IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 9, 2021, accepted May 4, 2021, date of publication May 18, 2021, date of current version May 27, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3081587

Security Verification for Cyber-Physical Systems Using Model Checking

CHING-CHIEH CHAN[1,2], CHENG-ZEN YANG[1], (Member, IEEE), AND CHIN-FENG FAN[1]
[1]Department of Computer Science and Engineering, Yuan-Ze University, Taoyuan 320315, Taiwan
[2]Telecommunication Laboratories Chunghwa Telecom Company Ltd., Taoyuan 326402, Taiwan

Corresponding author: Ching-Chieh Chan (ccchan.francis@gmail.com)

ABSTRACT A malicious attack may endanger human life or pollute environment on a cyber-physical system (CPS). However, successfully attacking a CPS needs not only the knowledge of information technology (IT) but also the domain knowledge of the system's operation technology (OT). Therefore, it is critical to identify the vulnerabilities of a CPS. This paper proposes a systematic method for the security verification of a CPS, focusing on OT by using model checking with UPPAAL, so as to enhance cyber security. In our security analysis, we considered unsafe situations to be the result of a potentially effective security attack. Thus, we suggested a systematic method to generate security constraints based on the safety constraints (or safety checks) of the CPS and then enhance these security constraints by security verification using model checking with UPPAAL. UPPAAL's simulation tool can perform a detailed search for each state in various possible model combinations and can explore human-computer interactions more deeply. The contributions of our method are as follows: First, a systematic method is proposed to generate security constraints based on the overall safety requirements at the OT level. Second, the security constraints thus generated can be used for run-time monitoring to identify the possible security attacks when they are violated. Third, this paper proposes to augment normal system modeling with a suggested Attack Module to simulate the potential OT attacks. Finally, the verification results may be used in the following twofold directions: to identify the vulnerabilities for possible design improvements and to suggest the further additions of security constraints.

INDEX TERMS Cyber-physical systems, model checking, operation technology, security constraint, security verification, UPPAAL.

I. INTRODUCTION

A cyber-physical system (CPS) is a system designed and constructed based on integrated computing and physical components [1]. CPS has been widely used in aviation, construction, agriculture, energy, environment, manufacturing, and transportation, including industrial control systems (ICS), the Internet of things (IoT), and critical information infrastructures. Most CPSs are safety-critical, and thus, malicious attacks on CPSs may have destructive consequences. In recent years, there have been several incidents on critical infrastructures through coordinated cyber and physical attacks (CCPA) [2], [3], threatening national security. For example, Stuxnet infected and destroyed the nuclear facilities in Natanz, Iran [4], and Ukraine's power network was attacked [5], leading to blackouts for hundreds of thousands of households in the Ivano-Frankivsk region.

Malicious attacks against, particularly, ICS may cause devastating losses, such as in the medical, financial, and transportation fields. The security threats of CPS systems come from a variety of possible sources [6], [7], such as malicious intruders, natural disasters, human errors or accidents, and equipment failure. The first security requirement of a CPS is to avoid attacks [8]. Therefore, the analysis of the weakness of CPSs in the operation technology (OT) phase is a critical issue. We believe that the integration of safety and security will help the CPS avoid major hazards, particularly in OT integration [9].

In our security analysis, we treated unsafe situations as the results of potentially effective security attacks. Thus, we suggest a systematic method to generate security constraints based on the safety constraints (or safety checks) of the CPS and then enhance these security constraints by security

The associate editor coordinating the review of this manuscript and approving it for publication was Cong Pu.

verification using model checking with UPPAAL. UPPAAL's simulation tool can perform a detailed search for each state in various possible model combinations and can explore human-computer interactions more deeply. Simulation using the UPPAAL model checker was used in our security verification. For the simulation, we augmented a normal system model with our Attack Module for generating the potential OT attacks. The CPS model includes software, hardware, and the operator subsystems. To sabotage a CPS physical system, not only the knowledge of information technology but also the domain knowledge of the system's operation technology is needed. However, current research rarely discusses the systematic generation of security constraints and seldom focuses on OT-level attacks on CPSs. Unlike such research, this study systematically generated a set of OT-level security constraints and utilized the formal verification of these constraints using model checking. The contributions of this method are as follows:

1) A systematic method is proposed to generate security constraints at the OT level.
2) The security constraints thus generated can be used for run-time monitoring to identify the possible security attacks when they are violated.
3) This paper proposes to augment a normal system model with a proposed Attack Module in order to simulate the potential OT attacks.
4) The verification results may be used in the following twofold directions:
    a) to identify vulnerabilities for possible design improvements, and
    b) to suggest the further additions of security constraints.

The rest of this paper is organized as follows: Section 2 introduces the background and related research, and Section 3 introduces our approach. In Section 4, two case studies are described, each including the generation of the safety restraint system, model construction, safety verification, and possible improvements. Finally, Section 5 presents the conclusions.

II. BACKGROUND AND RELATED WORK

A. CPS SECURITY

According to a survey by Positive Technologies in 2019 [10], most of the vulnerabilities in a CPS were in the human-machine interface (HMI)/supervisor control and data acquisition (SCADA) components in 2017, but in 2018, the CPS system focused on ICSs, and the vulnerabilities in HMI/SCADA were found. From the perspective of the CPS security incidents in the recent years, both within-the-network and dedicated facilities are likely to be attacked, and a variety of attack methods and technologies are used. Another report released by TechRepublic in 2020 pointed out that three-quarters of the 365 vulnerabilities in ICS systems discovered by operational technology (OT) security companies can be classified as high risk or serious risks [11]. Therefore, CPS security protection has become increasingly important.

There are many tools for model verification [12]. For example, from the perspective of modeling formalism, SpaceEx provides a network of hybrid automata, UPPAAL provides a network of time automata, and KeYmaera provides a hybrid program, but we are particularly concerned about the verification of the correlation between the possible running paths of the system and the state changes. Therefore, a subset of timed computation tree logic provided by UPPAAL was used for the verification.

B. UPPAAL MODEL CHECKING

We wanted to explore the various possible interactive operating conditions of the system state as much as possible; therefore, we adopted UPAAL, as UPPAAL's simulation tool can perform a detailed search for each state in various possible model combinations. By exploring the various possible combination states between the system models, we could explore the interaction between humans and machine states in more depth. UPPAAL can also support exhaustive state exploration/examination to deal with complex CPS applications.

UPPAAL is a formal model checking integration tool environment [13], which can be used for the development and verification of software and hardware systems, and has also been used for security reasoning [14], [15]. UPPAAL was developed by the Department of Information Technology of the University of Uppsala, Sweden, and the Department of Computer Science of the University of Aalborg, Denmark; it can be used for the modeling and verification of real-time systems. UPPAAL's model checker is based on the timing automata theory. Its modeling language includes the related functions such as bounded integer variables and provides a query language for system verification. Basically, a model must be provided when using a model checking tool. When used, the model can be designed to include the security requirements and design of the system. Then, the user adds the required specifications of the system, and finally, the model checker checks through the model to determine whether it can meet the given specifications. If not, the model checking tool can provide counter-examples.

The UPPAAL integration tool mainly provides three major functions: system editor, simulator, and verifier. The system editor is used to edit and construct modules. An UPPAAL module is composed of several components. The contents of each component have their own operating state and transitions between states. After the model is built, the simulator can be used to explore the model state space. The simulator provides a step-by-step tracking or a random execution, and it can retrospect each execution process. In the verifier, we can give a query to check the model we built. It may or may not be possible to hold the query specified conditions. The query is represented in temporal logic. The query notation is taken from the temporal logic field. For example, the notation "A[]" at the beginning means "this is true in all reachable

states," and the notation "E<>" at the beginning means "it is possible to reach a state." The query "A[] not deadlock" indicates that there is no "deadlock" for the state in all the reachable states of the system. Here, "deadlock" means that no outgoing transition is possible.

C. ACCIDENTS INVOLVING HUMAN-MACHINE INTERACTIONS

It can be found that some of the accidents were caused by human-machine interface errors by observing several disaster events in the past. The causes of these accidents revealed that the problem may be due to operation errors, false instructions and the confusion of the system mode. For example: the Northeast blackout of 2003 [16] occurred because the administrator did not know that the power plant was faulty, which caused the voltage to become very low and some circuits became overloaded and tripped; thus, proper scheduling was not possible. In New York City alone, the estimated cost of power outages is more than $500 million [17], and the cause of the power outage was incorrect instructions. In another example, the 2015 crash of TransAsia Airways Flight 235, the flight safety investigation committee's investigation report pointed out that the crew members did not follow the standard procedures in the implementation manual to identify the fault and did not follow the correct procedures to make corrections, which eventually caused the plane to crash [18]. Therefore, the crash could be attributed to operation error. In the case of the crash of China Airlines Flight 676 in 1998, the investigation report showed that the pilot accidentally detached from the autopilot of the aircraft but did not realize it. The pilot performed a go-around but did not know that the autopilot was disengaged. Therefore, he did not take control of the aircraft because he thought the autopilot would start the maneuver. This resulted in the aircraft being unmanned for up to 11 s [19]. This is a case of the confusion of the system mode. From the above examples, we can infer that there are some potential problems with the system and the human-machine interface. If these problems can be detected or prevented in advance, major hazards can be avoided or reduced.

D. RELATED WORK

The CPS becomes increasingly important in various critical information infrastructures such as energy, transportation, military, healthcare, and manufacturing. At present, the CPS has been applied to many key infrastructures, such as smart grids, industrial control systems, wearable or implanted medical devices, or small control system networks embedded in modern cars, and human interaction with the CPS is getting higher and higher and more frequent. Although there is no consistent definition of CPS that can be widely recognized, CPS is used in the physical world of supervisors and control [7]. From the abstract CPS architecture, we can classify CPS into three main components, namely communication, computing and control, and monitoring and manipulation. The communication may be wireless or wired to provide the connection between the various components of the CPS, the computing and control part can receive the sent control commands and receive the results detected by the physical world, and monitoring and manipulation can maintain and control the physical world through a variety of sensor equipment. However, the interaction between the various components of the CPS may cause different security concerns. For example, the communication and computing functions of the CPS components may be exploited to cause unexpected attacks. Similarly, the physical world may cause non-physical attacks in addition to CPS control and monitoring, such as sending misleading information to the CPS. Regarding the security verification of CPSs, many scholars have conducted research related to information security. The part of security verification can be divided into the security of the physical system, the cyber system, and the cyber-physical system interaction. However, most of the verification methods focus on IT-related technologies, and only a few, such as Akella's work [20], focus on both cyber and physical systems. We made relevant comparisons between our method and Akella's [20]. Both provide modeling methods and adopt formal verification for the security of CPS. The results are shown in Table 1. In [20], the authors took the information flow as the basic concept of CPS security, provided a formal modeling technology for the information flow, transformed it using mathematical models, and automatically analyzed it through process algebra specifications.

TABLE 1. Relevant comparisons between two approaches.

Approach	Concept	Verification	Tool	Check
R. Akella [20]	Information flow security	Information flow models provide reasoning	CoPS	Security process algebra
Our	Operation technology attacks, cyber-physical constraints	Modeling checking merits: system completeness and input coverage	UPPAAL	Temporal logic

In addition, there has been related research on CPS intrusion detection. The detection methods include host-based intrusion detection, which mainly monitors parts or activities of the host, network-based intrusion detection, vulnerability assessment-based intrusion detection including abuse/signature intrusion detection and anomaly-based intrusion detection, status protocol analysis, and process failure variables [21]–[23]. Most of the detection technologies analyze the network traffic from different perspectives to detect intrusions. Anomaly detection has been applied to the various layers of the system, such as physical, MAC, routing, and application. For the application of the physical layer, most of the current research uses the received signal strength indicator (RSSI) as a distance-based indicator, the power perceived by a given node on the network, and tries various

solutions in an attempt to mitigate the physical layer attacks in the wireless network based on the power perceived by a given node [24]. There are many studies on cyberattacks by using mathematics to model CPSs as linear systems to address various cyberattack issues [25], mainly from control engineering viewpoints. Yet, most of them are at a quite abstract level. They can be used to guide further development of pragmatic cybersecurity techniques. The idea of our study is the establishment of different perception points in the infrastructure to improve the run-time attack monitoring. The process invariants approach by Adepu and Mathur [23] is a method that converts the system security requirements into invariants and uses them to detect intrusions. At present, many scholars have proposed the use of invariants to conduct intrusions. Research on intrusion detection has been proven to be effective in detecting attacks [23], [26]–[28]. An invariant is an expression [29] in which the value of a program in a certain state does not change during the execution [22]. The design of process invariant detection does not consider attack methods. Therefore, it is not related to attack methods [23]. However, these invariants check mainly for informational technology attacks, not operational technology attacks.

In recent years, a few scholars have used invariants on distributed anomaly detectors as the detection of deviations from the expected behavior of the system, such as Sugumar's work [30]. Sugumar's work is mainly focused on testing the anomaly detectors deployed in the critical infrastructure, using timed automaton for the infrastructure under consideration, and simulating and modeling on UPPAAL for distributed anomaly detectors and possible mutations. The model integrates the tested anomaly detector with the normal model to simulate and establish a set of reference attacks. Then according to the state and command mutation operators for each reference attack, additional attacks are generated by changing the reference attack model. Each additional attack will be executed during simulated attacks. The detection results of the tested anomaly detectors will be evaluated and checked for deficiencies, and repeated tests and modifications will be made. However, we believe that attacks of the OT level may be complicated. Attack scenarios may also include a combination of various events involving information and physical attacks, and will not be limited to–state and command mutations. Moreover, understanding the safety of the physical system of CPSs is the first step to conduct cybersecurity analysis and verification. Therefore, based on our previous research [31], we generate security constraints based on system safety, and provide run-time monitoring to identify attacks. Our attack scenario generation uses an attack-tree based approach as follows: we design an attack tree template for possible OT-level attacks of CPS, and produce various possible attack scenarios by using–our general template of the attack tree. Compared to Sugumar's mutation-based approach, our attack-tree-based approach can generate more complicated attack scenarios.

Current research rarely pays attention to OT malicious attacks [23], [32]. This study, in contrast, focused mainly on the OT attacks on CPSs, because OT attacks usually target physical systems and may cause further severe consequences. If the system executes against CPS safety constraints, it may be a physical attack. In other words, the violation of safety constraints can be a result of malicious attacks. Therefore, we can generate the possible security attack scenarios from a sequence of unsafe events. This study systematically generated the system's security constraints based on the safety constraints and assertions and used these security constraints to detect the possible attack scenarios. Further, consider that OT attacks may cause different levels of disasters because of the different system states on the CPS. To provide the possibility of thoroughly exploring the state space and to find the potential problems that may not be discovered by the simulation itself, a combination of possible attacks and HMI was used in the model checking to explore the human-machine interactions more deeply. The study was based on our previous classification of safety assertions [31], [33] and systematically generated a set of security constraints for run-time monitoring, followed by the related security verification using model checking, assuming that the violation of these safety constraints was caused by malicious attacks. In summary, the contribution of this work is to develop a constraint-based model checking approach to CPS security design, verification, and run-time monitoring. The distinctive feature of our model checking is its focus on OT attacks, and thus, our approach can effectively enhance system security as compared to most CPS security designs and verification methods [34], [35].

III. OUR APPROACH

The framework of the proposed security verification approach has two main parts, namely the systematic security constraint generation based on a safety analysis and the security constraint verification. The main concept is to use the results of the systematic security constraint generation to perform a security analysis and then use model checking for the security verification. This study mainly focuses on the modeling method and verification framework of the security constraints. The framework of the security verification approach is shown in Fig. 1.

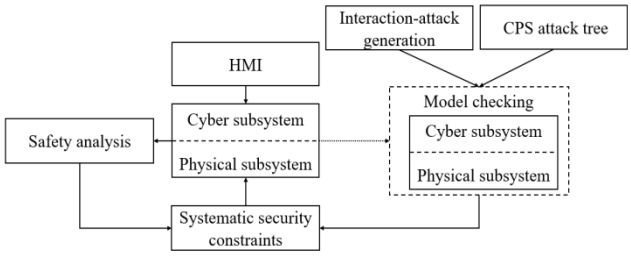


FIGURE 1. Framework of the security verification approach.

The process of the security verification approach is shown in Fig. 2. First, the security constraint generation step considers both safety and security. The relevant generation method

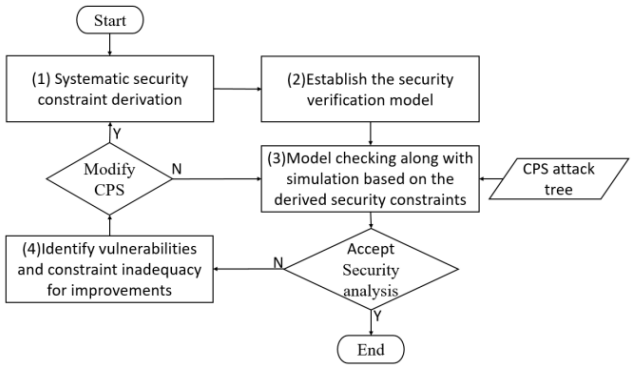


FIGURE 2. Process of the security verification approach.

will be in accordance with the safety constraint generation criteria that we previously proposed [31], [33]. According to the guidelines, the safety requirements of the CPS system will be used as the basis to initially generate the security constraints that are consistent with the safety requirements. Second, a behavioral model of the CPS can be established. Then, model checking is used to simulate the system and possible attacks so as to improve the system design or enhance our security constraints, and it can provide the possibility of thoroughly exploring the state space and exploring the human-machine interactions more deeply. Model checking further uses the above security constraints for security verification and modeling. It provides the possibility of discovering the potential problems not covered by the simulation. This can enhance the verification and simulation of security constraints. The method also adds the ability to generate various simulated scenarios that may be attacked in the modeling. This ability allows the model to be used to generate various simulated scenarios that may be attacked based on the CPS attack tree, and at the same time, can be used to simulate the security constraints of the system verification. If any problem is found, this method can be used to make improvements based on the problem and to verify compliance with the security constraints.

A. SYSTEMATIC SECURITY CONSTRAINT GENERATION

In this study, we assumed that the violation of CPS safety constraints was caused by the possible OT attacks. Thus, our safety constraints [31], [33] could easily be treated as the security constraints. They could be applied to the first step of the proposed method. These constraints could be used for run-time security monitoring, and their violation indicated possible security attacks.

Step 1: Systematic security constraint generation

The proposed safety/security constraints were as follows:

1) Value and range constraints: They check whether the values of the device states and the process variables (such as water level and pressure) in the software control are legal or within the legal ranges.
2) Dependency between device states and process variables: This refers to the temporal dependency, shown in the software control, between a device status and its affected process variables. For example, if the pump is turned on, the water level will increase.
3) Cyber and physical consistency constraints: They check whether the software variable in the software control or its displayed value is consistent with its represented physical entity. On-site human checking is needed for such constraints. For example, the software or the displayed water level value is equal to its physical reality.
4) Global invariants: They check the invariance relationship, for example, energy and mass conservation laws, physical restrictions, and material restrictions. They check whether such system invariants as energy conservation and mass conservation laws still hold in the control software. For example, the current amount of water is the initial amount minus the leaked amount. Energy conservation rules should be followed and quality conservation should be maintained in various controls.

Regarding the generation of security constraints, we propose that they be produced during the system design phase by using the following procedure:

1) When the cyber model design is completed:
   a) Consider adding constraints on the value and the range of the logical variables representing the physical sensors, actuators, equipment, and processes.
   b) Consider adding a dependency relationship between the values of the logical variables representing the device states and the process variables.
   c) Consider adding logical variables that represent the software control or a displayed value, which corresponds to the consistency constraint between the HMI display and the physical model.
2) When the physical model design is completed:
   a) Consider adding a dependency relationship between the value of the logical variable representing the physical sensor or device and the cyber model.
   b) Consider adding logical variables that represent the physical model, which corresponds to the consistency constraint on the cyber model.
3) When the HMI design is completed: Consider adding the software display and physical counterpart consistency constraints.
4) Considering the global variables of the system: Check the invariance relationship of the cyber, physical, and HMI, respectively.

B. MODELING PROCESS

In Steps 2–4 of the proposed approach, model checking is mainly used for the CPS security constraint verification. We assumed that the violation of security constraints was triggered by OT attacks. In the model checking step, normal

model templates and various possible attack model templates using UPPAAL are proposed. We used various combinations of the proposed model templates in the simulation, and the results of our model checking were used to improve the adequacy of the security constraint set. The advantage of using UPPAAL's simulation tools was that an exhaustive search could be achieved for each state in various possible model combinations and the human-machine interactions could be explored more deeply. If problems were found, the exemplified information provided by UPPAAL could be used to improve the security constraints. Steps 2–4 of the proposed approach are as follows:

Step 2: Establish the security verification model

Model the normal operation of the system and provide the required information for the abovementioned security constraints. Moreover, this method augments the system model by the HMI, the proposed Attack Module, and the suggested Simulation Module. The Attack Module provides the desired modification of the normal system model by changing the input or output information of each module, assuming malicious attacks on hardware devices, software control, or communication causing the false information. It also supports the Normal Module in making internal adjustments, including replacing the software control modules and simulating the implanting of malware, on the basis of a preselected attack scenario. The Simulation Module is mainly a selection module for controlling the simulation sequence of the model module or the simulation case. The details will be described in Sec. 3.3.2

Step 3: Perform model checking along with the simulation based on the generated security constraints

The generated security constraints are assumed to be used for the run-time monitoring to detect the potential security attacks. Therefore, the security verification is performed by using the abovementioned security constraints as a query and by running model checking. This step is further divided into the following sub-steps:

1) Generate security verification queries based on the abovementioned security constraints for model checking.
2) Run the model checker for a certain attack scenario, which is simulated by modifying the normal system behavior using the Attack Module and Simulation Module. For the design of the attack scenario, this study used the fault tree analysis template mentioned in [33], involving a combination and classification of cyber, physical, and human attacks.

Step 4: Identify vulnerabilities and constraint inadequacy for the improvements

The model checking result can be used in the following two directions:

1) To identify vulnerabilities for further design improvement.
2) To identify constraint inadequacy and suggest further constraint improvement. Design improvement can be such measures such as adding sensors and using hardware interlocks.

Whether the proposed constraint set is complete is a difficult issue. However, the improvement of the proposed constraint set is feasible. If an attack scenario cannot be detected by the current constraints in the simulation, additional constraints may be generated to expand the constraint set.

1) UPPAAL MODELING

A CPS can be modeled by the cyber subsystem, the physical subsystem, and the human-machine interface, as shown in Fig. 3.

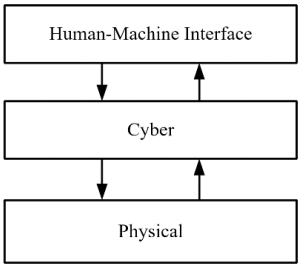


FIGURE 3. CPS system overview.

To model and simulate various attack scenarios, we propose adding two more modules to the Normal model, namely the Attack Module and the Simulation Module. Thus, a CPS is modelled in three modules in UPPAAL: The Attack Module, the Normal Module, and the Simulation Module. The Normal Module models the normal operation of the CPS, including the HMI, cyber, and physical parts. The Attack Module is used for simulating communication, software, and hardware sabotages in a selected attack scenario, such as logic errors, value errors, data transmission errors (input, output, or displayed data errors), device failures, and electromagnetic disturbance interference. The Simulation Module can be optional and be used to specify the process priority and select a simulation case, when the demand progress is set at the beginning or when there is a demand-specific execution order. The relationship among the UPPAAL modules is shown in Fig. 4.

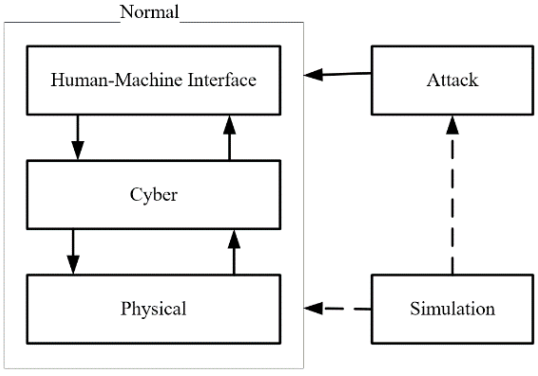


FIGURE 4. Proposed UPPAAL model.

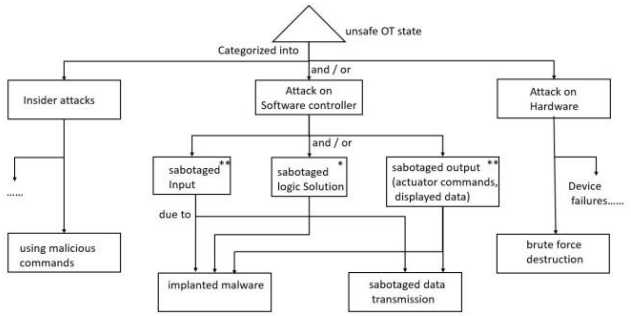


FIGURE 5. CPS attack tree templates.

2) ATTACK MODULE

The Attack Module can change the input and the output in the Normal Module, assuming malicious attacks on hardware devices, software control, or communication causing the false information, on the basis of the attack scenario under examination. In this work, the possible attack scenarios were first generated systematically according to the proposed template shown in Fig. 5. To begin with, an unsafe operational technology state, involving a risky combination of device and process states, was identified. For example, an OT state might include a faulty sensor along with a hacked display. As another example, an OT state involved a sabotaged control. Then, the lower parts of the attack tree template could be used to generate the possible causes and the orders of these faults. We categorized the possible attack causes into three categories: insider attack, software control attack, and hardware attack, as shown in Fig. 5. The possible attack events could be classified as follows [33]:

1) Insider attack event: The operator or insider attack may be brute force physical destruction or the use of malicious commands.
2) Software control attack events: Software control attacks can be further divided into inputs that are changed by malware, outputs that are changed by malware, and software logic solutions that have been tampered with by viruses.
3) Hardware attack event: A hardware attack may be an unsafe measure taken by a damaged physical device.

Among them, the software controller attack was our focus. Attacks on the software control could be further divided into input changed by malware, output changed by malware, and software logic scheme (logic solution) tampered with by the virus, that is, replaced by the implanted malware, as shown in the starred box in Fig. 5. The CPS attack tree templates of various types are shown in Fig. 5; they were used as a basis to generate various attack scenarios as the inputs to execute the model checker UPPAAL in this study. If software control sabotage due to implemented malware (the starred box in Fig. 5) was desired, we augmented the original normal model with an alternative malware Attack Model simulating the malware behavior (logic) in UPPAAL. If the input/output transmission attack (the double starred box in Fig. 5) was desired, the Attack Module was set to change any input or output among the components of the normal system model. All the attacks could be designed in advance, and the method of increasing attacks could be added to UPPAAL through the attack model. Moreover, system operations could be added when the attack had to be evaluated and simulated.

3) GENERATE OT-LEVEL ATTACK SCENARIOS BASED ON THE ABOVE ATTACK-TREE TEMPLATE

Considering the systematic generation of the potential attack scenarios at the OT level, we propose the following procedure at the stage of constructing the verification model:

1) Determine the OT attack goal and the associated final states: Situations causing serious consequences, and their associated physical and/or logical final states are first identified. For example, the OT attack goal is to sabotage the heat controller to cause serious physical damage such as explosion or fire, and the associated final state is the erroneous controller and misleading display.
2) Explore the feasible causal scenarios leading to the abovementioned final state by a systematic application of the attack tree template (Fig. 5) by the analyst. For example, in the above example, the attack tree template guides the analyst through the possible attack paths, namely input attacks, output attacks, and logical attacks, to form the various possible scenarios. For instance, an attack scenario can consist of an input data attack changing the heating behavior and an output attack of the display to mislead the operator; another attack scenario may involve a sabotaged logic solution so that both the heat behavior and the display output are wrong; the third attack scenario may be formed by a logic attack to change the heating behavior along with a hardware attack on the display to deceive the operator; and so on.
3) Set the attack scenario by first constructing a normal operation model and then creating the required attack modules on the basis of the selected scenario, and combine them into an attack scenario. The principles of creating Attack modules are as follows:
    a) A unique identifier is set for each attack in each case, and the identifier is used in the model to control whether to activate the attack.
    b) If the Normal Model needs to be modified (for example, to simulate a logic attack). The Attack Module must be integrated into the design in the normal template.
    c) Consider the template designed for each attack, and limit it to only being enabled when the attack is launched.
    d) Set all the attack activation settings in the Simulation Module.

We will show detailed implementation examples in Sec. 3.3.2.

C. SECURITY VERIFICATION APPROACH WITH AN EXAMPLE

To better explain our approach, the following simple example is used. Take a simple automatic injection system as an example. The system includes a water level sensor, an automatic controller, and a water level display. Its behavior is described as follows:

1) The level of the water tank must be less than or equal to 34 units.
2) The level of the water tank will automatically lose 1 unit of water each time.
3) The automatic controller will automatically start the injection when the level of the water tank is less than 34 units and inject 1 unit of water each time.
4) The system provides a level indicator to display the current level of the water tank.

1) SYSTEMATIC SECURITY CONSTRAINT GENERATION

The first step is to systematically generate the security constraints for this example on the basis of the proposed categories:

4) Value and range constraints: They check the working range in which the processing program variables and process variables are valid, such as the normal working ranges. In this example, the level of the water tank must be less than or equal to 34 units.
5) Dependency between device states and process variables: In this example, the automatic controller will automatically start the water injection when the level of the water tank is less than 34 units and inject 1 unit of water each time.
6) Cyber and physical consistency: In this case, the actual water level in the physical system is equal to the value of the water level variable in the automatic controller and equal to the value indicated by the water level sensor. The automatic controller should also be consistent with the value displayed on the HMI.
7) Global invariants: In this case, the constant energy conservation is the relationship between the water level rise and the automatic controller.

2) ESTABLISH THE SECURITY VERIFICATION MODEL

First, construct a normal model, including an HMI in the UPPAAL model. For example, as shown in Fig. 6, the physical model, the cyber model, and the HMI model are displayed in sequence. Use the channels in UPPAAL to communicate between the UPAAL models and use variables for the data exchange. For example, the cyber model and the physical model in Fig. 6 will be synchronously triggered through the channel pumpon, and the variable Tklevel is used for the water level data exchange. Channel disp will synchronously trigger the transfer of the water level data from the cyber model to the HMI module. Channel tanklevel will synchronously trigger the transfer of the water level data from the physical model to the cyber module.

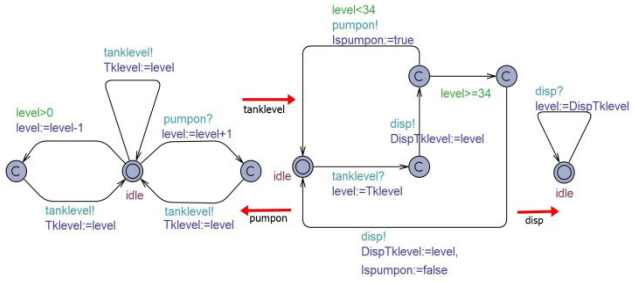


FIGURE 6. Communication and data exchange between UPAAL models.

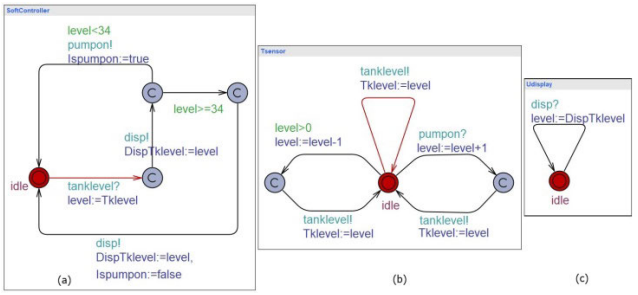


FIGURE 7. The normal module in UPPAAL.

The automatic control representing the cyber module is named SoftController, as shown in Fig. 7(a). Its main function is to receive the information sent by the water level sensor and control the automatic water injection when the level of the water tank is lower than 34 units; the injection will stop when the water level is higher than or equal to 34 units. At the same time, it notifies the HMI module to display the current water level. The temperature sensor used to represent the physical module is named Tsensor, as shown in Fig. 7(b). Its main function is to sense the degree of the physical water level, transmit the current water level to the automatic control, and sense the water level loss. Furthermore, the HMI module used to display the current water level obtained through the automatic control is named Udisplay, as shown in Fig. 7(c).

Attack scenarios are then generated on the basis of the abovementioned CPS attack tree templates (Fig. 5). The hypothetical simple attack belongs to the input/output transmission attack, and the target is to make the water level of the water tank exceed 34 units. To achieve this goal, we will reduce the actual value sent by the water level sensor to the software controller by 1 unit, which will cause the actual level of the water tank to exceed 34 units. To reduce the input water level of 1 unit to SoftController, first, add a malicious attack module named SoftCtrlAtkIn. SoftCtrlAtkIn will be added between SoftController and Tsensor to affect the input of SoftController, as shown in Fig. 8. Then, the communication channel (tanklevel) of SoftController is changed to receive the communication channel (tanklevelAtk) of SoftCtrlAtkIn, and the updated SoftController is named SoftControllerAtk, as shown in Fig. 9. Finally, use the Simulation Module to control two attack scenarios as shown in Fig. 10. The Normal

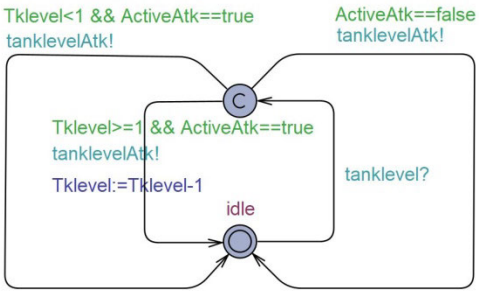


FIGURE 8. Malware input module.

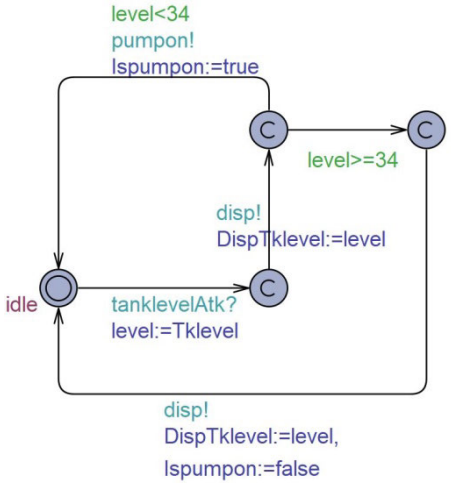


FIGURE 9. Updated SoftController.

FIGURE 10. The Simulation Module.

model, Attack Module and Simulation Module are established in UPPAAL, as shown in Fig. 11.

Next, we will continue the above example to demonstrate how to build a model with dynamically generated attacks. First of all, we will apply the first item of the previously mentioned design principle, which is to set a unique identification value for each single attack. According to the above example, we will set a unique identification name for this input attack as CIn[0] and preset the value of the variable CIn[0] to false. Secondly, we will apply the second item of the design principle to integrate the modified change with the original software control design in the same template, as shown in Fig. 12. In Fig. 12, we combine the normal template from Fig. 7 and Fig. 9, which is the modified software control, into a UPPAAL template and set it to execute different programs according to the different execution function requirements. Applying the third item of the design principle,

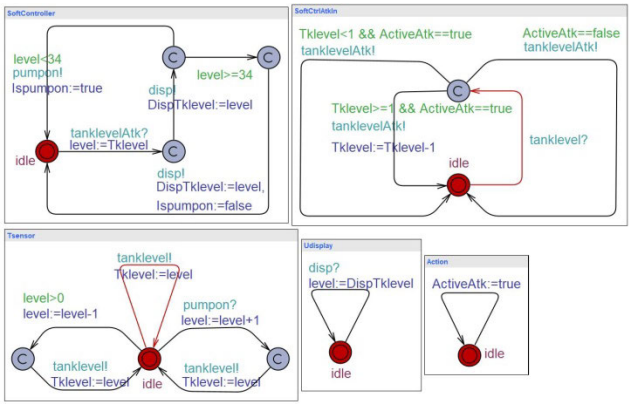


FIGURE 11. Security verification model in UPPAAL.

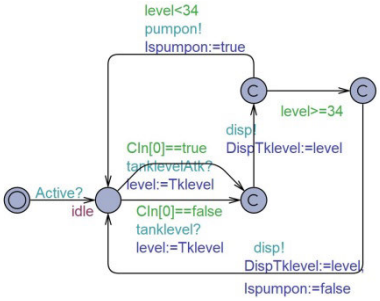


FIGURE 12. Integrating the modified change with the original software control design.

we modify the previous example of the malware input module (Fig. 8) to restrict the attack to be activated only when the attack is launched, as shown in Fig. 13.

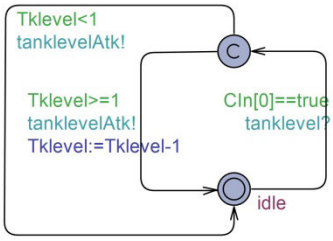


FIGURE 13. Restricting the attack to be activated in the malware input module.

We will add another input attack to the display part, which will increase the input attack template between the software controller and the HMI, and directly apply the abovementioned three principles. We set a unique identification name for this input attack as CIn[1] and preset the value of CIn[1] to false. The result of integrating the display part of the normal template in the same UPPAAL template is shown in Fig. 14. Restricting the input attack template will only be activated when the display attack is enabled, as shown in Fig. 15. Combine the abovementioned two attacks and apply the fourth item of the design principle to set all the attack activation settings in the Simulation Module, as shown

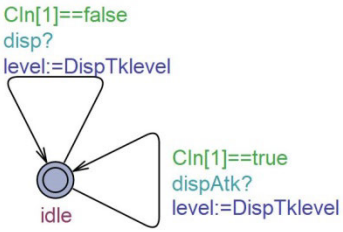


FIGURE 14. Integrating the display part of the normal template.

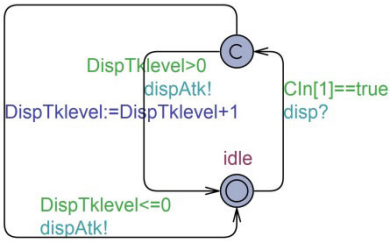


FIGURE 15. Restricting the input attack template will only be activated when the display attack is enabled.

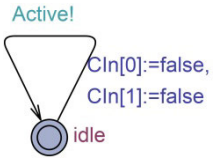


FIGURE 16. All the attack activation settings in the Simulation Module.

in Fig. 16. After completion, we can directly set which attacks will be combined to enable operation in the Simulation Module.

3) MODEL CHECKING ALONG WITH SIMULATION BASED ON THE GENERATED SECURITY CONSTRAINTS

This step is conducted to perform model checking based on the above-proposed security constraints. The security constraints for this case are given below:

1) Value and range constraints: For example, the level of the water tank must be less than or equal to 34 units. (A[] (Tsensor.level <= 34)).
2) Dependency between device states and process variables: When the level of the water tank is lower than 34 units, the controller will automatically start filling water. (((Tsensor.level < 34) && (Tsensor.level > 0) && (SoftController.idle)) --> (SoftController.Ispumpon == true)).
3) Cyber and physical consistency: For example, the level of the water detection sensor and the display must be consistent with the controller water level. (A[] (SoftController.level == Tsensor.level) imply true), (A[] (SoftController.level == Udisplay.level) imply true).
4) Global invariants: When water injection is turned on, the degree received by the system from the water level sensor must be less than 34 units. (((SoftController.Ispumpon == true) && (Tsensor.level < 34)) --> true), (((SoftController.Ispumpon == true) && (Tsensor.level > 34)) --> false).

The results of executing these security constraint queries using UPPAAL are shown in Fig. 17. This figure shows that the system will not violate any security constraints when operating in the normal mode.

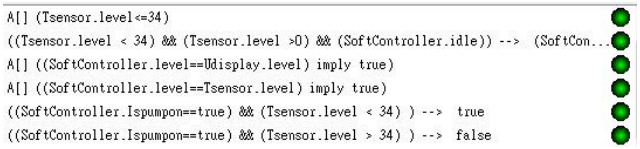


FIGURE 17. UPPAAL system security constraint verification query in the normal mode.

Enable the Attack Module and use the same security constraints to perform model checks in the UPPAAL model. The result is as shown in Fig. 18. Through inspection, it can be found that only when the system violates the value and the range and/or the invariant constraints will the system abnormalities be discovered. Such results are often not expected in system security, if we hope that this type of attack can be detected or warned about early. Thus, a further revision of the security constraints or the system design should be performed.

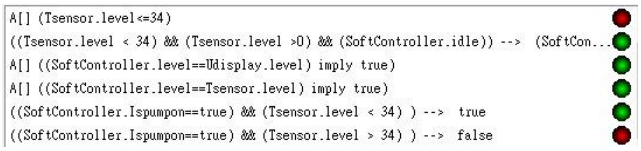


FIGURE 18. UPPAAL system security constraint verification query under the attack module enabled.

4) IDENTIFY VULNERABILITIES AND CONSTRAINT INADEQUACY FOR IMPROVEMENTS

The fourth step is to identify the vulnerabilities and the constraint inadequacy for making improvements, such as the situation in the above case. In this case, the consistency between the physical and the cyber values can be added; that is, the following checks are added: (SoftController.level == Tsensor.level) --> (SoftController.level == Udisplay.level) and (SoftController.level == Udisplay.level) --> (SoftController.level == Tsensor.level), which turns out to be effective in detecting security problems, as shown in Fig. 19. Repeat the above steps in the same way until an acceptable set of security constraints is obtained.

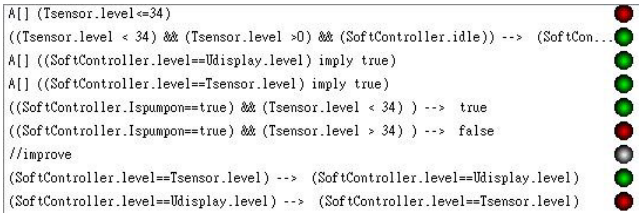


FIGURE 19. UPPAAL system security constraint verification query.

IV. CASE STUDY

To demonstrate the effectiveness of the proposed method, the method was applied to a heating control system.

A. CASE 1: HEATING CONTROL SYSTEM

We took a heating control system as a case study, whose behavior can be described as follows:

1) The operator uses a heating switch to control the heater, and the display will show the current heating temperature.
2) When the heating switch is turned off, the system starts to cool down, and when the heating switch is turned on, the system starts to warm up.
3) When the heating switch is turned on, the heating process starts from any temperature lower than 150°C to 200°C.

In this case, on the basis of the possible attacks of the CPS attack tree, we will show the verification results of the system simulation of the possible attack combinations, such as multiple attacks and single attacks.

1) SYSTEMATIC SECURITY CONSTRAINT GENERATION

First, we systematically generate the security constraints for this case on the basis of the above-proposed categories:

1) Value and range constraints: They check the working range such as the normal working range. In this case, the legal temperature range is from 150°C to 200°C when the switch is on.
2) Dependency between device states and process variables: In this case, the system is in a cooling state after the system heating switch is turned off, and in a heating state after the switch is turned on.
3) Cyber and physical consistency: In this case, the actual temperature in the physical system is equal to the value of the temperature variable in the control software and to the value indicated by the temperature sensor.
4) Global invariants: In this case, the pragmatic invariant is that the highest temperature legally allowed is 200°C; the energy conservation invariant is that there exists a relationship between the temperature increase and the generated heat.

2) ESTABLISH THE SECURITY VERIFICATION MODEL

The normal system model including an HMI is first constructed in UPPAAL. The Normal Module includes a user switch control module on the HMI module, which provides a simulation of the user pressing the switch, as shown in Fig. 20(a). We will use capital U to represent user-related events or variables. The temperature display module is shown in Fig. 20(b), where the variable TInHMI represents the currently displayed temperature. A cyber module representing the heating control is shown in Fig. 21(a). The heating control is mainly used to control the heating process. Variables with the capital T refer to the current temperature and those with the capital H refer to the heater. The physical module represents the temperature sensor, as shown in Fig. 21(b).

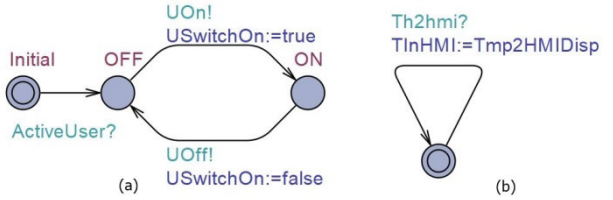


FIGURE 20. Human-machine interface modules.

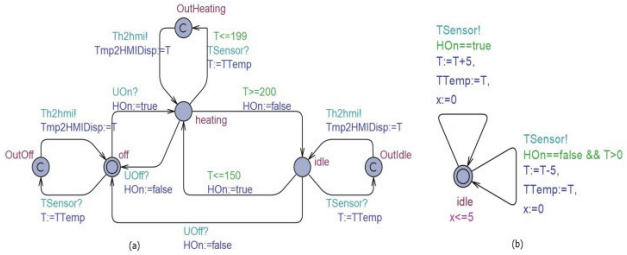


FIGURE 21. Cyber module and physical module.

Attack scenarios are then generated on the basis of the above CPS attack tree templates (Fig. 5). We demonstrate two cases of attacks on software controllers, which may possibly make the OT state unsafe, namely CaseL and CaseT. Some implementable malware models will be added to CaseL; a successful attack may mislead the operator and cause disaster. Here, we show a hypothetical attack in which the heater stops heating after reaching a temperature of 150°C. However, the display constantly shows that the temperature is increasing, thereby misleading the operator. To model CaseL, we first add a malware input module before the heating control. The module is named InAtkCaseL and controls and changes the heating control behavior. The malware input module that maliciously controls the input temperature is shown in in Fig. 22. Use the variable CIn[0] to control whether InAtkCtrlCaseL will be enabled. In addition, it is necessary to change the receiving and heating control behavior of the heating controller and to inject malicious attacks into the heating controller. The inject malicious module part is named MwAtkCtrlCaseL. The variable CM[0] is used to indicate

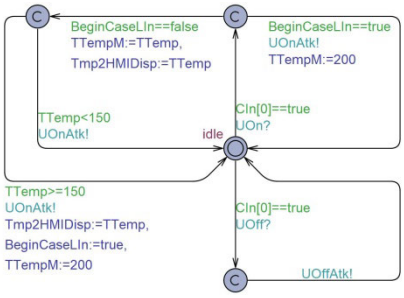


FIGURE 22. Malware input module in CaseL.

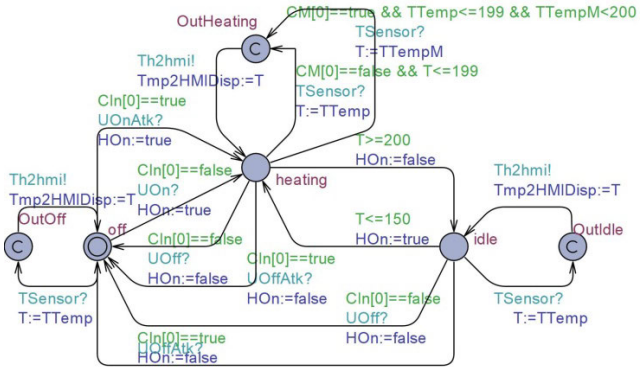


FIGURE 23. Malware heater module in CaseL.

whether to enable the injection attack, as shown in Figure 23. When the heater is not heated, the display can continue to display temperature changes, so add a malware input module to control the temperature display between the heater and the display, as shown in Fig. 24. The module is named InAtkDispCaseL. The variable CIn[1] is used to indicate whether to enable this input attack.

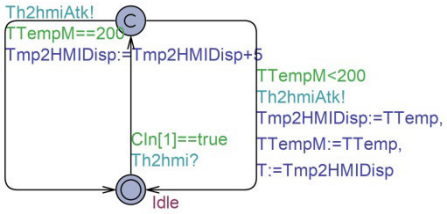


FIGURE 24. Malware input module in CaseL.

We will integrate Case T and Case L in a UPPAAL model. CaseT demonstrated that the malware simply modified the input value returned by the temperature sensor and received by the heating controller. Hypothetically, the attack shown here is to decrease the sensor reading by 10°C. A successful attack may result in an excessive error between the temperature of the heating controller and the actual temperature. This may result in asset loss. Other possible output attacks can be modelled similarly. Therefore, the change of the output from the sensor to the heating controller is shown in Fig. 25. The output attack module is named OutAtkSnsrCaseT. The heating controller will be modified to receive the message of the malicious output attacking module, as shown in Fig. 26. The variable COt[0] is used to indicate whether to enable

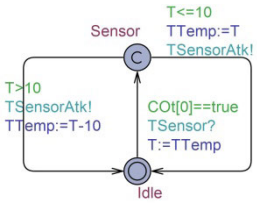


FIGURE 25. Malware output module in CaseT.

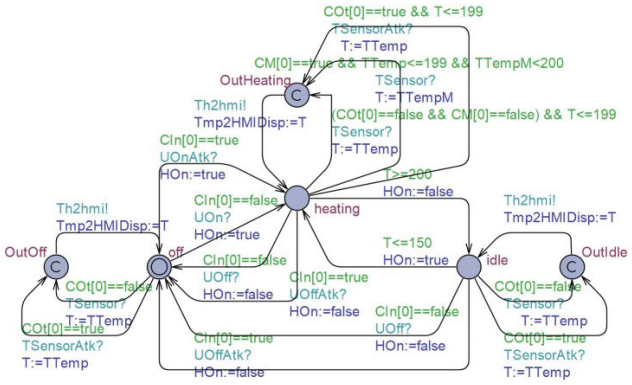


FIGURE 26. Malware heating controller include CaseT.

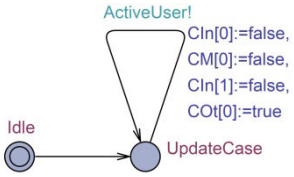


FIGURE 27. The Simulation Module for CaseT and CaseL.

OutAtkSnsrCaseT. Finally, we use the Simulation Module to control two attack scenarios, as shown in Fig. 27. The plot of Case T can be simulated by activating OutAtkSnsrCase. Similarly, the scenario of Case L can be simulated by simultaneously activating InAtkCtrlCaseL, MwAtkCtrlCaseL, and InAtkDispCaseL.

3) AUTHOR PHOTO MODEL CHECKING ALONG WITH SIMULATION BASED ON THE GENERATED SECURITY CONSTRAINTS

The next step is to perform a model check based on the security constraints suggested in the first step. For this set of security constraints, the model will be used to predict whether OT-level system damages will occur when various possible attacks are encountered and verify whether the set of security constraints is sufficient and can be used to detect any of the possible security issues currently known. At the same time, we can also check whether it is possible to detect these possible attacks. The security constraints in this case are given below:

1) Value and range constraints: For example, when the temperature is less than or equal to 150°C, heating must be started (ProcHeater. T <= 150 -> HOn == true), and the temperature must be stopped when it reaches 200°C. (ProcHeater.T > 200 -> HOn == false).
2) Dependency between device states and process variables: If the system heating switch is turned off, the system is in a cooling state. (ProcUser.USwitchOn == true -> ProcHeater.heating || ProcHeater.idle). Moreover, the system is in a heating state after being turned on. (ProcUser.USwitchOn == false -> ProcHeater.off).

3) Cyber and physical consistency: For example, the temperature of the temperature detection sensor must be consistent with the controller temperature. (A[] (ProcTprSensor.T == ProcHeater.T) imply true).
4) Global invariants: When the heating switch is turned off, the system starts to cool down and the maximum temperature cannot be higher than 200°C. A[] ((HOn == false) && (ProcHeater.T > 200) imply true)
5) The system starts to heat up, and the maximum temperature cannot be higher than 200°C. A[] ((HOn == true) && (ProcHeater.T > 200) imply false).

The results of executing these security constraint queries using UPPAAL are shown in Fig. 28. This figure shows that the system will not violate any security constraints when operating in the normal mode.

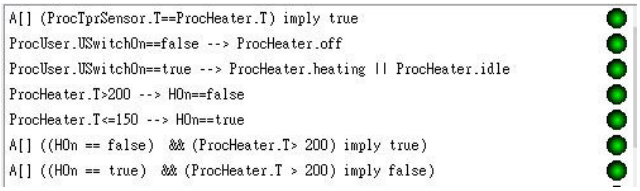


FIGURE 28. System security constraint verification with UPPAAL in the normal mode.

The abovementioned security constraints are used to perform model checks on the CaseL and CaseT models. The verified results were the same, as shown in Fig. 29. This implied that the implanted malware could effectively circumvent all of the abovementioned security constraints. Thus, the considered set of security constraints is not adequate. Thus, a further revision of the security constraints or system design should be performed.

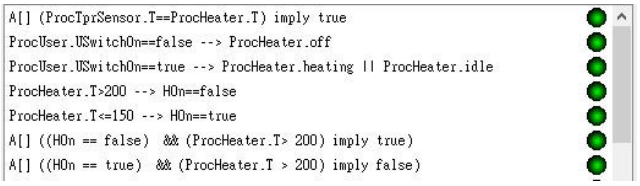


FIGURE 29. Results of security constraint detection.

4) IDENTIFY VULNERABILITIES AND CONSTRAINT INADEQUACY FOR IMPROVEMENTS

The fourth step is to identify vulnerabilities and constraint inadequacy for improvements, such as the situation in the above CaseL. In CaseL, the consistency between the physical and the cyber values can be added; that is, the following check can be added: A[] (ProcHeater.T == ProcHMI.TInHMI) imply (ProcTprSensor.T == ProcHeater.T), which turns out to be effective in detecting security problems, as shown in Fig. 30. This improvement has the same effect on CaseT. However, note that the check for the consistency between the physical entities and their cyber counterparts may need onsite human checking, which may not always be convenient or feasible.

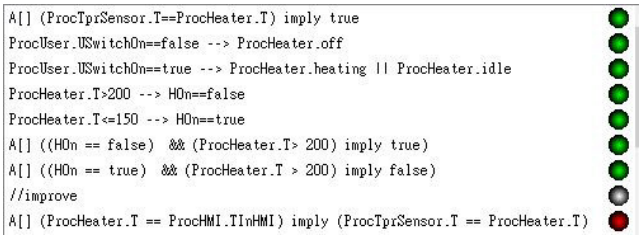


FIGURE 30. System security constraint verification with UPPAAL.

B. CASE 2: SAFETY INJECTION SYSTEM

For the safety injection system, if the operator receives incorrect system messages and cannot detect them in time, the most serious hazard may be the water tank dry-out, such as the Three Mile Island accident [36]. Therefore, the proposed approach will be used in this case to explore the possible various states of the system and the HMI interaction simulation, verify the security constraints, and predict the possible hazards. This result can be used to improve the design or improve the security constraints to avoid accidents.

1) SECURITY CONSTRAINTS AND MODELING PROCESS

A safety injection system is used as our case study. The system consists of Tank0, Tank1, and Tank2. Tank0 generates steam, cools it, and circulates the condensed water back to Tank0 through Tank1. If water leaks from Tank0, the leaked water will be collected in Tank2, as shown in Fig. 31. The normal water level of Tank0 ranks from L1 to L8.5; otherwise, the alarm will be on. When a leak occurs and the water level of Tank0 drops to L1.5, the software controller starts to inject the feed-water from Tank1 to Tank0 by opening the pump and the valve between them; however, if Tank1's water is below the threshold, the feed-water injection will come from Tank2 to Tank0 till Tank0's water level reaches L8. Note that in order to use the integer-based simulation tools provided by UPPAAL, we expand all the water levels by 10 times; for example, water level L8.5 will be represented as 85 units.

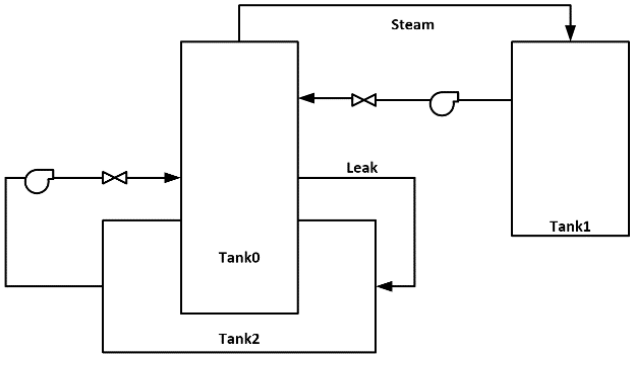


FIGURE 31. Overview of the safety injection system.

The most devastating OT attack is that which causes the dry-out of Tank0. In order to accomplish this goal, the attacker should stop any water injection when Tank0 is low in

water, and at the same time, the operator should not be alerted by the display or the alarm. Thus, multiple and coordinated attacks, consisting of a sabotage on the control logic, on the display, and on the alarm, can be achieved by the malware. Therefore, our modelled malware will not perform feed-water injection and deliberately indicate that Tank0 gradually rises to 82 units, and turn off the alarm, which eventually causes a dry-out disaster. We then design all the needed UPPAAL system components, namely a Normal Model, an HMI model, an Attack Module, and a Simulation Module. We also model the software controller being implanted with malware.

The software controller will switch between three states when it is executing. The first state is the normal operation of Tank0, as shown in Fig. 32; the second is the state of supplying water by Tank1, as shown in Fig. 33(a); and the last is the state of supplying water by Tank2, as shown in Fig. 33(b). The first state is named idle, the second state is named supply1, and the third state is named supply2. The display system and the operator manual mode are constructed in the HMI model, as shown in Fig. 34, wherein the left half of the module is used as the display mode and the right half is used as the manual operation mode. If the Tank0 display water level is greater than 80 units and a pump has been turned on, it will simulate the manual shutdown of the pump. In order to construct the above attack scenario, we added malware to the

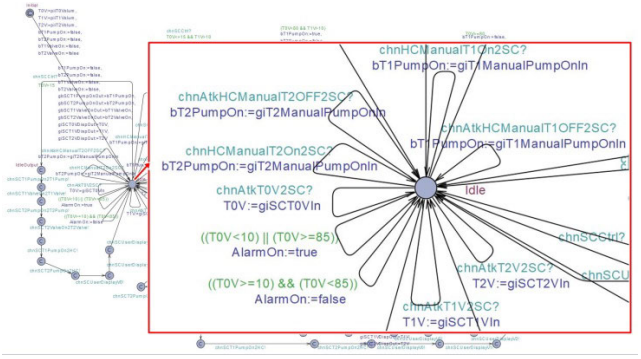


FIGURE 32. Normal operating state of Tank0 in the software controller.

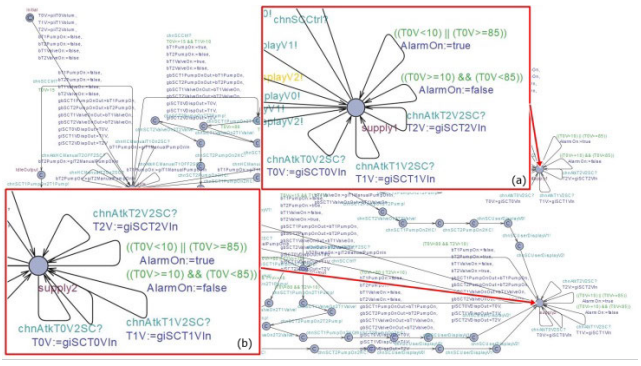


FIGURE 33. State of supplying water by Tank1 and the state of supplying water by Tank2 in the software controller.

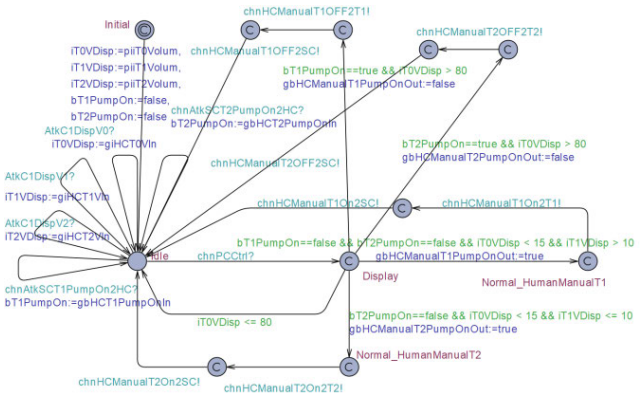


FIGURE 34. Display system and operator manual mode are constructed in the HMI model.

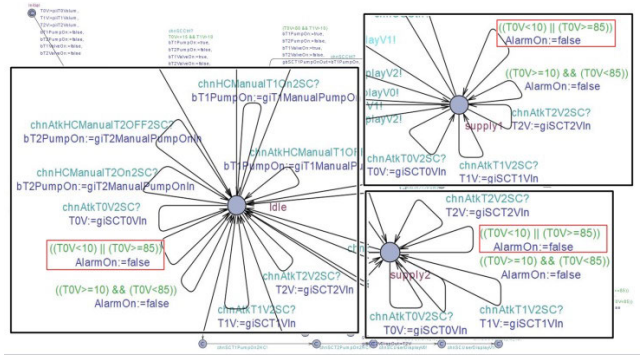


FIGURE 35. Malware will turn off the alarm in the software controller.

software controller. The malware will provide the function of turning off the alarm system, as shown in Fig. 35. An error in the water level display has also been added, which will provide an erroneous water level to deliberately deceive the operator, as shown in Fig. 36. The water level of Tank0 will be displayed below 85, the water level of Tank1 will be displayed below 26, and the water level of Tank2 will be displayed below 5. Each time, the water level changes by two units.

The next step is to design queries for model checking on the basis of the proposed categories of security constraints. The security constraints for this case can be described as follows:

1) Value and range constraints: The normal range of the Tank0 water level is from Level 1 to Level 8.5, that is, from 10 units to 85 units. ((SfCntr.IdleOutput && AlarmOn == false) || (SfCntr.T1Output && AlarmOn == false) || (SfCntr.T2Output && AlarmOn == false)) --> ((SfCntr.T0V >= 10) && (SfCntr. T0V <= 85))
2) Constraints of dependency between device states and process variables: If the Tank1 pump and valve are on, the water volume of Tank1 is being reduced; if the Tank1 pump and Tank1 valve are off, the water volume of Tank1 remains unchanged. A[] (T1.PumpOn == true && T1.ValveOn == true) imply

```
int C1V0Disp( ) {
    if (AtkC1==true) {
        if (xV0<82) {
            xV0:=xV0+2;
        } else {  xV0:=xV0-2;          }
    } else {  xV0:=giAtkT0VDispOut;  }
    return xV0; }//int C1V0Disp()

int C1V1Disp( ) {
    if (giSCT1VIn<=15) AtkC1:=true;
    if (AtkC1==true) {
        if (xV1<26) {
            xV1:=xV1+2;
        } else {  xV1:=xV1-2;          }
    } else {  xV1:=giAtkT1VDispOut;  }
    return xV1; }//int C1V1Disp()

int C1V2Disp( ) {
    if (AtkC1==true) {
        xV2:=xV2-1;
        if (xV2<5) {
            xV2:=5;                    }
    } else {  xV2:=giAtkT2VDispOut;  }
    return xV2; }//int C1V2Disp
```

FIGURE 36. Error water level display to deliberately deceive the operator.

```
((SfCntr.IdleOutput && AlarmOn==false) || (SfCntr.T1Output && AlarmOn==false) || (SfCntr.T2Output &&...
A[] (T1.PumpOn==true && T1.ValveOn==true) imply (T1.Volum<= T1.LastVolum)
A[] (T1.PumpOn==false && T1.ValveOn==false) imply (T1.Volum>= T1.LastVolum)
A[] (T2.PumpOn==true && T2.ValveOn==true) imply (T2.Volum<= T2.LastVolum)
A[] (T2.PumpOn==false && T2.ValveOn==false) imply (T2.Volum>= T2.LastVolum)
(T2.PumpOn==true && T2.ValveOn==true) --> (T2.Volum<= T2.LastVolum)
(T2.PumpOn==false && T2.ValveOn==false) --> (T2.Volum== T2.LastVolum)
A[] HMI.Display imply (T0.Volum==SfCntr.T0V) && (T0.Volum==HMI.iT0VDisp)
A[] HMI.Display imply (T1.Volum+T1.iSteamV==SfCntr.T1V) && (T1.Volum+T1.iSteamV==HMI.iT1VDisp)
A[] HMI.Display imply (T2.Volum+T2.LeakV==SfCntr.T2V) && (T2.Volum+T2.LeakV==HMI.iT2VDisp)
(T0.React && T1.React && T2.React) --> (T0.Volum+T0.T1FeedV+T0.T2FeedV+T1.Volum+T1.iSteamV+T2.Volum+...
(SfCntr.IdleOutput || SfCntr.T1Output || SfCntr.T2Output)  --> (SfCntr.T0V+SfCntr.T1V+SfCntr.T2V) ==110
(HMI.Display) --> (HMI.iT0VDisp+HMI.iT1VDisp+HMI.iT2VDisp)==110
```

FIGURE 37. Model checking security constraints.

```
((SfCntr.IdleOutput && AlarmOn==false) || (SfCntr.T1Output && AlarmOn==false) || (SfCntr.T2Output &&...
A[] (T1.PumpOn==true && T1.ValveOn==true) imply (T1.Volum<= T1.LastVolum)
A[] (T1.PumpOn==false && T1.ValveOn==false) imply (T1.Volum>= T1.LastVolum)
A[] (T2.PumpOn==true && T2.ValveOn==true) imply (T2.Volum<= T2.LastVolum)
A[] (T2.PumpOn==false && T2.ValveOn==false) imply (T2.Volum>= T2.LastVolum)
(T2.PumpOn==true && T2.ValveOn==true) --> (T2.Volum<= T2.LastVolum)
(T2.PumpOn==false && T2.ValveOn==false) --> (T2.Volum== T2.LastVolum)
A[] HMI.Display imply (T0.Volum==SfCntr.T0V) && (T0.Volum==HMI.iT0VDisp)
A[] HMI.Display imply (T1.Volum+T1.iSteamV==SfCntr.T1V) && (T1.Volum+T1.iSteamV==HMI.iT1VDisp)
A[] HMI.Display imply (T2.Volum+T2.LeakV==SfCntr.T2V) && (T2.Volum+T2.LeakV==HMI.iT2VDisp)
(T0.React && T1.React && T2.React) --> (T0.Volum+T0.T1FeedV+T0.T2FeedV+T1.Volum+T1.iSteamV+T2.Volum+...
(SfCntr.IdleOutput || SfCntr.T1Output || SfCntr.T2Output)  --> (SfCntr.T0V+SfCntr.T1V+SfCntr.T2V) ==110
(HMI.Display) --> (HMI.iT0VDisp+HMI.iT1VDisp+HMI.iT2VDisp)==110
```

FIGURE 38. Verification results of system security constraints in the attack scenario.

(T1.Volum <= T1.LastVolum) A[] (T1.PumpOn == false && T1.ValveOn == false) imply (T1.Volum >= T1.LastVolum). For Tank2, we have the same constraints.

3) Cyber and physical consistency constraints: The water volume of Tank0 must be equal to the value of the variable representing the water volume in the controller. A[] HMI.Display imply (T0.Volum == SfCntr.T0V) && (T0.Volum == HMI.iT0VDisp). The same constraints apply to Tank1 and Tank2.
4) Global invariants: The software variables of the three water tanks should add up to the initial total water amount of 110 units in our test case. (T0.React && T1.React && T2.React) --> (T0.Volum + T0.T1FeedV + T0.T2FeedV + T1.Volum + T1.iSteamV + T2.Volum + T2.LeakV) == 110. The same constraints apply to the software controller and HMI display part.

The resulting queries in UPPAAL, based on these security constraints, are shown in Fig. 37. Because the checks are performed on the normal system, all the checks pass with green lights. Next, these queries will be conducted on the attack model.

2) MODEL CHECKING MALWARE IMPLEMENTED SCENARIO

As mentioned above, the most devastating attack consequence is a dry-out disaster. This attack can be easily accomplished by an implanted malware producing multiple and concurrent attacks on the control logic along with false display data and a nonfunctional alarm. That is, the malware control will not inject feed-water when Tank0 has a water level lower than Level 1.5, while it displays a normal or high water level and disables the alarm. Thus, the operator may be misled and does not react or perform erroneous actions. The following scenario is modeled in our attack module. The malware sets Tank0's display at a high-water level when the water level is low. In our test run, Tank0's water level is set to 15 units (equivalent to Level 1.5) and the display starts to be attacked. During the attack, Tank1's water level display will gradually approach to 26 units and Tank0's water level display will gradually increase from the current water level to a maximum of 82 units. Tank2's water level display will be based on the displays of Tank1 and Tank0. The water level display gradually adjusts. The malicious water level display controller, which we designed for the first time, is shown in Fig. 36. Then, UPPAAL is used to perform the model check on the abovementioned security constraints on the simulation run of this attack scenario. The results are shown in Fig. 38. However, we tried to evade the security constraints and redesigned the Attack Module, as shown in Fig. 39. The test results are shown in Fig. 40. In the last query of security constraints, we obtained different detection results.

However, such a malware attack producing a false indication may probably cause operator mode confusion, leading to more complex consequences. We may further analyze the results through the UPPAAL simulation tool and track the execution steps to find the counterexample. In our test run, the actual water levels of Tank0, Tank1, and Tank2 in the UPPAAL modules are 0, 26, and 84 units, but the display levels of Tank0, Tank1, and Tank2 show 82, 25, and 3 units, as shown in Fig. 41. The possible consequences may be that the operator is misled and manually turns off the pump of Tank1 and then the pump of Tank2, which may eventually result in a dry-out disaster. After analysis, we identify that the

```
int C1V0Disp() {
    if (AtkC1==truc) {
        xV0:=iTotalWater-xV1-xV2;
        if (xV0<82) {
            xV0:=xV0+1;
        } else {  xV0:=xV0-1;        }
    } else {  xV0:=giAtkT0VDispOut;  }
    return xV0; }//int C1V0Disp()

int C1V1Disp() {
    if (giSCT1VIn<=15) AtkC1:=truc;
    if (AtkC1==truc) {
        if (xV1<26) {
            xV1:=xV1+1;
        } else {  xV1:=xV1-1;        }
    } else {  xV1:=giAtkT1VDispOut;  }
    return xV1; }//int C1V1Disp()

int C1V2Disp() {
    if (AtkC1==truc) {
        xV2:=iTotalWater-xV1-xV0;
    } else {  xV2:=giAtkT2VDispOut;  }
    return xV2; }//int C1V2Disp
```

FIGURE 39. Redesign of the attack module.

```
((SfCntr.IdleOutput && AlarmOn==false) || (SfCntr.T1Output && AlarmOn==false) || (SfCntr.T2Output &&...
A[] (T1.PumpOn==true && T1.ValveOn==true) imply (T1.Volum<= T1.LastVolum)
A[] (T1.PumpOn==false && T1.ValveOn==false) imply (T1.Volum>= T1.LastVolum)
A[] (T2.PumpOn==true && T2.ValveOn==true) imply (T2.Volum<= T2.LastVolum)
A[] (T2.PumpOn==false && T2.ValveOn==false) imply (T2.Volum>= T2.LastVolum)
(T2.PumpOn==true && T2.ValveOn==true) --> (T2.Volum<= T2.LastVolum)
(T2.PumpOn==false && T2.ValveOn==false) --> (T2.Volum== T2.LastVolum)
A[] HMI.Display imply (T0.Volum==SfCntr.T0V) && (T0.Volum==HMI.iT0VDisp)
A[] HMI.Display imply (T1.Volum+T1.iSteamV==SfCntr.T1V) && (T1.Volum+T1.iSteamV==HMI.iT1VDisp)
A[] HMI.Display imply (T2.Volum+T2.LeakV==SfCntr.T2V) && (T2.Volum+T2.LeakV==HMI.iT2VDisp)
(T0.React && T1.React && T2.React) --> (T0.Volum+T0.T1FeedV+T0.T2FeedV+T1.Volum+T1.iSteamV+T2.Volum+...
(SfCntr.IdleOutput || SfCntr.T1Output || SfCntr.T2Output)  --> (SfCntr.T0V+SfCntr.T1V+SfCntr.T2V) ==110
(HMI.Display) --> (HMI.iT0VDisp+HMI.iT1VDisp+HMI.iT2VDisp)==110
```

FIGURE 40. Verification results in the redesigned attack module.

set of constraints generated in our first try is not adequate. The global invariants are effective in many cases; however, for a carefully designed attack, the invariants may also fail to detect attacks. The remaining powerful security constraints are used to compare the consistency of the cyber images, such as software variables and display values, with their physical counterparts. However, note that these consistency checks of the cyber and physical counterparts may need human onsite checks, which may not be always convenient or even possible. In addition, system vulnerabilities can be identified by the above model checking. For example, the vulnerabilities of the above case consist of the HMI and network. Then, system redesign such as the addition of multiple sensors, multiple alarms, or hardware interlocks, along with a possible recommendation of periodic onsite human checking, may also be carried out to improve system security.

In summary, we successfully applied the proposed method to this case study. The proposed model checking step may reveal the inadequacy of the original constraint set, which can be further augmented, such as a lack of comparison of the consistency of the software variables and the display values with their physical counterparts. The enhanced security constraint set can be used for run-time monitoring in order to detect security attacks when the system is in actual operation. Moreover, the proposed model checking can

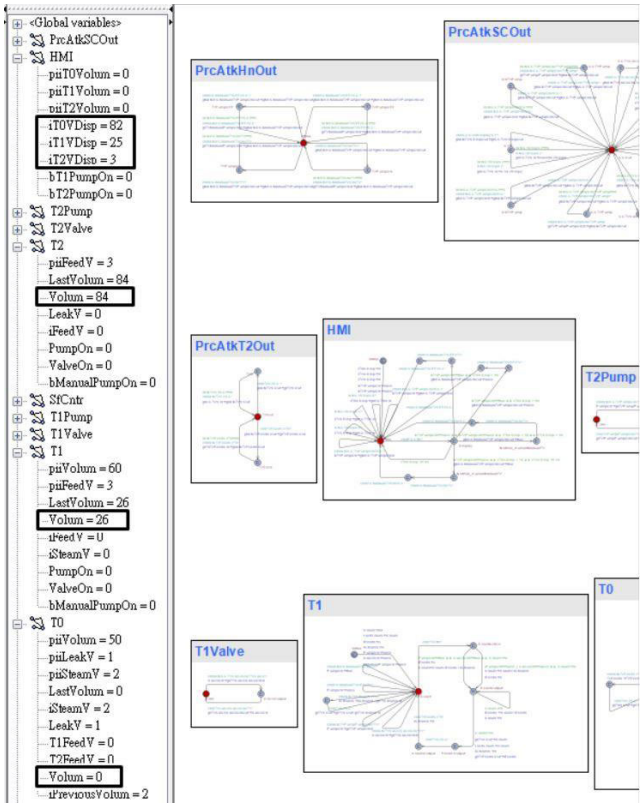


FIGURE 41. Results obtained using UPPAAL simulation tools.

identify the vulnerabilities of the system under analysis, such as the inability of the invariant to detect an attack in the case of a well-designed attack. Thus, design improvement may be required to alleviate these vulnerabilities.

V. SUMMARY AND CONCLUSION

Cyber security is critical for a CPS. We may ensure a CPS's security by preventing attacks at the design stage and detecting attacks at run time by constraint monitoring. To achieve these goals, we proposed a systematic approach for the security verification of a CPS design by using UPPAAL model checking. We found that an exhaustive search can be achieved for each state in various possible model combinations and the human-machine interaction could be explored more deeply. Different from most of the current research that focuses on IT attacks to control the CPS, without addressing the OT damage, our work particularly focused on the OT attacks. The categories of OT-level security constraints were first proposed. These constraints could be generated on the basis of safety requirements in a systematic manner. Then, they were represented in the UPPAAL queries for model checking. Both a normal model and an attack model were developed in UPPAAL. The attack model mainly represented an implanted malware for a selected attack scenario based on our attack tree template. Model checking using the generated security constraints was then performed on both the normal and the attack models to explore the potential OT attacks. The results of this method support the following two-fold functions:

1) Augmenting the set of security constraints. If the model checking shows that the original set of constraints is not effective in detecting malicious attacks, a more adequate set of security constraints can be developed to serve as run-time monitoring in the future when the system is in operation.
2) Redesigning the system. The model checking results can be further analyzed to identify vulnerabilities for a possible redesign and improvement of the system to enhance its cyber security.

According to our experiments, the global invariant is in general effective. However, for carefully designed attacks, the global invariant may fail to detect any anomaly. Then, the only useful constraints will be the consistency checking between the cyber and the physical counterparts, that is, onsite human checking, which may not always be convenient or feasible. Thus, when it is difficult to prevent or detect malicious attacks, the most rigorous recommendation is that the critical components of a CPS should be changed back to the analog design in order to reduce the attack surface [37]. In the future, the completeness of the constraint sets and the process of automatically revising constraints and proposing new constraint sets will be further investigated. Moreover, a hybrid simulation by combining UPPAAL with Ptolemy [38] will be conducted, as the latter can model the continuous behavior more accurately.

REFERENCES

[1] L. Monostori, "Cyber-physical systems," in *CIRP Encyclopedia of Production Engineering*, S. Chatti and T. Tolio, Eds. Berlin, Germany: Springer, 2018, pp. 1–8.

[2] P. B. L. D. Haegley, "Resilient industrial control systems (ICS) & cyber physical systems (CPS)," *J. Cyber Secur. Inf. Syst.*, vol. 7, no. 2, pp. 5–9, Sep. 2019.

[3] CyberX. (2019). *2019 Global ICS & IIoT Risk Report*. Accessed: Feb. 2, 2021. [Online]. Available: https://cyberx-labs.com/resources/risk-report-2019/#donwload-form

[4] A. Nourian and S. Madnick, "A systems theoretic approach to the security threats in cyber physical systems applied to stuxnet," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 1, pp. 2–13, Jan. 2018, doi: 10.1109/TDSC.2015.2509994.

[5] K. Zetter. (2016). *Inside the Cunning, Unprecedented Hack of Ukraine's Power Grid, Wired*. Accessed: Feb. 2, 2021. [Online]. Available: https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/

[6] V. Y. P. Keith, A. Stouffer, S. Lightman, M. Abrams, and A. Hahn, "Guide to industrial control systems (ICS) security," NIST, Gaithersburg, MD, USA, Tech. Rep. 800-82 Rev 2, 2015. [Online]. Available: https://www.nist.gov/publications/guide-industrial-control-systems-ics-security

[7] A. Humayed, J. Lin, F. Li, and B. Luo, "Cyber-physical systems security—A survey," *IEEE Internet Things J.*, vol. 4, no. 6, pp. 1802–1831, Dec. 2017, doi: 10.1109/JIOT.2017.2703172.

[8] Waterfall Security. *Whitepaper Emerging Consensus for Industrial Control System (ICS) Security*. Accessed: Feb. 2, 2021. [Online]. Available: https://waterfall-security.com/scada-security/whitepapers/emerging-consensus-on-ics-cybersecurity/

[9] G. Sabaliauskaite and A. Mathur, "Aligning cyber-physical system safety and security," in *Proc. CSDM*, 2014, pp. 41–53.

[10] Positive Technologies. *ICS Vulnerabilities: 2018 in Review*. Accessed: Feb. 2, 2021. [Online]. Available: https://www.ptsecurity.com/ww-en/analytics/ics-vulnerabilities-2019/

[11] TechRepublic. *Industrial Control System Cybersecurity Vulnerabilities are Rising in 2020*. Accessed: Feb. 2, 2021. [Online]. Available: https://www.ptsecurity.com/ww-en/analytics/ics-vulnerabilities-2019/

[12] Cyber-Physical Systems Virtual Organization. *Verification Tools Main Wiki Page*. Accessed: Mar. 21, 2021. [Online]. Available: https://cps-vo.org/node/32762

[13] G. Behrmann, A. David, and K. Larsen, "A tutorial on uppaal," in *Proc. Formal Methods Design Real-Time Syst.*, 2004, pp. 200–236.

[14] Q. Zhongsheng, L. Xin, and W. Xiaojin, "Modeling distributed real-time elevator system by three model checkers," *Int. J. Online Eng.*, vol. 14, no. 4, p. 94, Apr. 2018, doi: 10.3991/ijoe.v14i04.8383.

[15] Y. Lu and M. Sun, "Modeling and verification of IEEE 802.11i security protocol in UPPAAL for Internet of Things," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 28, nos. 11–12, pp. 1619–1636, Nov. 2018, doi: 10.1142/S021819401840020X.

[16] IEEE Spectrum. *The Blackout of 2003*. Accessed: Feb. 2, 2021. [Online]. Available: https://spectrum.ieee.org/energy/the-smarter-grid/the-blackout-of-2003

[17] History. *Blackout Hits Northeast United States*. Accessed: Feb. 2, 2021. [Online]. Available: https://www.history.com/this-day-in-history/blackout-hits-northeast-united-states

[18] Taiwan Transportation Safety Board. *Releases Final Report on TransAsia Airways Flight GE 235 Occurrence Investigation*. Accessed: Feb. 2, 2021. [Online]. Available: https://www.ttsb.gov.tw/english/16051/16113/16114/16531/post

[19] Aviation Accidents. *China Airlines-Airbus-A310-B4-622R (B-1814) Flight CI676*. Accessed: Jan. 7, 2021. [Online]. Available: https://www.aviation-accidents.net/china-airlines-airbus-a310-b4-622r-b-1814-flight-ci676/

[20] R. Akella, "Verification of information flow security in cyber-physical systems," Ph.D. dissertation, Dept. Comput. Sci., Missouri Univ. Sci. Technol., Rolla, MO, USA, 2013.

[21] G. Sugumar and A. Mathur, "Testing the effectiveness of attack detection mechanisms in industrial control systems," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Jul. 2017, pp. 138–145, doi: 10.1109/QRS-C.2017.29.

[22] S. Han, M. Xie, H.-H. Chen, and Y. Ling, "Intrusion detection in cyber-physical systems: Techniques and challenges," *IEEE Syst. J.*, vol. 8, no. 4, pp. 1049–1059, Dec. 2014, doi: 10.1109/JSYST.2013.2257594.

[23] S. Adepu and A. Mathur, "Using process invariants to detect cyber attacks on a water treatment system," in *ICT Systems Security and Privacy Protection*, J.-H. Hoepman and S. Katzenbeisser, Eds. Cham, Switzerland: Springer, 2016, pp. 91–104.

[24] P. Martins, A. B. Reis, P. Salvador, and S. Sargento, "Physical layer anomaly detection mechanisms in IoT networks," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Budapest, Hungary, Apr. 2020, pp. 1–9, doi: 10.1109/NOMS47738.2020.9110323.

[25] D. Ding, Q.-L. Han, X. Ge, and J. Wang, "Secure state estimation and control of cyber-physical systems: A survey," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 51, no. 1, pp. 176–190, Jan. 2021, doi: 10.1109/TSMC.2020.3041121.

[26] S. Kong, Y. Shen, and H. Zhou, "Using security invariant to verify confidentiality in hardware design," presented at the Great Lakes Symp. VLSI, Banff, AB, Canada, 2017.

[27] A. Choudhari, H. Ramaprasad, T. Paul, J. W. Kimball, M. Zawodniok, B. Mcmillin, and S. Chellappan, "Stability of a cyber-physical smart grid system using cooperating invariants," in *Proc. IEEE 37th Annu. Comput. Softw. Appl. Conf.*, Jul. 2013, pp. 760–769, doi: 10.1109/COMPSAC.2013.126.

[28] S.-W. Hsiao, Y. S. Sun, M. C. Chen, and H. Zhang, "Cross-level behavioral analysis for robust early intrusion detection," in *Proc. IEEE Int. Conf. Intell. Secur. Informat.*, May 2010, pp. 95–100, doi: 10.1109/ISI.2010.5484768.

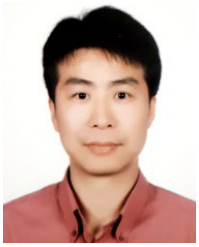
[29] Wikipedia. *Invariant*. Accessed: Feb. 2, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Invariant

[30] G. Sugumar and A. Mathur, "A method for testing distributed anomaly detectors," *Int. J. Crit. Infrastruct. Protection*, vol. 27, Dec. 2019, Art. no. 100324, doi: 10.1016/j.ijcip.2019.100324.

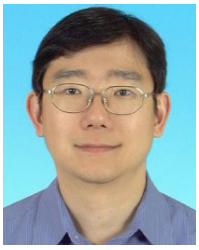
[31] C.-F. Fan, C.-C. Chan, H.-Y. Yu, and S. Yih, "A simulation platform for human-machine interaction safety analysis of cyber-physical systems," *Int. J. Ind. Ergonom.*, vol. 68, pp. 89–100, Nov. 2018, doi: 10.1016/j.ergon.2018.06.008.

[32] T. Wang, Q. Su, and T. Chen, "Formal analysis of security properties of cyber-physical system based on timed automata," in *Proc. IEEE 2nd Int. Conf. Data Sci. Cyberspace (DSC)*, Jun. 2017, pp. 534–540, doi: 10.1109/DSC.2017.44.

[33] M.-Z. Hong and C.-F. Fan, "Vulnerability analysis for cyber physical systems," presented at the Taiwan Acad. Netw. Conf. (TANET), Kaohsiung, Taiwan, 2019.
[34] S. Verma, P. Lee, and I. G. Harris, "Error detection using model checking vs. simulation," in *Proc. IEEE Int. High Level Design Validation Test Workshop*, Nov. 2006, pp. 55–58, doi: 10.1109/HLDVT.2006.319964.
[35] G. E. Gelman, "Comparison of model checking and simulation to examine aircraft system behavior," M.S. thesis, Aerosp. Eng., Georgia Inst. Technol., Atlanta, GA, USA, 2013.
[36] U.S.NRC. *Backgrounder on the Three Mile Island Accident*. Accessed: Feb. 2, 2021. [Online]. Available: https://www.nrc.gov/reading-rm/doc-collections/fact-sheets/3mile-isle.html
[37] S. G. Freeman, C. S. Michel, R. Smith, and M. Assante, "Consequence-driven cyber-informed engineering (CCE)," Idaho Nat. Lab., Idaho Falls, ID, USA, Tech. Rep. INL/EXT-16-39212, 2016, doi: 10.2172/1341416.
[38] S. Bogomolov, M. Greitschus, P. G. Jensen, K. G. Larsen, M. Mikucionis, T. Strump, and S. Tripakis, "Co-simulation of hybrid systems with SpaceEx and Uppaal," in *Proc. 11th Int. Modelica Conf.* Versailles, France: Linköping Univ. Electronic Press, pp. 159–169, doi: 10.3384/ecp15118159.



CHING-CHIEH CHAN was born in Taiwan, in 1968. He is currently pursuing the Ph.D. degree with the Computer Science and Engineering Department, Yuan-Ze University, under the supervision of Dr. Cheng-Zen Yang and Dr. Chin-Feng Fan. He works at Telecommunication Laboratories Chunghwa Telecom Company Ltd., Taiwan. His research interests include digital identity, safety analysis, and cyber security.



CHENG-ZEN YANG (Member, IEEE) received the B.S. and M.S. degrees from the Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan, in 1988 and 1990, respectively, and the Ph.D. degree from the Department of Computer Science and Information Engineering, National Taiwan University, in 1996. He is currently an Associate Professor with Yuan-Ze University. His research interests include software engineering, machine learning, text mining, and high-speed networking.

CHIN-FENG FAN received the Ph.D. degree in computer science from Southern Methodist University, Dallas, TX, USA. She is a Retired Professor of Yuan-Ze University, Taiwan. Her research interests include software engineering, safety analysis, and cyber security.