Rampo: A CEGAR-based Integration of Binary Code Analysis and System Falsification for Cyber-Kinetic Vulnerability Detection

Kohei Tsujio, Mohammad Abdullah Al Faruque, and Yasser Shoukry
Department of Electrical Engineering and Computer Science
University of California, Irvine
{ktsujio, alfaruqu, yshoukry}@uci.edu

***Abstract*—Cyber-physical systems (CPS) play a pivotal role in modern critical infrastructure, spanning sectors such as energy, transportation, healthcare, and manufacturing. These systems combine digital and physical elements, making them susceptible to a new class of threats known as *cyber kinetic* vulnerabilities. Such vulnerabilities can exploit weaknesses in the cyber world to force physical consequences and pose significant risks to both human safety and infrastructure integrity. This paper presents a novel tool, named Rampo, that can perform binary code analysis to identify cyber kinetic vulnerabilities in CPS. The proposed tool takes as input a Signal Temporal Logic (STL) formula that describes the kinetic effect—i.e., the behavior of the "physical" system—that one wants to avoid. The tool then searches the possible "cyber" trajectories in the binary code that may lead to such "physical" behavior. This search integrates binary code analysis tools and hybrid systems falsification tools using a Counter-Example Guided Abstraction Refinement (CEGAR) approach. In particular, Rampo starts by analyzing the binary code to extract symbolic constraints that represent the different paths in the code. These symbolic constraints are then passed to a Satisfiability Modulo Theories (SMT) solver to extract the range of control signals that can be produced by each of the paths in the code. The next step is to search over possible "physical" trajectories using a hybrid systems falsification tool that adheres to the behavior of the "cyber" paths and yet leads to violations of the STL formula. Since the number of "cyber" paths that need to be explored increases exponentially with the length of "physical" trajectories, we iteratively perform refinement of the "cyber" path constraints based on the previous falsification result and traverse the abstract path tree obtained from the control program to explore the search space of the system. To illustrate the practical utility of binary code analysis in identifying cyber kinetic vulnerabilities, we present case studies from diverse CPS domains, showcasing how they can be discovered in their control programs. In particular, compared to off-the-shelf tools, our tool could compute the same number of vulnerabilities while leading to a speedup that ranges from $3\times$ to $98\times$.**

I. Introduction

In an era where the backbone of modern society—power grids, transportation networks, healthcare facilities, and industrial control systems—rely extensively on cyber-physical systems (CPS), understanding and protecting against vulnerabilities has never been more critical. In particular, the convergence of physical and digital systems in critical CPS infrastructure has introduced new dimensions of risk, giving rise to a class of threats known as cyber kinetic vulnerabilities [1]–[3]. These vulnerabilities represent a unique breed of cyber-physical security challenges, where the exploitation of software vulnerabilities can lead to potentially catastrophic physical consequences. That is, cyber kinetic vulnerabilities blur the lines between digital and physical security, requiring a holistic approach that can analyze software controlling physical processes from the point-of-view of the induced physical or kinetic behavior.

This work was partially sponsored by the NSF award #2002405 and the C3.AI Digital Transformation Institute. The first author was partially supported by Japan Student Services Organization.

Unfortunately, there is currently a disconnect within the field of CPS formal verification and analysis. On the one side, the celebrated achievements in the software and hardware verification field [4]–[8] are inadequate for analyzing the dynamics of the underlying physical components of CPS. On the other hand, the techniques developed in the control theory community have focused mainly on the formal verification of the dynamics of physical/mechanical systems [9]–[13] while ignoring the complexity of the computational and cyber components. This paper aims to provide a comprehensive framework for tackling cyber kinetic vulnerabilities, one that not only identifies potential threats but also evaluates their physical consequences. In a rapidly evolving digital landscape, this integrated approach is crucial for protecting critical infrastructure and ensuring the safety and reliability of essential services for society at large.

This paper delves into the compelling realm of binary code analysis and falsification as an indispensable facet of safeguarding CPS against cyber kinetic vulnerabilities. Binary code, the low-level representation of software, is the critical interface where the virtual and physical worlds intersect. By subjecting binary code to rigorous scrutiny, researchers and security professionals can unearth vulnerabilities that are invisible at higher abstraction levels and thus lay the foundation for enhancing the resilience of critical infrastructure. In particular, compared to code-level, binary code analysis provides several advantages, including (i) *Visibility into Compiled Code:* Binary code analysis involves examining the compiled form of a software application, providing insights into the actual instructions executed by the processor. This level of analysis is essential for understanding how the software interacts with the hardware, as it focuses on the executable, machine-readable code (ii) *Reverse Engineering:* Reverse engineering binary code can reveal hidden or obfuscated vulnerabilities, which

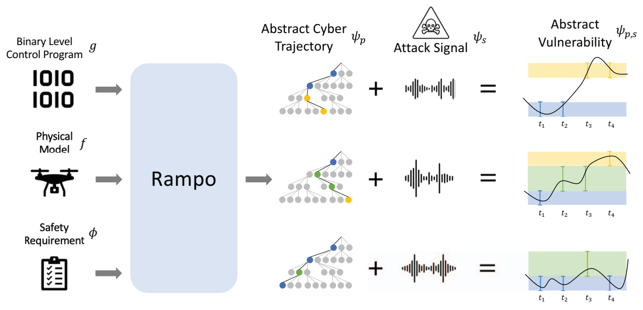
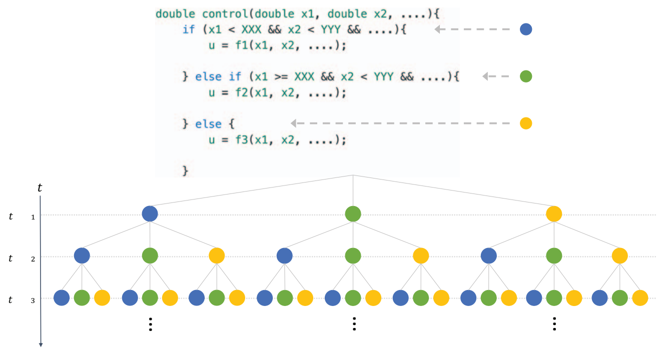

Binary Level Control Program g
1010
1010
Physical Model f
Safety Requirement ϕ
Rampo
Abstract Cyber Trajectory ψ_p
Attack Signal ψ_s
Abstract Vulnerability $\psi_{p,s}$
+
=
t_1 t_2 t_3 t_4

Fig. 1. Rampo takes as input the binary code of the control program, a black-box model that represents the physics of the system, and cyber-kinetic safety requirements captured in STL. The outputs are the cyber trajectories inside the binary code and the corresponding physical system behavior that can lead to violation of the STL requirements. Moreover, Rampo can also output attacks on the system's sensors that can also lead to kinetic vulnerabilities.


```
double control(double x1, double x2, ....){
    if (x1 < XXX && x2 < YYY && ....){
        u = f1(x1, x2, ....);

    } else if (x1 >= XXX && x2 < YYY && ....){
        u = f2(x1, x2, ....);

    } else {
        u = f3(x1, x2, ....);

    }
```

Fig. 2. An example of a code with three paths (marked with blue, green, and yellow). When considering the temporal evolution of a CPS, one needs to take into account the different combinations of these three paths across time, which leads to an exponential increase in the overall number of paths.

might not be readily apparent at the source code level. It is a valuable technique for understanding proprietary or closed-source software, and (iii) *Legacy Systems:* In several CPS applications, only the binary code is available, especially for older or proprietary software. Binary code analysis is the primary method for assessing the security of such systems.

This paper introduces a novel tool called Rampo. As shown in Figure 1, Rampo takes as an input a Signal Temporal Logic (STL) formula that describes the kinetic effects one would like to avoid. For example, this STL formula can encode the requirement that the voltages and currents of the electric distribution buses should be within acceptable safe ranges, drones should approach landing sites with specific velocity profiles to avoid a crash, or the machinery in the industrial control systems should operate within safe operation regimes. The objective of our tool is to simultaneously identify vulnerabilities in the binary code used to control this CPS along with physical attacks (e.g., changes in the sensor measurements) that can lead to a violation of the STL formula and, hence, an undesired kinetic vulnerability. In other words, Rampo is designed to simultaneously analyze the binary code and the dynamics of the physical system to identify vulnerabilities in both the binary code and the physical system that can lead to kinetic consequences.

The design of a tool that can simultaneously analyze the binary code and the dynamical behavior of physical systems within CPS faces two significant complexity challenges. First, performing such analysis necessitates tools that can reason about constraints defined over discrete and real variables, respectively. Unfortunately, tools like Satisfiability Modulo Theories (SMT) solvers and Mixed-Integer Programming (MIP) do not scale well for complex systems. Second, due to the dynamic behavior of CPS, one needs to analyze temporal trajectories over both the cyber and physical systems. That is, while existing binary code analysis focuses on analyzing all the possible paths inside the code within one execution of the code, in CPS, the control code is executed periodically where the path taken within the same binary code may differ from one time to another depending on the state of the physical system. Hence, consider, for the sake of an example, a simple code with three different possible paths (corresponding to different if-else conditions in the code). A classical binary code analysis will aim to cover all these three paths. Nevertheless, considering temporal trajectories of 10-time steps in a CPS application, one must consider all $3^{10} = 59049$ possible paths that may arise from computing the control signal 10 times within a trajectory, a daunting computational problem.

To solve the intertwined challenges mentioned above, Rampo integrates two state-of-the-art techniques from binary analysis and hybrid systems falsification into a unified framework using a Counter-Example Guided Abstraction Refinement (CEGAR) approach [14]. In particular, we use binary analysis tools to perform symbolic execution over the binary code to extract SMT constraints representing different paths inside the code. Using SMT solvers, we solve an optimization problem to identify the bounds on the control signal that can be generated from every possible path in the binary code. To avoid the combinatorial explosion resulting from considering different paths along a temporal trajectory of the CPS, we *abstract* several possibilities of code path executions and use this abstraction to *guide* a falsification of the physical system against the STL requirements. Falsification tools use stochastic optimization algorithms to find trajectories. If a trajectory of the physical system is found, we refine our abstraction using the information extracted from this trajectory (also referred to as an abstract counter-example). By refining the abstraction iteratively, we can ensure the coverage of all possible paths inside the binary code along the temporal evolution of the CPS. In summary, this paper introduces several novel contributions summarized as follows:

- We propose a novel framework to identify cyber-kinetic and cyber-physical-kinetic vulnerabilities in software used to control physical systems.
- We propose a Counter-Example Guided Abstraction Refinement approach to harness the exponential growth in the search space.
- We provide an ablation study that shows the benefits of the proposed framework, both in terms of the effectiveness of finding vulnerabilities and in terms of scaling more favorably to off-the-shelf tools.

II. Related works

Analyzing CPS software for correctness and safety has been an active area in the past few decades. For example, Fuzz testing has become an instrumental technique in the domains of software verification and security analysis. Several works address the fuzz testing of CPS software for automatic test generation [15] and for the discovery of security vulnerabilities [16]–[18]. Unfortunately, the work in [16]–[18] (and others) have focused either on simple physical systems or vulnerabilities with no kinetic consequences.

Another prominent technique for identifying vulnerabilities in software is through symbolic execution techniques. *KLEE* [19] and *angr* [20] are some of the most powerful analysis tools. By utilizing symbolic execution and SMT solvers, these tools can extract path information in a binary code and find several types of vulnerabilities. Unfortunately, these techniques focus mostly on traditional cyber vulnerabilities and do not generalize to reason about cyber-kinetic vulnerabilities due to their lack of analyzing models of physical systems.

One of the widely applied techniques to discover bugs in CPS is Falsification [12], [21], [22]. While traditional falsification engines do not guarantee to cover all possible paths inside the control software, new directions in this field have shown new techniques to provide such code coverage [23], [24]. Unlike the work in [23], [24], this paper focuses on coverage of code paths *along temporal evolution of the code*. As motivated in Figure 2, the temporal evolution of the code leads to an exponential growth in the space of all possible paths that a code can take over time. This challenge of dealing with the temporal evolution of code (or as we call it cyber trajectories), is one of the main motivations behind our CEGAR-based approach.

III. Problem Definition

A. Notation

We use the symbols $\mathbb{N}$, $\mathbb{R}$, and $\mathbb{B}$ to denote the set of natural, real, and Boolean numbers, respectively. Additionally, we use the symbol $\mathbb{FP}$ to denote the set of Floating Point numbers. For ease of notation, we do not distinguish between 32-bit (single) or 64-bit (double) floating point numbers. We use $\wedge$, $\vee$, and $\neg$ to represent the logical AND, OR, and NOT operators, respectively. Given a sequence $a = \{a_t\}_{t=0}^{N}$, we denote by $\texttt{Element}(a, i)$ the ith element of the sequence a.

B. Cyber-Kinetic Vulnerabilities

We consider nonlinear, discrete-time dynamical systems:

$$x_{t+1} = f(x_t, u_t), \quad y_t = h(x_t), \quad u_t = g(y_t), \tag{1}$$

where $x_t \in \mathbb{R}^n$ is the state of the system at time $t \in \mathbb{N}$, $u_t \in \mathbb{FP}^m \subset \mathbb{R}^m$ is the action at time t, and $y_t \in \mathbb{FP}^o \subset \mathbb{R}^o$ is the sensor (output) measurements at time t. As a cyber-physical system, we assume that the system is controlled by a control program $g : \mathbb{FP}^o \rightarrow \mathbb{FP}^m$. Without loss of generality, any program can be decomposed using the so-called path constraints [25] as:

$$\begin{aligned} g(y) = &[c^{(1)}(y) \Rightarrow u = g^{(1)}(y)] \\ &\wedge[c^{(2)}(y) \Rightarrow u = g^{(2)}(y)] \\ &\quad\vdots \\ &\wedge[c^{(k)}(y) \Rightarrow u = g^{(k)}(y)], \end{aligned}$$

where $c^{(i)} : \mathbb{FP}^o \rightarrow \mathbb{B}$ is the ith path constraint of the program g and $g^{(i)} : \mathbb{FP}^o \rightarrow \mathbb{FP}^m$ is the path function, i.e., the function that is executed at the ith path of g.

Example: Consider the following control program g:

```
double control(double y1, double y2){
  if (-1 <= y1 && y1 <= -0.5){
    if (y2 > 0){
        u = 4 * y1 - 6;
    }else{
        u = 0;
    }
  }else{
    u = -4 * y1 - 6;
  }
  return u;
}
```

This control program consists of 3 paths with the following path constraints:

$$\begin{aligned} c^{(1)}(y) &= (-1 \leq y1) \wedge (y1 \leq -0.5) \wedge (y2 > 0), \\ c^{(2)}(y) &= (-1 \leq y1) \wedge (y1 \leq -0.5) \wedge (y2 \leq 0), \\ c^{(3)}(y) &= \neg\left((-1 \leq y1) \wedge (y1 \leq -0.5)\right). \end{aligned}$$

The corresponding path functions are:

$$g^{(1)}(y) = 4y1 - 6, \quad g^{(2)}(y) = 0, \quad g^{(3)}(y) = -4y1 - 6.$$

It is important to note that we do not assume prior knowledge of the path constraints and the corresponding path functions, as they will be extracted automatically from the binary code representing the control program.

Given a control program g that consists of k paths, we denote by $\texttt{PATH} : \mathbb{FP}^o \rightarrow \{1, \ldots, k\}$ the function that returns the path index of an input y. That is:

$$\texttt{PATH}(y) = p \Longleftrightarrow c^{(p)}(y) = \texttt{True}. \tag{2}$$

Note that the function PATH is well defined since path constraints are mutually exclusive, and hence, an input y can only be processed by only one path inside the code.

Using this notation, we can now define the cyber trajectory of a system as the index of the control program's path taken at different time instances as follows.

Definition 3.1 (Cyber Trajectories): Given a control program g with k paths, a sequence $\psi_p = \{p_t\}_{t=0}^{H}$, with $p_t \in \{1, \ldots, k\}$, is called a cyber trajectory of the physical system f if there exists a corresponding trajectory of the physical system $\psi_x = \{x_t\}_{t=0}^{H}$ with $x_{t+1} = f(x_t, g(h(x_t)))$ $(t < H)$ such that $p_t = \texttt{PATH}(h(x_t))$ for all $t \in \{0, \ldots, H\}$.

We are interested in enumerating all cyber trajectories that can lead to physical system trajectories that violate a safety requirement φ. In this paper, we assume that the safety requirement φ is captured by a Signal Temporal Logic (STL) formula. For the formal definition of STL syntax and semantics, we refer the reader to [26]. Using this notation, the problem of interest can be defined as follows.

***Problem** 3.2 (Enumeration of Cyber-Kinetic Vulnerabilities):* Given an STL formula φ, a controller program g with k paths, and a physical system model f. Find the set Ψ_p^φ of cyber trajectories that may lead to cyber-kinetic vulnerabilities, i.e., Ψ_p^φ is defined as:

$$\begin{aligned}\Psi_p^\varphi =& \Big\{ \{p_t\}_{t=0}^H \in \{1,\ldots,k\}^{H+1} \mid \\ & \exists \psi_x = \{x_t\}_{t=0}^H . \big[x_{t+1} = f(x_t, g(h(x_t)))\ (t < H), \\ & \psi_x \not\models \varphi,\ p_t = \mathtt{PATH}(h(x_t)), \forall t \in \{0,\ldots,H\} \big] \Big\}. \end{aligned} \quad (3)$$

In other words, a cyber trajectory ψ_p is considered a cyber-kinetic vulnerability if there exists an associated trajectory of the physical system ψ_x that violates the safety requirement φ.

C. Cyber-Physical-Kinetic Vulnerabilities

While Problem 3.2 focuses on finding vulnerabilities in the controller program that lead to violation of safety requirements, we are also interested in generalizing this problem into scenarios when an attacker is capable of manipulating the physical sensor measurements $h(x_t)$ before it is processed by the control program g [27]–[32]. Typically, these sensor manipulations are of relatively small magnitude $\overline{s}$ compared to the sensor noise levels to avoid being detected. That is, we consider the system:

$$x_{t+1} = f(x_t, u_t), \quad y_t = h(x_t) + s_t, \quad u_t = g(y_t) \quad (4)$$

where s_t is the physical attack signal at time t with $\|s_t\| \le \overline{s}$. In such a setup, given a trajectory of the physical system ψ_x and a trajectory of the physical attack signal $\psi_s = \{s_t\}_{t=0}^H$, the corresponding cyber trajectory ψ_p is defined as $\psi_p = \{\mathtt{PATH}(h(x_t) + s_t)\}_{t=0}^H$.

***Problem** 3.3 (Enumeration of Cyber-Physical-Kinetic Vulnerabilities):* Given an STL formula φ, a controller program g with k paths, a physical system model f, and a limit on the physical attack signal $\overline{s}$. Find the set $\Psi_{p,s}^\varphi$ of cyber-physical-kinetic vulnerabilities defined as:

$$\begin{aligned}\Psi_{p,s}^\varphi = & \Big\{ \{(p_t, s_t)\}_{t=0}^H \in \{1,\ldots,k\}^{H+1} \times \mathbb{FP}^{H+1} \mid \\ & \exists \psi_x = \{x_t\}_{t=0}^H . \big[x_{t+1} = f(x_t, g(h(x_t) + s_t))\ (t < H), \\ & \psi_x \not\models \varphi,\ p_t = \mathtt{PATH}(h(x_t) + s_t), \\ & \|s_t\| \le \overline{s}, \forall t \in \{0,\ldots,H\} \big] \Big\}. \end{aligned} \quad (5)$$

IV. Framework

A. Overview of the Rampo Framework

As pictorially shown in Figure 3, given a binary-level control program g, our first step is to extract the path constraints $c^{(1)}, \ldots, c^{(k)}$ from g. To this end, we use off-the-shelf symbolic code execution engines that can extract SMT constraints representing different paths in the control program (Line 5 in Algorithm 1).

The next step is to analyze the extracted paths to find cyber trajectories that can lead to cyber-kinetic vulnerabilities. Nevertheless, and as motivated in Figure 2, the number of cyber trajectories increases exponentially as a function of the number of paths k and the length of the trajectory H. In particular, for trajectories of length H and a control program with k paths, our tool is now challenged with analyzing all k^H possible cyber trajectories. To harness the combinatorial explosion, we resort to an iterative Counter-Example Guided Abstraction Refinement (CEGAR) process. In this process, cyber trajectories go through several levels of abstraction that will be iteratively refined later in the process to ensure coverage of all possible cyber trajectories.

The first level of abstraction is to consider the "range" of the control signal u that each program path can produce. Without loss of generality, and for the sake of notation, we will assume that control signal u is scalar — nevertheless, our framework is designed to support multidimensional control signals. For such a case, the maximum and minimum of the control signal u within the ith path p of g, denoted by $\overline{u}_p$ and $\underline{u}_p$, can be computed by solving the following optimization problem:

$$\overline{u}_p = \max_{y \in \mathbb{FP}^o} g^{(p)}(y) \quad \text{subject to} \quad c^{(p)}(y) = \mathtt{True}, \quad (6)$$

$$\underline{u}_p = \min_{y \in \mathbb{FP}^o} g^{(p)}(y) \quad \text{subject to} \quad c^{(p)}(y) = \mathtt{True}. \quad (7)$$

Thanks to the current advances in SMT solvers (e.g., Z3 [33]), one can efficiently solve the optimization problem above to obtain the *path ranges* $R = \{(\overline{u}_1, \underline{u}_1), \ldots, (\overline{u}_k, \underline{u}_k)\}$. Given these path ranges, we can now abstract a cyber trajectory using the ranges of control signals at each time step. That is, for any cyber trajectory $\psi_p = \{p_t\}_{t=0}^H$, the corresponding "range" of control signals within this trajectory is defined as:

$$\overline{\mathtt{Range}}(\psi_p) = \Big\{ \overline{u}_{p_t} \Big\}_{t=0}^H, \quad \underline{\mathtt{Range}}(\psi_p) = \Big\{ \underline{u}_{p_t} \Big\}_{t=0}^H \quad (8)$$

The second level of abstraction combines multiple cyber trajectories into an "abstract cyber trajectory". Such abstraction is done by combining the ranges of control signals across different cyber trajectories. In particular, the first iteration of Rampo abstracts all cyber trajectories into one. The ranges of control signals along all cyber trajectories are used to augment the STL-based safety requirements φ as follows:

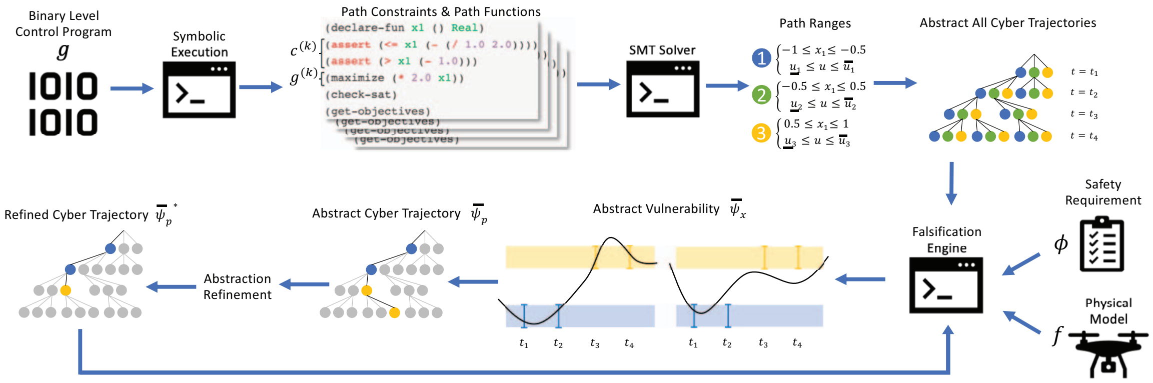
$$\neg\phi = \neg\varphi \wedge \left[\bigwedge_{t=0}^H \min_{p \in \{1,\ldots,k\}} \underline{u}_p \le u_t \le \max_{p \in \{1,\ldots,k\}} \overline{u}_p \right]. \quad (9)$$

The logical negation $\neg$ is motivated by the fact that falsification engines aim to find a violation of ϕ, and hence, the encoding above forces the control signal to be within the ranges of interest.

Next, Rampo uses these abstractions to guide a system falsification engine. System falsification refers to a set of tools that take as input a dynamical system f and an STL formula ϕ and find a physical-level trajectory $\psi_x = \{x_t\}_{t=0}^H$ that violates ϕ, i.e.,

$$\overline{\psi}_x = \mathtt{Falsify}(f, \phi). \quad (10)$$

If $\overline{\psi}_x$ is not empty, then the computed trajectory ψ_x is guaranteed to violate the STL requirements ϕ, i.e., $\psi_x \not\models \phi$.


Binary Level Control Program
g
1010
1010
Symbolic Execution
Path Constraints & Path Functions
$c^{(k)}$
$g^{(k)}$
SMT Solver
Path Ranges
1 $\begin{cases}-1 \le x_1 \le -0.5 \\ \underline{u}_1 \le u \le \overline{u}_1\end{cases}$
2 $\begin{cases}-0.5 \le x_1 \le 0.5 \\ \underline{u}_2 \le u \le \overline{u}_2\end{cases}$
3 $\begin{cases}0.5 \le x_1 \le 1 \\ \underline{u}_3 \le u \le \overline{u}_3\end{cases}$
Abstract All Cyber Trajectories
$t = t_1$
$t = t_2$
$t = t_3$
$t = t_4$
Safety Requirement
ϕ
Falsification Engine
Physical Model
f
Abstract Vulnerability $\overline{\psi}_x$
t_1 t_2 t_3 t_4
t_1 t_2 t_3 t_4
Abstract Cyber Trajectory $\overline{\psi}_p$
Abstraction Refinement
Refined Cyber Trajectory $\overline{\psi}_p^*$

```
(declare-fun x1 () Real)
(assert (<= x1 (- (/ 1.0 2.0))))
(assert (> x1 (- 1.0)))
(maximize (* 2.0 x1))
(check-sat)
(get-objectives)
```

Fig. 3. A pictorial representation of the proposed framework. Binary-level control programs are analyzed using symbolic execution to extract the path constraints, path functions, and the range of the control signals associated with each path in the program. Next, a Counter-Example Guided Abstraction Refinement (CEGAR) is used to abstract all cyber trajectories for a horizon H, and a falsification engine is used to search for trajectories of the physical system that violate the safety requirements. The cyber trajectories are refined around the falsifying trajectories until all concrete cyber-kinetic vulnerabilities are found.

Note that the bar in $\overline{\psi}_x$ is used to emphasize the fact that $\overline{\psi}_x$ is not a *concrete* vulnerability since it was computed based on the abstraction of cyber trajectories using control signal ranges (Lines 11-12, Algorithm 1). Since the abstraction of cyber trajectories using range of control signals is a sound approximation, then whenever $\overline{\psi}_x$ is empty, one can conclude that the system is free from all cyber-kinetic vulnerabilities. On the other hand, if $\overline{\psi}_x$ is not empty, then our tool will extract the cyber trajectory corresponding to $\overline{\psi}_x$ as follows:

$$\begin{aligned}\overline{\psi}_p &= \texttt{Extract-Path}(\overline{\psi}_x) \\ &= \left\{\texttt{PATH}(h(\texttt{Element}(\overline{\psi_x}, t)))\right\}_{t=0}^{H}. \end{aligned} \tag{11}$$

We refer to $\overline{\psi}_p$ as an abstract vulnerability (Lines 13-16, Algorithm 1).

Since the abstract vulnerability $\overline{\psi}_p$ is extracted from a physical trajectory that violated the safety requirements, it is likely a *concrete* vulnerability may exist in the system. To that end, Rampo will explore $\overline{\psi}_p$ by iteratively refining the control signal ranges around $\overline{\psi}_p$. In particular, Rampo provides two different strategies for the iterative abstraction refinement of $\overline{\psi}_p$ that are explained in detail in Section V and Algorithm 1. These abstraction refinement approaches can either find *concrete* cyber vulnerabilities ψ_p, find other abstract cyber trajectories $\overline{\psi}_p^*$ that need to be subsequently refined, or certify that a set of abstract trajectories $\underline{\psi}_p$ will not lead to a vulnerability (Lines 19-23, Algorithm 1).

Finally, Rampo will search for new abstract vulnerabilities over the unexplored abstract trajectories. This is achieved by forcing the falsification engine to search over the control signal ranges that are not considered yet by encoding them in the STL formula as:

$$\begin{aligned}\neg\phi = \neg\varphi \wedge \Big[\bigwedge_{\psi_p \in \underline{\Psi}_p} \bigwedge_{t=0}^{H} \neg \left(\texttt{Element}(\underline{\texttt{Range}}(\psi_p), t) \le u_t\right) \Big] \\ \wedge \Big[\bigwedge_{\psi_p \in \underline{\Psi}_p} \bigwedge_{t=0}^{H} \neg \left(u_t \le \texttt{Element}(\overline{\texttt{Range}}(\psi_p), t)\right) \Big], \end{aligned} \tag{12}$$

where φ is the original safety requirements and $\underline{\Psi}_p$ is the set of the abstract trajectories explored so far (Lines 25-30, Algorithm 1).

By refining the abstractions of the explored cyber vulnerabilities and searching over unexplored abstract trajectories, Rampo will iteratively identify new abstract cyber trajectories and search for concrete vulnerabilities within those abstract cyber trajectories until all cyber trajectories are covered. This process is summarized in Algorithm 1 and Figure 3.

B. *Correctness of Algorithm 1*

The soundness and completeness of Algorithm 1 relies on the soundness and completeness of the falsification engine (`Falsify`). This follows from the fact that Rampo uses sound abstraction of the cyber trajectories and relies on the falsification engine to discard abstract trajectories when no abstract physical-level trajectory violates the safety requirements. Unfortunately, off-the-shelf falsification engines rely on stochastic optimization to reason about non-convex constraints, and hence, these falsification engines are sound but not complete, rendering Rampo to be sound but not complete.

C. *Extension to Cyber-Physical-Kinetic Vulnerabilities*

Algorithm 1 can be directly extended to enumerate cyber-physical-kinetic vulnerabilities. As captured by Problem 3.3, the main difference between cyber-kinetic vulnerabilities and cyber-physical-kinetic vulnerabilities is the need to find additional sensor manipulation signals $\{s_t\}_{t=0}^{H}$ that can lead to violation of safety requirements. To that end, one can treat the signal s as an additional input signal to the falsification engine and rely on the falsification engine to simultaneously search for ψ_x and ψ_s. An example of this extension is provided in the numerical examples in Section VI-D.

V. Abstraction Refinement of Abstract Vulnerabilities

As shown in Algorithm 1, whenever an abstract vulnerability $\overline{\psi}_p$ is found, Rampo needs to refine this abstraction to identify concrete vulnerabilities, find other potential abstract vulnerabilities, or exclude some abstract trajectories from

Algorithm 1 Rampo

```
Input: Safety specification $\varphi$, System dynamics $f$, binary-level control program $g$
Output: Set of Cyber-Kinetic Vulnerabilities $\Psi_p^\varphi$
 1: Initialize $\Psi_p^\varphi :=$ Empty Set
 2: Initialize $\overline{\Psi}_p :=$ Empty Set
 3: Initialize $\underline{\Psi}_p :=$ Empty Set
 4: Step 1: Extract path constraints & ranges:
 5: $(c^{(1)}, g^{(1)}), .., (c^{(k)}, g^{(k)}) =$ Symbolic-Execution($g$)
 6: for Each path $p$ in $\{1, \ldots, k\}$ do
 7:     $\overline{u}_p = \max_{y \in \mathbb{PF}^o} g^{(p)}(y)$ subject to $c^{(p)}(y) =$ True
 8:     $\underline{u}_p = \min_{y \in \mathbb{PF}^o} g^{(p)}(y)$ subject to $c^{(p)}(y) =$ True
 9: end for
10: Step 2: Abstract all trajectories & falsify:
11: $\neg\phi \leftarrow$ Equation (9)
12: $\overline{\psi}_x =$ Falsify($f, \phi$)
13: if Not-EMPTY($\overline{\psi}_x$) then
14:     $\overline{\psi}_p =$ Extract-Path($\overline{\psi}_x$)
15:     $\overline{\Psi}_p = \overline{\Psi}_p \cup \{\overline{\psi}_p\}$
16: end if
17: while Not-EMPTY($\overline{\Psi}_p$) do
18:     Step 3: Refine abstract trajectories:
19:     Pop one $\overline{\psi}_p$ from $\overline{\Psi}_p$
20:     $(\psi_p, \overline{\psi}_p^*, \underline{\psi}_p) =$ Abstraction-Refinment($f, \varphi, \overline{\psi}_p$)
21:     $\Psi_p^\varphi = \Psi_p^\varphi \cup \psi_p$
22:     $\overline{\Psi}_p = \overline{\Psi}_p \cup \{\overline{\psi}_p^*\}$
23:     $\underline{\Psi}_p = \underline{\Psi}_p \cup \{\underline{\psi}_p\}$
24:     Step 4: Explore unexplored abstract trajectories:
25:     $\neg\phi \leftarrow$ Equation (12)
26:     $\overline{\psi}_x =$ Falsify($f, \phi$)
27:     if Not-EMPTY($\overline{\psi}_x$) then
28:         $\overline{\psi}_p =$ Extract-Path($\overline{\psi}_x$)
29:         $\overline{\Psi}_p = \overline{\Psi}_p \cup \{\overline{\psi}_p\}$
30:     end if
31: end while
32: return $\Psi_p^\varphi$
```

the search space. In this section, we provide two alternative approaches for the iterative abstraction refinement.

A. *Linear-Search Based Refinement*

In the first abstraction refinement approach, given an abstract vulnerability $\overline{\psi}_p$ of length $l \leq H$, our tool concretizes/refines the ranges around this specific $\overline{\psi}_p$. This can be encoded in the STL formula ϕ as follows:

$$\neg\phi = \neg\varphi \wedge \left[\bigwedge_{t=0}^{l} \texttt{Element}(\underline{\texttt{Range}}(\psi_p), t) \leq u_t \right] \wedge \left[\bigwedge_{t=0}^{l} u_t \leq \texttt{Element}(\overline{\texttt{Range}}(\psi_p), t) \right]. \quad (13)$$

The falsification engine then uses ϕ to search for a trajectory of the physical system. Note that even if a trajectory is found, this does not account for a concrete vulnerability since the behavior of the control program g is still abstracted as a range of control signals (recall the two levels of abstractions in Section 4.1). This requires another call to the Falsification engine, this time using the physical model f augmented with the control program g constrained to the path constraints $c^{(\texttt{Element}(\overline{\psi}_p, 1))}, \ldots, c^{(\texttt{Element}(\overline{\psi}_p, l))}$. If a concrete vulnerability ψ_x was found, then it is added to the set of concrete cyber-kinetic vulnerabilities. Regardless of whether a concrete vulnerability is found, the abstract trajectory $\overline{\psi}_p$ is added to the set of explored trajectories $\underline{\psi}_p$ (Lines 2-11, Algorithm 2).

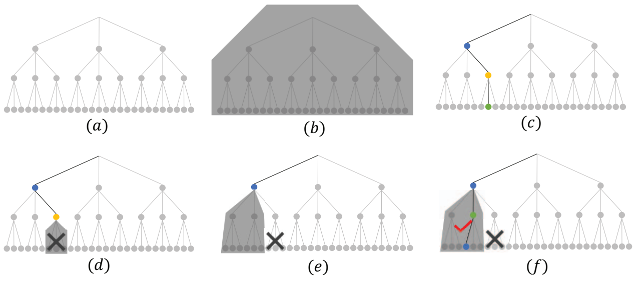


Fig. 4. A pictorial representation of the Counter-Example Guided Abstraction Refinement process. (a) A tree of all possible cyber trajectories for a controller code with 3 paths and $H = 3$. (b) In the first iteration, Rampo will abstract all the trajectories in the search tree by considering the control signals that can be produced by all trajectories. (c) The falsification engine finds a physical-level trajectory that violates the safety requirements and Rampo extracts the corresponding trajectory. (d) The control signal ranges are refined/concretized around the trajectory found from the previous step except for the last time step. The falsification engine could not find a corresponding physical-level trajectory that violates the requirements, and hence, this sub-tree is removed from the search space. (e) Rampo backtracks one level up and abstracts the cyber trajectories around the truncated trajectory found before. (f) The falsification engine reported a physical-level vulnerability in the search tree, and the vulnerability was checked to be a concrete one.

If a concrete vulnerability is not found by the process above, we iteratively reduce the abstraction refinement by removing the range constraints in ϕ (as pictorially illustrated in Figure 4). That is, starting from $i = 1$ until $i = l$, we will iteratively call the falsification engine to find a trajectory of the physical system that violates the safety requirements by encoding these range constraints in ϕ as follows:

$$\neg\phi = \neg\varphi \wedge \left[\bigwedge_{t=0}^{l-i} \texttt{Element}(\underline{\texttt{Range}}(\psi_p), t) \leq u_t \right] \wedge \left[\bigwedge_{t=0}^{l-i} u_t \leq \texttt{Element}(\overline{\texttt{Range}}(\psi_p), t) \right]. \quad (14)$$

Whenever the falsification engine fails to find a violation of ϕ (for $i = 1, \ldots, l$), the truncated version of $\overline{\psi}_p$ (i.e., $\{p_t\}_{t=0}^{i}$) is added to the set of explored trajectories $\underline{\psi}_p$. Similarly, whenever the falsification engine finds an abstract vulnerability, the newly discovered abstract vulnerability is added to $\overline{\psi}_p^*$. This process is summarized in Figure 4 and Algorithm 2.

B. *Binary-Search Based Refinement*

While Algorithm 2 relaxes the range constraints along $\overline{\psi}_p$ in a linear fashion, our second approach for abstraction refinement is to relax the range constraints in a binary-search fashion. That is, we replace the loop in Step 14 of Algorithm 2

```
Algorithm 2 Abstraction-Refinement-Linear
Input: Safety requirements $\varphi$, Physical System Model $f$,
    Abstract cyber trajectory $\overline{\psi}_p$
Output: Set of concrete cyber vulnerabilities $\psi_p$,
    Set of unexplored abstract vulnerabilities $\overline{\psi}^*_p$,
    Set of explored abstract trajectories $\underline{\psi}_p$
 1: Initialize $\psi_p, \overline{\psi}^*_p, \underline{\psi}_p :=$ Empty Set
 2: Step 1: Search for concrete vulnerability
 3: $\neg\phi \leftarrow$ Equation (13)
 4: $\overline{\psi}_x =$ Falsify$(f, \phi)$
 5: if Not-EMPTY$(\overline{\psi}_x)$ then
 6:     $f_{\mathrm{cl}} =$ Form-Closed-Loop-Model$(f, g)$
 7:     $\psi_x =$ Falsify$(f_{\mathrm{cl}}, \phi)$
 8:     if Not-EMPTY$(\psi_x)$ then
 9:         $\psi_p = \overline{\psi}_p$
10:         $\underline{\psi}_p = \underline{\psi}_p \cup \overline{\psi}_p$
11:     end if
12: else
13:     for $i = 1$ to $l$ do
14:         $\neg\phi \leftarrow$ Equation (14)
15:         $\overline{\psi}_x =$ Falsify$(f, \phi)$
16:         if Not-EMPTY$(\overline{\psi}_x)$ then
17:             $\overline{\psi}^*_p = \overline{\psi}^*_p \cup \{\{$Element$(\psi_p, t)\}_{t=0}^{l-i}\}$
18:             Exit the FOR loop
19:         else
20:             $\underline{\psi}^*_p = \underline{\psi}^*_p \cup \{\{$Element$(\psi_p, t)\}_{t=0}^{l-i}\}$
21:         end if
22:     end for
23: end if
24: return $\psi_p, \overline{\psi}^*_p, \underline{\psi}_p$
```

with one that starts with $i = l/2$ and then selects the next value of i based on the success/failure of finding an abstract vulnerability. Compared to the linear-search-based approach, this binary search is more suitable for traversing trees with high values of l (hence the height of the tree being explored by the abstraction refinement).

VI. Experimental Evaluation

We implemented Rampo in Python where we used *angr* [20] for symbolic code execution, *Z3* [33] to compute the ranges of the control signal in each path, and *S-TaLiRo* [12] as the falsification engine. In this section, we perform a series of numerical analyses to evaluate the effectiveness and the scalability of our tool. First, we conduct a series of studies to compare against state-of-the-art falsification of closed-loop systems and to study the effect of varying different parameters (e.g., the complexity of the physical system and the complexity of the cyber system). We utilize two metrics for this study: (i) the execution time and (ii) the number of identified cyber-kinetic vulnerabilities. Next, we will study the ability to identify both cyber-kinetic and cyber-physical-kinetic attacks using a sophisticated model for engine control systems.

A. *Experiment 1: Comparison Against Black-Box Falsification*

We start by comparing Rampo to *S-TaLiRo*, one of the state-of-the-art CPS falsification tools. While we use *S-TaLiRo* internally within Rampo to find falsifying trajectories of the open-loop system f (up to specified path constraints), *S-TaLiRo* can also be used to falsify the entire closed-loop system (the dynamical system f along with the controller g). In this experiment, we show that analyzing the controller g away from the dynamical system f (using our approach) will lead to identifying a more significant number of vulnerabilities in the system compared to the as-is use of *S-TaLiRo*.

To that end, we consider a model of a drone moving vertically. We use a simple double integrator dynamical system f to capture the vertical movement of the drone. The safety requirement is for the drone's position to be always below 0.98. We also consider the following control program, which implements a gain scheduling controller:

```
double control(double x1){
  if (x1 < -1) x1 = -1;
  if (x1 > 1) x1 = 1;

  if (-1 <= x1 && x1 <= -0.5){
    u = 4 * x1 - 6;
  }else if(-0.5 < x1 && x1 < 0.5){
    u = 2 * x1 + 9;
  }else if(0.5 <= x1 && x1 <= 1){
    u = -4 * x1 - 6;
  }
  return u;
}
```

Listing 1. Control program for Experiments 1-3

Note that *S-TaLiRo* aims to find only one falsifying trajectory while our aim in Problem 3.2 is to enumerate all possible cyber-kinetic vulnerabilities. To that end, we ran *S-TaLiRo* 50 times with random seeds to force it to explore different vulnerabilities. We ran Rampo while restricting the number of calls to the `Falsify` function to 50. Our experiments show that Rampo was able to identify 14 cyber-kinetic vulnerabilities while S-TaLiRo-alone was able to find only 9 of those vulnerabilities. The execution time of Rampo was 191 seconds, while for S-TaLiRo-alone was 147 seconds. This result shows one of the main benefits of combining falsification and code analysis tools. By splitting the analysis between the two domains (cyber and physical domains), one can achieve better coverage of the different cyber trajectories in the system.

B. *Experiment 2: Scalability with respect to complexity of the physical system model*

Our second experiment aims to evaluate the scalability of our tool with respect to the number of states in the dynamical model f. To that end, we use the same control program in Listing 1, and we vary the number of integrators in the system from 1 to 10. To obtain a baseline for comparison, we perform an exhaustive search by partitioning the space of initial states (used to find falsifying trajectories by *S-TaLiRo*) into hypercubes. By constraining *S-TaLiRo* to find falsifying trajectories that start from different initial states, we can increase the number of identified vulnerabilities. Nevertheless, and as shown in Figure 5 (left), this brute force search scales

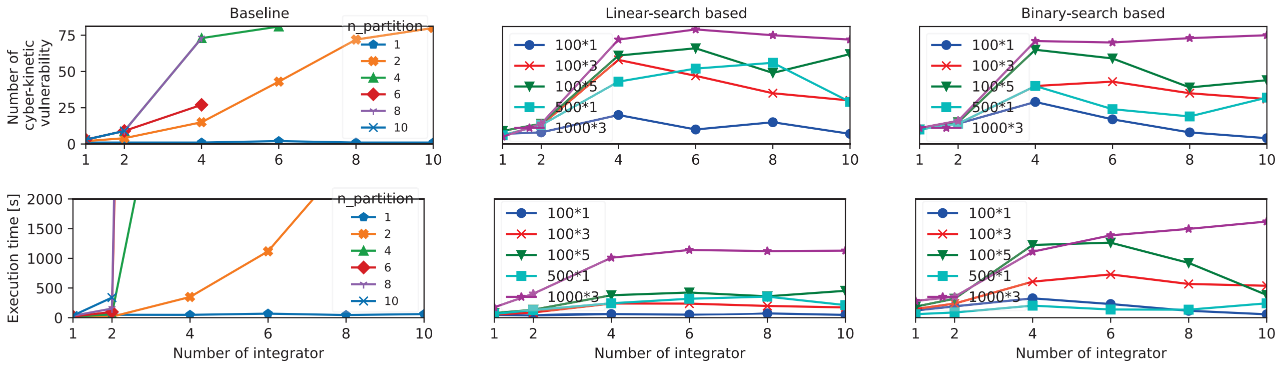


Fig. 5. The number of cyber-kinetic vulnerabilities found and the execution time. $n\_partition$ refers to the number of partitions among each dimension of the state space in the model. In the brute-force (baseline) approach, as the number of partitions increases, the execution time increases exponentially. For both the linear- and binary-search-based abstraction refinement approaches of Rampo, the execution time is almost constant as the number of integrators in the model increases. This leads to a speedup that ranges from $3\times$ to $98\times$ while computing the same number of vulnerabilities.

poorly as the number of states increases and as the granularity of the partitioning increases.

On the other hand, as shown in Figure 5 (middle and right), our tool with the linear- and binary-search-based abstraction refinement shows a similar trend to the baseline in terms of their ability to identify cyber-kinetic vulnerabilities while avoiding the exponential increase in the execution time. In general, the execution time of Rampo remained almost constant as the number of states increased. We repeated the same experiment multiple times while changing some of the internal parameters to the `Falsify` function on Rampo (which as explained above, is implemented using *S-TaLiRo* but with open-loop dynamics and path constraints). In particular, different plots in Figure 5 show different settings where the number before the multiplication operator indicates the number of optimization steps taken in one falsification, and the latter numbers are the number of calls to *S-TaLiRo* per one call to the `Falsify` function. For the same number of identified vulnerabilities, Rampo achieved a speedup that ranges from $3\times$ to $98\times$.

C. Experiment 3: Scalability with respect to the complexity of the control software

In this experiment, we study the scalability of Rampo as we increase the length of the cyber trajectories H. Recall that increasing H results in an exponential growth in the number of cyber trajectories that need to be explored. Figure 6 (top) shows the number of identified vulnerabilities as H increases for both the linear- and binary-search-based abstraction refinement approaches. As expected, the number of vulnerabilities increases as H increases. Moreover, the number of vulnerabilities found by the linear- and binary-search-based abstraction refinement approaches are comparable among the different settings for the `Falsify` function.

On the other hand, 6 (bottom) shows the execution time for both the linear- and binary-search-based abstraction refinement approaches. As expected, the benefits of the binary search start to grow as the depth H increases, resulting in an average of $2\times$ speedup for the binary-search-based abstraction refinement compared to the linear-search-based abstraction refinement

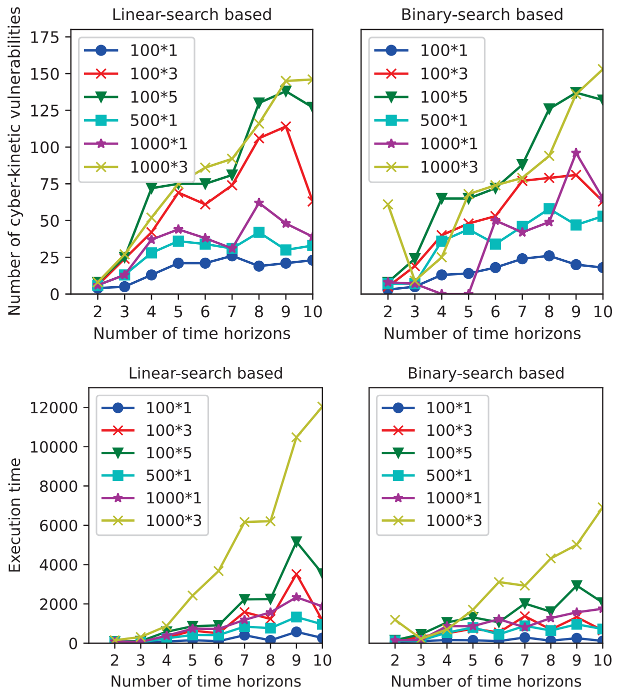


Fig. 6. Number of cyber-kinetic vulnerabilities found and execution times in different lengths of time horizons

approach. Moreover, and most importantly, the execution time of both the linear- and binary-search-based abstraction refinement approaches does *not* increase exponentially with H, albeit with the exponential growth in the number of cyber trajectories. We hypothesize that this pattern is due to the CEGAR approach used by Rampo which can reason about several cyber trajectories simultaneously, which harnesses the exponential growth in the number of cyber trajectories.

D. Experiment 4: Enumerating Cyber-Kinetic and Cyber-Physical-Kinetic Vulnerabilities

In this experiment, we evaluate the ability of Rampo to identify both cyber-kinetic and cyber-physical-kinetic vulner-

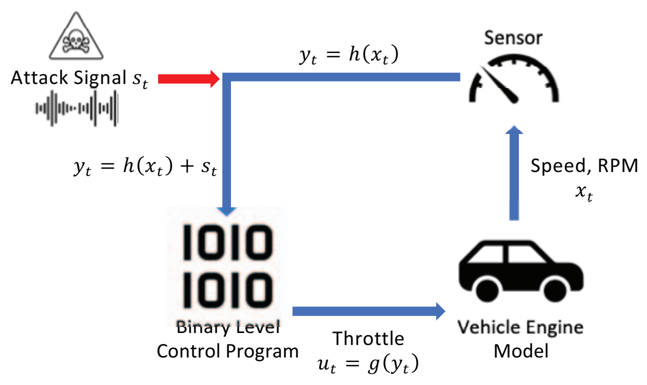


Fig. 7. Vehicle engine model

abilities in complex systems. To that end, we use a model for a vehicle control signal (shown in Figure 7). The model has two inputs, $Speed$ and RPM, and the one output, $Throttle$. The control program for this model is shown in Listing 2 which has a total of $k = 4$ paths. The safety requirement φ is "Either the $Speed$ or the RPM is below a specified threshold."

```
double control(double RPM, double Speed){
  if (RPM > 3300){
    if (Speed > 80){
      Throttle = -RPM*0.002 - Speed*1.1 + 183.0;
    } else {
      Throttle = - RPM*0.001 + Speed*0.6 + 19.0;
    }
  } else {
    if (Speed > 80){
      Throttle = RPM*0.001 - Speed*1.7 + 216.0;
    } else {
      Throttle = - RPM*0.001 - Speed*0.6 + 139.0;
    }
  }
  return Throttle;
}
```

Listing 2. Control program for the vehicle engine model

We started by searching for cyber-kinetic vulnerabilities using Rampo. Figure 8 shows the only cyber-kinetic vulnerability found by the binary-search-based abstraction refinement approach. This vulnerability corresponds to the cyber trajectory $(p_0 = 1, p_1 = 4, p_2 = 4, p_3 = 4)$. The corresponding trajectories of ψ_{x_1} and ψ_{x_2} are shown in Figure 8 where cross the bars that indicate corresponding paths.

Next, we examine whether a small amount of physical attack signals can lead to *other* kinetic attacks, i.e., can a small amount of physical attack signal force the control program to go through a different cyber trajectory that can lead to a new kinetic attack? To that end, we restrict the physical attack signal ψ_s to be $\leq 2.0\%$ of the original sensor signals ($Speed$ and RPM). We considered three different configurations. In the first one, the attacker can change only $Speed$, in the second one, the attacker can only change RPM, while in the last configuration, the attacker can affect both $Speed$ and RPM.

For the first two settings (attacking either $Speed$ alone or RPM alone), our tool was not able to find any *new* cyber trajectory that can lead to vulnerability (other than the one shown in Figure 8). On the other hand, by allowing the attacker

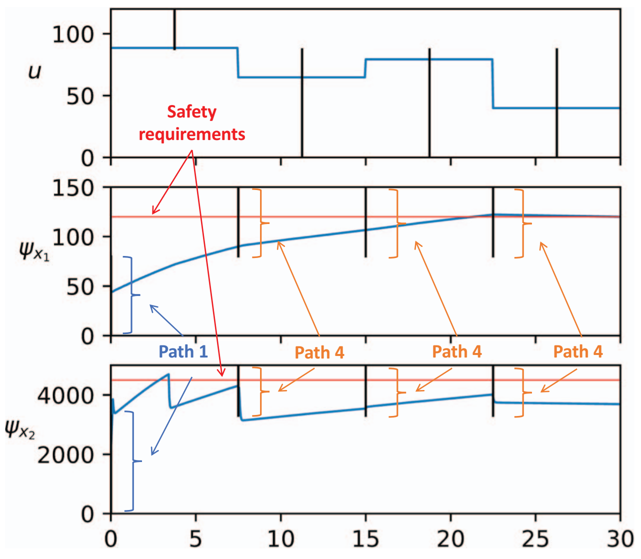


Fig. 8. Cyber-kinetic vulnerability found in the vehicle engine model

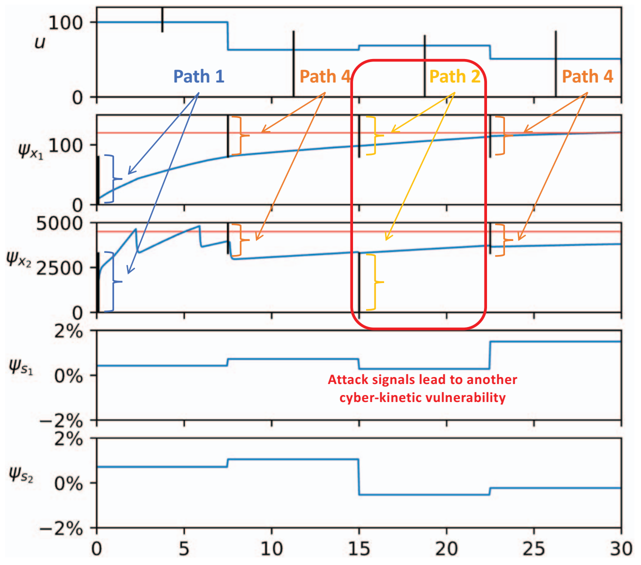


Fig. 9. Another cyber-kinetic vulnerability found only when attack signals are injected

to corrupt both the $Speed$ and RPM, our tool identified another cyber trajectory that $(p_0 = 1, p_1 = 4, p_2 = 2, p_3 = 4)$ that can lead to vulnerabilities. The cyber-physical-kinetic vulnerability is shown in Figure 9.

By analyzing the third time step in Figure 9, one can notice that the ψ_{x_2} trajectory is close to violating the constraints of $p = 2$. That is, a small perturbation in the sensor signal (either intentional by an attacker or by noise) can force the code to execute a different path in the control program g. This shows the effectiveness of our tool to identify these corner cases, which can be used to fix the control program to be more robust and secure.

VII. Conclusion

This paper proposed a novel tool, named Rampo, that can perform binary code analysis to identify cyber kinetic vulnerabilities in CPS. Our tool analyzes the binary level control program to extract the ranges of control signal inputs to the physical system. It uses this information to abstract the code and guide falsification tools to identify possible kinetic vulnerabilities. The tool iteratively refines the abstraction until concrete cyber-kinetic vulnerabilities are found or the entire space of cyber trajectories is covered. We provided different approaches for the abstraction refinement process and generalized the framework to find cyber-physical-kinetic vulnerabilities. Our numerical analysis shows that our tools scale more favorably compared to off-the-shelf tools and are able to harness the exponential growth in the search space when the temporal evolution of the code is considered leading to 3×-98× speed up in execution time while identifying the same number of vulnerabilities.

References

[1] S. D. Applegate, "The dawn of kinetic cyber," in *2013 5th International Conference on Cyber Conflict (CYCON 2013)*, 2013, pp. 1–15.

[2] S. R. Chhetri, J. Wan, and M. A. Al Faruque, "Cross-domain security of cyber-physical systems," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 200–205.

[3] S. R. Chhetri, A. Canedo, and M. A. Al Faruque, "Kcad: Kinetic cyber-attack detection method for cyber-physical additive manufacturing systems," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.

[4] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10/sup 20/ states and beyond," in *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990, pp. 428–439.

[5] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker: (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*. Springer, 2014, pp. 389–391.

[6] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ansi-c software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2011.

[7] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[8] J.-L. Boulanger, *Industrial use of formal methods: formal verification*. John Wiley & Sons, 2013.

[9] P. Tabuada, *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media, 2009.

[10] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer, 2013, pp. 258–263.

[11] S. Bansal, M. Chen, S. Herbert, and C. J. Tomlin, "Hamilton-jacobi reachability: A brief overview and recent advances," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, 2017, pp. 2242–2253.

[12] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-taliro: A tool for temporal logic falsification for hybrid systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, P. A. Abdulla and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 254–257.

[13] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 167–170.

[14] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer, 2000, pp. 154–169.

[15] S. Sheikhi, E. Kim, P. S. Duggirala, and S. Bak, "Coverage-guided fuzz testing for cyber-physical systems," in *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2022, pp. 24–33.

[16] D. Serpanos and K. Katsigiannis, "Fuzzing: Cyberphysical system testing for security and dependability," *Computer*, vol. 54, no. 9, pp. 86–89, 2021.

[17] L. J. Moukahal, M. Zulkernine, and M. Soukup, "Vulnerability-oriented fuzz testing for connected autonomous vehicle systems," *IEEE Transactions on Reliability*, vol. 70, no. 4, pp. 1422–1437, 2021.

[18] D. S. Fowler, J. Bryans, S. A. Shaikh, and P. Wooderson, "Fuzz testing for automotive cyber-security," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 239–246.

[19] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[20] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[21] T. Yamaguchi, B. Hoxha, D. Prokhorov, and J. V. Deshmukh, "Specification-guided software fault localization for autonomous mobile systems," in *2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2020, pp. 1–12.

[22] T. Yamaguchi, T. Kaga, A. Donzé, and S. A. Seshia, "Combining requirement mining, software model checking and simulation-based verification for industrial automotive systems," in *2016 Formal Methods in Computer-Aided Design (FMCAD)*, 2016, pp. 201–204.

[23] Q. Thibeault, T. Khandait, G. Pedrielli, and G. Fainekos, "Search based testing for code coverage and falsification in cyber-physical systems," in *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*, 2023, pp. 1–8.

[24] A. Dokhanchi, A. Zutshi, R. T. Sriniva, S. Sankaranarayanan, and G. Fainekos, "Requirements driven falsification with coverage metrics," in *2015 International Conference on Embedded Software (EMSOFT)*, 2015, pp. 31–40.

[25] F. S. de Boer and M. Bonsangue, "Symbolic execution formally explained," *Formal Aspects of Computing*, pp. 1–20, 2021.

[26] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 2004, pp. 152–166.

[27] Y. Shoukry, P. Martin, P. Tabuada, and M. Srivastava, "Non-invasive spoofing attacks for anti-lock braking systems," in *Cryptographic Hardware and Embedded Systems-CHES 2013: 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings 15*. Springer, 2013, pp. 55–72.

[28] Y. Shoukry, P. Martin, Y. Yona, S. Diggavi, and M. Srivastava, "Pycra: Physical challenge-response authentication for active sensors under spoofing attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1004–1015.

[29] Y. Shoukry, P. Nuzzo, A. Puggelli, A. L. Sangiovanni-Vincentelli, S. A. Seshia, and P. Tabuada, "Secure state estimation for cyber-physical systems under sensor attacks: A satisfiability modulo theory approach," *IEEE Transactions on Automatic Control*, vol. 62, no. 10, pp. 4917–4932, 2017.

[30] Y. Shoukry and P. Tabuada, "Event-triggered state observers for sparse sensor noise/attacks," *IEEE Transactions on Automatic Control*, vol. 61, no. 8, pp. 2079–2091, 2015.

[31] A. Barua and M. A. Al Faruque, "Hall spoofing: A non-invasive {DoS} attack on grid-tied solar inverter," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1273–1290.

[32] A. Barua, Y. G. Achamyeleh, and M. A. A. Faruque, "A wolf in sheep's clothing: Spreading deadly pathogens under the disguise of popular music," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 277–291.

[33] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.