2015

# Mission-Aware Vulnerability Assessment for Cyber-Physical System

Xiaotian Wang
*Wright State University*

# MISSION-AWARE VULNERABILITY ASSESSMENT FOR CYBER-PHYSICAL SYSTEM

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Engineering

By

XIAOTIAN WANG
M.S., Wright State University, 2012

August 2015
Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

August 25, 2015

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Xiaotian Wang ENTITLED Mission-aware Vulnerability Assessment for Cyber-Physical System BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Engineering.

Junjie Zhang, Ph.D.
Thesis Director

Mateen M. Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on
Final Examination

Junjie Zhang, Ph.D.

Adam Robert Bryant, Ph.D.

Michelle Andreen Cheatham, Ph.D.

Robert E. W. Fyffe, Ph.D.
Vice President for Research and
Dean of the Graduate School

## Acknowledgements

I would like to express my sincerest gratitude to my supervisor Dr. Junjie Zhang, who has supported and guided me throughout my research and thesis with his patience, knowledge, and encouragement. Without his support and mentoring, this thesis would not have been completed or written. Furthermore, I would like to thank my advisory committee Dr. Adam Robert Bryant, and Dr. Michelle Andreen Cheatham for reviewing this thesis and providing advice for improvement. Last but not the least, I would like to thank my family: my wife and my parents for the unconditional support and love throughout writing this thesis and my life in general.

# Abstract

Wang, Xiaotian. M.S.C.E., Department of Computer Science and Engineering,
Wright State University, 2015.
Mission-Aware Vulnerability Assessment for Cyber-Physical Systems

Designing secure cyber-physical systems (CPS) is fundamentally important. An indispensable step towards this end is to perform vulnerability assessment. This thesis discusses the design and implementation of a mission-aware CPS vulnerability assessment framework. The framework intends to accomplish three objectives including i) mapping CPS mission into infrastructural components, ii) evaluating global impact of each vulnerability, and iii) achieving verifiable results and high flexibility. In order to accomplish these objectives, a model-based analysis strategy is employed. Specifically, a CPS simulator is used to model dynamic behaviors of CPS components under different missions; the framework facilitates a bottom-up approach to traverse a holistic model of a CPS that aims at profiling relationships among all CPS components. In order to analyze the derived models, we have leveraged formal methods, including program symbolic execution, logic programming, and linear optimization. The framework first successfully identifies mission-critical components, then discovers all attack paths from system access points to mission-critical components, and finally recommends the optimized mitigation plan.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1
# Introduction

## 1.1 PROBLEM CONTEXT

A Cyber-Physical System (CPS) integrates the capabilities of computation, communication, and control to facilitate seamless interaction between cyber and physical worlds. Despite the fact that CPSs have been envisioned to revolutionize a wide range of areas, their practical deployment faces many challenges. Security concerns have been recognized as one of the most significant obstacles since attackers may tamper with the integrity, confidentiality, and availability of a CPS in order to cause irrecoverable, disastrous consequences to cyber systems as well as physical entities. Securing CPSs therefore becomes fundamentally important.

Mitigating vulnerabilities is essential for CPS security since vulnerabilities have been considered as the targets for cyber attacks. Ideally, every vulnerability that is discovered can be mitigated by various means such as testing, software verification, and vendors' reporting. However, this ideal solution may suffer from huge practical limitations. On the one hand, mitigating all vulnerabilities could incur prohibitively high cost, particularly considering the huge complexity for typical CPSs. On the other hand, it might actually be unnecessary. For instance, if a vulnerability can never be accessed by attackers, leaving it unpatched will not affect the security of the CPS at all. Therefore, we need an effective method that can first assess how each vulnerability impacts the CPS security and then generate mitigation plans accordingly. However, devising such a

method is faced with several significant challenges. First, since CPSs are usually designed to accomplish certain missions, CPS missions need to be considered into the loop of vulnerability assessment. To be more specific, a vulnerable component may impact the same CPS in distinct ways for different missions. For example, if an unmanned aerial vehicle (UAV) is always used for reconnaissance and will never carry any weapons, then a vulnerability that only impacts the weapon system can be assigned with low mitigation priority. Second, since a CPS is usually composed of a large number of networked, interacting components (a.k.a. the system-of-systems nature), the impact introduced by a vulnerable component might be propagated across or tolerated by the entire CPS. Therefore, the vulnerability assessment method needs to consider interactions among all components rather than focusing on individual components. Third, CPSs are considerably dynamic as a result of redefined missions, added components, and newly-discovered vulnerabilities. This requires the vulnerability assessment method to be able to incorporate new knowledge into the analysis process without modifying the analysis algorithm.

## 1.2 THESIS STATEMENT

### 1.2.1 Objectives

This thesis is to build a framework capable of performing automated mission-aware vulnerability assessment for CPSs by systematically addressing the aforementioned challenges. Specifically, this framework aims at accomplishing the following design objectives.

a) Mapping missions into infrastructural components: given a CPS and its corresponding mission(s), the framework can automatically identify those

components whose proper operations are indispensable for the accomplishment the CPS mission(s). The components are named as mission-critical components in this paper.

b) Evaluating how vulnerabilities impact the entire CPS: given a CPS, its mission-critical components, and interactions among CPS components, the framework can automatically evaluate how vulnerable components collectively impact the CPS' mission-critical components.

c) High verifiability and flexibility: the framework needs to generate verifiable results. The framework will be sufficiently flexible so that new knowledge of the studied CPS (e.g., emerging vulnerabilities), which might be represented using richer semantics, can be incrementally integrated into the analysis process without modifying the analysis algorithm.

### 1.2.2 Solutions

In order to accomplish these design objectives, the design follows a model-assisted analysis strategy. This study takes advantage of CPS simulators, which are typically available for CPS test and verification, to model how CPS components behave under different missions. A mission is also modeled using a set of values assigned to relevant program variables of the CPS simulator. As a consequence, mission-critical components can be identified as those components that will be used by a CPS simulator given a set of its variable values. In addition, the framework facilitates a bottom-up approach to traverse a holistic model of a CPS that aims at profiling relationships among all CPS components. Based on this model, it can iteratively evaluate how each vulnerable component or a set of vulnerable components affect the performance of other components

and eventually mission-critical components. This study has employed formal methods (Clarke & Wing, 1996) to analyze the derived models. Specifically, it has leveraged symbolic execution (Ou, Govindavajhala, & Appel, 2005) to identify mission-critical components based on the CPS simulator and mission models. In addition, logic programming (Clocksin, Mellish, & Clocksin, 1987) is adopted to express the relationships among all CPS components and impact of each vulnerability. Finally, linear optimization (Bertsimas, Tsitsiklis , & Tsitsiklis, 1997; Bowen, 1993; Bowen, 1993) has been adopted to generate mitigation strategies that can protect all mission-critical components and meanwhile yield minimum cost. The formal method-based design plays a central role to accomplish the design objectives. On the one hand, the framework inherits mathematical rigor from the formal methods, rendering results with verifiable accuracy. On the other hand, the declarative nature of logic programming makes the framework highly extensible to incorporate new knowledge without modifying the analysis algorithm. This paper outlines the work-in-progress on this work, illustrates key techniques used in the framework, and discusses some open challenges and potential solutions.

## 1.3 RELATED WORK

### 1.3.1 Identifying mission-critical components

A few methods have been proposed (Musman, Tanner, Temin, Elsaesser, & Loren, 2011; Jakobson, 2011) to identify mission-critical components of a complex system. However, these methods have two significant limitations. First, these methods mainly rely on manual efforts to perform analysis, which is time-consuming and error-prone. Second, the built system dependency description is static and thus fails to characterize the dynamic

behaviors of the system. For example, the Crown Jewels Analysis method (Musman, Tanner, Temin, Elsaesser, & Loren, 2011) leverages the dependency maps, which are manually constructed, to associate mission objectives with cyber assets (e.g., the components).

### 1.3.2 **Discovering attack paths in computer networks**

A few methods (Sheyner & Wing, 2004; Ou, Boyer, & McQueen, 2006; Ou, Govindavajhala, & Appel, 2005) have been proposed to evaluate software vulnerabilities in computer networks by identifying attack paths. Similar to this framework, those approaches also employ formal methods. Sheyner et. al. used model checking (2004) and Ou et. al. adopted logic programming (2005; 2006). In spite of the similarity, the framework in this study differs from those methods in several ways. First, existing methods have not considered mission contexts in the assessment loop. Second, compared to existing methods that extensively concentrate on profiling the consequence of software vulnerabilities, this framework incorporates richer semantics such as the impacts introduced by data integrity and physical interference. Finally, unlike existing methods that require significant modification of the program interpreters, this study takes advantage of the built-in features of the programming language, which implies higher compatibility and lower maintenance cost.

# Chapter 2
# Technical Background of Formal Methods

Our model-based analysis method extensively leverages formal methods. Formal methods are techniques and tools used to model complex software and hardware systems as mathematical entities. With the mathematically rigorous model of a complex system, a system's properties can be verified in a more thorough fashion than empirical testing. Specifically, the mathematical rigor of formal methods requires all the statements in this methodology to be well-formed with mathematical logic. Formal verifications are rigorously deducted from that logic, which means each step has to follow a rule of inference and can be checked through a mechanical process (Kling, 1996). This framework involves two types of formal methods, including symbolic execution (King, 1976) and logic programming (Sterling & Shapiro, 1994).

## 2.1 SYMBOLIC EXECUTION

Symbolic execution is a formal method-based technique that is extensively used for static program analysis. It can represent the input of a program as symbolic variables and the program will execute symbolically instead of using concrete values of the input. As a result, each path of the program will be associated with a constraint that has to be satisfied to make the path executable. The constraint can be derived from the input symbolic variables (Boyer, Elspas, & Levitt., 1975).

In Figure 1, an example illustrates the functionality of symbolic execution, and how executable paths are discovered based on user-defined symbolic variables. When a variable is set up as a symbolic value, it will be calculated by a constraint solver to check whether it can satisfy the symbolic constraint at each branching point or not. If there is at

least one feasible solution, the symbolic execution will continue to discover further paths along current branching point. Otherwise the execution will stop on the current path and recursively work on next available path until all paths are analyzed (Cadar, Dunbar, & Engler, 2008; King, 1975). In this example program, integer variables ***x, y*** and ***z*** are assigned as symbolic values. Without the assumptions of the variables, they could be any integers during symbolic execution. For instance, the first branching point, the "Path 1" can be satisfied if ***x*** is an integer that is bigger than zero. Four paths can be discovered without assumptions. If the user assumes ***x*** is less than or equal to zero, ***y*** is less than or equal to 10, and ***z*** is greater than zero, the constraints of "Path 1" and "Path 2" can not be satisfied.

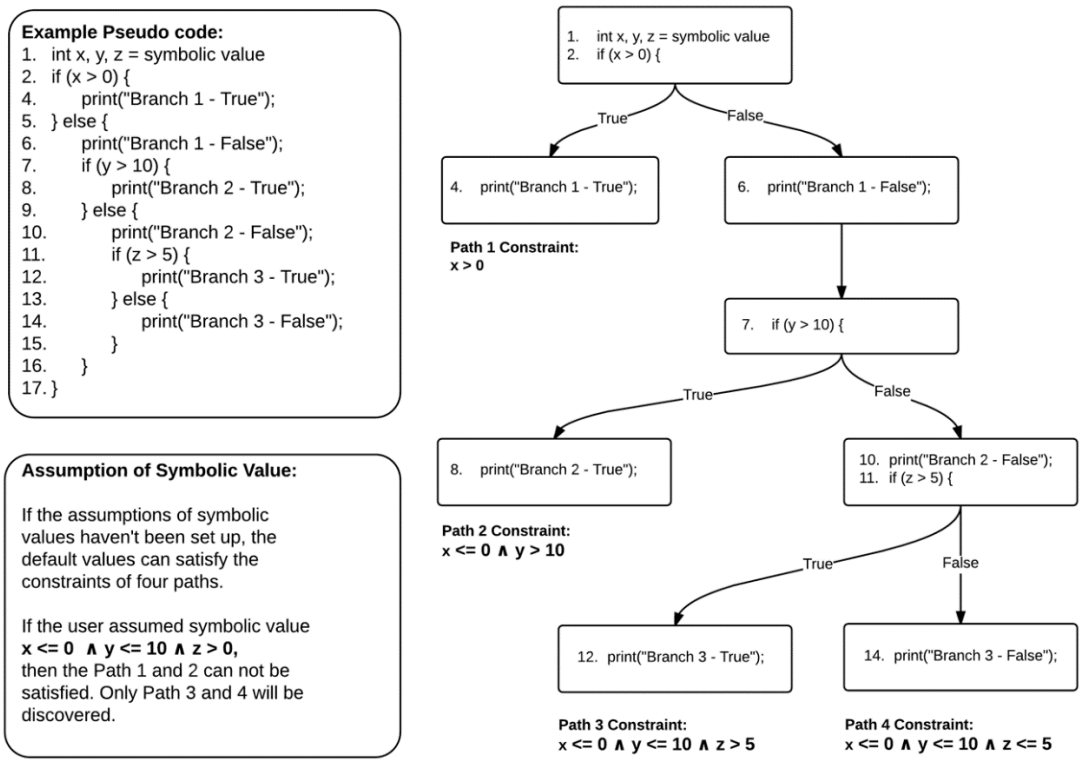


*Figure 1 A symbolic execution example*

Therefore, only "Path 3" and "Path 4" are executable with the assumptions, and functions including 'print ("Branch 1 – False")', 'print ("Branch 2 – False")', 'print ("Branch 3 –

True")' and 'print ("Branch 3 – False")' are associated with the symbolic constraint "$x<=0 \wedge y<=10 \wedge z>0$".

In this study, in order to identify mission-critical components and decrease the size of symbolic execution space, the specific mission assigns assumptions for the input variables of UAV simulator. Similar to the example in Figure 1, only executable paths and functions can be discovered and associated with its constraint, accordingly the corresponding components are identified as mission-critical components.

## 2.2 LOGIC PROGRAMMING

Logic programming is a declarative programming paradigm based on logical formulas. Prolog is a well-known implementation in the logic programming language family. An executable Prolog program normally consists of three parts including rules, facts and queries. A rule is a semantic representation written in the form of clauses, such as "*Head*: - *Body*". This can be understood as "If the *Body* is true, then the *Head* is true". It is a proposition which can be true or false depending on the facts. A fact is a special case of rules in which its *Body* is always true, so the facts are true.

Figure 1 presents an example of a Prolog program. Specifically, "David is the father of Solomon" and "Solomon is the father of Rehoboam" refer to facts, respectively. One rule is defined as "if A is the father of B and B is the father of C, then A is the grandfather of C". The query is expressed as "Who is the grandfather of Rehoboam?" and it is worth noting that "Who" (starting with capital "W") represents a variable. A Prolog interpreter can analyze the facts and perform automatic inference based on the given rules and queries automatically. In this particular example, the Prolog interpreter will identify the value (i.e., "david") for the variable "Who" (Sterling & Shapiro, 1994).

**Facts:**

father(david, solomon).
father(solomon, rehoboam).
father(rehoboam, abijah).

**Rule:**

grandfather(A, C) :- father(A, B), father(B, C).

**Query:**

grandfather(Who, rehoboam).

**Answer:**

Who = david.

*Figure 2 How does the Prolog run? - An example of a Prolog program*

For the purpose of searching in a knowledge base, a bottom-up or a top-down approach can be utilized to achieve a goal. The top-down approach starts processing information from the top of a predefined data structure to the bottom, but the bottom-up approach searches in the opposite way. To be more specific, the following example illustrates the difference between the top-down and bottom-up approach.

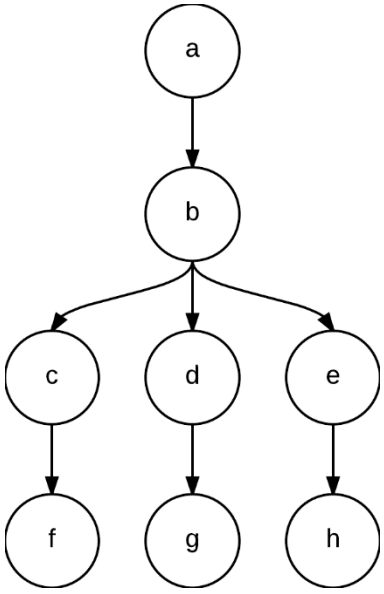


*Figure 3 The bottom-up approach V.S. the top-down approach - A family tree*

The Figure 3 presents a family tree that can be read as “a” is the father of “b”, “b” is the father of “c”, “d”, and “e”, and so on. Depending on these relations, if more semantic information is desired, the appropriate traversal strategy has to be defined. For instance, a query “Who is in the list of g’s predecessor?” can be asked. A top-down approach will traverse from “a” to “f”, from “a” to “g”, and from “a” to “h”, then the answer can be found. In this way, two redundant lists are discovered. However, the bottom-up approach can be facilitated to only generate desired answer. In Figure 4, both approaches are coded into a Prolog program.

```
%%     a, b, ..., and h are the names
%%     "father(a,b)" means a is b's father.
%%     Any values which begin with a capital letter are variables;
%%     Any values which begin with a lower case letter are concrete values;

Facts:
1.    father(a,b).
2.    father(b,c).
3.    father(b,d).
4.    father(b,e).
5.    father(c,f).
6.    father(d,g).
7.    father(e,h).

Rules:
1.    successor(Father,[Father,Succ]) :-
2.          father(Father,Succ).                  %% Is the Father the father of Successor?

3.    successor(Goal,[Goal|Successor_List]) :-
4.          father(Goal,Son),                     %% Is the Goal the father of Son?
5.          successor(Son,Successor_List).        %% Who is in Son's successor list?

6.    top_down_approach(Goal,Successor_List) :-
7.          successor(Goal,Successor_List),       %% Who is in Goal's successor list?
8.          last(Successor_List,Tail),            %% Who is the tail element in the list of successors?
9.          \+ father(Tail,_Son).                 %% The Tail of list has no son;

10.   predecessor(Son,[Predecessor,Son]) :-
11.         father(Predecessor,Son).              %% Is the Predecessor the father of Son?

12.   predecessor(Goal,Predecessor_List) :-
13.         father(Father,Goal),                  %% Is the Father the father of Goal?
14.         predecessor(Father,List),             %% Who is in the Father's successor list?
15.         append(List,[Goal],Predecessor_List). %% To append the Goal into the List;

16.   bottom_up_approach(Name,[Head|_Tail]) :-
17.         predecessor(Name, [Head|_Tail]),      %% Who is in Goal's predecessor list?
18.         \+ father(_Father, Head).             %% The Head of list has no father;
```

*Figure 4 The coded bottom-up and top-down approaches in a Program program*

The query guided by bottom-up approach achieves one desired list. Obviously the answers in Figure 5 demonstrate the bottom-up approach is more effective in this specific query.

**Query:**
top_down_approach(a, Successor_List) ?

**Answer:**
Successor_List = [a,b,c,f];
Successor_List = [a,b,d,g];
Successor_List = [a,b,e,h];

**Query:**
bottom_up_approach(g, Predecessor_List) ?

**Answer:**
Predecessor_List = [a,b,d,g];

*Figure 5 The queries and answers based on bottom-up and tom-down approaches*

Since the CPS's infrastructure model in this framework is transformed into a similar data structure, a bottom-up approach is facilitated to traverse a holistic model of a CPS that aims at profiling relationships among all CPS components. For instance, if the component *a* is a access point[1] and the component *g* is the target component[2] that the attacker wants to compromise, then the bottom-up approach discovers attack paths starting from the target component *g* up to the access point component *a*. Since the component *h* and *f* are not targets, traversing on them is wasting time. The bottom-up approach avoids traversing irrelevant paths in order to reduce the running time. In the meantime, the Prolog's backtracking mechanism guarantees that this approach can find all potential attack paths for the target component.

[1] The "Access point" in Figure 15 of Section 3.4.
[2] The "Component A" in Figure 15 of Section 3.4.

# Chapter 3
# The Mission-aware Vulnerability CPS Assessment Framework

## 3.1 ARCHITECTURE OVERVIEW

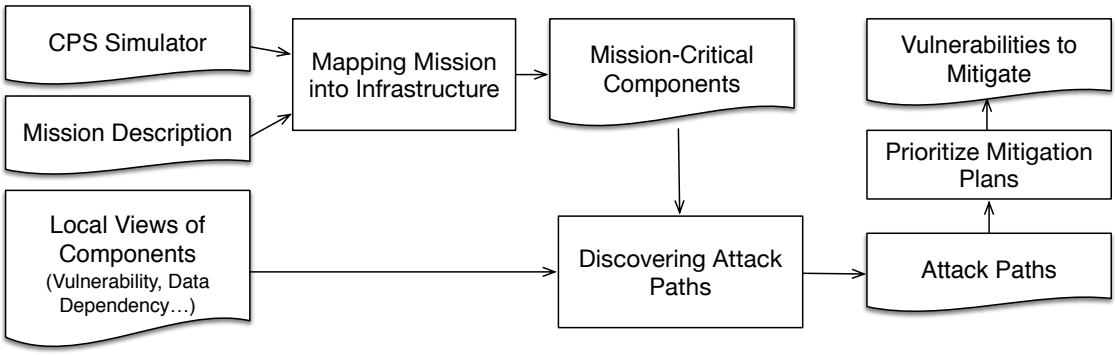


*Figure 6 The framework's architectural overview*

The architectural overview of this framework is presented in Figure 6. It takes the CPS simulator and the profile of each component as inputs. The input characterizes its internal security states and interactions with its neighbor components. The first phase of this framework aims at mining all mission-critical components. The second phase integrates the profile of each individual component and assesses how vulnerable components collectively disrupt mission-critical components. To be more specific, it discovers all potential attack paths, where each attack path represents a sequence of actions an attacker can take (e.g., exploiting a vulnerability) in order to affect the mission-critical components and ultimately disrupt the CPS' mission. Finally, given the cost to mitigate each vulnerable component, this framework will identify a set of vulnerable components to mitigate so that it can protect all mission-critical components while incurring minimum cost.

## 3.2 CONCEPTS AND DEFINITIONS

### *3.2.1 Cyber-physical system (CPS)*

A Cyber-Physical System (CPS) integrates computational elements, networking components, and physical processes. Salient examples of CPSs, include, but are not limited to, cardiac pacemakers, insulin pumps, automobiles with anti-lock braking system (ABS), and unmanned aerial vehicles (UAV). Different from traditional computing systems such as personal computers, a CPS is usually characterized by its systems-of-systems nature, (Lee, 2008), where a collection of (autonomous) components collaborate and interact to accomplish specific missions. This results in significant challenges for vulnerability assessment. Specifically, it is insufficient to assess vulnerabilities on each individual component. Instead, effective assessment needs to characterize the interaction of all components. In addition, missions have to be considered.

For example, an automobile usually consists of an embedded central controller, a speed sensor, an anti-lock braking system, a GPS, as well as a pump and a valve for monitoring and controlling the speed of the car. The central controller monitors the speed sensor at all times. If an out of ordinary deceleration is detected right before a wheel locks up, the wheel will be stopped quickly. The speed sensor and central controller may also interact with the GPS. If this car is a military vehicle and carries an automatic weapon system controlled by the central controller, it might be employed for a particular mission.

From the security perspective, vulnerabilities might exist on individual components, and might also exist on services between any two components. For instance, malicious users may have opportunities to compromise the central controller and GPS, in

order to disable the ABS during an anti-lock braking process. If a malicious user can achieve those successfully, there might exist attack paths from one component all the way down to the braking system. For particular missions, the weapon systems also have to be considered into the vulnerability assessment. However, if the weapon system will not be invoked in a mission, it will not affect the security of this CPS in this mission.

### *3.2.2 Mission*

A mission is a specific job or task which should be accomplished by the CPS. e.g., a UAV that takes 10 pictures at certain address. One challenge of this study is how to quantify a mission for a CPS. In the UAV case study, a mission is represented as a set of values which can be used as inputs for the UAV simulator. More details about missions can be found in Section 3.3.

### *3.2.3 Vulnerability*

In a CPS, a vulnerability is a weakness or a flaw which can be used by malicious users to compromise the integrity, availability, or confidentiality of the CPS (Microsoft, 2015). In this study, vulnerabilities can reside in services that are among components. In the automobile example, a malicious user might have an attack path from a GPS to a central unit, and from the central unit to a weapon system by compromising vulnerabilities among these components.

### *3.2.4 Service*

The interaction among multiple components can be characterized as services. In this framework, we consider two types of services: the control service and the data service. Control services can direct behaviors from one component to another. In Figure 7, a central processing unit sends a signal to turn on the camera. From the central

processing unit to the camera, there exists a control service. The data service can transfer information from one component to another, e.g., a temperature sensor can transfer temperature information to the central processing unit for calculation or to the database for storage. From a program analysis perspective, a service can be seen as a function or a module in a program which will be called by other components.

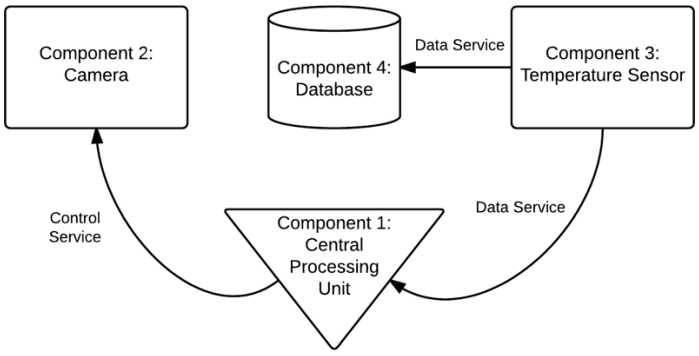


*Figure 7 Services and components in a CPS*

### *3.2.5 Component*

Any constituent element of a system should be considered as a component. The case study of a UAV in this thesis has a rough scope to define a component, so a few small components might be abstracted into single one. In this way, one component might have multiple services. However, the semantic meaning of the abstract model of the UAV will not be changed.

The UAV example in Figure 8 illustrates how the components are located on a UAV. Combining this with the UAV simulator, it can be abstracted into a model in Figure 9.

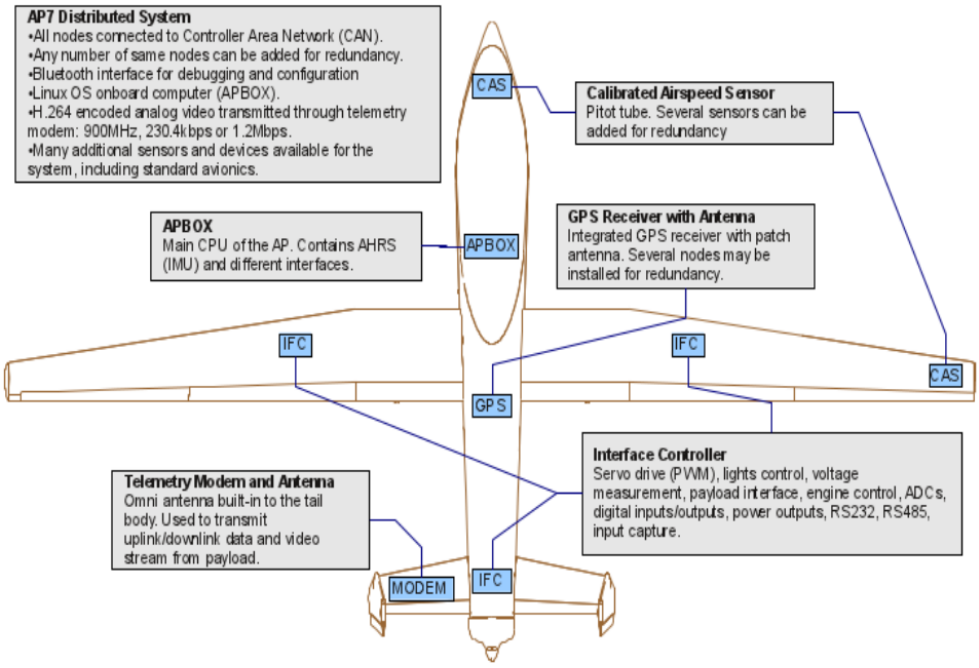


*Figure 8 The components on a example UAV CPS (UAVOS Company, 2014)*

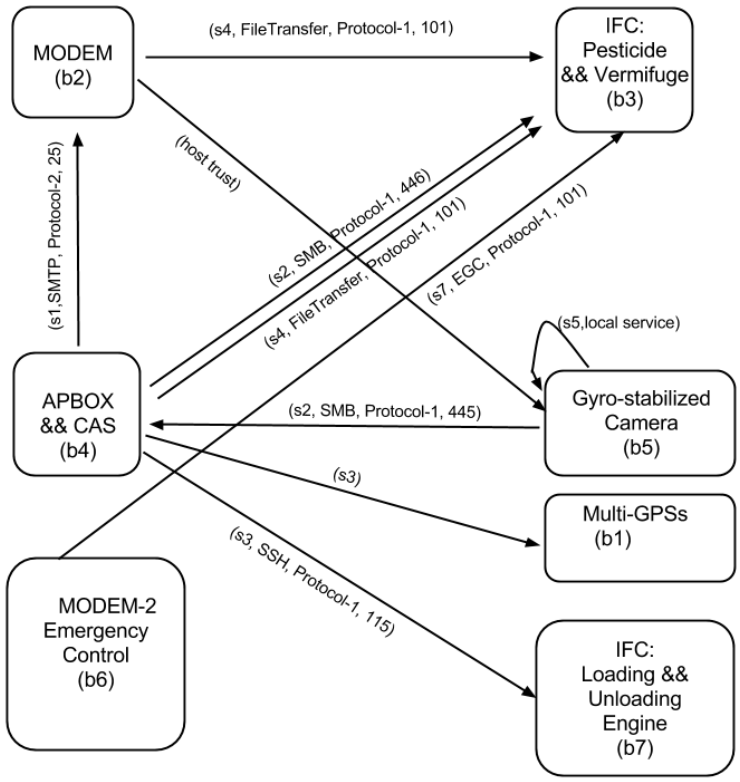


*Figure 9 The abstracted model of the UAV's services and components*

In Figure 9, only control and data services that exist among components are displayed. The physical impacts are also available among components in this study. Each component has its corresponding coordination on the UAV, from which the distance between two components can be calculated. When two components are physically close enough, one may affect another's functionality significantly. For example, if a GPS is over-heating and is close enough to a camera, and the camera is sensitive to high temperature, then the camera's capability may be undermined. A malicious user might take advantage of this interference to cause a failure of a mission.

#### *3.2.6 Mission-critical component*

As long as the component supports functionality to this mission and its failure will lead the mission failure or degradation, it will be considered as mission-critical component (DoD, 2013). For instance, if a camera on a UAV will be invoked in a mission, this camera must be one of the mission-critical components, since its damage or inaccuracy of its behavior may cause the failure of the mission.

### 3.3 MAPPING MISSION INTO INFRASTRUCTURE

In this thesis, a component is defined as mission-critical if it is used by the CPS to carry out a mission. Intuitively, if a mission-critical component malfunctions, the mission is highly likely to be disrupted. The objective of this phase is to discover all mission-critical components given a CPS simulator and the description of the mission(s). To this end, missions will be considered from a programmer's perspective, where a mission is corresponding to a set of values assigned to certain program variables of the CPS simulator. As a consequence, mission-critical components are identified as those

components that will be invoked by the CPS simulator given a set of values assigned to certain program variables.

The program variables for a CPS simulator can be generally categorized into two classes, namely the configuration parameters and the sensing signals. A configuration parameter refers to a parameter that is set by the CPS operator for a specific mission. In other words, the values of the configuration parameters are known before hand. In contrast, the value for a sensing-signal parameter can only be acquired at runtime. Consequently, all acceptable values are considered for sensing signal parameters.

Given program variables and their values, a straightforward approach to identify mission-critical components is to exhaustively execute the CPS simulator with all possible assigned values for all the program variables. Despite its simplicity, this approach may incur prohibitively high computation cost since all potential values for a set of program variables are likely to result in an enormous number of testing executions.

The program symbolic execution (King, 1976) is adopted to address this challenge. Specifically, this framework will first execute the CPS simulator symbolically. Each component in the simulator will be associated with a symbolic constraint, where a symbolic constraint is expressed by a collection of program variables (i.e., configuration parameters and sensing signals). A component will be invoked if any combination of the variable values make its associated constraint satisfied. To automate the satisfiability investigation process, a constraint solver is leveraged (e.g., a Satisfiability Modulo Theories solver). A pseudo-code snippet of an unmanned aerial vehicle (UAV) simulator presented in Figure 10 is used to demonstrate the design of this framework.

```
1.  ......
2.  // x_* and y_* defines the adversary area.
3.  static int x_low = 10000;
4.  static int x_high = 20000;
5.  static int y_low = 10000;
6.  static int y_high = 20000;
7.   ......
8.  start_camera(int x, int y){
9.       bool is_low_visibility;
10.      is_low_visibility = optic_sensor();
11.      if(is_low_visibility){
12.      IR_camera();
13.      } else{
14.           if(x >= x_low && x <= x_high && y >= y_low && y <= y_high){
15.                hi_res_camera();
16.           }else{
17.                regular_camera();
18.           }
19.      }
20. }
```

*Figure 10 The code snippet of an unmanned aerial vehicle (UAV) CPS*

The corresponding control flow graph and symbolic constraints of paths are displayed in Figure 11.

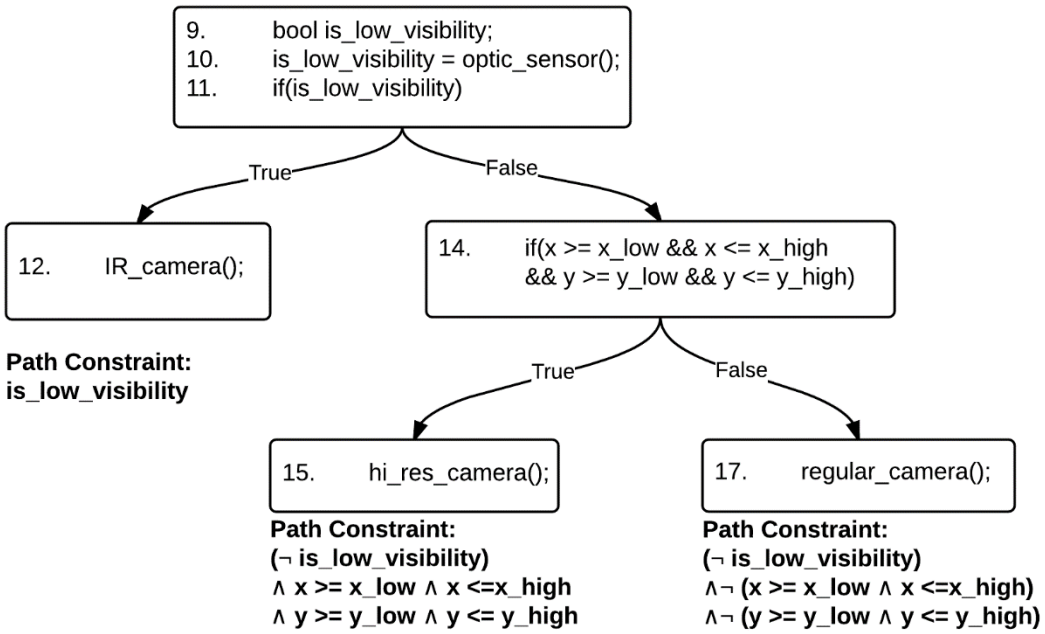


*Figure 11 Control flow graph and constraints of execution paths*

The function `start_camera(int x, int y)` is used to start the proper camera for the UAV based on the current coordinate of the UAV (i.e., `x` and `y`). Specifically, if the optical visibility is low (i.e., the "`optic_sensor()`" returns "true"), then the UAV will use its infrared camera (i.e., "`IR_camera()`"). Otherwise, it will use a high resolution optical camera or a regular optical camera. If the UAV flies over an adversarial area, which is defined by "`x_low`", "`x_high`", "`y_low`", and "`y_high`", it will use the high resolution camera (i.e., "`hi_res_camera()`"); otherwise, it will use a regular optical camera (i.e., "`regular_camera()`"). This study considers the scenario in which the adversarial area is stable. Therefore, the programming variables of this framework's concern include "`x`", and "`y`". For a specific mission, the UAV operator will assign values for "`x`" and "`y`". Comparatively, "`is_low_visiability`" will be initialized by the optical sensor at runtime.

Three missions are placed for the UAV, each mission representing a rectangular reconnaissance area. Each area is defined by the possible values for both "x" and "y", which are illustrated in Table 1 and further visualized in Figure 12. Since "`is_low_visiability`" is a sensing-signal variable, all possible values (i.e., "true" and "false") are provided. The advantage of using symbolic execution to identify mission-critical components becomes evident in this example. If the exhaustive execution strategy is followed, a large number of 9001×9001×2 execution paths are needed to explore all possible values for three variables. Comparatively, the symbolic execution technique adopted by this framework executes this program, and the build-in solver exams the three constraints (e.g., C1, C2, and C3 in Table 2) for three camera components, respectively, whether they can be satisfied by using the given symbolic

values “x” and “y” or not. Table 3 presents the decisions from a constraint solver which depends on the predefined symbolic values and possible values for “is_low_visiability” generated by the build-in solver as shown in Table 2.

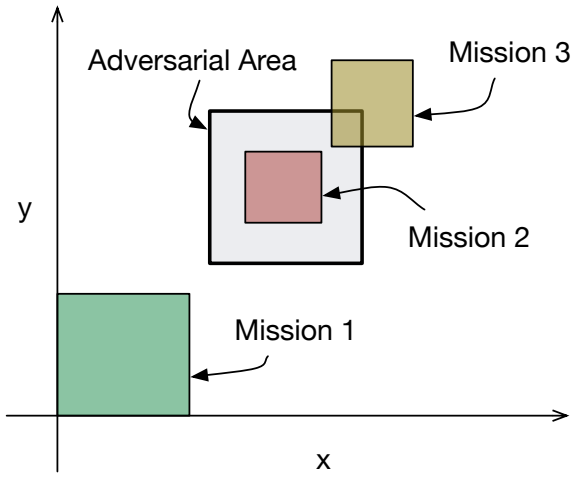


*Figure 12 Missions example*

| Mission | x's range | y's range |
|---|---|---|
| Mission 1 | x ∈ [0, 9000] | y ∈ [0, 9000] |
| Mission 2 | x ∈ [12500, 15000] | y ∈ [12500, 15000] |
| Mission 3 | x ∈ [19000, 25000] | y ∈ [19000, 25000] |

*Table 1 Configuration for three missions*

| Component | Constraint | Notation |
|---|---|---|
| IR_camera() | is_low_visiability | C1 |
| hi_res_camera() | x ∈ [10000, 20000] ∧ y ∈ [10000, 20000] ∧ ¬ is_low_visibility | C2 |
| regular_camera() | ¬(x ∈ [10000, 20000] ∧ y ∈ [10000, 20000]) ∧ ¬ is_low_visibility | C3 |

*Table 2 The constraint for each function*

| | C1 | C2 | C3 | Critical Components |
|---|---|---|---|---|
| Mission 1 | Y | N | Y | IR_camera() and regular_camera() |
| Mission 2 | Y | Y | N | IR_camera() and hi_res_camera() |
| Mission 3 | Y | Y | Y | IR_camera(), regular_camera(), and hi_res_camera() |

*Table 3 Constraints satisfied for each mission*

If a constraint of an execution path can be satisfied, its corresponding component will be invoked by the CPS system and therefore becomes the critical component. For example, since only C1 and C3 can be satisfied for Mission 1, "`IR_camera()`" and "`regular_camera()`" will be labeled as mission-critical functions, and its corresponding components are mission-critical components.

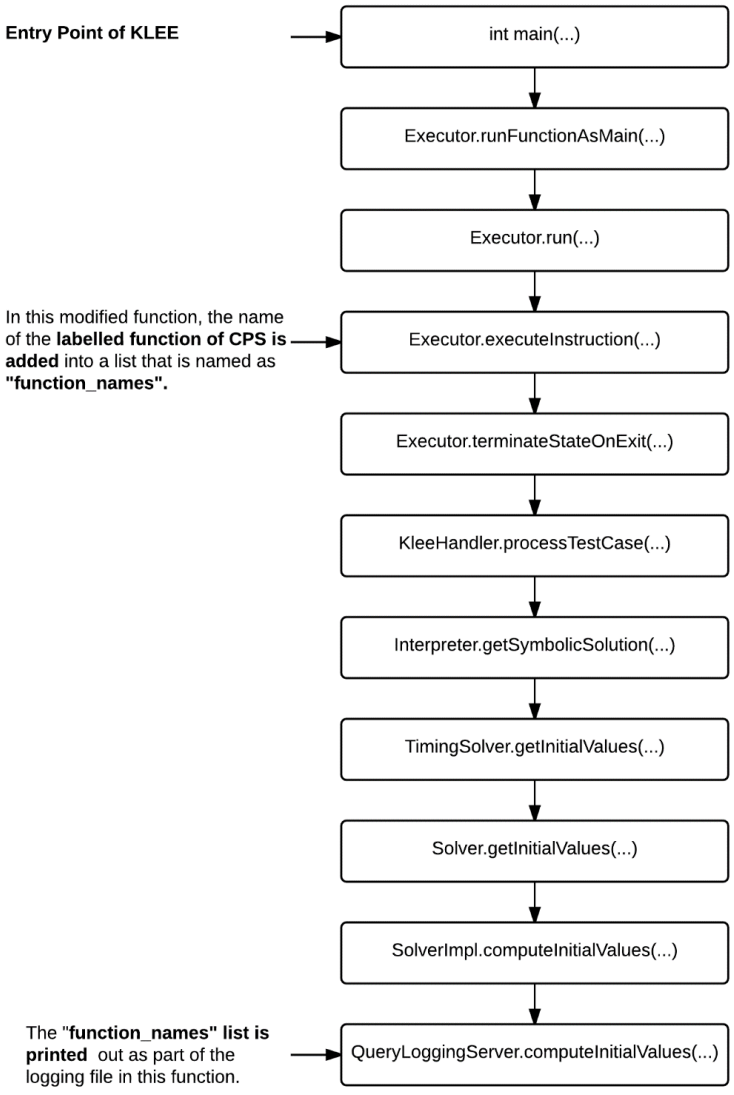


*Figure 13 The overview and call graph of the instrumented KLEE*

The KLEE (Cadar, Dunbar, & Engler, 2008; Baral & Gelfond, 1994), a symbolic virtual machine, is currently instrumented to implement this phase of this framework. Specifically, this framework augments KLEE so that it can label functions that represent CPS components. The instrumented KLEE can automatically collect constraints for each labeled function and subsequently perform verification using its built-in constraint solver. Figure 13 illustrates how the labeled functions which represent CPS components can be traversed and collected during symbolic execution.

**Instrumented Varialbe:**

A new variable "function_names" is created and updated with the Execution State. It will be used as storing the labelled function names of UAV simulator which are invoked during symbolic execution.

```
std::list<llvm::StringRef> function_names;
```

**Location:** /KLEE/include/klee/ExecutionState.h

**Instrumented Function:**

```
1426.  void Executor::executeInstruction(ExecutionState &state, KInstruction *ki) {
       ...
       // collecting the labelled function names of CPS
1428.  if (ki->function_name != NULL) {
1429.      state.function_names.push_back(*ki->function_name);
1430.  }
       ...
1444.  if (state.stack.size() <= 1) {
       ...
1446.      terminateStateOnExit(state);
       ...
}
```

**Location:** /KLEE/lib/Core/Executor.cpp

**Instrumented Function:**

```
183.       bool QueryLoggingSolver::computeInitialValues(..., std::list<llvm::StringRef> function_names) {
       ...
188.       //the functionality for this framework that outputs the labelled function names of CPS
189.       logBuffer << "Functions Executed:\n";
190.       for (std::list<llvm::StringRef>::const_iterator list_it = function_names.begin(), end = function_names.end(); list_it != end; ++list_it) {
191.           logBuffer << *list_it << "\n";
192.       }
193.       logBuffer << "\n";
194.       //continue with other functionality
       ...
}
```

**Location:** /KLEE/lib/Solver/QueryLoggingSolver.cpp

*Figure 14 Code snippet of instrumented functions of KLEE*

The inter-procedure call graph of instrumented KLEE illustrates how exactly the labelled function names are printed out from the entry point of the KLEE symbolic execution engine. In the instrumented KLEE, a list "`function_names`" that stores labelled function names is created and updated during symbolic execution process. In the call graph, some functions of KLEE are modified to fulfill the purpose of this framework. Two major instrumented functions are listed in Figure 14. The first modified function of KLEE is in charge of collecting each labelled functions name of CPS when each of them is invoked in an instruction. The second one is responsible of outputting the names of labelled functions in order to identify the corresponding mission-critical components. Accordingly, the mission is mapped into the infrastructure of the abstracted model of a CPS.

### 3.4 MINING ATTACK PATHS

After the mission-critical components have been identified, this framework evaluates how vulnerable components can collectively impact the mission-critical components. This thesis aims at discovering all attack paths that can lead an attacker from exploiting one or multiple access points to disrupting a mission-critical component via a sequence of malicious actions (e.g., exploiting security vulnerabilities). This framework aims at facilitating users toward following a bottom-up strategy to discovery attack paths.[3] Components are first individually characterized and then aggregated for

[3] The general idea of bottom-up strategy is demonstrated in Section 2.2.

holistic analysis. The current model profiles a CPS from the following two perspectives: internal security states and their relationships.

An attacker's privilege has been widely used by existing network-based vulnerability assessment methods (Sheyner & Wing, 2004; Ou, Govindavajhala, & Appel, 2005; Ou, Boyer, & McQueen, 2006) to represent security states of a computing node. Such representation reflects the consequence of successfully exploiting a software vulnerability (e.g., a buffer overflow vulnerability). However, it becomes insufficient for CPS vulnerability assessment since actions an attacker can take to disrupt a CPS mission are no longer limited by exploiting software vulnerabilities. For example, a mission-critical component, which is sensitive to high temperature, could be physically interfered with by another component, which is instructed by an attacker to generate an extensive amount of heat. In response, three types of security states are considered which include adversarial privilege, data integrity, and physical safety.

Similar to existing methods, the adversarial privilege state indicates the attacker's privilege, such as the typical "USER" or "ROOT" privilege. Comparatively, the new states, data integrity and physical safety, demonstrate the trustworthiness of the generated data and the physical condition, respectively. Since the internal states might not be independent, how an internal state affects another within a component is also characterized. For example, an attacker who obtains "ROOT" privilege on a component can contaminate its data integrity. Interactions with Other Components. The internal states of a component could be affected by other components. For example, if a component *A* takes output from another component *B* as its input and B's data is

contaminated by the attacker, then *A*'s data integrity will be effected and its missions are highly likely to be subsequently disrupted.

Similar scenarios are applicable to those components that interfere with each other physically. Vulnerable services can also serve as stepping stones for an attacker to gain privilege on a component from its victimized neighbor.

The BProlog (Zhou, 2014) language is adopted, a popular logic programming (Clocksin, Mellish, & Clocksin, 1987) language, to express the model and implement the analysis algorithm. The Prolog-based implementation builds two important features into this framework. First, the declarative nature of Prolog facilitates us to incorporate new semantics (e.g., semantics to describe physical safety with finer granularity) into the model. Second, the method in this framework generates verifiable, accurate results due to BProlog's optimized design for exhaustive search (i.e., it can reveal all attack paths).

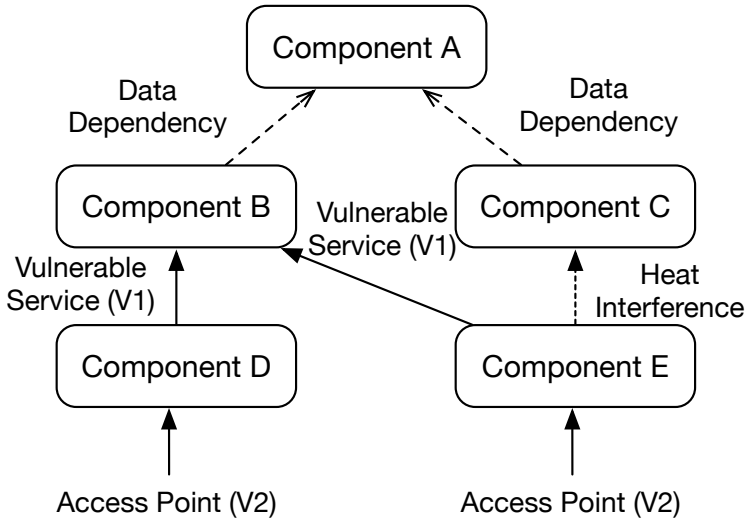


*Figure 15 Modeling an example CPS*

It is worth noting that this implementation takes advantage of language-level features of Prolog rather than manipulating its internal implementation to produce attack

paths (Sheyner & Wing, 2004; Ou, Boyer, & McQueen, 2006; Ou, Govindavajhala, & Appel, 2005). As a result, this implementation can naturally benefit from the evolution of Prolog interpreters such as the integration of distributed or parallel computing infrastructures (Côrte-Real, Dutra, & Rocha, 2013), implying low engineering and maintenance cost in the practical deployment.

Similar to Figure 7, Figure 15 represents a fraction of a CPS to illustrate the design of this framework, where the relationships among components are annotated in it. For example, the component *A* relies on the output from both *B* and *C* for operation; *B* offers a vulnerable service to *D*; *C*'s operation can be interfered with by high temperature from *E*. In Figure 16, the Prolog code presents the Prolog-based model of this UAV CPS example. "PART 1 – Facts" of the code refers to the structure of the CPS and the vulnerabilities of the system, which are presented as Prolog "facts;" "PART 2 – Rules" describes the impact of vulnerabilities, data dependency and physical interferences, which are presented as Prolog "rules." Rules can either be applicable for a specific component (e.g., line 12 - 14) or all components (e.g., line 15 - 36). A Prolog rule is a Horn clause represented as "$H$ :- $B_1, B_2, ..., B_n$", which means that "if $B_1$, $B_2$, ...and $B_n$ are all true, then $H$ is true". In the context of vulnerability assessment, $B_1$, $B_2$, ...and $B_n$ indicate the conditions necessary for an attacker manipulate internal security states of a component; $H$ indicates the consequence of the manipulation. For instance, line 23 - 26 means that the data integrity of a component (denoted by the variable Target) is compromised if this component has data dependency on another component (denoted by the variable Source) which means Source's data integrity has been contaminated. It is worth noting that Prolog's backtracking mechanism is leveraged to discover attack paths.

The backtracking mechanism is for finding all answers for a query. In this example, "Path" in line 25 indicates the attack path that lead to the data damage on the Source component and "[(Target, 'data-dependency') | Path]" recursively attach the current vulnerability to the head of the attack path.

```
1.  % PART 1 -- Facts:
2.  % The CPS structure
3.  data_dependency(a,b).
4.  data_dependency(a,c).
5.  service(b, d, vuln1).
6.  service(b, e, vuln1).
7.  heat_interference(c, e).
8.  access_point(d, vuln2).
9.  access_point(e, vuln2).

10. % PART 2 -- Rules:
11. % The relationships among internal security states of a component
12. emit_heat(Target, [(Target, 'emit-heat')|Path]) :-
13.     attacker(Target,root,Path),
14.     \+ member((Target,'emit-heat'),Path).

15. data_damage(Target, [(Target,'heat-data')|Path]) :-
16.         data_dependency(Target, Source),
17.         over_heat(Source, Path),
18.         \+ member((Target,'heat-data'), Path).

19. data_damage(Target, [(Target, 'root-data')|Path]) :-
20.         attacker(Target, root, Path),
21.         \+ member((Target,'root-data'),Path).

22. % The relationships across different components
23. data_damage(Target, [(Target, 'data-dependency')|Path]) :-
24.         data_dependency(Target, Source),
25.         data_damage(Source, Path),
26.         \+ member((Target,'data-dependency'), Path).

27. attacker(Target, root, [(Target,'vuln2'), accessPoint]) :-
28.         access_point(Target,vuln2).

29. attacker(Target, root, [(Target, 'vuln1')|Path]) :-
30.         service(Target,Source,vuln1),
31.         attacker(Source, root, Path),
32.         \+ member((Target,'vuln1'),Path).

33. over_heat(Target, [(Target,'heat-interference')|Path]) :-
34.         heat_interference(Target, Source),
35.         emit_heat(Source, Path),
36.         \+ member((Target,'heat-interference'), Path).
```

*Figure 16 An illustrative model - Prolog Code*

**Query:**

data_damege(a, Path).

**Answer:**

Path = [(a,'heat-data'),(c,'heat-interference'),(e,'emit-heat'),(e,vuln2),accessPoint]

Path = [(a,'data-dependency'),(b,'root-data'),(b,vuln1),(d,vuln2),accessPoint]

Path = [(a,'data-dependency'),(b,'root-data'),(b,vuln1),(e,vuln2),accessPoint]

*Figure 17 Query and answer*

In order to discover all attack paths, a user can first define the disruption state for a mission-critical component and execute a query. For example, if define the disruption of the component A as the violation of its data integrity, then a user can execute the query "data_damage(a, Path)", where Path is a variable that contains all attack paths. In this specific example, Path contains the attack paths which are shown in Figure 17. The second path is

"Access Point -v2→ Com. D -v1→ Com. B -root-data→ Com. B -data-depen.→ Com. A"

## 3.5 Optimizing Mitigation Plans

In this phase, the objective of this framework is to identify a set of vulnerable components so that mitigating them can i) protect all mission-critical components from being disrupted and ii) minimize the mitigation cost. The discovered attack paths with 100% coverage in the previous phase greatly facilitate the mitigation process. If mitigating a set of vulnerable components can block all attack paths, then all mission-critical components will be protected. Hence, this framework formulates this problem based on the derived attack paths. The variable $v_i$ is used to denote a vulnerable component and variable $c_{v_i}$ represents the cost to mitigate this vulnerability. Each attack

path (e.g., $p_k$) is represented by a set of its composing vulnerable components (i.e., $p_k = \{v_m^k, v_n^k, \ldots, v_q^k\}$). The $P$ is used to represent all attack paths, where $P = \{p_1, p_2, \ldots, p_m\}$. The $V$ is used to denote a set of vulnerable components selected for mitigation. A function, namely *count(p)*, is introduced to count the number of vulnerable components to be mitigated in an attack path $p$, where $count(p_k) = |\{p_k \cap V\}|$. Then, the objective of this phase can be formulated as an optimization problem described as follows:

$$Given: \quad \{v_m^k, v_n^k, \ldots, v_q^k\}, \quad \left\{c_{v_1^k}, c_{v_2^k}, \ldots\right\}, \quad \exists p_k \in P \tag{1}$$

$$Find: \quad V \tag{2}$$

$$Minimize: \quad \sum_{v_i^k \in V} c_{v_i^k} * v_i^k, \qquad \forall v_i^k \in p_k, \ \exists p_k \in P \tag{3}$$

$$Subject\ to: \quad count(p_k) \geq 1 \tag{4}$$

$$count(p_k) = \sum_{v_i^k \in V} v_i^k \ , \ \forall v_i^k \in p_k, \quad \exists p_k \in P \tag{5}$$

This is a typical linear optimization problem and the BProlog's build-in solver is leveraged to seek a feasible optimum solution for the mitigation plans.

### 3.6 GRAPHIC USER INTERFACE

In order to improve the usability and flexibility, a graphic user interface (GUI) is designed to integrate the functionalities of this framework. This interface facilitates that a user can easily extend the knowledge base of the infrastructure and the vulnerabilities of CPS without modifying the reasoning strategy. Five sections including configuration editor, component editor, trust relation editor, access point editor and service editor are employed to read infrastructure information from the user. The vulnerability editor section reads the details of each vulnerability from the user. The configuration editor is

used to describe the services and relationships among components.

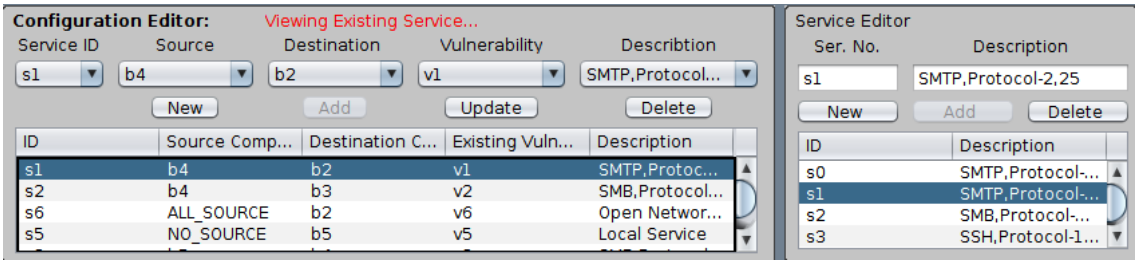


*Figure 18 Configuration editor screenshot*

In Figure 18, the first row on the table can be read as there exists a service *s1* from component *b4* to component *b2*, and the service *s1* has a vulnerability *v1*. The service editor can read service description from the user. (e.g., s1 is SMTP service which has an opening port 25). Similar to the service editor, the component editor records the component's description, the services running on it, and its trusting components.

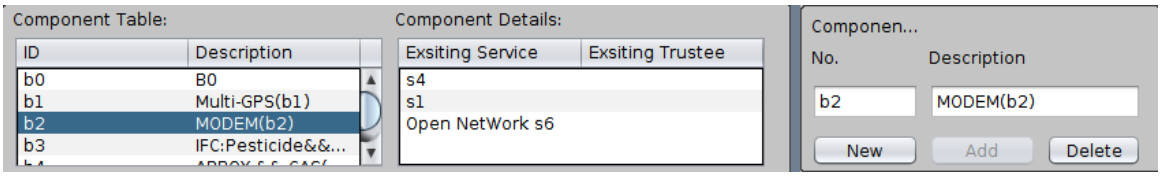


*Figure 19 Component editor screenshot*

In Figure 19, component b2 is a MODEM that has three services including *s4, s1*, and *s6*. The trusting relation can be defined by the trust relation editor in Figure 20.

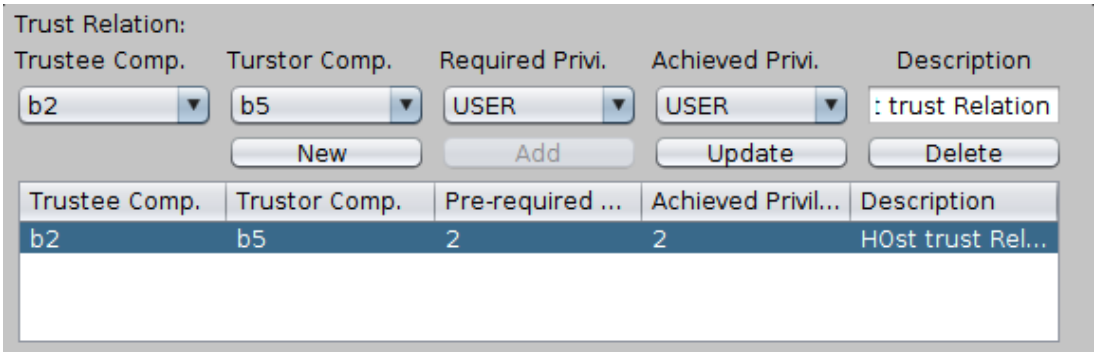


*Figure 20 Trust relation editor screenshot*

The first row of the trust relation table represents that if a user achieves USER privilege on component *b2*, then he or she can immediately get USER privilege on *b5* by this host trust relation.

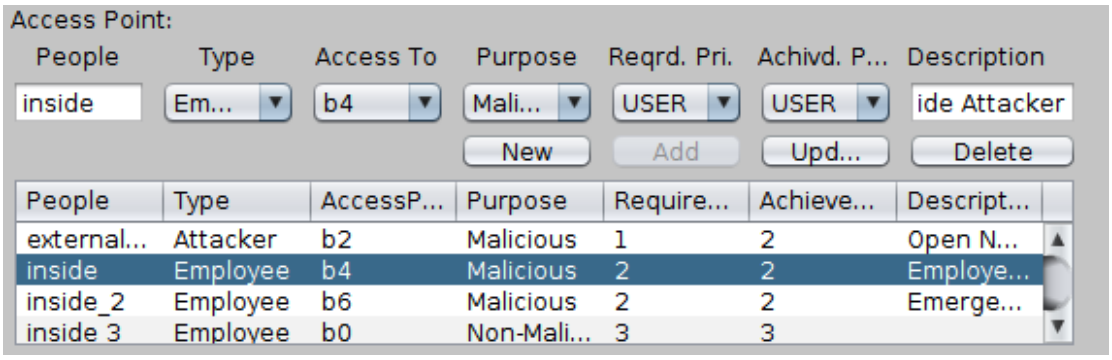


*Figure 21 Access point editor screenshot*

Users can also assume the access points of a system in the access point editor. In Figure 21, the user assumes that a malicious employee could achieve USER privilege on component *b4*, if and only if when this employee has USER privilege in this system. All above editors are designed to build up and extend the infrastructure of a system. The vulnerability editor allows the vulnerability to be integrated into the analysis process. In Figure 20, the user inserted a vulnerability *v1* that can be compromised by the malicious users to achieve a USER privilege if he or she already had GUEST privilege. In the configuration editor, a vulnerability can be attached to its corresponding services.

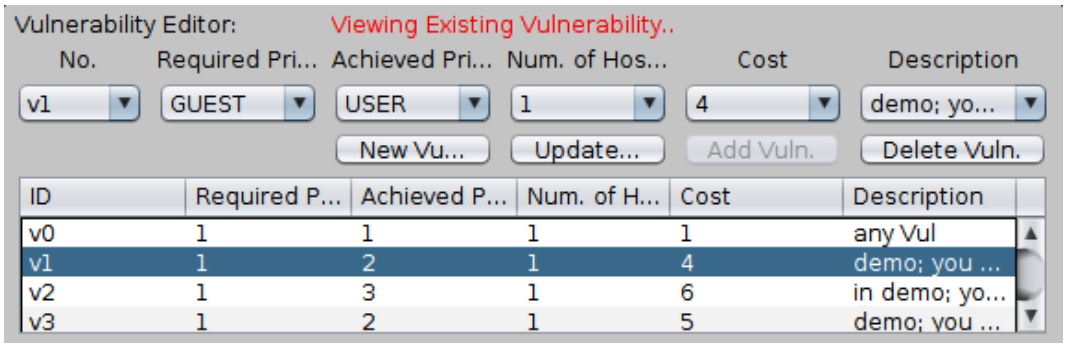


*Figure 22 Vulnerability editor screenshot*

The cost of mitigating each vulnerability can be defined by the user. It is used to determine the minimum cost mitigation plan. For example, in Figure 22, to mitigate a vulnerability *v1* requires 4 units cost.

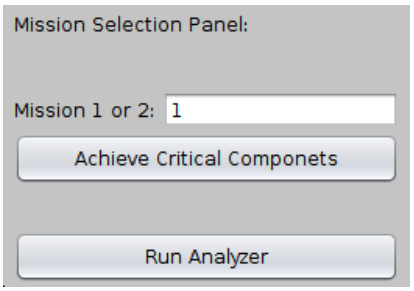


*Figure 23 Control panel screenshot*

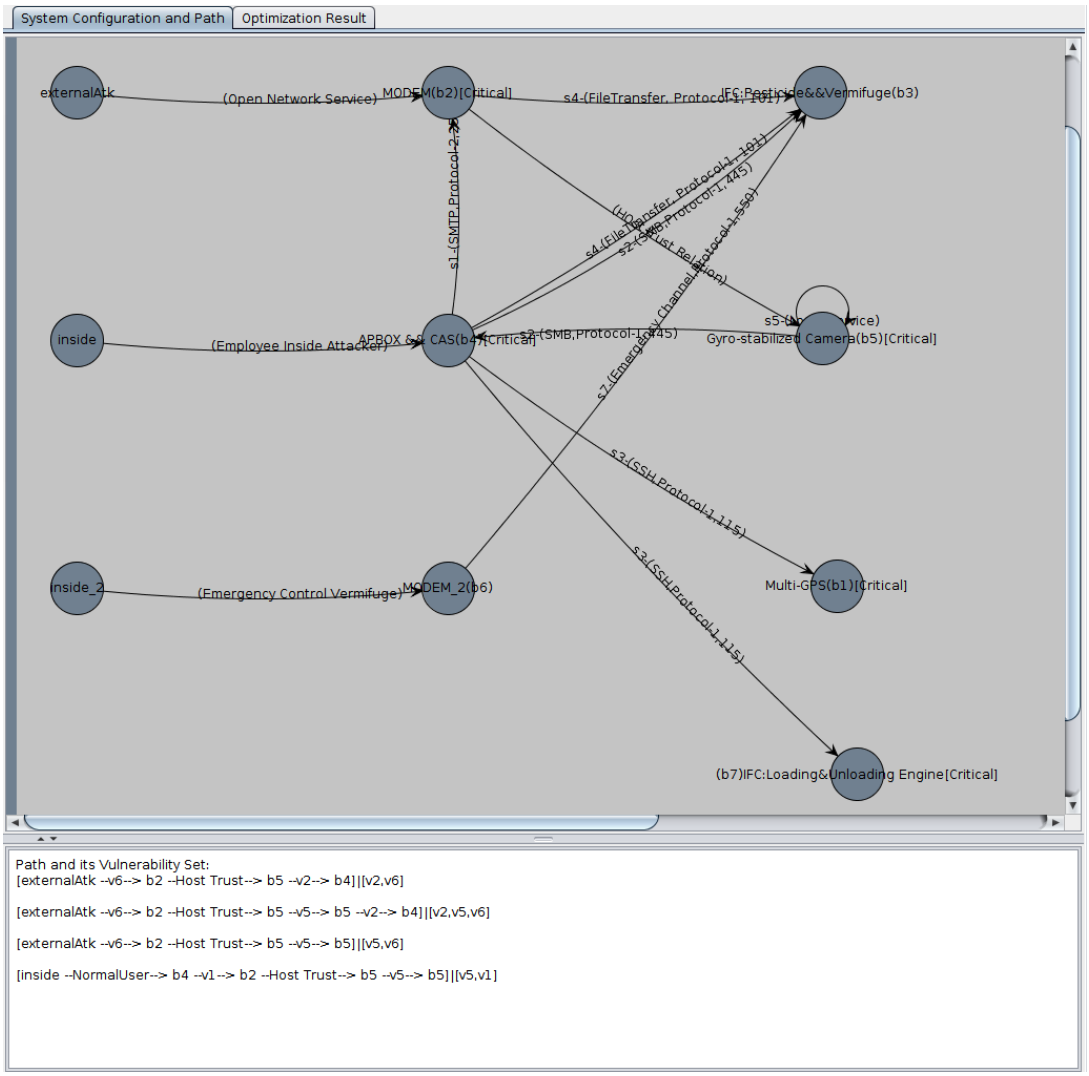


*Figure 24 Visualized system configuration and attack paths*

Once all infrastructure and vulnerability information is inserted into this framework, the user can choose the mission and run analyzer to achieve the results. In this demo, two missions are pre-defined to illustrate the design idea[4]. After clicking the "Run Analyzer" button in Figure 23, the visualized system configuration is displayed on the output panel. In Figure 24, the system configuration illustrates the same abstracted model as the UAV model in Figure 27. The attack paths are displayed at bottom of Figure 24. This interface also output the optimized mitigation plan(s) if there exists at least one feasible solution. Based on the cost assumption (e.g., random integers between 0 and 10) in this study, one feasible optimum solution is achieved by the build-in solver of BProlog in Figure 25. Patching vulnerabilities *v2* and *v5* is the minimum cost mitigation plan that needs 8 units cost. This mitigation plan can destroy all existing attack paths.

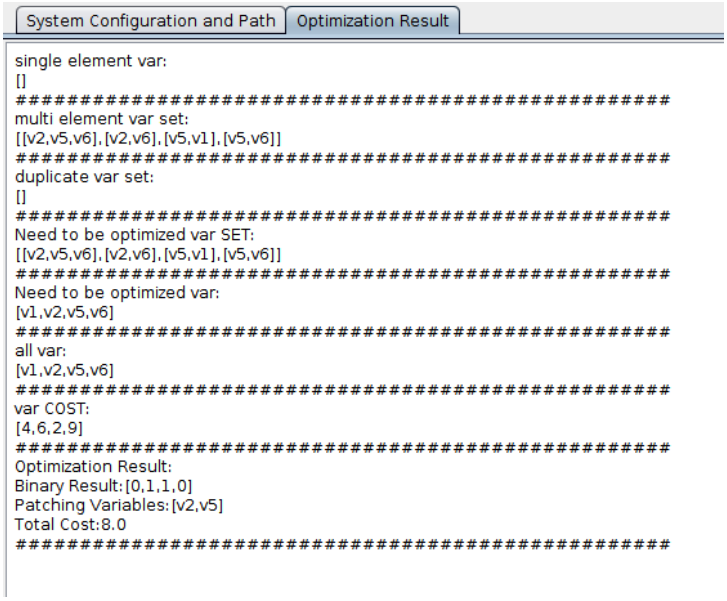


*Figure 25 Optimized mitigation plan screenshot*

[4] These two missions can be found in Section 3.7.1.

The panoramic screenshot of this interface is displayed in Figure 26. Cooperating with this GUI benefits the user to build up and enrich the infrastructure information and achieve desired results without worries about the underlying technologies.

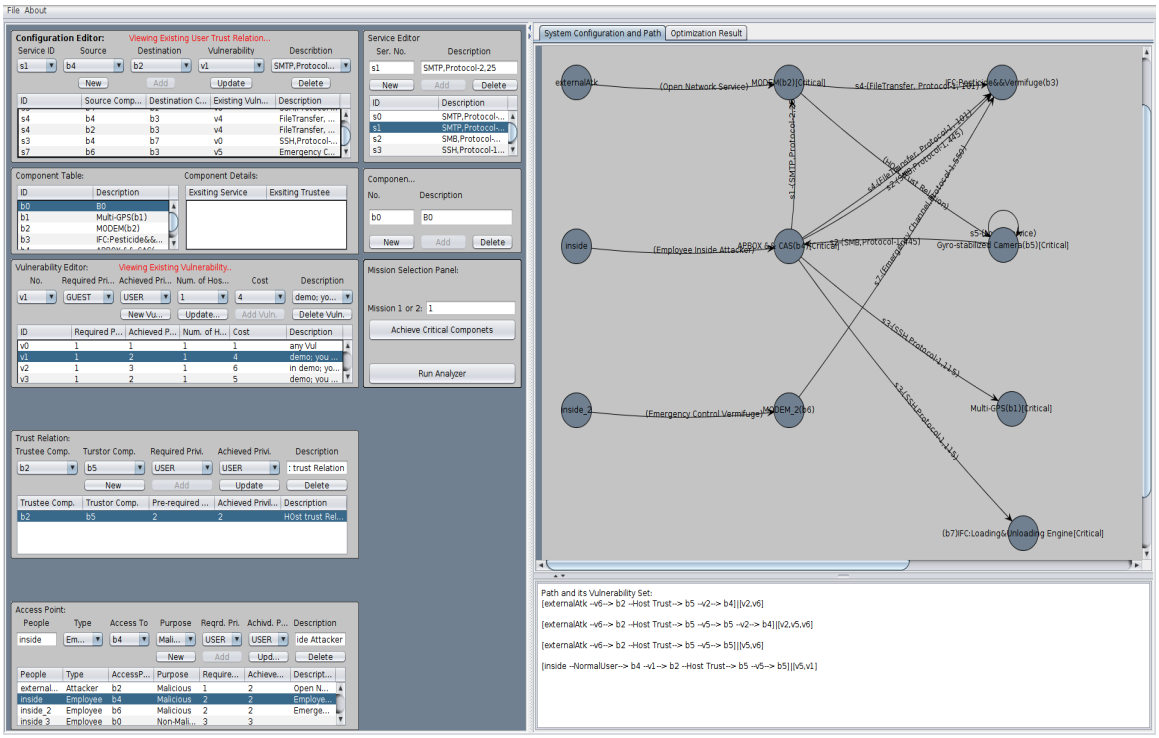


*Figure 26 Panoramic screenshot of GUI*

## 3.7 FRAMEWORK EVALUATION

### 3.7.1 Effectiveness

A UAV simulator has been build to evaluate the vulnerability assessment framework. The simulator, written in C++, represents the integration of the interacting components. It can be abstracted into a model which is illustrated in Figure 27. Its proper operation relies on both the configuration parameters and sensing signals. The configuration parameters describe the cruise routes of the UAV and adversarial areas; the sensing signals capture the weather conditions, which are randomly generated during the simulation. Different components will be used when the combination of the current

position of the UAV, its associated adversarial status, and the weather condition. The route map and mission list can be found in Figure 28.

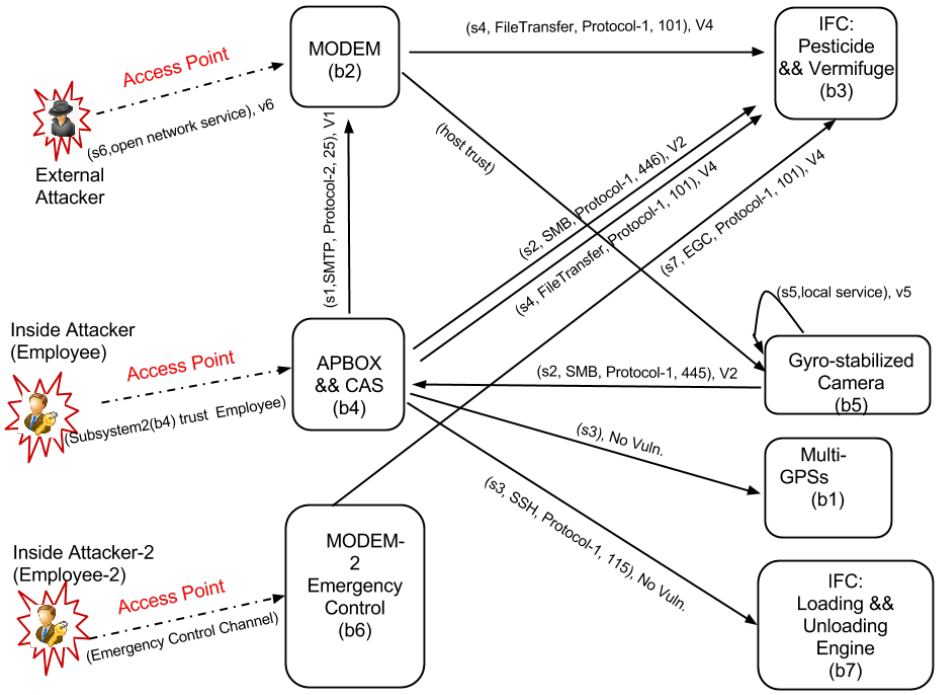


*Figure 27 Model of the UAV simulator*

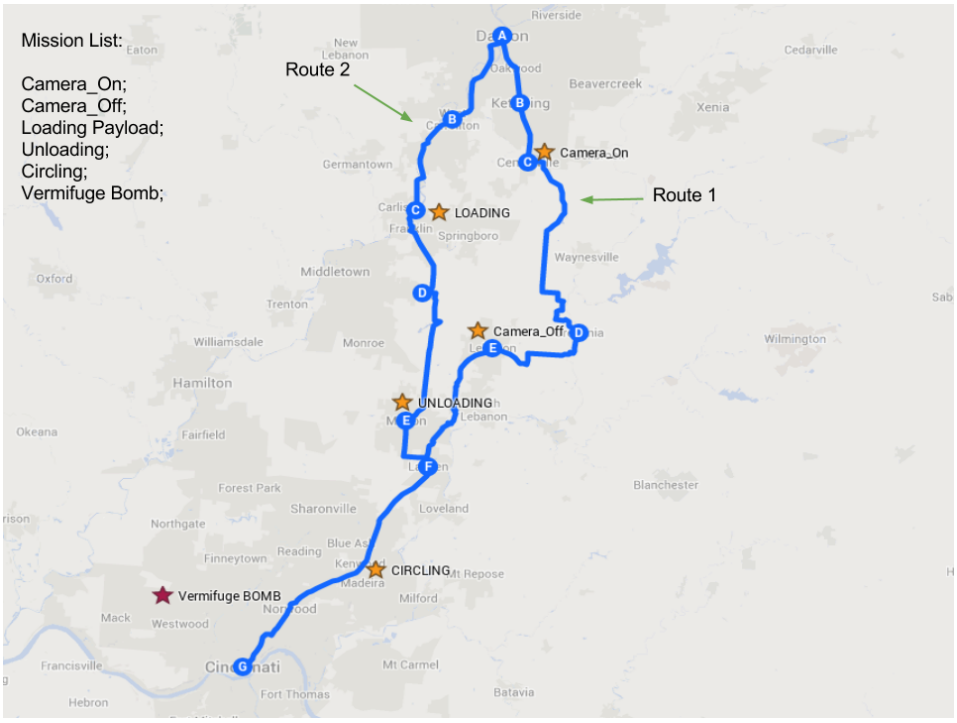


*Figure 28 Missions and route map for the UAV simulator*

A mission is associated with a set of possible values for both the cruise route and the coordinates of adversarial areas; all possible values are assigned to each sensing signal variable. In Table 4 and 5, the input values of missions can be found for both Route 1 and 2. The assume ranges of coordinates for Position 2, 3, 4, and 5 are also listed in Table 4 and 5.

| Position No. | Latitude | Longitude | Altitude |
|---|---|---|---|
| 1 | 397589478 | -841916069 | 225 |
| **2** | **396894936 < v0_0 < 396895136** | **-841688374 < v0_1 < -841688174** | **296 < v0_2 < 316** |
| 3 | **396283828 < v1_0 < 396284028** | **-841593918 < v1_1 < -841593718** | **301 < v1_2 < 321** |
| **4** | **394508837 < v2_0 < 394509037** | **-840960552 < v2_1 < -840960352** | **202 < v2_2 < 222** |
| 5 | **394353273 < v3_0 < 394353473** | **-842030022 < v3_1 < -842029822** | **224 < v3_2 < 244** |
| **6** | 393120031 | -842829935 | 247 |
| 7 | 391908926 | -843635507 | 233 |
| **8** | 391644798 | -844542798 | 186 |
| 9 | 391031182 | -845120196 | 167 |

*Table 4 Coordinates of route 1*

| Position No. | Latitude | Longitude | Altitude |
|---|---|---|---|
| 1 | 397589478 | -841916069 | 225 |
| **2** | **396722712 < v0_0 < 396722912** | **-842521732 < v0_1 < -842521532** | **208 < v0_2 < 228** |
| 3 | **395786685 < v1_0 < 395786885** | **-842979273 < v1_1 < -842979073** | **196 < v1_2 < 216** |
| **4** | **394928267 < v2_0 < 394928467** | **-842896714 < v2_1 < -842896514** | **258 < v2_2 < 278** |
| 5 | **393600486 < v3_0 < 393600686** | **-843099490 < v3_1 < -843099290** | **236 < v3_2 < 256** |
| **6** | 393120031 | -842829935 | 247 |
| 7 | 391908926 | -843635507 | 233 |
| **8** | 391644798 | -844542798 | 186 |
| 9 | 391031182 | -845120196 | 167 |

*Table 5 Coordinates of route 2*

The instrumented KLEE is used to identify all mission-critical components given a mission description and the the Prolog-based analysis engine subsequently enumerate all attack paths. While this framework by design inherits the mathematical rigor of SAT and Prolog, the manual analysis further verified that all mission-critical components and attack paths were correctly identified. All critical functions and critical components have been identified successfully which are listed in Table 6. After critical components are

identified, attack paths can be found by the Prolog program of this framework. The attack paths and corresponding vulnerability set are listed in Figure 29.

| **Route 2 Mission-Critical Functions** | | **Route 1 Mission-Critical Functions** | |
|---|---|---|---|
| (passive_antenna). | (engine_mode_normal). | (passive_antenna). | (engine_mode_normal). |
| (radio_modem). | (engine_voltage_sensor). | (bluetooth_signal). | (engine_voltage_sensor) |
| (bluetooth_signal). | (magnetometer). | (klee_assume). | (magnetometer). |
| (klee_assume). | (klee_div_zero_check). | (accZ). | (gps). |
| (accZ). | (gps). | (accY). | (gps_magnetometer). |
| (accY). | (gps_magnetometer) | (solar_energy_off). | (speed_detector). |
| (solar_energy_off). | (speed_detector). | (roll). | (system_voltage_sensor). |
| (roll). | (claw_engine). | (accX). | (yaw). |
| (accX). | (system_voltage_sensor). | (klee_make_symbolic). | (turn_off_Camera). |
| (radio_decoder). | (radar_dish_engine). | (solar_energy_on). | (camera_remote_off). |
| (klee_make_symbolic). | (yaw). | (weather_detector). | (accelerometer). |
| (radar_processor). | (puts). | (gyroscope_sensor). | (engine_mode_APBOX). |
| (ground_defense_signal) | (radar). | (normal_shutter). | (camera_remote_on). |
| (solar_energy_on). | (gyroscope_sensor). | (memcpy). | (pitch). |
| (accelerometer). | (memcpy). | (turn_off_night_mode). | (turn_on_Camera). |
| (engine_mode_APBOX). | (radio_battery_control). | (static_press._sensor). | (gps_STM32_microcontroller). |
| (pitch). | (backup_battery). | (circling). | (radio_signal). |
| (unloading). | (radio_wave_generator). | (moveTo). | (gps_1). |
| (gps_STM32_microcontroller) | (time). | (puts). | (is_night_mode). |
| (gps_1). | (static_pressure_sensor). | | |
| (circling). | (radio_analog_sensor). | | |
| (moveTo). | (rand). | | |
| (weather_detector). | (srand). | | |
| (pick_up). | | | |
| **Route 2 Mission-Critical Components** | | **Route 1 Mission-Critical Components** | |
| [apbox, cas, claw, gps, modem] | | [apbox, camera, cas, gps, modem] | |
| [b1, b2, b3, b4] | | [b1, b2, b4, b5] | |

*Table 6 Critical functions and components generated by instrumented KLEE*

**Route 1:**
**Critical Components: [b1,b2,b4,b5]**

**Path and its Vulnerability Set:**

**[people --v6--> b2 --Host Trust--> b5 --v2--> b4]|[v2,v6]**

**[people --v6--> b2 --Host Trust--> b5 --v5--> b5 --v2--> b4]|[v2,v5,v6]**

**[people --v6--> b2 --Host Trust--> b5 --v5--> b5]|[v5,v6]**

**[employee --NormalUser--> b4 --v1--> b2 --Host Trust--> b5 --v5--> b5]|[v5,v1]**

**Route 2:**
**Critical Components: [b1,b2,b3,b4]**

**Path and its Vulnerability Set:**

**[people --v6--> b2 --Host Trust--> b5 --v2--> b4]|[v2,v6]**

**[people --v6--> b2 --Host Trust--> b5 --v5--> b5 --v2--> b4]|[v2,v5,v6]**

*Figure 29 Critical components and attack paths*

### 3.7.2 Performance

The performance of finding attack paths by BProlog in this study has not encountered scalability issues. However, experiments that are more complicated are tested for evaluating the framework's scalability.

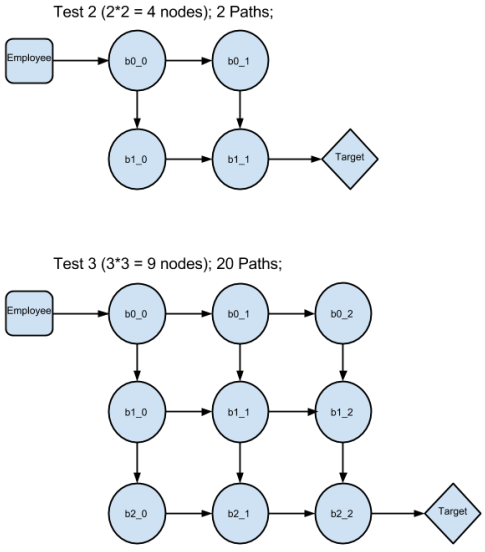


*Figure 30 The example topology of testing cases*

Test experiments are generated for testing the scalability of finding attack paths using the same bottom-up traversal algorithm. The sample topology is displayed in Figure 30. In this figure, an employee has the capability to compromise the target through an attack path, “Employee”, “b0_0”, “b1_0”, “b1_1” and “Target.”

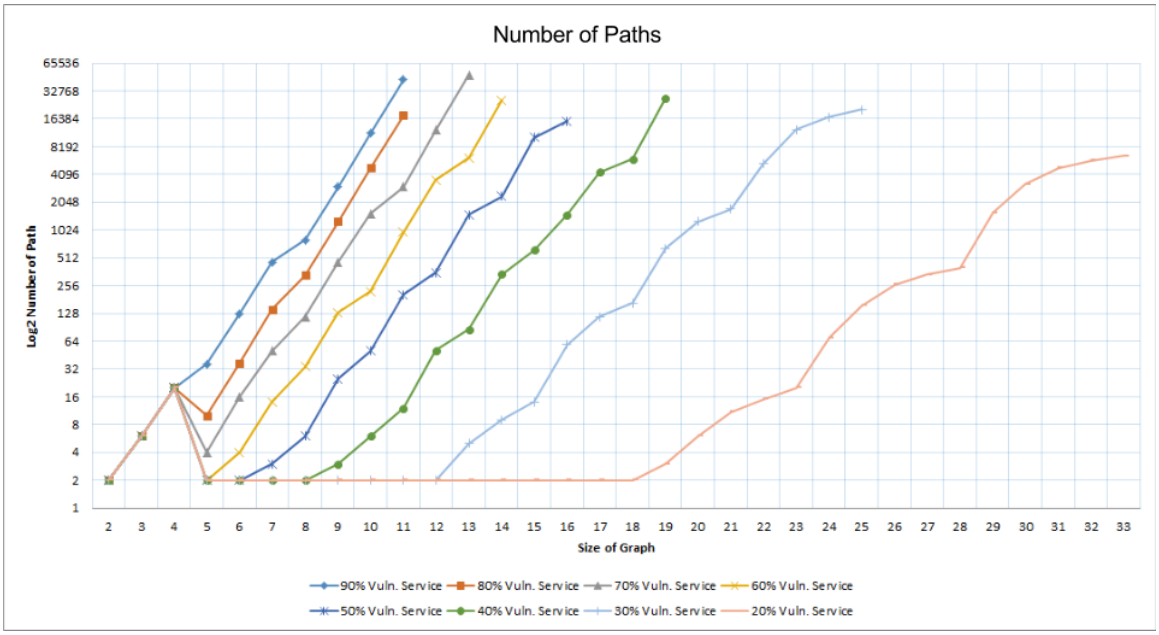


*Figure 31 Number of attack paths for the scalability experiments*

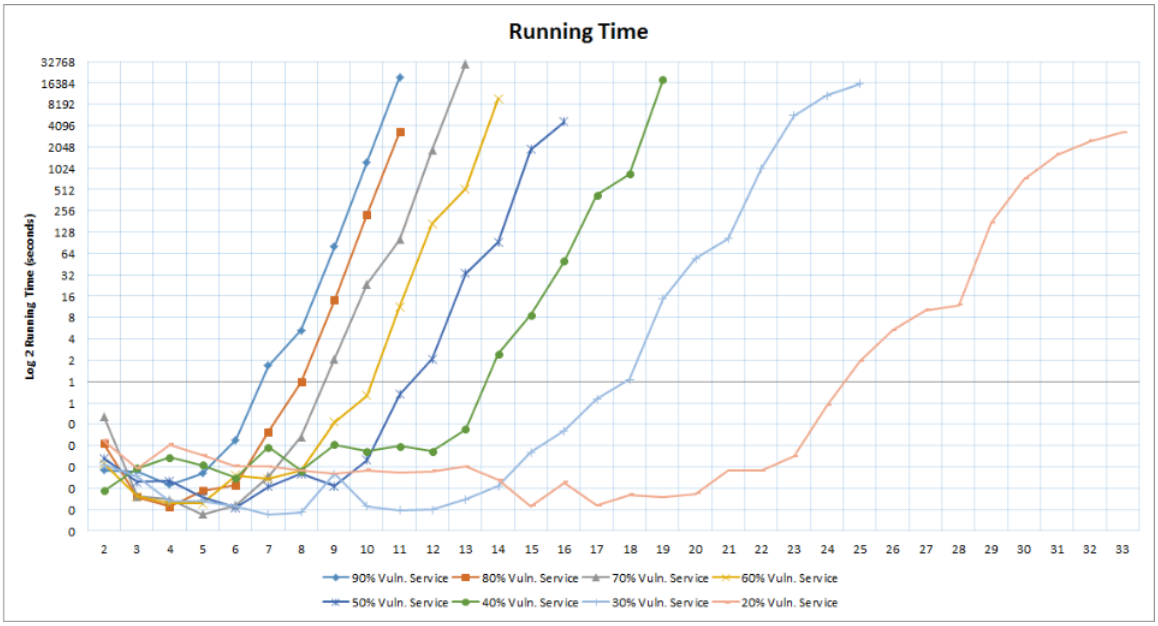


*Figure 32 Running time of finding attack paths for the scalability experiments*

As the number of nodes increases, the number of paths increases exponentially, which is shown in Figure 31. In order to demonstrate the number of vulnerabilities on components may also affect the scalability. The comparable experiments are created, so the number of vulnerable service is decreased in each topology by 10%, 20%, and so on. For example, there are 100 components in a topology. This topology with 10% vulnerability services represents that 10 of the 100 components contain vulnerable services. This decreasing strategy is also applied in the running time experiments. For instance, the left-most curve in Figure 31 represents that the number of attack paths increases significantly with the size of graph increases. Its topology contains 90% vulnerable services. The left-most curve in Figure 32 is the corresponding running time.

# Chapter 4
# Discussion and Further Work

This framework has systematically integrated program symbolic execution, logic programming, and linear optimization in order to perform mission-aware CPS vulnerability assessment. Despite the fact that this framework has demonstrated high accuracy and automaticity, a few open challenges have been observed. The discovery of mission-critical components will face great challenges if a CPS simulator yields complex constraints (e.g., sophisticated arithmetic constraints) that stay beyond the capabilities of constraint solvers. In response, a few solutions can be considered. First, if complex constraints involve a small number of variables that have small value spaces, this framework can still adopt execution-based approach. Second, domain knowledge (e.g., the relationships among variables) might be used to simply the model (e.g., reducing the value space for sensing signal sensors). For example, if an UAV takes a certain route for a mission, the wind speed might fall into a small range. In addition, the evolving techniques can be adopted for constraint solvers. Despite the fact that this framework offers high flexibility to incorporate richer semantics, exploiting such potential highly desires an interactive model development environment that can facilitate i) the communications among users responsible for different CPS components and ii) the sharing of rules applicable for multiple components. For example, if more fine grained security states are desired for a component, other relevant components need to be notified to redefine their relationships based on new states.

Model-driven design is gaining increasing popularity in building safety critical systems including CPS. Several languages have been defined to model various systems such as Unified Modeling Language (UML) (Warmer & Kleppe, 1999) and Architecture

Analysis & Design Language (AADL) (Feiler, Gluch, & Hudak, 2006) . How to extend this framework so that it can perform automated analysis using models in described in these languages is an area for the future work.

The solution to mitigation optimization focuses on reducing the cost of mitigations. However, more factors may influence the mitigation decisions. The likelihood for a vulnerable component to be taken advantage of by attackers is among the most important factors. For example, if a vulnerable component is associated with high mitigation cost but it is highly likely to be victimized, then it deserves high priority to be protected. Performing effective mission-aware vulnerability assessment plays a fundamentally important role for the massive deployment of CPSs. This framework outlines the work-in-progress towards this direction, where various formal method-based approaches have been systematically integrated to develop an effective solution. Future work will develop solutions to the aforementioned open challenges.

# REFERENCE


1) Baral, C., & Gelfond, M. (1994). Logic Programming and Knowledge Representation. *Journal of Logic Programming , 19*, 73--148.

2) Bertsimas, D., Tsitsiklis , J. N., & Tsitsiklis, J. (1997). *Introduction to Linear Optimization* (Vol. 6). Belmont, MA: Athena Scientific Belmont, MA.

3) Bowen, J. (1993). Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal , 8* (4), 189--209.

4) Boyer, R. S., Elspas, B., & Levitt., K. N. (1975). SELECT -- a formal system for testing and debugging programs by symbolic execution. *Proceedings of the International Conference on Reliable Software* (pp. 234-245). New York: ACM.

5) Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. *The 8th USENIX Conference on Operating Systems Design and Implementation* (pp. 209--244). Berkeley: USENIX Association.

6) Clarke, E. M., & Wing, J. M. (1996). Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR) , 28* (4), 626--643.

7) Clocksin, W. F., Mellish, C. S., & Clocksin, W. (1987). *Programming in PROLOG* (Vol. 5). Springer.

8) Collins, M. (1998). *Formal Methods.* Carnegie Mellon University. Pittsburgh: Carnegie Mellon University.

9) Côrte-Real, J., Dutra, I., & Rocha, R. (2013). Prolog Programming with a Map-reduce Parallel Construct. *Proceedings of PPDP'13.* Madrid, Spain: ACM.

10) DoD. (2013). Chapter 13 - Program Protection Plan. In DoD, *Defense Acquisition Guidebook.* Huntsville: Defense Acquisition University.

11) Feiler, P. H., Gluch, D. P., & Hudak, J. J. (2006). *The Architecture Analysis & Design Language (AADL): An Introduction.* Software Engineering Institute, Carnegie Mellon University. Pittsburgh, PA: Software Engineering Institute.

12) Jakobson, G. (2011). Mission cyber security situation assessment using impact dependency graphs. In *Information Fusion (FUSION), 2011 Proceedings of the 14th International Conference on* (pp. 1--8). IEEE.

13) King, J. C. (1975). A new approach to program testing. *The International Conference on Reliable Software. 10*, pp. 228--233. New York: ACM.

14) King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM , 19* (7), 385--394.

15) Kling, R. (1996). Systems safety, normal accidents, and social vulnerability. In R. Kling, *Computerization and Controversy (2nd ed.)* (pp. 746-763). Orlando, FL, USA: Academic Press, Inc.

16) Lee, E. A. (2008). *Cyber Physical Systems: Design Challenges.* EECS Department, University of California, Berkeley. Berkeley: University of California.

17) Microsoft. (2015, 06). *Definition of a Security Vulnerability*. Retrieved 06 01, 2015, from Mirosoft: https://msdn.microsoft.com/en-us/library/Cc751383.aspx

18) Musman, S., Tanner, M., Temin, A., Elsaesser, E., & Loren, L. (2011). A systems engineering approach for crown jewels estimation and mission assurance decision making. In *Computational Intelligence in Cyber Security (CICS), IEEE Symposium* (pp. 210--216). Paris: IEEE Symposium.

19) Ou, X., Boyer, W. F., & McQueen, M. A. (2006). A scalable approach to attack graph Generation. *Proceedings of the 13th ACM CCS* (pp. 336--345). Alexandria, Virginia, USA: ACM.

20) Ou, X., Govindavajhala, S., & Appel, A. W. (2005). MuLVAL: A Logic-based Network Security Analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium* (pp. 8--8). Berkeley, CA, USA: USENIX Association.

21) Sheyner, O., & Wing, J. (2004). Tools for Generating and Analyzing Attack Graphs. In *Formal Methods for Components and Objects* (Vol. 3188, pp. 344--371). Pittsburgh, PA, USA: Springer Berlin Heidelberg.

22) Sterling, L., & Shapiro, E. (1994). *The Art of Prolog (2nd ed): Advanced Programming Techniques.* Cambridge: MIT Press.

23) UAVOS Company. (2014, July 17). *Home*. Retrieved July 17, 2014, from Open System UAV: http://wiki.uavos.com/start

24) Warmer, J., & Kleppe, A. (1999). *The Object Constraint Language: Precise Modeling with UML.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

25) Zhou, N.-F. (2014, Feb 23). *B-Prolog User's Manual (Version 8.1) Prolog, Agent, and Constraint Programming.* Retrieved Feb 28, 2014, from B-Prolog: http://www.picat-lang.org/bprolog/