Automated Security Test Case Generation

Jeremy Gonzalo Arellano   
*Dept. of Computer Science and Engineering*  
*University of Notre Dame*Notre Dame, IN, USA  
jarella2@nd.edu

***Abstract* — This paper addresses the challenge of enhancing software security through automated security test case generation. With the growing prevalence of software vulnerabilities, there is an urgent need for efficient and effective security testing methods. This research aimed to develop a framework that leverages evolutionary algorithms for generating security test cases and evaluates these cases based on a risk assessment model. The methodology involved developing a vulnerability parsing algorithm using data from the Common Weakness Enumeration (CWE) catalog and testing it against various software repositories. Although the project encountered challenges in creating a fully functional algorithm, it yielded valuable insights into the complexities of automated security testing. The findings suggest that a more specialized approach, targeting specific types of vulnerabilities, may be more effective than a generalized security test generator. This pivot from a broad-scope tool to specialized security testing tools opens new avenues for future research in software security.**

**Index Terms — Automated Security Testing, Vulnerability Parsing, Evolutionary Algorithms, Software Vulnerabilities, Common Weakness Enumeration (CWE), Specialized Security Tools.**

# Introduction

In today’s modern world, software is constantly being faced with the threat of security breaches. Often, these breaches can be severe, leaving data compromised and services possibly suspended in the process [1]. While we have developed ways to prevent such breaches from happening thanks to the many static and dynamic approaches to security analysis, there still lies the problem of diagnosing and exposing any security threats that may still be lying around after this analysis [4]. Therefore, it is critical that some form of automated test case generation exists, where these generated tests will uncover potential vulnerabilities that often would be overlooked in normal analysis [4]. While current security testing methods are proficient at uncovering bugs/vulnerabilities, the amount of manual effort and time that these methods waste are not very efficient for large DevOps and agile environments [6]. Thus, the need for automated security test case generation is needed crucially in the world of technology today.

Software vulnerabilities stem from a plethora of sources, stemming from both low-level coding bugs that are generated from when the developer is creating the source code, to high-level design flaws that have to do with the architecture of a software piece [2][3]. While several tools and frameworks exist for identifying coding bugs and linting, detecting security bugs is another issue that requires different approaches and different scopes. In many cases, no security tool is perfect since there are many different vulnerabilities to consider and many more that emerge with evolving technology. Beyond just programming, a software’s or system’s architecture can also play a role in the security of itself, often discovering flows and potential security weaknesses based on design choices [5]. And beyond the architecture or a program/system, the evolution of a project exacerbates the challenge of maintaining a robust security posture throughout the development lifecycle [6].

This project aspires to bridge this gap by proposing an automated framework for security test case generation. Leveraging different algorithms, the framework aims to autonomously generate a suite of security test cases, prioritized on an integrated risk assessment system [4]. Thus, this approach not only augments the existing security testing process but also facilitates a more proactive stance towards software security, aiding in the early detection and remediation of security flaws [5].

Thus, throughout this research project, the aim is to answer the following questions:

### How can we use evolutionary algorithms to automatically create security test cases?

### How does ranking test cases based on risk affect the detection of serious vulnerabilities?

### How does our automated framework compare to current security testing methods in effectiveness and adaptability? How does our method compare to manual testing?

By analyzing these questions throughout the scope of this research project, the goal of creating and providing a new approach to automated solution for security test case generation. This project aims to contribute to the wider range of the software development ecosystem, providing a solution that will enhance the capability to uncover and resolve software vulnerabilities in everyday development.

# Related Work

While the core of this project serves to improve the landscape of security tools and techniques, several efforts have already been made in the realm of automated test case generation. In the realm of automated security test cases, various approaches have been taken at mitigating the risks of security breaches. Some have been more focused on the architectural perspective of a software, while others have taken the approach of using established but continuously evolving algorithms for generating security test cases.

Marksteiner, Ramler, and Sochor (2019) approach the problem in a unique way through integrating threat models within their automated test case generation. While the study’s focus was surrounding the use of these test cases with IoT applications, their methodology emphasizes the transition from threat identification to test case development [1]. From this transition of identification to test case development, the takeaway can be used to form an improved tool that ambitiously expands away from IoT application into other developer applications such as web apps. Thus, this integration is important for ensuring comprehensive coverage of potential vulnerabilities within the early development of an application, and important to serve as a foundation when developing an automated framework for security tests [1].

Building upon architectural flaws, further research has determined a shift within security on software’s security design flaws. The IEEE Center for Security Design’s report (2014) especially speaks on this shift through examining the move from bug detection to identifying security design flaws in software applications [2]. The study analyzes and points to the conclusion that security analysts in general must move away from the practice of focusing on an application's bug, but rather the design flaws of an application in general. Thus, this perspective given by the report is critical in understanding the role that architecture plays in software security. Through their analysis of the top 10 security design flaws, their research and findings provide an important guideline to building an analysis algorithm that can target these design issues rather than just code-level bugs [2].

Other works such as McGraw’s research provide a framework that builds upon existing software security practices, establishing a set of best practices that should be practiced in the field [3]. Throughout the research “Software Security: Building Security In” (2006), McGraw emphasizes the necessity of considering security at every level of an applications development, not just at the end of its development [3]. By ensuring that security is not just looked at as a last stage item in a software’s development, the focus is given to these practices to ensure that security is a fundamental aspect of software development. Thus, the research done will hopefully give a new light to different test case approaches by integrating security considerations in different stages of software development.

# Methodology

To discover the answers to our proposed questions, a framework first needs to be established for creating the automated security test case generation. By creating a system that utilizes evolutionary algorithms and by using architectural analysis in this system, the goal is to tackle an answer to our first proposed question, following suit with its performance by answering the following questions. Through this process of building and correcting our framework, the objective is to pave a path to test case generation that will be trivial on building on previous research.

## Framework Development

Our aim to develop our test case generation is to split the process to create a thorough framework. To integrate both architectural analysis and evolutionary algorithms into these test cases, we need a basis to work on and a starting point to see how these approaches can work together to produce anticipated results.

### Data Collection: The first step would be to gather any data necessary to evaluate our test cases through looking at different database sources and software repositories. Databases such as vulnerability databases would be useful in this search, as we can compare some of the most common security flaws within this database when evaluating our test suite. Utilizing the CWE catalog as a primary source of software weaknesses, we can further build a more detailed approach to different security breaches, and document how these security breaches could play a role in the development of our framework. A possible parsing mechanism would be developed to extract and retrieve relevant data from the list, extracting elements from the “Weakness” category of a CWE. By parsing these different XML files and excluding non-relevant information, we can further build a foundation for identifying vulnerabilities. And with different views on security breaches, the goal of studying these databases would hopefully bring a different perspective and testing points for our test cases.

Software Repositories would be another credible source of data when building our framework. Not only would this give real-world examples to develop and test our test-cases on, but it would also allow us to confirm common security patterns and flaws that are found in this process. By gathering a diverse range of real-world software projects, we can test the effectiveness of our framework and adjust as needed. While the selection of a repository might be complex, the choice will mostly be selected upon whether the repository can be tested and whether the codebase offers some software vulnerabilities.

### Evolutionary Algorithm Implementation: Once our database has been selected and has been solidly tested, our algorithmic approach can be developed. By drawing inspiration from genetic algorithms (GAs) and from parallel evolutionary methods, the hope is to form algorithms that prove efficient when deploying test cases [4]. Thus, our objective when developing these security algorithms is to optimize our defined test coverage criteria (such as branch or path coverage) and evolve our initial test cases into effective ones.

While this process of developing these algorithms plays out, some form of parallelization should be used to improve the speed and efficiency of our test generation process. Since parallelization offers some form speed improvements and effectiveness improvements when generating tests, it is important for us to develop this when creating/testing our algorithm [8]. Thus, our design should implement our evolutionary algorithm that will work on generating test cases concurrently, allowing for shorter generation times and more data for evaluation when testing the effectiveness and ability of our algorithm in the testing phase.

### Architectural Analysis: As mentioned before, the generation and analysis of the architecture of a software is important for us to analyze in the perspective of our first research question. By conducting analysis on different architectural documents from software projects, we can isolate the important steps when looking at a code-base and recognizing design flaws/vulnerabilities. Through exploring model-based testing approaches and using different architectural analysis tools, the aim here is to identify and document any architectural vulnerabilities that might have been found in this analysis [9]. And through looking at these common patterns, integrating them into our algorithm approach.

To integrate these test cases with our architectural analysis, the primary objective is to find common occurrences from our data sources and integrate them into our algorithm findings. By mapping different architectural flaws into our security evaluations, we can map them to certain test case scenarios and hopefully cover more vulnerabilities by considering this. Thus, with architectural analysis, the aim is to ensure that our test-suite covers both code-level bugs and architectural flaws along the way.

## Evaluation Framework

The next step once we have established a framework for test-case generation is to develop a framework for testing the effectiveness of our algorithm. By rigorously testing the effectiveness and efficiency of our automated security test case framework, the algorithm will have some metrics that can be used to improve the structure and analysis.

### Test Case Applications: By pinning our algorithm and test-case generation on real world software bases, we can test the performance metrics of our algorithm. Generating these test cases will also show whether the performance changes in different practical environments, and how widely the algorithm can be used. Pulling different open-source projects from GitHub will give a base to this research goal, allowing for the identification of flaws in the algorithm when looking at already identified security flaws in the GitHub repository.

### Risk Assessment: In answering our second proposed question, we must develop some type of model that will assess the risk associated with certain security vulnerabilities into our framework [10]. This model will serve the purpose of categorizing and prioritizing test cases based on the severity of a security vulnerability, integrating a type of priority queue when reporting these issues back as metrics. Through this, a prioritization will be given when executing the test cases, addressing the most critical vulnerabilities first and efficiently using resources on the faults that will most likely take the longest to assess and fix.

### Performance Metrics: As mentioned before, we need a way to test our algorithm and approach to see if it is any good when comparing it to previous tools. Not only would this provide the answer to our third research question, but it will serve to guide our algorithm to better metrics once flaws are identified and fixed. For one, coverage must be essentially covered for all code bases tested with our testing-suite. This would include paths such as branch coverage, paths, and other things that are normally called in the program being analyzed. Once identified, these will be pinned against what is already known from that database, giving way for a percentage to see how much of known vulnerabilities our algorithm covered. Efficiency would also be another metric worth measuring here, as it would give us the time and resources that our algorithm uses in its run time. The focus here would be to optimize our performance to be less time consuming and less resource intensive as previously mentioned tools. Thus, a timer and possibly resource monitor would serve useful to integrate into our program to evaluate its efficiency metrics. False Positive/Negative reporting would also serve vital to measure throughout the process of this project, since it would be important to see if our algorithm is performing correctly. By keeping track of the rate of false positives and false negatives, we can create a metric when pinned against found vulnerabilities, making it easier to assess the effectiveness of the testing suite. Thus, from these metrics, tweaks and refinements can be made to the algorithm and be reported later when discussing its performance when compared to other tools.

# Results

While the scope and ambition of this project were large in comparison to the time constraint allotted, some results were extracted from the research process. The focus of the research primary centered on addressing the first research question (RQ1), which explores the efficiently and effectiveness of an algorithm designed to parse and identify potential vulnerabilities within software repositories. While RQ2 and RQ3 are partially answered through the development of our parser, no definite conclusion was reached when attempting to develop answers for these questions. However, this section will walk through and detail the results that were achieved from answering RQ1, which will offer a foundational understanding of the potential and challenges inherent in automated security test case generation. The following subsections delineate the specific outcomes, observations, and reflections gathered from the project’s implementation.

## RQ1 Results: Efficiency and Effectiveness of the Vulnerability Parsing Algorithm

1. Algorithm Development and Implementation
   1. **Initial Steps**: The development process initiated with designing an algorithm capable of parsing Common Weakness Enumeration (CWE) data. This was critical in identifying prevalent software vulnerabilities and their characteristics.
   2. **Python Scripting and CWE Integration**: Using Python, the algorithm was crafted to efficiently parse through the CWE database. This integration allowed for the identification of vulnerability patterns relevant to various software repositories.
2. Performance Metrics and Evaluations:
   1. **Accuracy Analysis**: Due to the time constraints of the project, a proper analysis wasn’t measured due to the lack of proper identification of the vulnerabilities from the test repositories
   2. **Datasets**: A couple of known vulnerable repositors were used such as the OWASP Multillidae II, Vulnerable WordPress edition, and WebGoat. These repositories are known to be vulnerable and are used for testing purposes when testing security.
   3. **Scalability Insights**: The algorithm demonstrates some form of adaptability when ran through these repos, having the efficient measures to go through and identify the proper file extensions associated with the target pattern vulnerability.
   4. **Efficiency**: Parallelization was introduced, which makes the runtime of processing a repository much faster and produces results quick when attempting to run through a larger repository.
3. Encountered Challenges
   1. **Technical Hurdles**: The development phase presented several technical challenges, particularly in refining the algorithm's accuracy and handling complex data structures. This mainly was seen when analyzing the repository due to the vastness and complexity of some of the CWEs present in the creation of the database. There is also some CWE’s that had descriptions but no code to run a scan on, leading to these being skipped in the scan
   2. **Algorithmic Limitations**: Current limitations of the algorithm, including occasional false positives/negatives, were noted, signaling areas for future enhancement. For description CWE, no developed way of parsing of checking for these vulnerabilities was found during development. However, tweaks can be done to parse a potential example of these CWEs that will make this scan possible
4. Preliminary Conclusions and Forwards Path:
   1. **Effectiveness Assessment**: The preliminary analysis suggests that the algorithm holds promising potential in automating the detection of software vulnerabilities, albeit with room for improvement.
   2. **Prospects for Enhancement**: Future work will focus on refining the algorithm's accuracy and expanding its scope to address RQ2 and RQ3 more comprehensively.

## RQ2 Results: Impact of Risk-Based Test Case Ranking

Due to the complexities encountered in the initial development phases and time constraints, the evaluation and application of risk-based test case ranking (RQ2) remain inconclusive. The intended goal was to integrate a model for assessing the risk associated with identified vulnerabilities, thereby enabling the prioritization of test cases based on severity. However, given the current challenges with the primary algorithm's functionality, this aspect of the research could not be fully realized. The incomplete state of the primary algorithm has deferred the implementation of risk assessment models and their subsequent integration into the test case generation process. Future work in this area will aim to establish a robust model for risk assessment and seamlessly integrate it with the test case generation framework once the foundational algorithm achieves a stable and functional state.

## RQ3 Results: Comparative Analysis with Current Methods and Manual Testing

While this question also remains inconclusive, the results gathered from RQ1 do shed some light into this question on the significant challenges in developing a centralized security test case generator.

1. **Challenges of Broad-Scope Vulnerability Analysis**:
   1. Our initial approach aimed to create a comprehensive tool capable of identifying a wide array of security vulnerabilities across different software domains.
   2. However, the vast and diverse nature of security vulnerabilities in various fields presents a substantial challenge. The complexity and specificity of each type of vulnerability make it difficult for a single tool to effectively cover all aspects.
2. **Limitations of a Generalized Tool**:
   1. Our observations suggest that a one-size-fits-all solution in security test case generation may not be as effective as initially anticipated.
   2. The attempt to integrate a broad spectrum of security analysis into one framework led to complexities that hindered the tool's development and effectiveness.
3. **Advocating for Specialized Tools**:
   1. Based on our research and the encountered challenges, we propose that specialized tools, each targeting specific types of vulnerabilities or domains, might be more effective.
   2. This specialized approach allows for a deeper, more focused analysis of particular vulnerability types, enhancing the accuracy and efficacy of security testing.

# Conclusion

This research paper, while ambitious, explored the avenue of automated security test case generation, driven by the pressing need to enhance software security in an increasingly digital world. The research was guided by three pivotal questions: the feasibility of using evolutionary algorithms for test case generation, the impact of risk-based test case ranking, and the comparative analysis of this automated approach against current security methods and manual testing.

The core development centered around the first research question (RQ1), leading to the creation of a vulnerability parsing algorithm. This algorithm, designed to identify potential vulnerabilities in software repositories using data from the Common Weakness Enumeration (CWE) catalog, highlighted the complexity and the vast scope of security vulnerabilities. Despite the algorithm not reaching a fully functional state, the development process revealed critical insights into the challenges of creating a generalized security test generator. The diverse nature of software vulnerabilities suggested that a more effective approach might lie in developing specialized tools targeted at specific vulnerability types.

For RQ2 and RQ3, the research did not reach a conclusive stage due to the foundational challenges encountered with the algorithm. However, the initial exploration into these areas underscored the importance of specificity in security testing tools and the limitations of attempting to integrate a broad spectrum of security analysis into a single framework.

While the project did not fulfill all its intended objectives, the findings contribute significantly to the understanding of automated security test case generation. The research underscores the necessity of specialized tools for distinct security vulnerabilities, pivoting away from the initial goal of a comprehensive, all-encompassing solution. This paper's exploration opens pathways for future research, particularly in the realm of specialized security testing tools, each adept at tackling specific types of security threats in software development.

# References

Link to GitHub Repository for Project: <https://github.com/jarella2/Secured_Software_Engineering>

[1]S. Markstiener, R.Ramler. Integrating Threat Modeling and Automated Test Case Generation into Industrialized Software Security Testing, 2019

[2] T. I. C. S. C. for Secure Design. Avoiding the top 10 security design flaws, 2014.

[3] G. McGraw. Software security: building security in, volume 1. Addison-Wesley Professional, 2006

[4] Fraser, G., & Arcuri, A. (2014). A large-scale evaluation of automated unit test generation using EvoSuite.

[5] B.Chess, & G.McGraw. Static analysis for security, (2004)

[6] B. Boehm & R. Turner. Management Challenges to Implement Agile Processes in Traditional Development Organization, 2005

[7] Zalewski, M. (2011). American fuzzy lop

[8] Weiwei Wang, Shumei Wu, Zheng Li, Ruilian Zhao, Parallel evolutionary test case generation for web applications, Information and Software Technology, Volume 155, 2023

[9] Mohd-Shafie, M.L., Kadir, W.M.N.W., Lichter, H. et al. Model-based test case generation and prioritization: a systematic literature review. Softw Syst Model 21, 717–753 (2022). https://doi.org/10.1007/s10270-021-00924-8

[10] Felderer, Michael & Haisjackl, Christian & Pekar, Viktor & Breu, Ruth. (2014). A Risk Assessment Framework for Software Testing. 10.1007/978-3-662-45231-8\_21.