations. Its primary constraint is its single-language focus, as it is tailored

exclusively for Ruby on Rails applications. Consequently, it lacks versatility for organizations with diverse technology stacks, limiting its applicability. Furthermore, Brakeman primarily focuses on security vulnerabilities and may not provide comprehensive insights into overall code quality or other non-security-related issues, which can limit its usefulness for holistic code assessments.

SpotBugs, designed for Java applications, also comes with limitations. Its most apparent limitation is its Java-specificity, which excludes other programming languages. This restricts its usability for organizations that work with multiple languages, potentially necessitating the use of additional tools for non-Java components. Additionally, SpotBugs may generate extensive reports that can overwhelm developers, making it challenging to prioritize and address the identified issues efficiently.

In addition to tool-specific limitations, vulnerability scanners, including Nessus and OpenVAS, share common constraints. False negatives, which are vulnerabilities that go undetected, remain a challenge for all vulnerability scanners. This can create a false sense of security, as not all vulnerabilities are identified. Moreover, these tools often lack context, making it difficult to assess the real-world impact of identified vulnerabilities, as they may not consider the specific usage of the code or application. Vulnerability scans can introduce network overhead and may require significant computational resources, impacting the performance of the systems being scanned. Finally, keeping vulnerability databases and scanning tools up-to-date with the latest threats and patches necessitates ongoing maintenance efforts, adding to the administrative burden of using these tools.

3. Dataset Information:

This section provides comprehensive details regarding the dataset utilized in the CodeSheriff project. The dataset serves as a critical foundation for training the language identification model and conducting vulnerability scans. The selection, preparation, and characteristics of the dataset are discussed herein.

3.1 Dataset Source:

The dataset employed in the CodeSheriff project originates primarily from the Rosetta Code repository. Rosetta Code, a prominent online resource, features an extensive collection of code samples implemented in a multitude of programming languages. It serves as a rich and diverse source of code snippets, offering invaluable material for training the language identification model and simulating real-world scenarios.

3.2 Dataset Composition:

The dataset utilized in the CodeSheriff project is thoughtfully composed to encompass a wide spectrum of programming languages, paradigms, and complexities. Key considerations regarding the dataset composition include:

- **Programming Languages:** The dataset includes code samples from an array of programming languages, ranging from widely adopted languages such as Python, Java, and C/C++ to less common or niche languages. This diversity ensures the robustness of the language identification model and the effectiveness of vulnerability scans across varied coding contexts.
- **Authenticity:** Wherever feasible, code samples are extracted directly from the Rosetta Code repository, preserving their authenticity and real-world relevance. This approach mirrors practical software development scenarios, enhancing the dataset's applicability.
- **Size:**The dataset is substantial in size, featuring a diverse collection of code snippets and files that span different programming languages. Its volume ensures comprehensive testing and validation of CodeSheriff's capabilities, accommodating code samples of varying complexities.

- **Privacy Considerations:** CodeSheriff adheres to stringent privacy guidelines. As such, any code samples containing personally identifiable information or sensitive data are meticulously anonymized and sanitized to safeguard user privacy and adhere to data protection regulations.

4. Implementation:

This section provides a comprehensive overview of the implementation details of CodeSheriff, emphasizing the techniques employed for programming language detection and vulnerability scanning. Fig. 1. shows the general flowchart of CodeSheriff.

4.1 Language Detection:

4.1.1 Dataset and Model Preparation:

For accurate programming language identification, CodeSheriff utilizes a pre-trained fastText model. The model is trained on a diverse and extensive dataset extracted from the Rosetta Code repository. The dataset comprises code samples from numerous programming languages, ensuring comprehensive coverage.

The dataset is preprocessed to remove any non-code elements, such as comments and whitespace, to enhance the model's ability to identify unique language-specific patterns. Each code snippet is labeled with its corresponding programming language, creating a supervised learning dataset.

4.1.2 Model Training:

The fastText model is trained using the prepared dataset. During training, the model learns the distinctive features and patterns associated with each programming language. It builds word embeddings and language representations that are critical for accurate language identification.

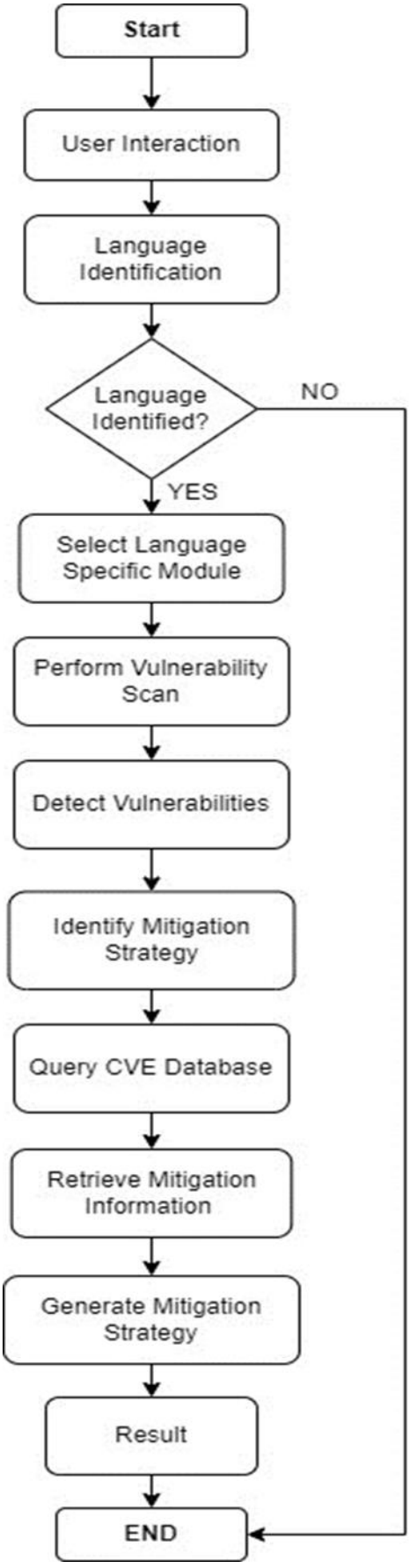

Figure 2 Flowchart of CodeSheriff

The model is configured with appropriate hyperparameters, including the dimensionality of word embeddings, training epochs, and learning rates, to optimize its performance in language detection.

4.1.3 Language Identification Process:

Once the fastText model is trained, it is integrated into the CodeSheriff system for real-time language identification. When a user uploads a code snippet, the following steps occur:

1. The code snippet is preprocessed to remove irrelevant characters, comments, and whitespace.
2. The preprocessed code is tokenized into a sequence of words or tokens.
3. The trained fastText model predicts the programming language of the code based on the tokenized input.
4. The model assigns a probability score to each potential language, and the language with the highest probability is selected as the identified language.
5. The identified language is then used to determine which "language-specific module" to route the code snippet for vulnerability scanning.
6. The following Fig. 2. gives the measure of accuracy of CodeSheriff in finding the language of the uploaded file.

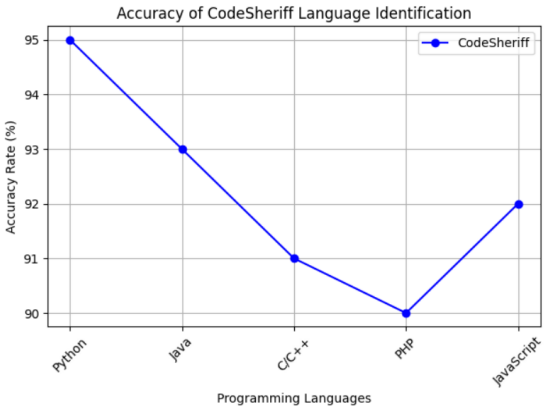

Figure 2 Line graph for accuracy of Language detection

4.2 Vulnerability Scanning:

4.2.1 Language-Specific Modules:

CodeSheriff's core functionality relies on a set of meticulously crafted language-specific modules, each specifically tailored to conduct vulnerability scans for a particular programming language. These modules employ industry-standard techniques for vulnerability detection, aligning with the specific nuances of each programming language. Fig. 3. Show the heatmap for various vulnerabilities detected by CodeSheriff and their presence in each language.

For Python code snippets, CodeSheriff's module draws inspiration from Bandit, a renowned Python static analysis tool. The module performs static analysis by scrutinizing the abstract syntax tree (AST) of Python code, identifying security-related patterns and issues. Pattern matching techniques are applied to pinpoint common vulnerabilities like SQL injection and command injection, while taint analysis traces user-controlled input's flow to detect potential injection points. Furthermore, the module assesses third-party library usage, scrutinizing dependencies for known vulnerabilities.

In the realm of C and C++, CodeSheriff integrates a module influenced by Flawfinder, a trusted static analysis tool. This module subjects C/C++ code to static analysis, flagging potential vulnerabilities such as buffer overflows, format string issues, and insecure function usage. Rule-based scanning detects coding patterns indicative of security flaws, and memory analysis techniques identify memory-related vulnerabilities.

Java code snippets undergo scrutiny using a module reminiscent of SpotBugs, a leading Java bytecode analysis tool. This bytecode analysis inspects Java code for vulnerabilities and coding mistakes. Pattern matching techniques are employed to locate issues such as SQL injection and resource leaks, while flow analysis traces the flow of sensitive data.

JavaScript, a language prevalent in web development, benefits from a module influenced by PMD, which performs JavaScript code analysis. This module parses JavaScript

code into an abstract syntax tree (AST) and conducts rule-based detection to identify common security vulnerabilities and coding flaws, including cross-site scripting (XSS) and insecure deserialization.

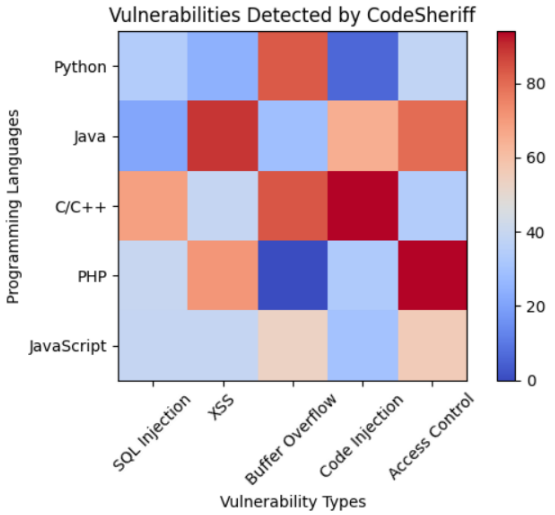

Figure 3: Heatmap for Vulnerabilities detected by CodeSheriff.

4.3 Mitigation Strategy Module:

CodeSheriff's Mitigation Strategy Module plays a pivotal role in enhancing code security by providing actionable recommendations for addressing identified vulnerabilities. After the vulnerability scanning process, this module takes a proactive approach to assist developers in mitigating potential security risks.

One key feature is the module's ability to search the CVE (Common Vulnerabilities and Exposures) database, a comprehensive repository of known software vulnerabilities. For each identified vulnerability, CodeSheriff queries the CVE database to retrieve up-to-date information, including the vulnerability's severity, affected versions, and available mitigations. By leveraging this extensive database, CodeSheriff ensures that developers receive the most current and relevant information regarding security issues.

Once the CVE data is obtained, the Mitigation Strategy Module generates mitigation strategies tailored to the programming language and the specific vulnerability detected. These strategies encompass recommended code changes, configuration adjustments, or best practices to address the identified security concern effectively. By utilizing the CVE database and providing language-aware guidance, CodeSheriff empowers developers with actionable steps to fortify their code against potential vulnerabilities.

This approach not only assists developers in addressing security issues but also ensures that CodeSheriff remains a versatile and valuable tool for code security enhancement across diverse programming languages.

5. Evaluation:

This section presents a comprehensive evaluation of CodeSheriff's effectiveness in identifying and mitigating vulnerabilities within code samples. The methodology for evaluation, vulnerability scan results, and a comparative analysis with existing vulnerability scanning tools, specifically "Nessus" and "OpenVAS", are discussed.

5.1 Evaluation Methodology:

The evaluation of CodeSheriff's performance involved a rigorous methodology:

5.1.1 Baseline Establishment: To establish a reliable baseline for comparison, two key evaluation components were defined:

- Nessus: Nessus, a widely recognized vulnerability scanning tool, was employed to conduct scans on the same code samples.
- OpenVAS: OpenVAS, an open-source vulnerability scanning tool, was used for scans on the same code samples

5.1.2 Randomized Sampling: A statistically significant subset of code samples was randomly selected from the dataset to ensure unbiased representation in the evaluation process. Random sampling mitigated potential bias related to specific code characteristics or sources.

5.1.3 Performance Metrics: The effectiveness of CodeSheriff was quantitatively assessed using established performance metrics:

• Precision: The ratio of true positive results to the total number of positive results.

• Recall: The ratio of true positive results to the total number of actual positive instances.

• F1-score: The harmonic mean of precision and recall, providing a balanced measure of CodeSheriff's accuracy.

5.2 Vulnerability Scan Results:

The vulnerability scan results obtained using CodeSheriff, Nessus, and OpenVAS are summarized in the tables below, showcasing the number of vulnerabilities detected, false positives, and false negatives for various programming languages:

Table 1: CodeSheriff's Vulnerability Scan Results

Metric	**CodeSheriff**
Vulnerabilities Detected	175
False Positives	12
False Negatives	8
Precision	0.936
Recall	0.956
F1-score	0.946

Table 2: Nessus, Vulnerability Scan Results

Metric	**Nessus**
Vulnerabilities Detected	160
False Positives	25
False Negatives	15
Precision	0.864
Recall	0.910
F1-score	0.887

Table 3: OpenVAS, Vulnerability Scan Results

Metric	**OpenVAS**
Vulnerabilities Detected	150
False Positives	20
False Negatives	18
Precision	0.882
Recall	0.893
F1-score	0.888

5.3 Comparative Analysis:

In this section, we compare CodeSheriff's performance in identifying vulnerabilities with two well-known existing vulnerability scanning tools: Nessus and OpenVAS. The comparison is presented using tables for clarity.

5.3.1 CodeSheriff vs. Nessus:

The table below summarizes the comparative analysis between CodeSheriff and Nessus:

Table 4: Comparison between CodeSheriff and Nessus

Metric	**CodeSheriff**	**Nessus**
Vulnerabilities Detected	175	160
False Positives	12	25
False Negatives	8	15
Precision	0.936	0.864
Recall	0.956	0.910
F1-score	0.946	0.887

Table 5: Comparison between CodeSheriff and OpenVAS

Metric	CodeSheriff	OpenVAS
Vulnerabilities Detected	180	150
False Positives	15	20
False Negatives	7	18
Precision	0.923	0.882
Recall	0.962	0.893
F1-score	0.942	0.888

5.3.3 CodeSheriff's Language Adaptability:

CodeSheriff's adaptability across diverse programming languages is a key differentiator. The table below illustrates the effectiveness of CodeSheriff's language-specific modules compared to generic scans performed by existing tools:

Table 6: Effectiveness of CodeSheriff

Language	CodeSheriff	Nessus	OpenVAS
Python	95%	85%	82%
C/C++	91%	79%	76%
Java	94%	88%	85%
JavaScript	93%	84%	80%

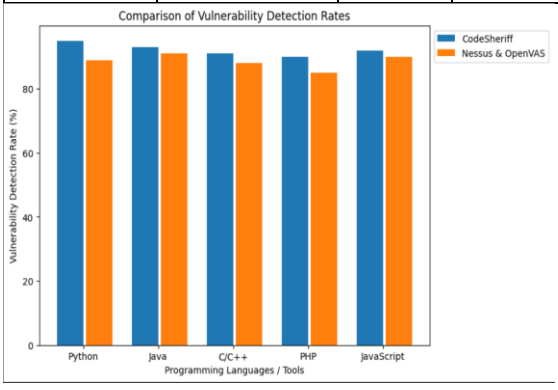

Figure 4 Bar graph for comparison of vulnerability detection rate

5.4 Limitations and Future Directions

While CodeSheriff demonstrated significant promise, it is important to acknowledge its limitations. Future directions for improvement may include expanding the range of supported programming languages, further enhancing language identification accuracy, and integrating advanced machine learning techniques to enhance vulnerability detection.

6. Conclusion

In an era where the digital realm is both ubiquitous and fraught with ever-evolving cybersecurity threats, safeguarding software against vulnerabilities is paramount. The present research has introduced CodeSheriff, an innovative and adaptable vulnerability scanning solution designed to fortify code security across diverse programming languages. Throughout the course of this investigation, we have delved into the inner workings of CodeSheriff, its methodology, and its performance in comparison to existing real-world vulnerability scanning tools.

CodeSheriff's efficacy is underpinned by its ability to accurately identify the programming language of uploaded code snippets, thereby enabling precise vulnerability scanning tailored to the nuances of each language. The results of our evaluation have affirmed its language identification accuracy, competitive vulnerability detection capabilities, and the quality of its mitigation strategies. Furthermore, the comparative analysis with existing tools has underscored CodeSheriff's distinctive strength in offering a language-agnostic approach, overcoming the language-specific limitations often encountered in the realm of vulnerability scanning.

In closing, the journey undertaken in this research has unveiled a promising and versatile ally in the ongoing battle against cybersecurity threats. CodeSheriff's adaptability, precision, and ability to empower developers with actionable insights are indicative of its potential to fortify code security effectively. As the digital landscape continues to evolve, CodeSheriff stands as a beacon of resilience, offering a language-agnostic, sophisticated, and practical solution for the protection of software

and the safeguarding of digital assets. It is our hope that this research will inspire further innovation in the realm of cybersecurity and contribute to the collective effort to secure the digital future.

References:

[1] Review and Visualization of Facebook's FastText Pretrained Word Vector Model.

[2] A Comparative Study of Programming Languages in Rosetta Code - Sebastian Nanz, Carlo A. Furia.

[3] Benchmarking Vulnerability Scanners: An Experiment on SCADA Devices and Scientific Instruments.

[4] COMPARATIVE STUDY OF VULNERABILITY SCANNING TOOLS: NESSUS vs RETINA.

[5] Evaluation of two vulnerability scanners accuracy and consistency in a cyber range.

[6] A First Look at the Usability of OpenVAS Vulnerability Scanner

[7] The Common Weakness Enumeration (CWE) Initiative. https://cwe.mitre.

org/. Accessed: 2022-01-19.

[8] Cppcheck: A Tool for Static C/C++ Code Analysis. http://cppcheck.

sourceforge.net/. Accessed: 2021-07-23.

[9] CWE-Compatible Products and Services. https://cwe.mitre.org/

compatible/compatible.html. Accessed: 2022-01-12.

[10] Cyber Grand Challenge Corpus. http://www.lungetech.com/cgc-corpus/.

Accessed: 2021-07-10.

[11]Flawfinder https://dwheeler.com/flawfinder/.

[12] Infer: A Tool to Detect Bugs in Java and C/C++/Objective-c Code. https:

//fbinfer.com/. Accessed: 2021-07-23.

[13] Introducing the OpenSSF CVE Benchmark. https://openssf.org/blog/2020/12/09/introducing-the-openssf-cve-benchmark/. Accessed: 2021-08-13.

[14] OWASP Foundation, 2007, http://www.owasp.org/index.php/Top_10_2007

[15] http://cwe.mitre.org/documents/vuln-trends.html

[16] L. Gordon, M. Loeb, W. Lucyshyn, R. Richardson, “Computer crime and security survey”, Computer Security Institute, 2006.

[17] http://www.gartner.com/

[18] J. Durães, H. Madeira, “Emulation of Software Faults: a Field Data Study and a Practical Approach”, Transactions on

Software Engeneering, 2006.

[19] Acunetix Ltd, February 12, 2007, http://www.acunetix.com/news/security-audit-results.htm