XXX-X-XXXX-XXXX-X/XX/$XX.00 ©2023 IEEE

A method based on Behavior Driven Development (BDD) and System-Theoretic Process Analysis (STPA) for verifying security requirements in critical software systems

Vitor Rubatino
Computer Science Department
Universidade Federal de Alfenas (UNIFAL-MG)
Alfenas, Brazil
vitor.rubatino@sou.unifal-mg.edu.br

Alice Batista
Computer Science Department
Universidade Federal de Alfenas (UNIFAL-MG)
Alfenas, Brazil
alice.nogueira@sou.unifal-mg.edu.br

Fellipe Guilherme Rey de Souza
Computer Science Department
Universidade Federal de Alfenas (UNIFAL-MG)
Alfenas, Brazil
fellipeguilhermerey@gmail.com

Rodrigo Martins Pagliares
Computer Science Department
Universidade Federal de Alfenas (UNIFAL-MG)
Alfenas, Brazil
rodrigo.pagliares@unifal-mg.edu.br

Abstract— **Security failures in critical software systems can lead to severe economic, environmental, and human consequences. To ensure the security of these systems, it is necessary to identify and document security requirements as part of the software development process. Although the System-Theoretic Process Analysis (STPA) technique can be used to identify security requirements, it is challenging to verify their accuracy, completeness, and consistency. We propose a method based on STPA and Behavior Driven Development (BDD) for verifying software security requirements. BDD establishes a common language between business analysts and software developers. We evaluate the method through examples related to preserving the Confidentiality, Integrity, and Availability (CIA) of information. The application of the method to the examples produces automated test cases written using Gherkin syntax, which are used to verify the requirements in the examples. The method proposed in this work has the potential to generate automated test cases that can be used to verify whether the software solution built meets the security requirements identified through an STPA analysis.**

Keywords— STAMP; STPA; BDD; Security requirements; Requirements verification.

I. INTRODUCTION

Our society has become increasingly dependent on critical software systems, such as those in energy, transportation, financial, healthcare, and communication. Security failures in these systems can result in serious consequences, including economic losses, human casualties, and environmental impacts [1] [2]. The development of critical software systems demands that security requirements be identified and documented as part of the software development process.

Verification of security requirements is crucial to ensure that the software meets its security objectives. This is accomplished by assessing the design and implementation of the software in relation to the specified security requirements, aiming to verify their completeness, consistency, and accuracy. However, due to the lack of systematic approaches, conducting security verification of critical software systems is a task that demands a significant amount of time and effort [3].

We propose a method for verifying software security requirements in critical systems. The method is based on STPA (Systems-Theoretic Process Analysis) and BDD (Behavior-Driven Development).

STPA is a technique for analyzing hazards and vulnerabilities in critical systems. This technique can be used to facilitate the identification safety/security requirements and test cases [2], enabling organizations to identify potential vulnerabilities before they can be exploited by malicious actors. While STPA technique can be used to identify security requirements, it is challenging to verify the completeness, consistency, and accuracy of the identified requirements.

BDD is a technique that aims to establish a common language between business analysts and software developers. One of the practices of BDD is the creation of test cases in natural language that describe the expected behavior of the software [4].

The remainder of this work is organized as follows: in Section II, the literature review and related work are presented; Section III introduces our method for verifying security requirements in critical software systems. Examples of the method's use are discussed in Section IV; Section V summarizes the results obtained in this work; discussion, conclusions and future work are presented in Sections VI and VII, respectively.

II. LITERATURE REVIEW AND RELATED WORK

Wang and Wagner [5] present a study, in the context of agile methods, to assess the use of BDD compared to the traditional User Acceptance Testing (UAT) aiming safety verification with STPA. The results of their work indicate that BDD is more effective than UAT for verifying safety requirements, taking into consideration the effectiveness of communication.

However, productivity, test rigor, and fault detection effectiveness did not show statistically significant differences.

Hirata et al. [3] propose a systematic approach for the semi-automatic generation of safety requirements and software test cases for critical systems. The authors combine requirements identified through STPA with a model-based approach called CoFI (Conformance and Fault Injection) for generating test cases. The use of the approach is exemplified with an insulin pump controlled by a smartphone system. Differently from their work, we utilize STPA analysis combined with BDD to create test scenarios and subsequently generate test cases.

Okubo et al. [6] address a common issue: defining the necessary security levels and privacy behaviors, as well as acceptance criteria for BDD, during the use of agile software development practices. The proposal of the method called BehaveSafe makes use of a Threat and Countermeasure graph (T&C graph) to establish acceptance criteria. The efficiency of this method was evaluated through a web-based system. Contrary to their method, we propose utilizing components of STPA risk analysis before practicing BDD.

Alves et al. [7] propose an approach for verification and validation of essential behaviors of a system to ensure its reliability. The approach uses state diagrams to represent the dynamic behavior of the system and runtime monitoring data. Both the state diagrams and monitoring data are verified and validated with test scenarios written using the Junit test framework. Meanwhile in our method, for verification, we use BDD, which not only covers essential system behaviors but also integrates business rules with programming language, focusing on software behavior.

Ghazel M. et al. [8] present an approach for specifying temporal requirements in complex systems. They also propose a verification method that integrates the specification process, enabling requirement verification. The authors present a case study in the field of railway operations. Contrary to your approach, the use of STPA results in our method simplifies the creation of test scenarios to verify security requirements, as through this feature, it becomes viable to execute test scenario mapping using the language Gherkin.

Purkayastha et al. [9] describe the use of a unit testing framework in Python to implement a formal security testing method. The authors employ a metric named Common Vulnerability Scoring System (CVSS) to represent the security state of a deployed system. The work outlines a series of BDD scripts for testing authentication and availability in an Electronic Health Records System. They proposed that BDD test scenarios written in Gherkin syntax serve as project documentation and to automate the tests. In a different way, we use STPA's UCAs and controller constraints to map to Gherkin syntax.

Tsoukalas et al. [10] present a mechanism that automates the identification of key concepts in security requirements expressed in natural language through syntactic and semantic analysis. Additionally, the authors propose a mechanism for the verification and validation of requirements by comparing them to a list of established security requirements, identifying inconsistencies, and suggesting refinements. Both mechanisms are implemented as standalone web services and are demonstrated through test cases aiming at facilitating software security specification and assurance. Meanwhile, we propose a method where the verification using BDD allows anyone, from engineers to product owners, to write BDD scenarios, thereby further enhancing the desired system behavior.

The work of Wang and Wagner [5] is the closest to our method. Similar to our work, the approach used by the authors combines the results of an STPA analysis with BDD for requirement verification. However, our work differs from Wang and Wagner's in several aspects, including our focus on security requirements (not safety). Additionally, in Wang and Wagner's work, STPA analysis is used as an artifact developed during their approach, while in our work, we adopt STPA analysis as input for our method, which provides more flexibility in its applicability. Furthermore, our method incorporates a set of resources and tool support that assist in the automation and verification of test cases, as described in Section V.

III. A METHOD FOR VERIFYING SOFTWARE SECURITY REQUIREMENTS IN CRITICAL SYSTEMS

Figure 3.1 presents the method proposed in this work. The method is divided into two main stages: security analysis (STPA) and security verification (BDD). Each stage, in turn, is divided into one or more activities that consume and produce artifacts and are performed by roles.

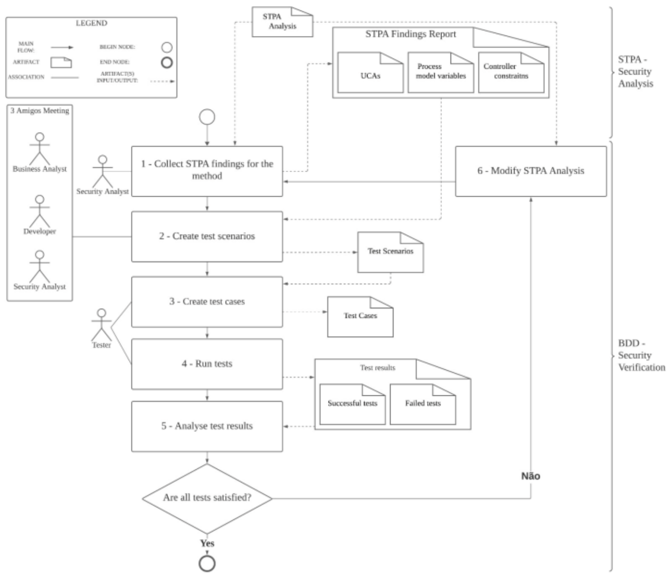


Figure 3.1 Method based on BDD and STPA for verifying security requirements in critical software systems

The role *Security Analyst* performs the first activity (*Collect STPA findings for the method*) which receives as input the artifact "*STPA Analysis*". The activity extracts, from steps 2 and 3 of the *STPA Analysis*, the information relevant to the method (process model variables from step 2 and UCAs and controller constraints from step 3). The activity produces the *STPA findings* artifact as output.

In the second activity, called "*Create test scenarios*", a meeting known as “Three amigos" takes place. The Business Analyst, Security Analyst and Tester roles participate in the meeting with the objective of generating BDD test scenarios written in the Gherkin syntax. The scenarios represent events that possibly trigger the UCAs identified in the previous activity.

In "Three Amigos" meeting, each role plays a crucial role, leveraging three distinct perspectives. The security analyst contributes insight into security requirements, the developer assesses implementation feasibility, while the business analyst ensures the need to meet project requirements.

The development of test scenarios uses Gherkin syntax. To this end, we use each UCA and corresponding *controller constraint* within the *STPA findings*. We decompose the UCAs into 5 parts, as defined by Leveson and Thomas [2] (see top of Figure 3.2). Each of the parts of a UCA is mapped to Gherkin syntax (*Given*, *When*, *Then*) as follows: 'Given [Context], When [Source + Type + Control Action], Then [Controller Constraint]'. This mapping is illustrated at the bottom part of Figure 3.2.

The "*Context*" is derived from the UCA description and is inserted after the 'Given' clause of Gherkin syntax, establishing the initial conditions for the test scenario. The *'Source'* and *'Type'* are combined with the *'Control Action*' and are declared after the *'When'* clause, indicating the specific action being declared. Lastly, the *'Then'* clause is followed by the *'Controller constraint*', describing the expected behavior of the system after executing the test scenario.

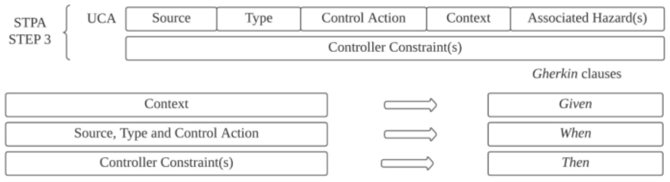


Figure 3.2 Mapping UCAs and *Controller constraints* to Gherkin Syntax

The next activity in our approach, named *'Create test cases*', is performed by the Tester based on the test scenarios previously created in the second activity. In this activity, '*Test cases*' are produced as output artifact.

In the fourth activity, the Tester executes the test cases created in the previous activity. This activity produces the artifact "*Test results*", which is made up of *'Tests that passed*', and '*Tests that failed*'.

In the next activity "Analyze test results", we analyze the results of previously executed tests. When 'Failed Tests' are identified, it is necessary to proceed to the 'Modify STPA analysis' activity, to correct the inconsistencies found. Then the cycle begins again, going through all method activities, until all tests are successful and satisfied.

IV. EXAMPLES

In this section, we present two examples to verify the feasibility of the proposed method. The purpose of these examples is to illustrate how we can use our method to verify security requirements.

For both examples, we performed an STPA analysis to be used as input for the first activity of our method (see Figure 3.1). In particular, the STPA analysis created focuses on issues associated with preserving the Confidentiality, Integrity, and Availability (CIA) of information [11].

The examples illustrate a role-based, user authentication and authorization system. We used our method to generate test cases in order to guide the implementation of the system.

We chose the Java programming language and the Spring Boot framework [12] to develop the system, subdivided into logical layers (presentation, control, business, and persistence). The implementation includes, in the presentation layer, a simple web form with the `username` and `password` fields and a button named `login`.

After implementation, we execute the fifth activity of our method (*Analyze test results*) to verify whether the implemented system complies with the security requirement and the test cases obtained within activity 3 (*Create test cases*). In what follows, we provide more details about the examples.

To better illustrate the examples, we show in Figure 4.1 the STPA control structure used as input for the first activity of our method. It has a controlled process (Repository) and four controllers: User, Login Controller, Authenticator, and Authorizer. The User controller is responsible for issuing the control action "provide credentials" to try to gain access to the system. Therefore, it sends the variables "`username`" and "`password`" for validation in the Repository. The Login Controller is responsible for intermediating access between the User and Authenticator controllers through the "request access" control action. The Authenticator is in charge of performing authentication through the "fetch user" control action (validation of the "`username`" and "`password`" stored in the Repository).

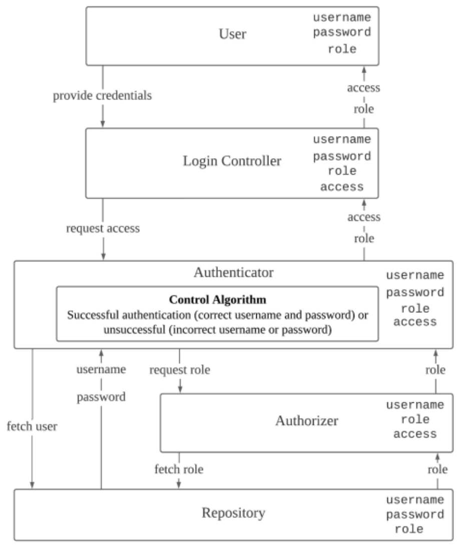


Figure 4.1 Control Structure for user authentication and authorization system

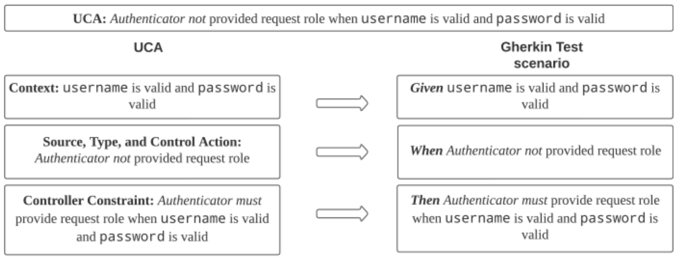
If both "`username`" and "`password`" provided are not found in the Repository, the Authenticator returns to the Login Controller the "access" feedback with the value "Not Allowed" and the "role" feedback with the value "Unknown". If both "username" and "password" are found in the Repository, the Authenticator issues the "request role" control action to the

Authorizer, which in turn provides the "fetch role" control action to the Repository. The Repository's response is the role found for the provided "username". The Authorizer returns the content of the "role" variable to the Authenticator, which in turn completes the authorization, granting access to the Login Controller and User (when `"username"` is `"Valid"`, `"password"` is `"Valid"` and `" role"` is `"Admin"` or `"User"`) or denying access (when `"username"` is `"Valid"`, `"password"` is `"Valid"` and `"role"` is `"Unknown"`).

A. Confidentiality and integrity

For this first example, by using the *STPA analysis* as input, we derive the *UCA's*, *Process Model Variables* and *Controller Constraints*, generating the *STPA Findings Report* artifact (first activity of our method).

We then use the *STPA Findings Report*, generated in the previous activity, to create test scenarios using Ghekin syntax (second activity). As a practical example, suppose the following UCA: "*Authenticator not provided request role when* `username` *is valid and* `password` *is valid*", together with the *Controller Restriction*: "*Authenticator must provide request role when* `username` *is valid and* `password` *is valid*". Figure 4.2 shows the test scenario created using Gherkin syntax for the UCA used as example.



UCA: Authenticator not provided request role when username is valid and password is valid
UCA
Gherkin Test scenario
Context: username is valid and password is valid
Given username is valid and password is valid
Source, Type, and Control Action: Authenticator not provided request role
When Authenticator not provided request role
Controller Constraint: Authenticator must provide request role when username is valid and password is valid
Then Authenticator must provide request role when username is valid and password is valid

Figure 4.2 Example of converting a UCA and a *Controller constraint* into a test scenario using Gherkin syntax

In the third activity of our method, we develop the automated test case for the previously created acceptance and integration test scenario. This was accomplished using the Cucumber [13], JUnit [14], and Selenium [15] frameworks, as exemplified in Figure 4.3.

The `@Given` annotation (Line 01) describes the method `usernameIsValidAndPasswordIsValid`. This method configures the Chrome browser and directs the navigation from the home page to the system's login page (Lines 06 to 08). Then, the method populates the username and password fields with invalid values (Lines 11 to 14) and acts by locating and clicking the login button present in the form (Lines 15-17).

In lines 19 to 24, the `@When` annotation describes the ongoing action: *when the authenticator does not provide the control action to request role*. This action is represented by clicking a button that grants access to the user-specific page, provided the user has the "USER" role associated. In Line 23, we ensure, through the `verifyNoInteractions` method, that no interaction is occurring.

The `@Then` annotation and subsequent code (Lines 25 to 37) establish the desired system behavior to prevent UCA. In this scenario, when the Authenticator does not provide the expected action, the system must ensure the action is successfully completed. To achieve this, we locate the user page access button and complete the action by clicking on it. We then verify if the current URL matches the expected URL, ensuring that the user has gained access to the desired page.

After executing the tests, we proceed to the Analyze Test Results activity using the Gauntlt tool [16]. Our goal is to detect failures that may indicate vulnerabilities in the system. These identified failures serve as indicators that require a detailed review of the STPA Analysis and possibly the STPA findings report, as seen in Figure 3.1.

To correct the failures identified in the fifth activity (*Analyze test results*), we move on to the next activity (*Modify STPA Analysis*), in which we undertake an iterative process. This involves reviewing the STPA analysis and teste cases implementation, making necessary modifications, and returning to the first activity of the method, updating the *STPA findings report*. From this report, we develop new test scenarios and test cases, execute the tests, analyze the results, and make necessary corrections. We repeat this cycle until no more test failures occur.

```
01  @Given("username is valid and password is valid")
02  public void usernameIsValidAndPasswordIsValid(){
03    ChromeOptions options = new ChromeOptions();
04    driver = new ChromeDriver(options);
05    driver.get("http://localhost:8080");
06    WebElement authenticate =
07      driver.findElement(By.id("login-button"));
08    authenticate.click();
09    WebElement username =
10      driver.findElement(By.name("username"));
11    WebElement username =
12      driver.findElement(By.name("password"));
13    username.sendKeys("validUser@email.com");
14    password.sendKeys("validPassword@123");
15    WebElement login =
16      driver.findElement(By.id("login-button"));
17    login.click();
18  }
19  @When("authenticator not provided request role")
20  public void notRequestRole(){
21    driver.get("http://localhost:8080");
22    WebElement mock = mock(WebElement.class);
23    VerifyNoInteractions(mock);
24  }
25  @Then(" authenticator must provided request role when
26  username is valid and password is valid")
27  public void mustRequestRoel(){
28     WebElement mock = mock(WebElement.class);
29    driver.get("http://localhost:8080");
30    WebElement user = driver.findElement(By.id("user"));
31    user.click();
32    String currentUrl = driver.getCurrentUrl();
33    assertThat(currentUrl).isEqualTo("http://localhost:8080/home-
34  user");
35    verifyNoMoreInteractions(mock);
36    driver.quit();
37  }
```

Figure 4.3 Automated test case for the test scenario "Authenticator provides request role when username and password are not valid"

B. *Availability*

In order to verify security requirements related to information availability, we followed our method and conducted automated tests with the aid of the Karate [17] (for API tests) and Gatling (for loading tests) [18] tools. Our objective is to evaluate the availability of information of the user authentication and authorization system.

In the first activity of our method, we create the *STPA findings report*. Then, in the subsequent activity, using the mapping to Gherkin syntax, we use the artifact generated in the previous activity to create test scenarios.

In this example, we use the UCA: "*Authorizer provides fetch role too early when username is provided and password is provided*" and the *Controller Constraint*: "*Authorizer should not provide fetch role too early when username is provided and password is provided*". The purpose of this test is to verify whether the Authorizer is granting system access level before the authentication process is properly completed. The mapping for the Gherkin syntax corresponding to this scenario is shown in Figure 4.4.

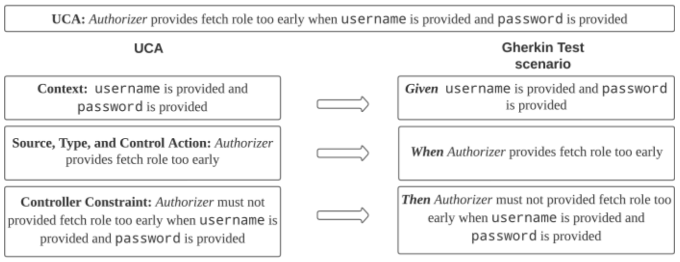

UCA: Authorizer provides fetch role too early when username is provided and password is provided
UCA
Gherkin Test scenario
Context: username is provided and password is provided
Source, Type, and Control Action: Authorizer provides fetch role too early
Controller Constraint: Authorizer must not provided fetch role too early when username is provided and password is provided
Given username is provided and password is provided
When Authorizer provides fetch role too early
Then Authorizer must not provided fetch role too early when username is provided and password is provided

Figure 4.4 Example of converting a UCA into a test scenario using Gherkin syntax

With the test scenarios already defined, we proceed to the third activity, using the Karate tool to develop the test cases. An example of code using Gherkin syntax for the Karate tool can be seen in Figure 4.5. The purpose of the code for this example is to describe the expected behavior of the *Authorizer* controller in a web system, as the example described in this section.

Lines 02 to 05 contain global settings applicable to all test cases. In this snippet, we configure the base URL as `'http://localhost:8080/'` and set the read and connection timeouts (`readTimeout` and `connectTimeout`) limits to 1000 milliseconds.

The scenario indicates the specific test scenario to be tested (Lines 06 and 07). From Line 08 to Line 10, the `Given` clause establishes the initial context of the test, including defining the path as `'login'` and filling in the form with specific values. The `When` clause, in Lines 11 and 12, represents the action to be performed, that is, navigate to the path `'home-user'`, that requires authorization for access.

Finally, the `Then` clause, on Line 13, establishes the expected result of the previous action, indicating that the system should return an `HTTP 401` status code, used to denote unauthorized authentication.

```
01  Feature: Authorizer
02  Background:
03    * url 'http://localhost:8080/'
04    * configure readTimeout = 1000
05    * configure connectTimeout = 1000
06  Scenario: Authorizer provides fetch role too early When
07  username is provided and password is provided
08    Given path 'login'
09    And form field username = 'user@email.com'
10    And form field password = 'password123'
11    When path 'home-user'
12    And method get
13    Then status 401
```

Figure 4.5 Example of converting a UCA into a test scenario using Gherkin syntax

In the fourth activity, which involves executing tests, we implement a program using the Gatling tool. This program is written in the Scala programming language and its purpose is to perform the requests defined by the test case. An excerpt from the program is described in Figure 4.6.

The code snippet provided is part of a availability of information testing script using Gatling. It sets up a scenario called `authScenario`, in which 10 virtual users are gradually injected over the course of 5 seconds. Additionally, it specifies the HTTP configuration defined in `httpConf` for the test's HTTP requests.

```
setUp(
   authScenario.inject(rampUsers(10) during (5
seconds)).protocols(httpConf))
```

Figure 4.6 Code snippet to perform requests and generate performance data for the role-based authentication and authorization system.

Furthermore, the program generates application performance data, as can be seen in Figure 4.7.

```
 ----Global Information-----------------------------------------------------
> request count                            10 (OK=10    KO=0  )
> min response time                         6 (OK=6     KO=-  )
> max response time                       129 (OK=129   KO=-  )
> mean response time                       88 (OK=88    KO=-  )
> std deviation                            50 (OK=50    KO=-  )
> response time 50th percentile           125 (OK=125   KO=-  )
> response time 75th percentile           127 (OK=127   KO=-  )
> response time 95th percentile           129 (OK=129   KO=-  )
> response time 99th percentile           129 (OK=129   KO=-  )
> mean requests/sec                         2 (OK=2     KO=-  )
----Response Time Distribution-------------------------------------------
> t < 800 ms                               10 ( 100%)
> 800 ms < t <1200 ms                       8 (  0%)
> t > 1200 ms                               0 (  0%)
> failed                                    0 (  0%)
```

Figure 4.7 Performance report for a sample of requests of size n = 10

As a result of the simulations carried out, we generate a performance report that provides a view of the application's behavior during the test (Figure 4.7). The 'request count' indicates that 10 requests were made, all of which were successful (OK=10, KO=0). The 'min Response Time' represents the minimum response time of the application, which is equal to 6 milliseconds, while the 'max Response Time' indicates the maximum response time (129 milliseconds). The 'average response time' corresponds to the average response times, which was 88 milliseconds during the test. The remaining lines provide information about the distribution of response times.

After executing the tests, we analyze the performance report results (fifth activity) and proceed to the sixth activity (Modify STPA Analysis) to make the necessary modifications to the STPA analysis. This allows us to enter the iterative process of the method, continuing until no more test failures occur.

V. RESULTS

We believe that the adoption of the proposed method described in this work can assist security analysts in identifying inconsistencies in the security requirements found in an STPA analysis. The method provides a sequence of activities that facilitate the verification of the correctness of the requirements, which, in turn, contributes to preventing scenarios of losses that may initially not appear critical, but can prove crucial in the context of the analyzed system.

The method can be important for the field of security, especially in a scenario where threats and vulnerabilities are becoming increasingly complex and subtle to be identified. Identifying all potential risks and adequately protecting the system is becoming a challenging task.

Our method has the potential to assist in verifying the integrity and completeness of security requirements, as it allows for a more comprehensive and systematic approach to security verification.

VI. DISCUSSIONS

In recent years, we have observed a significant increase in the importance attributed to the security requirements verification of critical software systems. By incorporating security verification controls during software development, it is possible to identify and mitigate potential vulnerabilities and security risks before the software is implemented.

The early implementation of a security requirements verification method for critical software systems allows hazards to be considered from the outset of a software development project, rather than being addressed later. In this way, potential hazards or risks that could negatively impact the system's operation or integrity can be identified and verified.

We believe our approach is more practical and flexible compared to other similar articles. In the work of Wang and Wagner [5], the STPA analysis is developed during the approach and in this method, the analysis is an input artifact, providing greater flexibility and facilitating its application.

Contrary Hirata et al. [3], our emphasis is on verifying security requirements, establishing a common language among those involved to create test scenarios. This approach simplifies the efficient generation of automated test cases to verify security requirements more effectively.

Furthermore, utilizing a method for security requirement verification enables vulnerabilities traceability, which facilitates the verification and correction of security issues. The method helps to ensure that the system meets the objectives set by security analysts and has the potential to prevent potential failures in critical software systems.

VII. CONCLUSION, RECOMMENDATION, AND FUTURE WORK

The proposed method has the potential to aid security analysts in verifying security requirements in a systematic way.

In future research, it may become promising to explore the use of the control algorithm as an improved approach. Its application can extend the verification of security requirements, identifying potential flaws in this domain more comprehensively. This perspective may represent an opportunity to strengthen the integrity of security requirements in critical software systems.

However, it is important to emphasize that the efficiency and effectiveness of this method still need to be demonstrated through additional examples and applications. In particular, we are interested in evaluating our method with STRIDE [19] examples.

REFERENCES

1. I. F. Sommerville, "Software Engineering," 10th ed., Essex, UK: Pearson Education, 2016.
2. N. G. Leveson and J. P. Thomas, "STPA Handbook," Cambridge, MA, USA: Cambridge The MIT Press, 2018.
3. C. M. Hirata and A. M. Ambrosio, "Combining STPA With CoFI to Generate Requirements and Test Cases for Safety-Critical System," IEEE Systems Journal, vol. 16, no. 4, pp. 6635-6646, Dec. 2022.
4. "Cucumber Docs," 2023. Available at: https://cucumber.io/docs/gherkin. Last access: August 29, 2023.
5. Y. Wang and S. Wagner, "Combining STPA and BDD for Safety Analysis and Verification in Agile Development: A Controlled Experiment," in Agile Processes in Software Engineering and Extreme Programming, J. Garbajosa, X. Wang, and A. Aguiar (eds.), New York: Springer, 2018, vol. 314, pp. 37-53.
6. T. Okubo, Y. Kakizaki, T. Kobashi, H. Washizaki, S. Ogata, H. Kaiya, and N. Yoshioka, "Security and privacy behavior definition for behavior driven development," in Product-Focused Software Process Improvement: PROFES 2014, A. Jedlitschka, P. Kuvaja, M. Kuhrmann, T. Männistö, J. Münch, and M. Raatikainen (eds.), Helsinki, Finland: Springer, 2014, vol. 8892, pp. 306-309.
7. M. Alves, D. Drusinsky, and M. Shing, "A practical formal approach for requirements validation and verification of dependable systems," in 2011 Fifth Latin-American Symposium on Dependable Computing Workshops, 2011, São José dos Campos, New York: IEEE, 2011, pp. 47-51.
8. M. Ghazel, M. Masmoudi, and A. Toguyeni, "Verification of temporal requirements of complex systems using UML patterns, application to a railway control example," in 2009 IEEE International Conference on System of Systems Engineering, 2009, Albuquerque, New York: IEEE, 2009, pp. 1-6.
9. S. Purkayastha, S. Goyal, T. Phillips, H. Wu, B. Haakenson, and X. Zou, "Continuous Security through Integration Testing in an Electronic Health Records

System," in 2020 International Conference on Software Security and Assurance (ICSSA), 2020, pp. 26-31.

10. D. Tsoukalas, M. Siavvas, M. Mathioudaki, and D. Kehagias, "An Ontology-based Approach for Automatic Specification, Verification, and Validation of Software Security Requirements: Preliminary Results," in 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2021, pp. 83-91.
11. P. Bourque and R. E. Fairley, "SWEBOK: Guide to the Software Engineering Body of Knowledge," 4th ed., Los Alamitos, CA: IEEE Computer Society, 2014.
12. C. Walls, "Spring in Action," 6th ed., Shelter Island, NY: Manning, 2022.
13. J. F. Smart, "BDD in Action: Behavior-driven development for the whole software lifecycle," 1st ed., Shelter Island, NY: Manning, 2014.
14. C. Tudose, "JUnit in Action," 3rd ed., Shelter Island, NY: Manning, 2020.
15. B. Garcia, "Hands-On Selenium Webdriver with Java: A Deep Dive Into the Development of End-To-End Tests," 1st ed., Sebastopol, California, USA: O'Reilly Media, 2022.
16. "Gauntlt Getting-started," 2023. Available at: http://gauntlt.org/index.html#getting-started. Access on July 25, 2023.
17. B. Bischof, "Writing API Tests with Karate: Enhance your API testing for improved security and performance," 1st ed., Birmingham: Packt Publishing, 2023.
18. "Gatling Docs," 2023. Available at: https://gatling.io/docs. Access on August 1, 2023.
19. Microsoft, "Uncover security design flaws using the STRIDE approach," MSDN Magazine, November 2006.