Hierarchical Colored Petri Nets for Vulnerability Detection in Software Architectures

Maya Benabdelhafid[1], Kamel Adi[1], Omer Landry Nguena Timo[1] and Luigi Logrippo[1,2]
[1]*Computer Security Research Laboratory, Université du Québec en Outaouais, Gatineau, Canada*
[2]*School of Electrical Engineering and Computer Science, University of Ottawa, Ontario, Canada*
fi

Keywords: Software Architecture, Security, Vulnerability Detection, Access Control, Hierarchical Colored Petri Nets, HCPN, Model Checking, ASK-CTL, CPN Tools.

Abstract: Hierarchical Colored Petri Nets (HCPNs) are a powerful formalism for modeling complex systems. This paper presents a formal approach based on HCPN for vulnerability detection in software architecture. By incorporating model checking and the enhanced computing-timing logic of ASK-CTL queries, the proposed approach enables rigorous security property verification. Through a case study of a hypothetical small library system, we demonstrate how this automated process effectively identifies a critical Access Control vulnerability: a regular user gaining unauthorized access to a function reserved for librarians.

1 INTRODUCTION

In response to the growing number of Access Control (AC) attacks, as well as other types of attacks, a wide range of methods for vulnerability detection have been proposed in recent years. Among the proposed methods, we find the formal methods that offer a rigorous approach to address cyber threats by employing mathematically precise model-based techniques. Formal models, representing software, hardware, or both, contribute to ensuring system consistency through rigorous formal proofs. They complement traditional testing methods, which are typically limited to specific execution scenarios.

Within the domain of formal methods, theorem proving relies on computer-based proofs to verify system properties, while model checking exhaustively ensures that a finite-state model meets specified requirements (Kulik et al., 2022). Software model checking, an automated approach for identifying defects in software, transforms source programs into models and explores their state space to detect counterexamples that violate specified properties. The presence of a counterexample indicates a vulnerability in the source program (Zhong et al., 2023).

Recent studies have highlighted the effectiveness of model checking as a powerful technique for analyzing security vulnerabilities (Rouland et al., 2024). This paper is positioned within the broader context of identifying design-level vulnerabilities with a particular emphasis on mitigating AC attacks using ASK-CTL based model checking. It proposes a formal framework based on Hierarchical CPN (HCPNs) for modeling, enabling modular, maintainable design and offering a structured approach to build secure software architectures. HCPNs provide a formal foundation that enables rigorous verification of system properties, while their expressive primitives allow for precise modeling of security-related interactions. Once the architecture is modeled, we show how the model checking technique can be employed to verify if the model satisfies security properties formulated in ASK-CTL formulas. Vulnerabilities (namely, AC flows) in software architecture are modeled, enabling early detection of security flaws before deployment. Our work also demonstrates that, during the detection process, valuable insights can be gathered regarding the instances of detected vulnerabilities, making it possible to later integrate security patterns to mitigate vulnerabilities.

The remainder of this paper is structured as follows: Section 2 covers preliminaries; Section 3 outlines the software architecture and AC challenges; Section 4 details our HCPN-based vulnerability detection approach; Section 5 reviews related work; and Section 6 concludes with a summary and future directions.

Benabdelhafid, M., Adi, K., Timo, O. L. N. and Logrippo, L.
Hierarchical Colored Petri Nets for Vulnerability Detection in Software Architectures.
DOI: 10.5220/0013638700003979
In *Proceedings of the 22nd International Conference on Security and Cryptography (SECRYPT 2025)*, pages 523-530
ISBN: 978-989-758-760-3; ISSN: 2184-7711

2 PRELIMINARIES

2.1 Colored Petri Nets

CPNs allow the definition of tokens using data types and complex data manipulations (Jensen et al., 2007). Each token carries an associated data value, referred to as a colored token that can be examined and modified by the transitions occurring in the model, enabling dynamic behavior representation.

Definition 1 (CPN syntax)**.** *A **CPN** is defined as a 9-tuple: $CPN = (\Sigma, P, T, I, O, C, G, V, M_0)$, where:*

- Σ *is a finite set of color sets, which are non-empty types;*
- $P = \{p_1, p_2, \ldots, p_m\}$ *is a finite set of places and* $T = \{t_1, t_2, \ldots, t_n\}$ *is a finite set of transitions, such that* $P \cap T = \emptyset$*;*
- I *is an input function* $I : P \times T \to \mathbb{N}$ *that defines directed arcs from places to transitions;*
- O *is an output function* $O : T \times P \to \mathbb{N}$ *that defines directed arcs from transitions to places;*
- $C : P \to \Sigma$ *is a color function that assigns a color set to each place;*
- G *is a guard function* $G : T \to Expr_{Bool}$ *such that* $\forall t \in T$, $Type(G(t)) = Bool$ *and the types of variables in* $G(t)$ *are subsets of* Σ*;*
- V *is a finite set of variables* $v \in V$ *where each variable has a color type* $c \in C$*. Arc expressions and guards can contain such variables;*
- M_0 *is the initial marking function* $M_0 : P \to Bag(C(p))$*, specifying the initial distribution of colored tokens in each place* $p \in P$*.*

Definition 2 (CPN semantics)**.** *The semantics of a CPN is described by a transition system called State Space :* $SS = (\mathcal{M}, M_0, \to)$ *:*

- *the configurations of* $\mathcal{M}$ *are markings* M*, with* $M(p) \in Bag(C(p))$ *for each place* $p \in P$*,*
- *the initial configuration is the initial marking* M_0*,*
- *the transition relation* $\to$ *is defined as follows: Let* t *be a transition and let* v *be a valuation of variables. We write* $v^-(p,t) \in Bag(C(p))$ *and* $v^+(t,p) \in Bag(C(p))$ *for the respective values of* $E(p,t)$ *and* $E(t,p)$ *for* $p \in P$*. The transition* $M \xrightarrow{t,v} M'$ *is possible if, for each place* $p \in P$*,* $M(p) \geq v^-(p,t)$ *and in this case,* $M'(p) = M(p) - v^-(p,t) + v^+(t,p)$ *for each* $p \in P$*.*
- *To fire a transition, all variables must be associated with specific colors. An* execution *starting from* M *is a sequence of firings* $M \xrightarrow{t_1,v_1} M_1 \xrightarrow{t_2,v_2} M_2 \ldots$*. A marking* M' *is reachable from* M *if there exists a finite execution (called Path)* $M \xrightarrow{t_1,v_1} M_1 \xrightarrow{t_2,v_2} M_2 \ldots \xrightarrow{t_n,v_n} M_n$ *starting from* M *such that* $M' = M_n$*.*

2.2 Hierarchical Colored Petri Nets

HCPNs extend CPNs to support modular modeling of complex systems (Jensen and Kristensen, 2009). They use substitution transitions to encapsulate modules and port places as interfaces for token exchange.

Definition 3 (CPN module)**.** *A **CPN module** is a 4-tuple:* $CPN_{module} = (CPN, T_{sub}, P_{port}, PT)$ *, where:*

- $CPN = (\Sigma, P, T, I, O, C, G, V, M_0)$ *is a model without hierarchical properties;*
- $T_{sub} \subseteq T$ *represents a set of substitution transitions;*
- $P_{port} \subseteq P$ *denotes a set of port places;*
- $PT : P_{port} \to \{In, Out, In/Out\}$ *is a port-type function that assigns a type to each port place.*

Definition 4 (HCPN)**.** *A **HCPN** is a 4-tuple:* $CPN_H = (S, SM, PS, FS)$ *where:*

- S *is a finite set of CPN modules. For each* $CPN_{module} = (CPN, T^s_{sub}, P^s_{port}, PT^s)$ *it must satisfy the requirement:* $(P^{s1} \cup T^{s1}) \cap (P^{s2} \cup T^{s2}) = \emptyset \quad \forall s_1, s_2 \in S$ *such that* $s_1 \neq s_2$*;*
- SM *is a submodule function* $T_{sub} \to S$ *that assigns a submodule to each* $T_{sub} \subseteq T$*, with the requirement that the module hierarchy is acyclic;*
- PS *is a port-socket function assigning a port-socket relation:* $PS(t) \subseteq P_{sock}(t) \times P^{SM(t)}_{port}$ *to each substitution transition* $t \in T_{sub}$*, satisfying:* $ST(t) = PT(p'), \quad C(p) = C(p'), \quad I(p)() = I(p')() \quad \forall (p,p') \in PS(t), \forall t \in T_{sub}$*;*
- $FS \subseteq 2^P$ *is a set of nonempty fusion sets such that:* $C(p) = C(p')$ *and* $I(p)() = I(p')() \quad \forall (p,p') \in fs, \forall fs \in FS$*.*

CPN Tools (Ratzer et al., 2003) is an environment for editing, simulating, and analyzing CPNs and HCPNs, with built-in state-space exploration and an ASK-CTL model checker.

3 SOFTWARE ARCHITECTURE AND ACCESS CONTROL IN WEB APPLICATIONS

AC stands out as the most critical vulnerability in the Open Web Application Security Project (OWASP) Top 10 - 2021 (Simplice et al., 2023).

Poorly enforced AC mechanisms expose software systems to unauthorized access. The architecture of a

web application defines how its components interact to maintain functionality. Typically, such applications comprise client browsers, web servers, and databases, which communicate to enable data exchange. However, as web applications become increasingly prevalent for remote access, they also face increased security threats. Exposing interactive features online increases the risk of unauthorized access, making AC a critical concern.

To illustrate these challenges, we consider a college library web application based on the OWASP framework (OWASP, 2024). This system allows students, staff, and librarians to search for books and manage resources. The potential vulnerabilities include the Common Weakness Enumeration "CWE-285: Improper Authorization". Authorization determines whether a user with a specific identity has the necessary privileges to access a resource, based on defined permissions and AC policies. When AC checks are not applied consistently - or not at all - users are able to access data or perform actions that they should not be allowed to perform. This can lead to a wide range of problems of improper authorization. Library resources may be inadvertently exposed due to misconfigured access permissions or system design flaws. The security of such a system hinges on effective AC enforcement. If vulnerabilities exist, attackers can bypass authentication, escalate privileges, or manipulate requests, leading to data breaches. If the system fails to validate user roles properly, a regular user may inadvertently gain access to librarian-only resources. In both cases, a misconfigured control leads to an unauthorized disclosure of resources, potentially compromising the system's security.

Let us consider a simple scenario: While regular users can log in to explore and request books (`getBooks`), librarians hold administrative privileges, including updating book records and managing user accounts (`getReqBooks`). A weak AC policy could allow an attacker posing as a student to execute `getReqBooks` — intended for librarians — by crafting a manipulated API request or exploiting session vulnerabilities. This could result in unauthorized access to the database, such as adding unauthorized users. To go to the essence of the problem, we assume that the software architecture consists of two user roles and defines four communication ports:

- *User Roles:* Two main user types are defined: `Admin` and `RegularUser`, each with specific privileges and access rights within the system.
- *Communication Ports:* The system regulates interactions and data exchange through ports connecting Browser, WebServer, and Database:
  - `B2WPort`: Browser toWebServer, handling user requests.
  - `W2BPort`: WebServer to Browser, delivering responses.
  - `W2DPort`: WebServer to Database, retrieving stored data for operations like `getBooks` and `getReqBooks`.
  - `D2WPort`: Database to WebServer, returning query results.

This simplified example serves to illustrate our formal approach.

4 VULNERABILITY DETECTION APPROACH

Our formal approach for vulnerability detection provides a process for identifying security weaknesses within software architectures. It begins with the manual construction of a HCPN model, which captures the system's structure. The hierarchical nature of HCPNs promotes modularity and scalability. Using CPN Tools, the model's state space is generated to explore all possible execution paths and interactions. Next, ASK-CTL properties are formally specified to express security requirements, enabling thorough verification of AC and other critical security properties. Once the specifications are defined, the verification process can be executed.

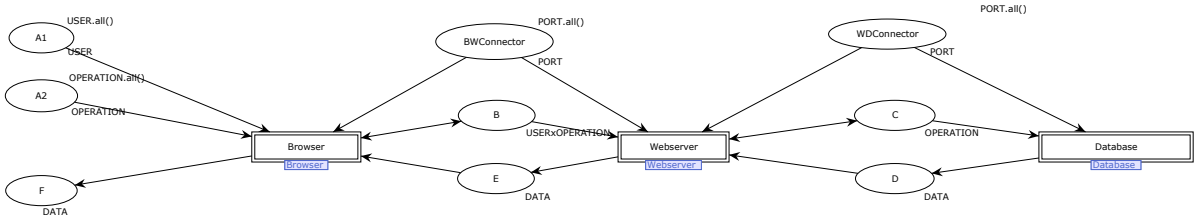
The following subsections detail each step of this approach.

4.1 HCPN Modeling

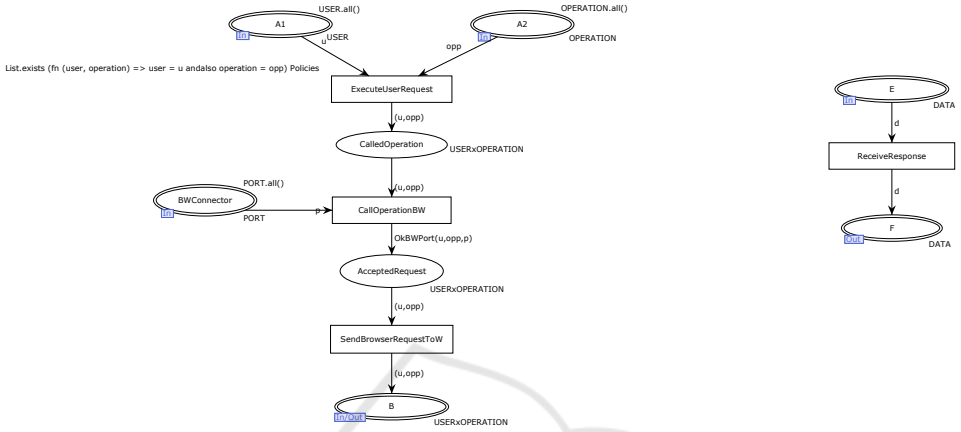
Based on the software architecture specified in an semi-formal language (e.g., Unified Modeling Language), we develop a formal model for each component using CPN and HCPN formal specifications given in Section 2. Notably, several studies have proposed approaches to transform UML diagrams into CPN and HCPN (Von Borstel et al., 2022), (Soares, 2017). Building on the simplified college library Web application introduced in Section 3, our complete model is manually constructed and organized into a modular structure with two levels of abstraction (Level 1 and Level 2).

Figure 1 depicts the overall software architecture CPN_H (a) (Level 1), and three components: $CPN_{Browser}$ (b), $CPN_{Webserver}$ (c), and $CPN_{Database}$ (d) (Level 2):

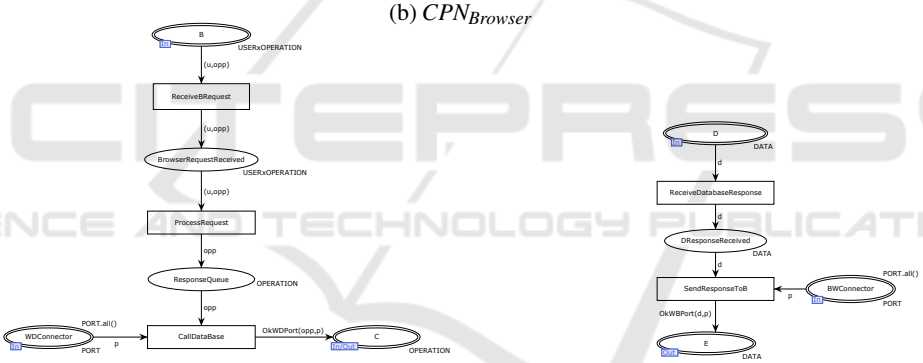
- Components are developed as a set of CPN modules, $s \in S$, represented by substitution transitions T_{sub} (rectangles in (a)) to which they are as-



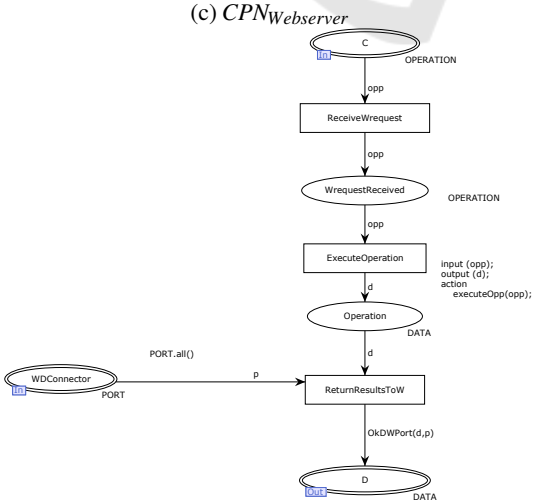
(a) CPN_H



(b) $CPN_{Browser}$



(c) $CPN_{Webserver}$



(d) $CPN_{Database}$

Figure 1: Software architecture for the college library system : CPN_H.

signed (SM, PS functions) and composed together to form CPN_H,

- Library states are represented by port places P_{port} (ovals in (a)) and tagged (PT function) with input and/or output, used for exchanging tokens (blue and small rectangle in (b-d)) between components,
- Exchanged message types are encoded in the color set of tokens Σ of the places,
- The initial marking represents the state of user requests and communication ports. Specifically, user requests are defined as tokenized pairs in the form `USER.all(), OPERATION.all()`, structured according to the `USER` and `OPERATION` color sets. This marking indicates that both the `Admin` and `RegularUser` roles have submitted requests for the `getBooks` and `getReqBooks` operations.
- CPN_H defines color sets Σ and variables V:

```
colset USER = with Admin | RegularUser;
var u : USER;
colset OPERATION = with getBooks | getReqBooks;
var opp : OPERATION;
colset USERxOPERATION = product USER *
OPERATION;
colset DATA = STRING;
var d: DATA;
colset PORT = with B2WPort | W2BPort | W2DPort
| D2WPort;
var p: PORT;
```

  As shown, `USER` color set categorizes users as either `Admin` or `RegularUser`, while `OPERATION` includes `getBooks` and `getReqBooks`. `DATA` is defined as a string type. Concerning `PORT`, it represents communication channels (`B2WPort`, `W2BPort`, `W2DPort`, `D2WPort`). Composite color sets such as `OPERATION×DATA` represent tuples combining these elements.

- CPN_H defines also different functions developed for regulating interactions :

```
fun executeOpp (opp: OPERATION) =
if opp = getReqBooks then "Allbooks"
    else "UserBooks";
fun OkBWPort (u: USER, opp: OPERATION, p: PORT)
= if p=B2WPort then 1`(u,opp)
else empty
fun OkWDPort(opp:OPERATION, p:PORT)
= if p=W2DPort then 1`opp
else empty
fun OkDWPort(d:DATA, p1:PORT)
= if p1=D2WPort then 1`d
else empty
fun OkWBPort (d: DATA, p: PORT)
= if p=W2BPort then 1`d
else empty
```

  As illustrated, `executeOpp` returns `"Allbooks"` for `getReqBooks` and `"UserBooks"` otherwise. Additional functions (`OkBWPort`, `OkWDPort`, `OkDWPort`, `OkWBPort`) validate data transmission through the corresponding communication ports.

- The AC policies are also given. They define which user roles can perform specific operations:

```
val basePolicies = [ (Admin, getReqBooks),
(Admin, getBooks), (RegularUser, getBooks)]
val Policies = if roleExtension then
basePolicies ++ [(RegularUser, getReqBooks)]
else basePolicies;
```

  Each policy pairs a role `u` with an operation `opp`. Verifying AC ensures that sensitive operations like `getReqBooks` are restricted to authorized roles, preventing unauthorized access and maintaining system security.

Parts (b–d) of Figure 1 detail how user requests are processed and how policies are enforced in the Browser (`ExecuteUserRequest` transition) through a guard function G:

```
List.exists (fn (user, operation) => user = u
andalso operation = opp) Policies
```

As the software architecture is being modeled, simulations are simultaneously conducted to analyze the behavior and identify potential inefficiencies. These simulations enable testing of various scenarios, optimizing performance, and ensuring seamless processing of user requests across the entities. By monitoring token flow and state transitions, we can iteratively refine CPN_H model. However, simulations alone are not sufficient to identify system vulnerabilities. While they help in analyzing functional correctness and performance, they do not guarantee security against potential threats such as unauthorized access.

To ensure security, verification techniques such as model checking can be employed to detect and mitigate vulnerabilities. For example, verifying the CWE-285 vulnerability involves checking whether a `RegularUser` is improperly allowed to perform the `getReqBooks` operation. By analyzing the system's behavior across all possible states (state space SS), we can confirm that unauthorized access is consistently prevented, or identify any potential misconfigurations.

4.2 State Space Generation

Figure 2 represents the state space of CPN_H model that will be used for analyzing the security of the software architecture (SS). Each node in SS corresponds to a unique system state or marking M, characterized by colored token distributions across places.

The directed edges between nodes indicate transitions fired by system events. The dense interconnections between states highlight the system's complexity by showing multiple execution paths and potential security vulnerabilities. This visualization enables formal verification through *SS* exploration, allowing the detection of security flaws, such as unauthorized access, by checking for unexpected transitions or unreachable states. To detect the CWE-285 vulnerability, we perform model-checking of the system against an ASK-CTL property that specifies the absence of this vulnerability. This property refers to a safety property that ensures a bad marking is unreachable from the initial marking, regardless of the execution path taken in the system's state space. A bad marking occurs when a token appears in an unintended or unexpected place.

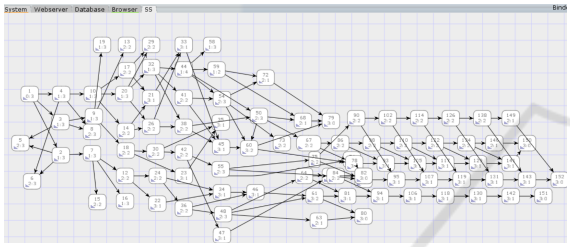


Figure 2: Generated state space: *SS*.

4.3 ASK-CTL Based Model Checking

Figure 3 evaluates the college library web application by verifying whether unauthorized users can execute restricted operations using Standard ML (SML) functions. The `isAllowed` function is defined to check user permissions based on predefined `Policies` while the `AC` function is used to simulate access attempts. If a user is authorized, the operation executes; otherwise, access is denied. The `allUserOpPairs` function generates all possible `(u,opp)` pairs and systematically tests them using `verify_vulnerability`, ensuring that unauthorized actions are correctly blocked. If an operation intended for librarians (e.g., `getReqBooks`) is executed by a regular user, the system identifies an AC flaw, highlighting potential privilege escalation risks. A 'true' result confirms that the operation can be executed in violation of the policies. A model checking procedure answers the following question: Given a state ASK-CTL formula and a CPN/HCPN, does the initial marking satisfy the formula? Different from Figure 3, which applies a predefined policy verification, the second SML listing given in Figure 4 applies ASK-CTL and model checking to formally verify security properties. Instead of relying on predefined rules, it defines logical formulas that express the AC security property and evaluates them over the system's state space.

```
val isAllowed = fn : USER * OPERATION -> bool
val AC = fn : USER * OPERATION -> string
val verify_vulnerability = fn : USER list * OPERATION list -> bool
val users = [RegularUser] : USER list
val operations = [getReqBooks] : OPERATION list
val result = true : bool

fun isAllowed (u: USER, opp: OPERATION) : bool =
    List.exists (fn (user, operation) => (user = u andalso operation = opp)) Policies

fun AC (u: USER, opp: OPERATION) : string =
    if isAllowed (u, opp) then
        "Access "
    else
        "No Access "

fun verify_vulnerability (users: USER list, ops: OPERATION list) : bool =
    List.all (fn (u, opp) =>
        if isAllowed(u, opp) then String.isPrefix "Access" (AC(u, opp))
        else String.isPrefix "No Access" (AC(u, opp))
    ) (allUserOpPairs(users, ops));


val users = [RegularUser];
val operations = [getReqBooks];

val result = verify_vulnerability(users, operations);
```

Figure 3: AC verification using SML.

```
val TokenInPlace = fn : Node -> bool
val A1 = NF ("Access",fn) : A
val myASKCTLformula = EXIST_UNTIL (TT,NF ("Access",fn)) : A
val result = true : bool
Vulnerability detected .

fun TokenInPlace n =(Mark.Webserver'BrowserRequestReceived 1 n =1`(RegularUser,getReqBooks));

val A1 = NF("Access", TokenInPlace);

val myASKCTLformula = POS(A1);

val result = eval_node myASKCTLformula InitNode;

val _ = if result then
print "Vulnerability detected .\n"
else       print "No vulnerability.\n";
```

Figure 4: Formal Verification Using ASK-CTL.

In ASK-CTL, path properties are described using two primary logical operators: `EXIST_UNTIL` and `FORALL_UNTIL`. These fundamental operators serve as the basis for deriving additional ones, such as `POS` and `EV`. For our example, we focus on `POS` operator:

$$\texttt{POS}(M) \equiv \texttt{EXIST\_UNTIL}(TT, M)$$

where `TT` denotes a truth value. This operator returns `true` if there exists at least one path from an initial state that leads to a state where `M` holds true. The listing uses the syntax `$Mark.<SubNet>'<Place>NM$` to retrieve tokens in place `<Place>` of the Nth instance of page `<SubNet>` within the marking M. It then applies an ASK-CTL formula with the `POS` operator to verify whether all paths avoid a given condition. A positive result indicates the presence of a vulnerability—specifically, a situation where an unauthorized user gains access to a resource before the required token becomes valid, ex. a `RegularUser` attempting to execute `getReqBooks` illustrates a potential exploit path that could lead to unauthorized access.

5 RELATED WORK

The rigorous formalization of software architecture models for verifying and validating system security

has become a central research topic (Kulik et al., 2022). Among the most effective approaches, model checking stands out for its ability to exhaustively analyze a system's state space to detect potential vulnerabilities. By applying first-order logic and temporal logic, such as CTL and Linear Temporal Logic (LTL), model checking can enable the automatic verification of whether certain security policies hold in a given system. For instance, in (Rouland et al., 2024), the authors introduce a first-order logic based method for specifying software architecture models, detecting vulnerabilities, and identifying associated anchor points for applying security controls. Unlike our approach, which utilizes ASK-CTL, their paper employs first-order logic to express structural constraints and the behavior of signatures forming an Alloy model. Different from Alloy, CPNs provide a graphical and token-based representation, making them particularly useful for analyzing concurrent and distributed systems. Also, HCPN model data flows between ports by explicitly representing information movement through tokens. They enable direct visualization of asynchronous and synchronous communication and support hierarchical structuring and message transformations, making them ideal for tracking data interactions in software architecture.

In the literature, recent studies reveal the promising use of CPNs/HCPNs for vulnerability detection. In (Wang et al., 2024), the authors introduce a new concept called vulnerability nets, designed to detect security vulnerabilities in source code. Specifically, this approach is based on CPN and helps to identify taint-style vulnerabilities, such as buffer overflows, injection flaws, and cross-site scripting issues. Vulnerability nets can perform automated analysis, although the analyst must assist in identifying sanitizations to minimize false positives. Also, in (Zhou et al., 2020), the authors present a CPN-based approach for modeling and analyzing attack tolerance in Web services, particularly in cloud environments where service continuity is critical. Their framework is abstract and generic and uses monitors in CPN Tools. It focuses on formalizing attack-network interactions and detection mechanisms in order to develop effective tolerance solutions. Recently, in (Al-Azzoni and Iqbal, 2024), a formal approach is introduced for verifying AC in smart contracts written in the Digital Asset Modeling Language (DAML), using CPN and CPN Tools. The approach is model-driven and employs a new meta-model to capture AC requirements within DAML contracts. It automates the entire verification process, from parsing the DAML code and generating model instances to transforming these models into CPN models and performing model checking. On the same issue of smart contracts, in (He et al., 2023), the authors introduce a formal approach to model and analyze smart contract security using CPN, with a focus on detecting re-entrancy bugs, a major vulnerability that has led to significant financial losses. This method employs HCPN modeling to represent smart contracts at the source code level and applies formal analysis techniques, including correlation matrices, state space reports, and state space graphs generated via CPN Tools, to identify potential security flaws. The research described in (Gowdanakatte et al., 2024) is more specific in its application area. It presents a holistic methodology for assessing asset criticality and system resiliency in Industrial Control Systems within the energy sector. It utilizes CPN modeling for formal verification and automated analysis of system behavior under cyber attacks. Applied to a wind farm system, this work enables state-based analysis, identifies vulnerable assets, and proposes mitigation strategies. Similar to the aforementioned contributions, our approach explores CPNs and HCPNs for comprehensive software architecture modeling. In (Tikhonov and Novikov, 2021), the authors use CPN and CPN Tools to dynamically model and verify AC systems, aiming to reduce vulnerabilities. They highlight ASK-CTL logic for improving security analysis by explicitly modeling actions and rules, a focus also explored in our work. In (Amthor and Rabe, 2019), the authors proposed a new heuristic approach to handle dynamic dependencies, which are common in complex models such as type enforcement mechanisms. They demonstrated the practical impact of this analysis problem and discussed the implications of their findings for the design and analysis of AC models.

6 CONCLUSIONS

In an era marked by increasingly complex software architectures, ensuring robust protection against cybersecurity threats remains a critical challenge in application design. This work has presented a formal framework for vulnerability detection based on HCPNs and model checking techniques. Situated within the broader domain of formal verification, the proposed approach demonstrates the relevance and effectiveness of HCPNs and model checking in performing rigorous security analyses, particularly in the context of web applications. We demonstrate how HCPNs offer a concise yet expressive graphical notation, making them accessible to both experts and non-specialists. Supported by mature tools such as CPN Tools, HCPNs enable efficient modeling, simu-

lation, and analysis of software architectures. Their formal semantics provide a strong foundation for rigorous verification of system properties, while their rich modeling primitives facilitate precise representation of complex security interactions. Additionally, we utilize ASK-CTL logic to formally express and verify security properties. Through this analysis, we identify several directions for future research. These include enhancing policy violation detection through counterexample generation and addressing the state space explosion problem via optimization techniques, aiming to support more scalable analyses. These enhancements will facilitate the integration of security design patterns into system architectures, promote automated vulnerability mitigation, and reinforce resilience against evolving cyber threats. By bridging formal verification with security-aware design, our approach not only identifies vulnerabilities but also supports the development of more robust systems.

REFERENCES

Al-Azzoni, I. and Iqbal, S. (2024). Access control verification in smart contracts using colored petri nets. *Computers*, 13(11):274.

Amthor, P. and Rabe, M. (2019). Command dependencies in heuristic safety analysis of access control models. In *International Symposium on Foundations and Practice of Security*, pages 207–224. Springer.

Gowdanakatte, S., Abdelgawad, M., and Ray, I. (2024). Assets criticality assessment of industrial control systems: A wind farm case study. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*, pages 352–363. IEEE.

He, Y., Dong, H., Wu, H., and Duan, Q. (2023). Formal analysis of reentrancy vulnerabilities in smart contract based on cpn. *Electronics*, 12(10):2152.

Jensen, K. and Kristensen, L. M. (2009). Formal definition of hierarchical coloured petri nets. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*, pages 127–149.

Jensen, K., Kristensen, L. M., and Wells, L. (2007). Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3):213–254.

Kulik, T., Dongol, B., Larsen, P. G., Macedo, H. D., Schneider, S., Tran-Jørgensen, P. W., and Woodcock, J. (2022). A survey of practical formal methods for security. *Formal aspects of computing*, 34(1):1–39.

OWASP (2024). Application threat modeling. https://owasp.org/www-community/Threat_Modeling_Process. Accessed: 2024-03-17.

Ratzer, A. V., Wells, L., Lassen, H. M., Laursen, M., Qvortrup, J. F., Stissing, M. S., Westergaard, M., Christensen, S., and Jensen, K. (2003). Cpn tools for editing, simulating, and analysing coloured petri nets. In *International conference on application and theory of petri nets*, pages 450–462. Springer.

Rouland, Q., Adi, K., Nguena Timo, O., and Logrippo, L. (2024). Detecting information disclosure vulnerability in software architectures using alloy. In *19th International Conference on Risks and Security of Internet and Systems (CRiSIS)*. Springer.

Simplice, I., Fidel, O., Kennedy, C. G., Okokpujie, K., and Gabriel, S. (2023). Enhancing information system security: A vulnerability assessment of a web application using owasp top 10 list. In *International conference on smart computing and cyber security: strategic foresight, security challenges and innovation*, pages 385–397. Springer.

Soares, J. A. C. (2017). Automatic model transformation from uml sequence diagrams to coloured petri nets. Master's thesis, Universidade do Porto (Portugal).

Tikhonov, V. and Novikov, V. (2021). Verification of access control systems based on modeling with colored petri nets. *High-Tech Technologies in Earth Space Research*, 13(6):50–59.

Von Borstel, F. D., Villa-Medina, J. F., and Gutiérrez, J. (2022). Development of mobile robots based on wireless robotic components using uml and hierarchical colored petri nets. *Journal of Intelligent & Robotic Systems*, 104(4):70.

Wang, P., Liu, S., Liu, A., and Jiang, W. (2024). Detecting security vulnerabilities with vulnerability nets. *Journal of Systems and Software*, 208:111902.

Zhong, W., Zhou, J.-t., and Sun, T. (2023). Concurrent software fine-coarse-grained automatic modelling by coloured petri nets for model checking. *IET Software*, 17(1):55–75.

Zhou, W., Dague, P., Liu, L., Ye, L., and Zaïdi, F. (2020). A coloured petri nets based attack tolerance framework. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 159–168. IEEE.