Relational Analysis of Sensor Attacks on Cyber-Physical Systems

Jian Xiang*, Nathan Fulton†, Stephen Chong*
*SEAS, Harvard University *{jxiang, chong}@seas.harvard.edu*
†MIT-IBM Watson AI Lab. *nathan@ibm.com*

2021 IEEE 34th Computer Security Foundations Symposium (CSF) | 978-1-7281-7607-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CSF51468.2021.00035

***Abstract*—Cyber-physical systems, such as self-driving cars or autonomous aircraft, must defend against attacks that target sensor hardware. Analyzing system design can help engineers understand how a compromised sensor could impact the system's behavior; however, designing security analyses for cyber-physical systems is difficult due to their combination of discrete dynamics, continuous dynamics, and nondeterminism.**

This paper contributes a framework for modeling and analyzing sensor attacks on cyber-physical systems, using the formalism of hybrid programs. We formalize and analyze two relational properties of a system's robustness. These relational properties respectively express (1) whether a system's safety property can be influenced by sensor attacks, and (2) whether a system's high-integrity state can be affected by sensor attacks. We characterize these relational properties by defining an equivalence relation between a system under attack and the original unattacked system. That is, the system satisfies the robustness properties if executions of the attacked system are appropriately related to executions of the unattacked system.

We present two techniques for reasoning about the equivalence relation and thus proving the relational properties for a system. One proof technique decomposes large proof obligations to smaller proof obligations. The other proof technique adapts the *self-composition technique* from the literature on secure information-flow, allowing us to reduce reasoning about the equivalence of two systems to reasoning about properties of a single system. This technique allows us to reuse existing tools for reasoning about properties of hybrid programs, but is challenging due to the combination of discrete dynamics, continuous dynamics, and nondeterminism.

To validate the usefulness of our relational properties and proof techniques, we present three case studies motivated by real design flaws in existing cyber-physical systems.

I. Introduction

Cyber-physical systems, which consist of both physical and cyber components, are often safety and security critical [1]–[4]. Designing secure cyber-physical systems is difficult because adversaries benefit from a broad attack surface that includes both software controllers and physical components. Sensor attacks often allow an adversary to directly control the system under attack. For example, Cao et al. demonstrate how to manipulate an autonomous vehicle's distance measurements by shining a laser into its Light Detection and Ranging (LiDAR) sensors [5], Humphreys et al. demonstrate how spoofing Global Positioning System (GPS) signals may allow an attacker to force a yacht autopilot to deviate from a designated course [6], and Davidson et al. demonstrate a GPS-based hijacking attack on unmanned aircraft [7]. The breadth of the cyber-physical attack surface affords adversaries a range of attack modalities even when hijacking control is not possible. For example, Son et al. demonstrate how to crash a quadcopter using a magnetic attack on a quadcopter's gyroscopic sensors [8].

Testing-based approaches are insufficient to guarantee the safety of a cyber-physical system, even when the system is not under attack. In a 2016 study on autonomous vehicles, Kalra et al. conclude that a self-driving fleet would need to drive hundreds of millions or sometimes hundreds of billions of miles to provide a purely testing-based reliability case [9]. Driving these miles in a representative set of road conditions would take tens or hundreds of years depending on the size of the test fleet. The intractability of testing-based approaches is also confirmed by incompleteness results [10]. Establishing security is even more difficult than establishing safety.

The importance and difficulty of ensuring the safety of cyber-physical systems motivate a growing body of work on formal verification for embedded and hybrid systems [11]–[15]. However, relatively little work considers formal verification of such systems in the presence of sensor attacks. Some recent work emphasize timing aspects of sensor-related attacks [16], [17]; however, the work model the system's dynamics as a deterministic discrete time dynamical system, whereas most cyber-physical systems are best modeled with a nondeterministic combination of discrete and continuous dynamics.

It is important for cyber-physical system designers to understand whether a compromised sensor can result in undesired behavior, such as violating a safety property or corrupting a critical state. For example, the designer of an adaptive cruise control system might want to verify that the car's minimum following distance is not affected by a compromised GPS sensor.

Understanding the impact of compromised sensors requires us to reason about *relational properties* [18], that is, the relationship between executions of the original uncompromised system and executions of the system where some of the sensors have been compromised. Relational properties are often harder to reason about than functional properties, as they require reasoning simultaneously about multiple executions. And there is less tool support for formal verification of relational properties, compared to functional properties.

In this work, we define and explore two relational properties that characterize the robustness of cyber-physical systems under sensor attacks. Our threat model assumes a powerful attacker that may compromise a subset of sensors and arbitrarily manipulate those sensors' values. We do not model or discover the mechanisms by which an attacker manipulates sensor values; we simply assume they are able to do so.

Our first relational property is *robustness of safety*, which intuitively holds when the compromised sensors are unable to affect whether a given safety property holds in the attacked system. Note that this is not the same as requiring that the attacked system satisfies the safety property. Indeed it may be beyond current verification techniques to determine whether the safety property holds in the uncompromised system, let alone the compromised system. Nonetheless, even in such cases it can be possible to verify that compromised sensors do not affect the safety property. Robustness of safety implies that if the uncompromised system satisfies the safety property then the compromised system will too. Reasoning about robustness of safety separates reasoning about the implications of sensor attacks from reasoning directly about functional properties.

Our second relational property is *robustness of high-integrity state*, which requires that high-integrity parts of a system cannot be influenced by the attacker. For example, returning to our autonomous vehicle example, parts of the system pertaining to steering and braking should be regarded as high integrity and independent from low-integrity sensors such as the interior thermometer. Robustness of high-integrity state is similar to noninterference [19], [20], which requires that low-integrity inputs can not influence high-integrity outputs.

We work within the formalism of *hybrid programs* [10], [21], [22] and their implementation in the theorem prover KeYmaera X [23]. Hybrid programs model cyber-physical systems as hybrid-time dynamical systems, with the discrete time component of the system modeling software components and the continuous time component of the system modeling physical phenomenon.

To define our two relational properties, we introduce the $\mathcal{H}$*-equivalence* relation over hybrid programs, where $\mathcal{H}$ is a set of variables. Intuitively, two hybrid programs are $\mathcal{H}$-equivalent if they agree on the values of all variables in $\mathcal{H}$ at appropriate times. In particular, we define our two relational properties as $\mathcal{H}$-equivalence between the original system and the compromised system (for suitable sets of variables $\mathcal{H}$).

We introduce two sound and tractable techniques to reason about $\mathcal{H}$-equivalence (and thus to prove that robustness of safety and robustness of high-integrity state hold). The first technique decomposes reasoning about $\mathcal{H}$-equivalence of two large programs to reasoning about $\mathcal{H}$-equivalence of their subprograms. The second technique reduces reasoning about $\mathcal{H}$-equivalence of two programs A and B to reasoning about safety properties of a single program that represents both A and B. This reduction allows us to prove relational properties using KeYmaera X, an existing theorem prover for hybrid programs that does not directly support relational reasoning. This technique is inspired by the self-composition technique [24] used to prove noninterference in imperative and deterministic programs. A key challenge we faced in adapting the self-composition technique for hybrid programs is reasoning about nondeterminism and physical dynamics, and in particular, ensuring that certain nondeterministic choices are resolved the same in both executions.

The main contributions of this paper are the following:

1. We introduce a threat model of sensor attacks in the context of hybrid programs that model cyber-physical systems. We show that these sensor attacks can be formalized in terms of syntactic manipulations of hybrid programs. We introduce robustness of safety and robustness of high-integrity state, two relational properties that express security guarantees in the presence of sensor attacks. (Section III)
2. We introduce $\mathcal{H}$-equivalence, an equivalence relation over hybrid programs, and express our relational properties in terms of $\mathcal{H}$-equivalence. (Section IV)
3. We present two techniques for reasoning about $\mathcal{H}$-equivalence and prove their soundness. (Section V)
4. We validate the approach developed throughout the paper through three case studies of non-trivial cyber-physical systems: an anti-lock braking system, the Maneuvering Characteristics Augmentation System (MCAS) of the Boeing 737-MAX, and an autonomous vehicle with a shared communication bus. (Section VI)

We introduce some background about hybrid programs in Section II. Section VII discusses related work.

II. Background

Hybrid programs [22] are a formalism for modeling *cyber-physical systems*, i.e., systems that have both continuous and discrete dynamic behaviors. Hybrid programs can express continuous evolution (as differential equations) as well as discrete transitions.

Figure 1 gives the syntax for hybrid programs. Variables are real-valued and can be deterministically assigned ($x := \theta$, where θ is a real-valued arithmetic term) or nondeterministically assigned ($x := *$). Hybrid program $x' = \theta \& \phi$ expresses the continuous evolution of variables: given the current value of variable x, the system follows the differential equation $x' = \theta$ for some (nondeterministically chosen) amount of time so long as the formula ϕ, the *evolution domain constraint*, holds for all of that time. Note that x can be a vector of variables and then θ is a vector of terms of the same dimension.

Hybrid programs also include the operations of Kleene algebra with tests [25]: sequential composition, nondeterministic choice, nondeterministic repetition, and testing whether a formula holds. Hybrid programs are models of systems and typically over-approximate the possible behaviors of a system.

Real-valued terms θ

x	Real-valued program variable
c	Constant
$\theta_1 \oplus \theta_2$	Computation on terms $\oplus \in \{+, \times\}$

Hybrid Program α, β, P

$x := \theta$	Deterministic assignment of real arithmetic term θ to variable x
$x := *$	Nondeterministic assignment to variable x
$x' = \theta \& \phi$	Continuous evolution along the differential equation system $x' = \theta$ for an arbitrary real duration within the region described by formula ϕ
$?\phi$	Test if formula ϕ is true at the current state
$\alpha; \beta$	Sequential composition of α and β
$\alpha \cup \beta$	Nondeterministic choice between α and β
α^*	Nondeterministic repetition, repeating α zero or more times

Differential Dynamic Logic ϕ, ψ

$\theta_1 \sim \theta_2$	Comparison between real arithmetic terms ($\sim \in \{<, \leq, =, >, \geq\}$)
$\neg\phi$	Negation
$\phi \wedge \psi$	Conjunction
$\phi \vee \psi$	Disjunction
$\phi \rightarrow \psi$	Implication
$\forall x.\ \phi$	Universal quantification
$\exists x.\ \phi$	Existential quantification
$[\alpha]\phi$	Program necessity (true if ϕ is true after each possible execution of hybrid program α)

Fig. 1: Syntax of hybrid programs and dL

Differential dynamic logic (dL) [21], [22], [26] is the dynamic logic [27] of hybrid programs. Figure 1 also gives the syntax for dL formulas. In addition to the standard logical connectives of first-order logic, dL includes primitive propositions that allow comparisons of real-valued terms (which may include derivatives) and *program necessity* $[\alpha]\phi$, which holds in a state if and only if after any possible execution of hybrid program α, formula ϕ holds.

The semantics of dL [21], [26] is a Kripke semantics in which the Kripke model's worlds are the states of the system. Let $\mathbb{R}$ denote the set of real numbers and $\mathbb{V}$ denote the set of variables. A state is a map $\omega : \mathbb{V} \mapsto \mathbb{R}$ assigning a real value $\omega(x)$ to each variable $x \in \mathbb{V}$. The set of all states is denoted by STA. The semantics of hybrid programs and dL are shown in Figure 2. We write $\omega \models \phi$ if formula ϕ is true at state ω. The real value of term θ at state ω is denoted $\omega[\![\theta]\!]$. The semantics of a hybrid program P is expressed as a transition relation $[\![P]\!]$ between states. If $(\omega, \nu) \in [\![P]\!]$ then there is an execution of P that starts in state ω and ends in state ν.

We are often interested in partial correctness formulas of the form $\phi \rightarrow [\alpha]\psi$: if ϕ is true then ψ holds after any possible execution of α. The hybrid program α often has the form `(ctrl;plant)`*, where `ctrl` models atomic actions of the control system and does not contain continous parts (i.e., differential equations); and `plant` models evolution of the physical environment and has the form of $x' = \theta\ \&\phi$. That is, the system is modeled as unbounded repetitions of a controller action followed by an update to the physical environment.

Term semantics

$$\begin{aligned} \omega[\![x]\!] &= \omega(x) \\ \omega[\![c]\!] &= \mathrm{c} \\ \omega[\![\theta_1 \oplus \theta_2]\!] &= \omega[\![\theta_1]\!] \oplus \omega[\![\theta_2]\!] \text{ for } \oplus \in \{+, \times\} \end{aligned}$$

Program semantics

$$\begin{aligned} [\![x := \theta]\!] &= \{(\omega, \nu) \mid \nu(x) = \omega[\![\theta]\!] \text{ and for all other variables } z \neq x,\ \nu(z) = \omega(z)\} \\ [\![x := *]\!] &= \{(\omega, \nu) \mid \nu(z) = \omega(z) \text{ for all variables } z \neq x\} \\ [\![?\phi]\!] &= \{(\omega, \omega) \mid \omega \models \phi\} \\ [\![x' = \theta\ \&\phi]\!] &= \{(\omega, \nu) \mid \text{iff exists solution } \varphi : [0, r] \mapsto \text{STA of } x' = \theta \text{ with } \varphi(0) = \omega \text{ and } \varphi(r) = \nu, \text{ and } \varphi(t) \models \phi \text{ for all } t \in [0, r]\} \\ [\![\alpha \cup \beta]\!] &= [\![\alpha]\!] \cup [\![\beta]\!] \\ [\![\alpha; \beta]\!] &= \{(\omega, \nu) \mid \exists \mu, (\omega, \mu) \in [\![\alpha]\!] \text{ and } (\mu, \nu) \in [\![\beta]\!]\} \\ [\![\alpha^*]\!] &= [\![\alpha]\!]^* \text{ the transitive, reflexive closure of } [\![\alpha]\!] \end{aligned}$$

Formula semantics

$$\begin{aligned} \omega \models \theta_1 \sim \theta_2 &\text{ iff } \omega[\![\theta_1]\!] \sim \omega[\![\theta_2]\!] \text{ for } \sim \in \{=, \leq, <, \geq, >\} \\ \omega \models \phi \wedge \psi &\text{ iff } \omega \models \phi \wedge \omega \models \psi, \text{ similar for } \{\neg, \vee, \rightarrow, \leftrightarrow\} \\ \omega \models \forall x.\phi &\text{ iff } \nu \models \phi \text{ for all states } \nu \text{ that agree with } \omega \text{ except for the value of } x \\ \omega \models \exists x.\phi &\text{ iff } \nu \models \phi \text{ for some state } \nu \text{ that agrees with } \omega \text{ except for the value of } x \\ \omega \models [\alpha]\phi &\text{ iff } \nu \models \phi \text{ for all state } \nu \text{ with } (\omega, \nu) \in [\![\alpha]\!] \end{aligned}$$

Fig. 2: Semantics of hybrid programs and dL

Consider, as an example, an autonomous vehicle that needs to stop before hitting an obstacle.[1] For simplicity, we model the vehicle in just one dimension. Figure 3 shows a $\mathcal{HP}$ *model* (hybrid program model) of such an autonomous vehicle. [2] Let d be the vehicle's distance from the obstacle. The *safety condition* that we would like to enforce (ϕ_{post}) is that d is positive. Let v be the vehicle's velocity towards the obstacle in meters per second (m/s) and let a be the vehicle's acceleration (m/s^2). Let t be the time elapsed since the controller was last invoked. The hybrid program `plant` describes how the physical environment evolves over time interval ϵ: distance changes according to $-v$ (i.e., $d' = -v$), velocity changes according to the acceleration (i.e., $v' = a$), and time passes at a constant rate (i.e., $t' = 1$). The differential equations evolve only within the time interval $t \leq \epsilon$ and if v is non-negative (i.e., $v \geq 0$).

The hybrid program `ctrl` models the vehicle's controller. The vehicle can either accelerate at A m/s^2 or brake at $-B$ m/s^2. For the purposes of the model, the controller chooses nondeterministically between these options. Hybrid programs `accel` and `brake` express the controller accelerating or braking (i.e., setting a to A or $-B$ respectively). The controller can accelerate only if condition ψ is true, which captures that the vehicle can accelerate for the next ϵ seconds only if doing so would still allow it to brake in time to avoid the obstacle.

The formula to be verified is presented on the last line of the $\mathcal{HP}$ model. Given an appropriate precondition ϕ_{pre}, the axioms and proof rules dL can be used to prove that the

[1] Platzer introduces this autonomous vehicle example [22].

[2] Syntax of hybrid programs used in this paper is similar to the syntax used in KeYmaera X, but revised for better presentation.

```
1  Definitions.              /* cannot change over time */
2   R ϵ.          /* time limit for control */
3   R A.          /* acceleration rate */
4   R B.          /* braking rate */
5   B φ_pre       ≡ A ≥ 0 ∧ B ≥ 0 ∧ 2Bd > v^2
6   B φ_post      ≡ d > 0
7   B ψ           ≡ 2Bd > v^2 + (A + B)(Aϵ^2 + 2vϵ)
8   HP accel      ≡ ?ψ; a := A
9   HP brake      ≡ a := −B
10  HP ctrl       ≡ ((accel ∪ brake); t := 0)
11  HP plant      ≡ d' = −v, v' = a, t' = 1 & (v ≥ 0 ∧ t ≤ ϵ)
12 ProgramVariables.        /* may change over time */
13  R t.          /* clock variable */
14  R d.          /* distance to obstacle */
15  R v.          /* vehicle velocity */
16  R a.          /* acceleration of the vehicle */
17 Problem.                 /* dL formula to be proven */
18  φ_pre → [(ctrl; plant)*]φ_post
```

Fig. 3: $\mathcal{HP}$ model of an autonomous vehicle

safety condition ϕ_{post} holds. The tactic-based theorem prover KeYmaera X [23] provides tool support for automating the construction of these proofs.

To present some of our definitions, we need to refer to the variables that occur in a hybrid program [22], [26]. The *free variables* of hybrid program P, denoted $\mathrm{FV}(P)$, is the variables that may potentially be read by P. Values of $\mathrm{FV}(P)$ won't be modified during executions of program P. The *bound variables* of program P, denoted $\mathrm{BV}(P)$, is the set of variables that may potentially be written to by P.[3] We write $\textsc{Var}(P)$ for the set of all variables of P, and have $\textsc{Var}(P) = \mathrm{BV}(P) \cup \mathrm{FV}(P)$. For example, let P be the hybrid program modeling an autonomous vehicle with sensors shown in Figure 3, then $\mathrm{FV}(P) = \{A, B, \epsilon, v, d\}$, $\mathrm{BV}(P) = \{t, v, d, a, t', v', d'\}$, and $\textsc{Var}(P) = \{A, B, \epsilon, t, v, d, a, t', v', d'\}$. Formal definitions of $\mathrm{BV}(P)$, $\mathrm{FV}(P)$, and $\textsc{Var}(P)$ are included in Appendix A.

III. Modeling Sensor Attacks

In this section, we explain how we model the sensor attacks in hybrid programs. In particular, we introduce how sensor readings are modeled and describe our threat model.

A. Modeling Sensor Readings

Hybrid programs typically conflate the values of variables in the physical model and the values ultimately perceived by the sensor. For example, in Figure 3, the hybrid program contains a single continuous variable v that represents the value measured by a sensor; the model does not separate the model's representation of the value of v in the physical model from the software component's representation of v. Therefore, our analysis begins with a hybrid program P_{orig} in which sensor reads are not explicitly modeled. We construct a program P that is equivalent to P_{orig} but separately represents sensor reads and requires that variables holding sensor reads are equal to the underlying sensor's value. For example, v_p may represent the actual physical velocity of a vehicle and it changes according to laws of physics, and v_s may represent

[3]We follow the naming convention of related work on hybrid programs by using the names of free variables and bound variables [26].

```
1  Definitions.
2   R ϵ.          /* time limit for control */
3   R A.          /* acceleration rate */
4   R B.          /* braking rate */
5   B φ_pre       ≡ A ≥ 0 ∧ B ≥ 0 ∧ 2Bd_p > v_p^2
6   B φ_post      ≡ d_p > 0
7   B ψ           ≡ 2Bd_s > v_s^2 + (A + B)(Aϵ^2 + 2v_sϵ)
8   HP accel      ≡ ?ψ; a := A
9   HP brake      ≡ a := −B
10  HP ctrl       ≡ v_s := v_p; d_s := d_p; (accel ∪ brake); t := 0
11  HP plant      ≡ d_p' = −v_p, v_p' = a, t' = 1 & (v_p ≥ 0 ∧ t ≤ ϵ)
12 ProgramVariables.
13  R t.          /* clock variable */
14  R d_p.        /* distance to obstacle (physical) */
15  R v_p.        /* vehicle velocity (physical) */
16  R d_s.        /* distance to obstacle (sensed) */
17  R v_s.        /* vehicle velocity (sensed) */
18  R a.          /* acceleration of the vehicle */
19 Problem.
20  φ_pre → [(ctrl; plant)*]φ_post
```

Fig. 4: $\mathcal{HP}$ model of an autonomous vehicle with sensors

the variable in the controller into which the sensor's value is read. In model P we have the constraint $v_s = v_p$. From P we can derive additional models that allow sensed values to differ from actual physical values. For example, a model that represents the compromise of the velocity sensor would be identical to P except that the constraint $v_s = v_p$ is removed, allowing v_s to take arbitrary values. Similar modifications to P can represent the compromise of other sensors, or of multiple sensors at the same time.

As an example, Figure 4 shows a $\mathcal{HP}$ model of an autonomous vehicle introduced in Figure 3 whose hybrid program separates physical and sensed values: v_p and d_p are physical values of velocity and distance, while v_s and d_s are the corresponding sensed values. Note that the `ctrl` program sets the sensed values equal to the physical values (line 10).

B. Threat Model

We allow attackers to arbitrarily change sensed values. We are not concerned with the physical mechanisms by which an attacker compromises a sensor. Instead, we model sensor attacks as assignments to variables that represent sensed values. Let P be a hybrid program, $S_A \subseteq \mathrm{BV}(P)$ be a set of distinguished variables corresponding to sensors that may be vulnerable to attacks, the sensor attack on P is defined as follows:

Definition 1 (S_A-sensor attack)**.** *For a hybrid program P of the form $(\texttt{ctrl;plant})^*$ and a set of variables $S_A \subseteq \mathrm{BV}(P)$, the S_A-*sensor attack *on program P, denoted $\textsc{Attacked}(P, S_A)$, is the program obtained from P by replacing all assignments to variable $v \in S_A$ with assignment $v := *$.*

For example, let P be the hybrid program $(\texttt{ctrl;plant})^*$ modeling an autonomous vehicle with separate physical and sensed values shown in Figure 4. If the velocity sensor v_s is under attack, program $\textsc{Attacked}(P, \{v_s\})$ would be $(\texttt{ctrl}';\texttt{plant})^*$ where $\texttt{ctrl}'$ is the following:

$$v_s := *;\ d_s := d_p;\ (\texttt{accel} \cup \texttt{brake});\ t := 0.$$

```
1  ...
2  HP voting  ≡ v_{s1} := v_p; v_{s2} := v_p; v_{s3} := v_p;
3               (  (?v_{s1} = v_{s2}; v_s := v_{s1})
4                ∪ (?v_{s1} = v_{s3}; v_s := v_{s1})
5                ∪ (?v_{s2} = v_{s3}; v_s := v_{s2}) )
6  HP ctrl    ≡ voting; d_s := d_p; (accel ∪ brake); t := 0
7  ...
```

Fig. 5: $\mathcal{HP}$ model of an autonomous vehicle with sensor voting

Note that with such a threat model, only the `ctrl` part of a program `(ctrl;plant)*` is modified by an attack, i.e., ATTACKED(`(ctrl; plant)`, S_A) = (ATTACKED(`ctrl`, S_A)); `plant`. Intuitively, it means a sensor attack does not *directly* affect the physical dynamics with which the system interacts.

C. Robustness to Sensor Attacks

We explore the impact of an S_A-sensor attack by studying two relational properties that characterize the robustness of the system to the attack: (1) whether a S_A-sensor attack affects the safety of the system and (2) whether a S_A-sensor attack affects the system's high-integrity state.

a) Robust Safety: Safety is critical in many cyber-physical systems, e.g., a vehicle should not collide with obstacles and pedestrians. We first present the definitions of safety and our relational property robust safety, and then show an example.

Definition 2 (Safety)**.** *A hybrid program P of the form* `(ctrl;plant)*` *is* safe for ϕ_{post} assuming ϕ_{pre}*, denoted* SAFE$(P, \phi_{pre}, \phi_{post})$*, if the formula* $\phi_{pre} \rightarrow [P]\phi_{post}$ *holds.*

This definition says P is safe if for any execution of P whose starting state satisfies ϕ_{pre}, its ending state satisfies safety condition ϕ_{post}.

A system is *robustly safe* if compromise of sensors S_A does not affect whether the system is safe. Note that robust safety does not require that the attacked system is safe; instead it requires that *if* the original system is safe, then the attacked system is also safe. The distinction is important: it allows us to separate the task of reasoning about safety from the task of reasoning about sensor attacks. Indeed, as we will see in a case study in Section VI, it is possible to prove robust safety even when it is beyond current techniques to prove safety.

Definition 3 (Robust safety)**.** *For a hybrid program P of the form* `(ctrl;plant)*` *and a set of variables* $S_A \subseteq \text{BV}(P)$*, P is* robustly safe for ϕ_{post} assuming ϕ_{pre} under the S_A-sensor attack*, denoted* ROBUST$(P, \phi_{pre}, \phi_{post}, S_A)$*, if* SAFE$(P, \phi_{pre}, \phi_{post})$ *implies* SAFE(ATTACKED$(P, S_A), \phi_{pre}, \phi_{post})$.

For example, let P be the hybrid program modeling an autonomous vehicle with sensors shown in Figure 4. P is safe for ϕ_{post} assuming ϕ_{pre} (i.e., SAFE$(P, \phi_{pre}, \phi_{post})$). However, P is not robustly safe for ϕ_{post} assuming ϕ_{pre} under S_A-sensor attack where S_A is $\{v_s\}$, since SAFE(ATTACKED$(P, S_A), \phi_{pre}, \phi_{post})$ doesn't hold.

The system can be modified so that it does satisfy robust safety. For example, we can modify the system to use three velocity sensors (perhaps measuring velocity by different mechanisms) and use a voting scheme to determine the current velocity. Figure 5 shows a model of such a modified system. The physical velocity v_p is sensed by three sensors (line 2), and voting performed to determine the final reading v_s (lines 3–5). The contents elided in Figure 5 are the same as Figure 4.

```
1  ...
2  R T.        /* target temperature */
3  HP ctrl_t   ≡ temp_s := temp_p;
4                (  (?temp_s > T; thermo := -1)
5                 ∪ (?temp_s < T; thermo := 1)
6                 ∪ (?temp_s = T) )
7  HP ctrl     ≡ ctrl_t;
8                 v_s := v_p; d_s := d_p; (accel ∪ brake); t := 0
9  HP plant    ≡ d'_p = -v_p, v'_p = a, temp'_p = thermo; t' = 1
10                & (v_p ≥ 0 ∧ t ≤ ε)
11 ProgramVariables.
12 R temp_s.   /* interior temperature (sensed) */
13 R temp_p.   /* interior temperature (physical) */
14 R thermo.   /* thermostat command */
15 ...
```

Fig. 6: $\mathcal{HP}$ model of an autonomous vehicle with interior temperature control

Let P be the hybrid program modeling an autonomous vehicle with duplicated sensors shown in Figure 5. For any set $S_A \in \{\{v_{s_1}\}, \{v_{s_2}\}, \{v_{s_3}\}\}$, program P is robustly safe under S_A-sensor attack, i.e., P is robustly safe if at most one of the velocity sensors is compromised. Intuitively, this is because $v_s = v_p$ holds after running program `voting`, even if up to one of the velocity sensors is compromised. A systematic approach for proving robustness safety is presented in Section V.

b) Robustness of High-Integrity State: Sensors that may be compromised are low integrity: the sensed values might be under the control of the attacker. By contrast, parts of the system state might be deemed to be high integrity: their values are critical to the correct and secure operation of the system. Low-integrity sensor readings should not be able to affect a system's high-integrity state. For example, an attacker with access to a car's interior temperature sensor should not be able to affect the control of the car's velocity.

We can state this requirement as a relational property: we say the high-integrity state is robust if, for any execution of the system with its low-integrity sensors compromised, there is an execution of the non-compromised system that can achieve the same values on all high-integrity variables. We delay formal definition of robustness of high-integrity state to Section IV.

Let's consider an example. Figure 6 presents a $\mathcal{HP}$ model of an autonomous vehicle with sensors shown in Figure 4 but added with interior temperature control (elided contents in Figure 6 are the same as Figure 4). The vehicle has sensor readings of interior temperature ($temp_s$). The physical temperature ($temp_p$) changes according to $thermo$ that is set by $\texttt{ctrl}_t$ after comparing $temp_s$ with target temperature T (lines 4–6). In this example, the temperature sensor is low-integrity and may be compromised.

A system designer may want to understand if such an attack

can interfere with the vehicle's high-integrity state such as its velocity. Let P be the model of an autonomous vehicle with interior temperature control shown in Figure 6. Intuitively, its velocity (i.e., variable v_p) is robust with respect to sensor $temp_s$: for any execution of ATTACKED(P, $\{temp_s\}$), we have an execution of P that can produce the same values of v_p at every control iteration. The system does satisfy robustness of high-integrity state, and we will prove it in Section V.

IV. $\mathcal{H}$-EQUIVALENCE

This section introduces $\mathcal{H}$-equivalence, a notion of equivalence that allows us to reason about our relational properties.

A. Equivalence of Hybrid Programs

Intuitively, $\mathcal{H}$-equivalence of two programs means that for every execution of one program, there exists an execution of the other program such that the two executions agree on set $\mathcal{H}$ initially and at the end of every control loop iteration, where $\mathcal{H}$ is a set of high-integrity variables.

The formal definition of $\mathcal{H}$-equivalence of programs builds on $\mathcal{H}$-equivalence of program states.

Definition 4 ($\mathcal{H}$-equivalence of program states)**.** *For states $\omega_1, \omega_2 \in$ STA and a set of variables $\mathcal{H}$, states ω_1 and ω_2 are $\mathcal{H}$-equivalent, denoted $\omega_1 \approx_{\mathcal{H}} \omega_2$, if they agree on valuations of all variables in the set $\mathcal{H}$; i.e., $\forall x \in \mathcal{H}, \omega_1(x) = \omega_2(x)$.*

Definition 5 ($\mathcal{H}$-equivalence of programs)**.** *For hybrid programs $P_1 = \alpha^*$, $P_2 = \beta^*$, and a set of variables $\mathcal{H}$, P_1 and P_2 are* $\mathcal{H}$-equivalent*, denoted $P_1 \approx_{\mathcal{H}} P_2$, if they satisfy the following:*

$$
\begin{aligned}
&\forall n : \mathbb{N} \\
&\quad \forall \omega_0, \omega_1 \ldots \omega_n : \text{STA } \textit{such that } \forall i \in 0...(n-1), \\
&\qquad (\omega_i, \omega_{i+1}) \in [\![\alpha]\!] \ (\textit{respectively } [\![\beta]\!]) \\
&\qquad\quad \exists \nu_0, \nu_1 ... \nu_n : \text{STA } \textit{such that } \forall j \in 0...(n-1), \\
&\qquad\qquad (\nu_j, \nu_{j+1}) \in [\![\beta]\!] \ (\textit{respectively } [\![\alpha]\!]) \\
&\qquad\qquad \textit{and } \forall k \in 0...n, \ \omega_k \approx_{\mathcal{H}} \nu_k
\end{aligned}
$$

In the definition, the number n corresponds to an arbitrary number of loop iterations, and the last line indicates that the two executions agree on $\mathcal{H}$ at the beginning and end of every loop iteration. The definition is symmetric.

This definition can be readily adjusted for loop-free programs.

Definition 6 ($\mathcal{H}$-equivalence of loop-free programs)**.** *For two loop-free hybrid programs α and β, and a set of variables $\mathcal{H}$, α and β are* $\mathcal{H}$-equivalent*, denoted $\alpha \approx_{\mathcal{H}} \beta$, if they satisfy the following:*

$$
\begin{aligned}
&\forall \omega_0, \omega_1 : \text{STA } \textit{such that } (\omega_0, \omega_1) \in [\![\alpha]\!] \ (\textit{respectively } [\![\beta]\!]) \\
&\quad \exists \nu_0, \nu_1 : \text{STA } \textit{such that} \\
&\qquad (\nu_0, \nu_1) \in [\![\beta]\!] \ (\textit{respectively } [\![\alpha]\!]) \wedge \omega_0 \approx_{\mathcal{H}} \nu_0 \wedge \omega_1 \approx_{\mathcal{H}} \nu_1
\end{aligned}
$$

Note that Definition 5 is defined in lock-step, i.e., both loops iterate exactly the same number of times [29]. As pointed out by previous work [30], a lock-step approach is sometimes not flexible enough to express and verify some properties, e.g., properties that may hold for two programs that execute for different numbers of iterations. However, such a lock-step definition is reasonable in our setting. According to the threat model, we are comparing a system with compromised sensors and a system with uncompromised sensors and so the attack should not affect the rate of a system's control (i.e., how frequently the system's control loop executes). Thus, the robustness of a system is correctly encoded by a lock-step definition, in which states of a system with and without compromised sensors are consistent after every loop iteration. An additional benefit of this definition is that it is more tractable for verification, which we will explore in Section V.

B. Reasoning about Robustness using $\mathcal{H}$-equivalence

The $\mathcal{H}$-equivalence relation can be used to reason about our two relational properties.

a) Reasoning about Robustness of Safety: Robustness of safety can be established by proving $\mathcal{H}$-equivalence with the help of the following theorem, which states that if program P is $\mathcal{H}$-equivalent to ATTACKED(P, S_A) where $\mathcal{H}$ is the free variables of formulas ϕ_{pre} and ϕ_{post}, then P is robustly safe for ϕ_{post} assuming ϕ_{pre} under the S_A-sensor attack.

Theorem 1 ($\mathcal{H}$-equivalent programs are robustly safe)**.** *For a hybrid program P of the form `(ctrl;plant)*`, a set of variables $S_A \subseteq \text{BV}(P)$, and formulas ϕ_{pre} and ϕ_{post}, if $P \approx_{\text{FV}(\phi_{pre} \wedge \phi_{post})}$ ATTACKED(P, S_A), then*

$$\text{ROBUST}(P, \phi_{pre}, \phi_{post}, S_A)$$

A proof is in Appendix B. Intuitively, the theorem holds because if there were an execution of attacked program such that ϕ_{pre} held at the beginning but ϕ_{post} did not hold at the end of a loop, then there must be an execution of P where the same is true, contradicting the assumption that P is safe.

Note that the converse of Theorem 1 does not hold, i.e., if ROBUST$(P, \phi_{pre}, \phi_{post}, S_A)$, it is not always true that $P \approx_{\text{FV}(\phi_{pre} \wedge \phi_{post})}$ ATTACKED(P, S_A). For example, let P be the program $(b := 1;\ a := b)^*$, formula ϕ_{pre} be $a > 0$, ϕ_{post} be $b > 0$, and S_A be $\{a\}$. Then ROBUST$(P, \phi_{pre}, \phi_{post}, S_A)$ holds, but $P \approx_{\{a,b\}}$ ATTACKED(P, S_A) does not hold since some executions of ATTACKED(P, S_A) (i.e., $(b := 1; a := *)^*$) do not have a matching execution of P.

Theorem 1 reduces proving robustness of safety to proving $\mathcal{H}$-equivalence, which can be achieved by the techniques introduced in Section V.

b) Reasoning about Robustness of High-Integrity State: $\mathcal{H}$-equivalence directly expresses robustness of high-integrity state by letting $\mathcal{H}$ be the set of high-integrity variables. Therefore, proving robustness of high-integrity state is the *same* as proving $\mathcal{H}$-equivalence of the high-integrity state. The following definition makes this clear.

Definition 7 (Robustness of high-integrity state)**.** *For program P of the form `(ctrl;plant)*` and a set of variables S_A*

$\subseteq \mathrm{BV}(P)$, *and set of variables* $\mathcal{H}$, P *satisfies* robustness of high-integrity state $\mathcal{H}$ under the S_A-sensor attack *if* $P \approx_{\mathcal{H}} \textsc{attacked}(P, S_A)$.

V. Proving $\mathcal{H}$-equivalence

We present two sound techniques for reasoning about $\mathcal{H}$-equivalence.

A. Decomposition Approach

Our first approach proves $\mathcal{H}$-equivalence of programs by *decomposing* the proof obligation into simpler obligations for components of the programs. This relies on various compositional properties of $\mathcal{H}$-equivalence, stated here and proven in Appendix B.

Theorem 2. *For all loop-free hybrid programs A, B, C, D and sets $\mathcal{H}$ and $\mathcal{H}'$ of variables, the following properties hold:*

1. $A \approx_{\mathcal{H}} A$;
2. *If* $\mathcal{H} \subseteq \mathcal{H}'$ *and* $A \approx_{\mathcal{H}'} B$, *then* $A \approx_{\mathcal{H}} B$;
3. *If* $A \approx_{\mathcal{H}} B$ *and* $(\textsc{Var}(A) \cup \textsc{Var}(B)) \cap \mathcal{H}' = \emptyset$, *then* $A \approx_{\mathcal{H} \cup \mathcal{H}'} B$;
4. *If* $\mathrm{FV}(C) \cup \mathrm{FV}(D) \subseteq \mathcal{H}$, $A \approx_{\mathcal{H}} B$, *and* $C \approx_{\mathcal{H}} D$, *then* $(A; C) \approx_{\mathcal{H}} (B; D)$;
5. *If* $\mathrm{FV}(A) \cup \mathrm{FV}(B) \subseteq \mathcal{H}$ *and* $A \approx_{\mathcal{H}} B$, *then* $A^* \approx_{\mathcal{H}} B^*$.

Sequential composition (Property 4) is particularly useful. The condition $\mathrm{FV}(C) \cup \mathrm{FV}(D) \subseteq \mathcal{H}$ ensures that $\mathcal{H}$ includes all variables that might affect the evaluation of programs C and D. We use this property when considering $\mathcal{H}$-equivalence of `ctrl; plant` and $\textsc{attacked}(\texttt{ctrl; plant}, S_A) = \textsc{attacked}(\texttt{ctrl}, S_A)\texttt{; plant}$. In particular, if $\mathcal{H}$ includes the actuators by which the controller interacts with the physical environment, then $\texttt{ctrl} \approx_{\mathcal{H}} \textsc{attacked}(\texttt{ctrl}, S_A)$ ensures that the physical dynamics (i.e., program `plant`) can evolve identically in both the attacked and unattacked systems.

Consider the previously presented model of an autonomous vehicle with three velocity sensors shown in Figure 5, and let P be its hybrid program $(\texttt{ctrl; plant})^*$ and α be program P with `voting` excluded, i.e., $P = (\texttt{voting}; \alpha)^*$ and $\textsc{attacked}(P, \{v_{s_1}\}) = (\textsc{attacked}(\texttt{voting}, \{v_{s_1}\}); \alpha)^*$. Here, $\mathrm{FV}(\texttt{voting}) = \mathrm{FV}(\textsc{attacked}(\texttt{voting}, \{v_{s_1}\})) = \{v_p\}$, $\mathrm{FV}(\alpha) = \{v_s, v_p, d_p, A, B, \epsilon\}$, and $\mathrm{FV}(\texttt{voting}; \alpha) = \mathrm{FV}(\textsc{attacked}(\texttt{voting}, \{v_{s_1}\}); \alpha) = \{v_p, d_p, A, B, \epsilon\}$.

By definition of $\approx_{\mathcal{H}}$, we know $\texttt{voting} \approx_{\{v_s, v_p\}} \textsc{attacked}(\texttt{voting}, \{v_{s_1}\})$. By Property 3, we get

$$\texttt{voting} \approx_{\mathrm{FV}(\alpha)} \textsc{attacked}(\texttt{voting}, \{v_{s_1}\})$$

Then by Property 2,

$$(\texttt{voting}; \alpha) \approx_{\mathrm{FV}(\texttt{voting};\alpha)} (\textsc{attacked}(\texttt{voting}, \{v_{s_1}\}); \alpha)$$

Since $\alpha \approx_{\mathrm{FV}(\texttt{voting};\alpha)} \alpha$ (Property 1), by Property 4 we know,

$$\texttt{voting}; \alpha \approx_{\mathrm{FV}(\texttt{voting};\alpha)} \textsc{attacked}(\texttt{voting}, \{v_{s_1}\}); \alpha$$

By Property 5, we get

$$(\texttt{voting}; \alpha)^* \approx_{\mathrm{FV}(\texttt{voting};\alpha)} (\textsc{attacked}(\texttt{voting}, \{v_{s_1}\}); \alpha)^*$$

The free variables of $\phi_{pre} \wedge \phi_{post}$ (shown in Figure 4) are $\{v_p, d_p, A, B, \epsilon\}$, the same as $\mathrm{FV}(\texttt{voting}; \alpha)$. Thus, by Theorem 1, we have $\textsc{robust}(P, \phi_{pre}, \phi_{post}, \{v_{s_1}\})$.

B. Self-Composition Approach

The second approach toward proving $\mathcal{H}$-equivalence is inspired by self-composition [24], [31], a proof technique often used for proving noninterference [19], [20]. Noninterference is a well-known strong information security property that, intuitively, guarantees that confidential inputs do not influence observable outputs, or dually guarantees that low-integrity inputs of a system do not affect high-integrity outputs. Noninterference is a relational property: it compares two executions of a program with different low-integrity inputs.

To develop an intuition for how the self-composition technique is used to prove noninterference, consider the problem of checking whether low-integrity inputs of a deterministic program affect high-integrity outputs. Construct two copies of the program, renaming the program variables so that the variables in the two copies are disjoint. Set the high-integrity inputs in both copies to identical values but allow the low-integrity inputs to take different values. Now, sequentially compose these two programs together. If the composed program can terminate in a state where the corresponding high-integrity outputs differ, then the original program does not satisfy noninterference; conversely, if in all executions of the composed program, the high-integrity outputs are the same, then the original program satisfies noninterference. Intuitively, the composition of the two copies allows a single program to represent two executions of the original program, reducing checking a relational property of the original problem to checking a safety property of the composed program.

Using the same insights, we develop a self-composition technique for hybrid programs, allowing us to use existing verification tools such as KeYmaera X (which can reason about safety properties of hybrid programs) to reason about $\mathcal{H}$-equivalence of two hybrid programs.

It is non-trivial to adapt the self-composition approach to hybrid programs due to the nondeterminism in hybrid programs. In particular, to show that two executions of the same hybrid program are in an appropriate relation, it may be necessary to force (some of) the nondeterminism in the two executions to resolve in the same way. For example, a nondeterministic choice in a hybrid program may represent a decision by a driver to brake or accelerate; the driver's decision is assumed to be a high-integrity input, and so the resolution of the nondeterministic choice should be the same in both executions. The self-composition must somehow couple the nondeterministic choices to ensure this. Nondeterministic assignment must be similarly handled, i.e., resolution of high-integrity nondeterminism must be coupled in the two executions.

An additional source of nondeterminism in hybrid programs is the duration of physical evolution. The program construct for physical dynamics, $x' = \theta \& \phi$, specifies that the variable(s)

evolve according to the differential equation system $x' = \theta$ for an *arbitrary duration* within the region described by formula ϕ. The duration is chosen nondeterministically.

Our self-composition technique takes as input a program P and set of sensor variables S_A and creates a program that represents an execution of each of P and $\text{ATTACKED}(P, S_A)$. We ensure that the composed program (1) resolves high-integrity nondeterministic choices and assignments the same in both executions; and (2) has the same duration for corresponding physical evolutions.

To ensure that the two executions are appropriately related, we produce a formula that encodes that the two executions have the same values for high-integrity variables; we assume this formula holds at the beginning of the executions, and require the formula to hold at the end of every control iteration. If we can prove that this is the case, then we have proved that if the two executions (1) have the same values for high-integrity inputs at the beginning of their executions, (2) follow the same decisions on high-integrity nondeterminism during their executions, and (3) evolve for the same duration, then the two executions have the same values for high-integrity variables at the end of every control iteration.

Our self-composition approach has some limitations on the hybrid programs to which it applies. First, it is applicable only for hybrid programs of the form `(ctrl; plant)*`. Second, it is applicable only for hybrid programs that have *total* semantics for low-integrity inputs. Intuitively, it means if a program has a *valid* execution for an input state ω (i.e., exists a state ν such that $(\omega, \nu) \in [\![P]\!]$), then the program has a valid execution for every input state that differs with ω only on low-integrity inputs. The reason for this requirement is that self-composition uses a single program to represent two executions; this composed program has a valid execution only if both executions are valid. Since the two executions differ only on low-integrity inputs, our technique works only if semantics of the unattacked program is total on low-integrity inputs. A straightforward syntactic checker can be developed to check whether a hybrid program meets this requirement. More discussion about the limitation and the syntactic checker can be found in Appendix C.

The rest of this section describes in detail our self-composition approach: how to construct a single program that represents an execution of P and $\text{ATTACKED}(P, S_A)$, and then prove it correct. At a high level, our approach works by (1) converting program P to a canonical form P_{canon} that makes high-integrity nondeterministic choices and assignments explicit; and then (2) composing P_{canon} and $\text{ATTACKED}(P_{canon}, S_A)$ to ensure that the values of high-integrity nondeterministic choices and assignments, and evolution durations are the *same* for both executions.

a) Canonical Form for Hybrid Programs: Given a hybrid program of the form $(ctrl; plant)^*$, we rewrite it to a canonical form $(choices; ctrl'; plant)^*$ such that (1) each high-integrity

```
...
HP choices ≡ c:= *
HP ctrl    ≡ choices; ctrl_t; v_s:= v_p; d_s:= d_p;
                 (if (c) then accel else brake); t := 0
...
ProgramVariables.
B c.       /* choice variable */
...
```

Fig. 7: $\mathcal{HP}$ model of an autonomous vehicle with interior temperature control shown in Figure 6 whose hybrid program is rewritten to canonical form with a choice variable c

nondeterministic choice $\alpha \cup \beta$ in $ctrl$ is turned into a construct *if* c *then* α *else* β[4] in $ctrl'$, and (2) each high-integrity nondeterministic assignment $x := *$ in $ctrl$ is turned into $x := c$ in $ctrl'$, where c is a fresh variable, and $choices$ contains a nondeterministic assignment $c := *$. The program fragment $choices$ consists solely of a sequence of these nondeterministic assignments to these *choice variables*. Note that $(ctrl; plant)^*$ is semantically equivalent to $(choices; ctrl'; plant)^*$.

The goal of the canonical form is to make it easier to share the same nondeterministic choices and assignments between the two executions: when we compose the two programs, they will essentially share the same $choices$ program.

For example, Figure 7 shows the previously presented model of an autonomous vehicle with interior temperature control shown in Figure 6 whose hybrid program is rewritten to the canonical form (elided contents in Figure 7 are the same as Figure 6). The program has a nondeterministic choice variable c that represents a decision to brake or accelerate. This choice is considered high-integrity.

b) Hybrid Program with Renaming: Note that program P and $\text{ATTACKED}(P, S_A)$ have the same set of variables. To compare executions of P and $\text{ATTACKED}(P, S_A)$ in a composition, we need to rename *bound variables* in one of the two programs. Renaming is needed only for bound variables, since their values may differ during executions. Other variables are read-only and their values will be the same for executions of program P and $\text{ATTACKED}(P, S_A)$. Thus, these variables can be shared by both programs, and renaming is not needed.

To help us with renaming, we define *renaming functions* that map all and only the bound variables of a program to fresh variables.

Definition 8 (Renaming function)**.** *For hybrid program P, function $\xi : \text{VAR}(P) \to V$ (where V is a set of variables) is a* renaming function for P *if:*

1) *ξ is a bijection;*
2) *For all $x \in \text{BV}(P)$, $\xi(x) \notin \text{VAR}(P)$;*
3) *For all $x \in \text{VAR}(P) \setminus \text{BV}(P)$, $\xi(x) = x$.*

We write $\xi(P)$ for the program identical to P but whose variables have been renamed according to function ξ. We

[4]Construct *if* ϕ *then* α *else* β is syntactic sugar for $(?\phi; \alpha) \cup (?\neg\phi; \beta)$.

also apply renaming functions to states and formulas, with the obvious meaning.

c) Interleaved Composition: We develop an *interleaved composition* that composes two programs so their executions have the same values for high-integrity nondeterministic choices and assignments, and last the same evolution duration.

Definition 9 (Interleaved composition)**.** *Given a hybrid program $P = (choices;\ ctrl;\ (x' = \theta\,\&\,\phi))^*$ in canonical form, a renaming function ξ for P, a set of variables $S_A \subseteq \mathrm{BV}(P)$, the* interleaved composition of P under S_A attack with renaming function ξ, *denoted* $\mathrm{IC}(P, S_A, \xi)$*, is the following program:*

$$(choices;\ ctrl; \textsc{sub}(choices, \xi);\ \xi(\textsc{attacked}(ctrl, S_A));\ (x' = \theta, \xi(x' = \theta)\,\&\,\phi \wedge \xi(\phi)))^*$$

Where function $\textsc{sub}(choices, \xi)$ *replaces* $c_i := *$ *in program choices with* $\xi(c_i) := c_i$ *for all variables* c_i *in* $\mathrm{BV}(choices)$.

The composition has the following properties: (1) control components from two programs are executed sequentially (i.e., $choices;\ ctrl;\ \textsc{sub}(choices, \xi);\ \xi(\textsc{attacked}(ctrl, S_A))$); (2) plants are executed in parallel (i.e., $x' = \theta, \xi(x' = \theta)$) [32]; (3) the evolution constraint is a conjunction of the two evolution constraints (i.e., $\phi \wedge \xi(\phi)$), and (4) nondeterministic choices in $choices$ used by $ctrl$ and their counterparts used by $\xi(\textsc{attacked}(ctrl, S_A))$ have the same values.

For example, let P be the previously presented hybrid program (in canonical form) of an autonomous vehicle with interior temperature control shown in Figure 7. Figure 8 shows $\mathrm{IC}(P, \{temp_s\}, \xi)$, where function ξ renames bound variables in $\textsc{attacked}(P, S_A)$ with subscript 1. Program `ctrl'` and `Plant'` compose two programs as described in Definition 9 (lines 27–30). Line 18 shows the effect of function $\textsc{sub}(choices, \xi)$: substituting $c = *$ with $c_1 = c$ in $choices$. The choice represents a decision to accelerate or brake, which is high-integrity. The resolution of this choice should be the same in both executions.

d) Proving $\mathcal{H}$-equivalence with an Interleaved Composition: Given an interleaved composition $\mathrm{IC}(P, S_A, \xi)$, to prove that two programs are $\mathcal{H}$-equivalent on a set $\mathcal{H}$, we need to first identify a set η of high-integrity variables on which the evaluation of variables in $\mathcal{H}$ depend. Then we construct a formula to express that the two program executions have the same values for variables in set η, and finally prove that, for any execution of the composition, if the formula holds initially, it would hold at the end of every control loop iteration of the execution.

Definition 10 (Equivalence formula)**.** *For a set η of variables, a renaming function ξ such that $\eta \subseteq dom(\xi)$, the equivalence formula of η and ξ, denoted eq_η^ξ, is defined as:*

$$eq_\eta^\xi \equiv \bigwedge_{x \in \eta} (x = \xi(x))$$

Then the desired property is, for any execution of the composition, if the equivalence formula holds at the beginning of an execution, it holds at the end of every control loop iteration of the execution. That means, we want to prove the following:

$$eq_\eta^\xi \rightarrow [\mathrm{IC}(P, S_A, \xi)] eq_\eta^\xi$$

```
1  Definitions.
2   R ϵ.                  /* time limit of control */
3   R A.                  /* acceleration rate */
4   R B.                  /* braking rate */
5   R T.                  /* target temperature */
6   B eq_η          ≡ v_p = v_p1 ∧ d_p = d_p1
7   B ψ             ≡ 2Bd_s > v_s^2 + (A + B)(Aϵ^2 + 2v_sϵ)
8   HP choices      ≡ c:= *
9   HP ctrl_t       ≡ temp_s:= temp_p;
10                    ( (?temp_s > T; thermo:= -1)
11                    ∪ (?temp_s < T; thermo:= 1)
12                    ∪ (?temp_s = T) )
13  HP accel        ≡ ?ψ; a := A
14  HP brake        ≡ a := −B
15  HP ctrl         ≡ ctrl_t; v_s:= v_p; d_s:= d_p;
16                    (if (c) then accel else brake); t := 0
17  B ψ1            ≡ 2Bd_s1 > v_s1^2 + (A + B)(Aϵ^2 + 2v_s1ϵ)
18  HP choices_1    ≡ c_1:= c
19  HP ctrl_t1      ≡ temp_s1:= *;
20                    ( (?temp_s1 > T; thermo_1:= -1)
21                    ∪ (?temp_s1 < T; thermo_1:= 1)
22                    ∪ (?temp_s1 = T) )
23  HP accel_1      ≡ ?ψ_1; a_1 := A
24  HP brake_1      ≡ a_1 := −B
25  HP ctrl_1       ≡ ctrl_t1; v_s1:= v_p1; d_s1:= d_p1;
26                    (if (c_1) then accel_1 else brake_1); t_1 := 0
27  HP ctrl'        ≡ choices; ctrl; choices_1; ctrl_1
28  HP plant'       ≡ d'_p = −v_p, v'_p = a, temp'_p = thermo, t' = 1
29                    d'_p1 = −v_p1, v'_p1 = a_1, temp'_p1 = thermo_1, t'_1 = 1
30                    & (v_p ≥ 0 ∧ v_p1 ≥ 0 ∧ t ≤ ϵ ∧ t_1 ≤ ϵ)
31  ProgramVariables.
32  R c, c_1.             /* choice variables */
33  R t, t_1.             /* clock variables */
34  R d_p, d_p1.          /* distance to obstacle (physical) */
35  R d_s, d_s1.          /* distance to obstacle (sensed) */
36  R v_p, v_p1.          /* vehicle velocity (physical) */
37  R v_s, v_s1.          /* vehicle velocity (sensed) */
38  R a, a_1.             /* acceleration of the vehicle */
39  R temp_s, temp_s1.    /* interior temperature (sensed) */
40  R temp_p, temp_p1.    /* interior temperature (physical) */
41  R thermo, thermo_1.   /* rates of change for temperature */
42  Problem.
43   eq_η → [(ctrl'; plant')*]eq_η
```

Fig. 8: Interleaved composition of the hybrid program (in canonical form) modeling an autonomous vehicle with interior temperature control shown in Figure 7

For example, eq_η in Figure 8 (line 6) encodes that the two executions have the same position ($d_p = d_{p_1}$) and velocity ($v_p = v_{p_1}$). The desired property is shown at line 43.

We have proven this property using Keymaera X. Intuitively, proving this property means that for any execution of the autonomous vehicle model, whether or not its temperature sensor is compromised, if the vehicle starts with the same position and velocity, makes the same control decisions for acceleration and brake, and runs for the same duration, it would end with the same position and velocity.

e) Soundness: The soundness theorem links the self-composition approach with proving $\mathcal{H}$-equivalence. Proof of this theorem is based on trace semantics of hybrid pro-

grams [33], [34] and can be found in the extended version of this paper.

Theorem 3 (Soundness of the self-composition approach)**.** *For hybrid program P and P_c, a set $S_A \subseteq \mathrm{BV}(P)$, a renaming function ξ of P_c, a set of variables $\eta \subseteq \mathrm{BV}(P)$, and a set $\mathcal{H} \subseteq \eta$, if P_c is P in canonical form, $S_A \cap \eta = \emptyset$, and $eq_\eta^\xi \rightarrow [\mathrm{IC}(P_c, S_A, \xi)]eq_\eta^\xi$, then $P \approx_{\mathcal{H}} \mathrm{ATTACKED}(P, S_A)$.*

Note that the condition $S_A \cap \eta = \emptyset$ indicates that the adversary cannot compromise high-integrity variables.

f) Applicability: Our self-composition technique applies to a subset of problems of interest rather than general problems. In particular, our technique requires that two executions having the same duration at every control iteration for the plant, and identical values for high-integrity nondeterministic assignments. Our self-composition technique *cannot* be applied to compare two executions that evolve for different durations or that resolve high-integrity nondeterministic choices and high-integrity nondeterministic assignments differently. However, these restrictions arise naturally for many systems. First, requiring the same duration of evolution for both executions corresponds to the control system having the same frequency of operation. That is, the rate of the the system's control can't be influenced by the attacker. Second, high-integrity non-deterministic choices and high-integrity non-deterministic assignments are used to model exactly the nondeterminism that cannot be influenced by the attacker. As such, they should be resolved the same in both executions. For example, when considering how a corrupted temperature sensor can affect a (non-autonomous) vehicle, the driver's decisions (i.e., whether to accelerate or brake) would be modeled with a high-integrity nondeterministic choice, since we are concerned with understanding the relationship between two executions where the driver makes the same decisions but in one execution the sensor is corrupted. If in the two executions the driver is making different choices, the two executions might diverge almost arbitrarily, even if the corrupted sensor has no security impact. If, on the other hand, we want to use this technique to determine whether an autonomous vehicle's driving subsystem can be influenced by a corrupted temperature sensor, we would need a more precise model of the system that does not use nondeterministic choice between accelerating and braking to model the driving subsystem's decisions. That is, high-integrity nondeterministic choices are by assumption choices that cannot be influenced by the attacker.

VI. Case Studies

To demonstrate the feasibility and efficacy of our approach, we conduct three case studies of non-trivial systems. The first two case studies analyze robustness of safety with the decomposition approach, and the third one proves robustness of high-integrity state with the self-composition approach.

A. Case Study: an Anti-lock Braking System

System designers may wonder if the system is robustly safe against sensor attacks or if their countermeasures are effective.

```
1  Definitions.
2   HP ctrl   ≡ ω_s := ω_p; v_s := v_p;
3               λ_c := (v_s − ω_s ∗ R)/v_s; λ_p := (v_p − ω_p ∗ R)/v_p;
4               μ_p := C_1(1 − e^{−C_2 λ_p}) − C_3 λ_p;
5               ( (?λ_c < λ_ref; BRAKE := 0; T_b := 0)
6                ∪ (?λ_c = λ_ref; ?True)
7                ∪ (?λ_c > λ_ref; BRAKE := 1; T_b := 1200) ); t := 0
8   HP Plant  ≡ v_p' = (−μ_p F_N)/m, ω_p' = (μ_p F_N R − T_b)/J, t' = 1
9                & v_p ≥ 0 ∧ ω_p ≥ 0 ∧ t ≤ ε
10  B φ_pre     ≡ (v_p = 100 ∧ ω_p ≥ 0)
11  B φ_post    ≡ (v_p > 25 → ω_p ≥ 1)
12  R ε.          /* control interval */
13  R C_1, C_2, C_3.  /* constant for computing μ */
14  R J, R.       /* wheel inertia and wheel radius */
15  R F_N, m.     /* normal force and vehicle mass */
16  R λ_ref.      /* reference value of wheel slip ratio */
17 ProgramVariables.
18  R BRAKE.      /* brake status */
19  R T_b.        /* braking torque */
20  R ω_p, ω_s.   /* wheel speed (physical and sensed ) */
21  R v_p, v_s.   /* vehicle speed (physical and sensed) */
22  R λ_p, λ_c.   /* wheel slip (physical and calculated) */
23  R μ_p.        /* adhesion coefficient */
24  R t.          /* clock variable */
25 Problem.
26   φ_pre → [(ctrl; plant)*]φ_post
```

Fig. 9: $\mathcal{HP}$ model of an ABS system

This case study demonstrates analyzing robustness of safety with the decomposition approach in an Anti-lock Braking System (ABS). An ABS is a safety braking system used on aircraft and vehicles. It operates by preventing the wheels from locking up during braking, thereby maintaining tractive contact with the road surface. ABS monitors the speed of wheels using the wheel-speed sensors. If the controller sees that one wheel is decelerating at a rate that couldn't possibly correspond to the vehicle's rate of deceleration, it reduces the brake pressure applied to that wheel, which allows it to turn faster. Once the wheel is back up to speed, it applies the brake again [35].

a) Modeling ABS: Figure 9 shows a model of an ABS system [36], [37]. The model assumes a single wheel and uses a simple controller that turns on and off maximum braking torque. Intuitively, ABS systems are designed to achieve the maximum friction under certain circumstances (e.g., braking on icy road surface). They achieve this by maintaining an ideal slip ratio (e.g., λ_{ref} in Figure 9). Our controller switches the brake on and off based on the calculated slip ratio (λ_c) and reference slip ratio (lines 5–7). The calculated slip ratio is computed using sensed wheel speed and vehicle speed (lines 2–3). The physical slip ratio (λ_p) depends on physical wheel speed (v_p) and vehicle speed (w_p), which are affected by braking torque (T_b) and adhesion coefficient (μ_p) that depends on the physical slip ratio (line 4).

The initial condition of the ABS system (ϕ_{pre}) is that the vehicle is moving at a high speed and its wheel speed is not negative (line 10). The safety condition (ϕ_{post}) is that the vehicle's wheel should not lock if the current vehicle speed is large (line 11) [38].

b) Modeling Non-invasive Attack on ABS: Previous research has demonstrated attacks on ABS through physical channels [39]. By placing a thin electromagnetic actuator near the ABS wheel-speed sensors, an attacker can inject magnetic fields to both cancel the true measured signal and inject a malicious signal, thus spoofing the measured wheel speeds. Such an attack is a S_A-sensor attack, where $S_A = \{\omega_s\}$, on the wheel-speed sensor. Let P be the hybrid program modeling an ABS system shown in Figure 9. Then $\text{ATTACKED}(P, \{\omega_s\})$ is program P with line 2 changed into the following:

$$\texttt{ctrl} \equiv \omega_s := *;\ v_s := v_p$$

Program P is not robustly safe when the sensor ω_s is compromised: assuming $\text{SAFE}(P, \phi_{pre}, \phi_{post})$ holds, $\text{SAFE}(\text{ATTACKED}(P, \{\omega_s\}), \phi_{pre}, \phi_{post})$ doesn't necessarily hold, since ω_s can be an arbitrary value.

c) Designing Robustly Safe ABS System: System designers, in attempts to make ABS system modeled in Figure 9 safer, would be confident in their design if the system with countermeasures can be proven to be robustly safe against the attack.

Assume that designers deploy three wheel-speed sensors and a majority voting scheme in the ABS system modeled in Figure 9. The countermeasure can be modeled by changing line 2 in Figure 9 into the $\texttt{ctrl} \equiv \texttt{voting}; v_s := v_p;$, where $\texttt{voting}$ is the following:

$$\begin{aligned}\texttt{voting} \equiv\ & \omega_{s_1} := \omega_p; \omega_{s_2} := \omega_p; \omega_{s_3} := \omega_p; \\ & \texttt{if}\ (\omega_{s_1} = \omega_{s_2} \vee \omega_{s_1} = \omega_{s_3}) \\ & \texttt{then}\ \omega_s := \omega_{s_1}\ \texttt{else}\ \omega_s := \omega_{s_2}\end{aligned}$$

Using the decomposition approach, we can prove that such an ABS system is robustly safe if only one wheel-speed sensor is compromised. The proof can be found in Appendix B.

B. Case study: Boeing 737-MAX

Robustness of safety is a relational property: if the original system is safe then the attacked system will be safe too. Importantly, this separates reasoning about the implications of sensor attacks from reasoning directly about safety properties. Proving a system's safety is often labor-intensive and may even be epistemically problematic. For example, many systems must be verified and validated empirically because their correctness properties are not possible to state in a formal language. However, when it is not easy or even impossible to formally verify safety, it is often still possible to prove that compromised sensors do not affect the safety property.

To demonstrate this advantage of relational reasoning, we present a case study inspired by the Boeing 737-MAX Maneuvering Characteristics Augmentation System (MCAS) [40]. The 737-MAX's dynamics are extremely complicated, and proving properties about similar stabilization systems is an open challenge in hybrid systems verification [41]. Nonetheless, we are able to analyze robustness of safety against faults or attacks on the angle of attack (AOA) sensor used by the 737-MAX MCAS, even without an analysis of the system's overall safety property or the MAX's flight dynamics.

```
1  Definitions.
2   B φ_pre          /* preconditions (abstract) */
3   B φ_post         /* functional safety property (abstract) */
4   HP plant.        /* plane's dynamics (abstract) */
5   HP MCAS.         /* MCAS actuation (abstract) */
6   HP ctrl_aoa      ≡ (  (s_L := aoa_p; s_R := *)
7                         ∪ (s_L := *; s_R := aoa_p) );
8                         (aoa_s := s_L ∪ aoa_s := s_R)
9   HP ctrl          ≡ ctrl_aoa; MCAS(aoa_s)
10 ProgramVariables.
11  R aoa_p.         /* physical AOA */
12  R s_L, s_R.      /* left and right AOA sensor */
13  R aoa_s.         /* AOA used by MCAS */
14 Problem.
15  φ_pre → [(ctrl;plant)*]φ_post
```

Fig. 10: A Simple Model of Boeing737 Max flawed MCAS.

a) Modeling MCAS: The MCAS caused at least two deadly crashes in 2019 [42]. MCAS was added to compensate for instability induced by the 737-MAX's new engines. Adding new engines to an existing airframe resulting in an aircraft whose nose tended to pitch upward, risking stalls. The MCAS adjusts the plane's horizontal stabilizer in order to push the nose down when the aircraft is operating in manual flight at an elevated angle of attack (AOA). In many 737-MAX planes, the MCAS is activated by inputs from only one of the airplane's two angle of attack sensors. In both 2019 crashes, the MCAS was triggered repeatedly due to a failed AOA sensor. These false readings caused the MCAS software to repeatedly push the plane's nose down, ultimately interacting with manual inputs in a way that caused violently parabolic flight paths terminating in lost altitude and an eventual crash.

Figure 10 shows a simplified model of the original MCAS. The controller, plane's flight dynamics, and manual control inputs are all left abstract: the model focuses only on how values read by the left and right AOA sensors are used in MCAS. On each control iteration, one of the two AOA sensors is randomly chosen ($\texttt{ctrl}_{aoa}$) and the MCAS is activated using the value of the chosen sensor (line 9). In this model, we intentionally omit details about the flight controller, MCAS system, and flight dynamics. Even with a high-fidelity model [43], proving correctness for the 737-MAX MCAS requires advances in state-of-the-art reachability analysis for hybrid time systems; fortunately, relational reasoning allows us to nonetheless analyze robustness of the system against faults or attacks on the AOA sensors.

b) Reasoning for Robustness of Safety: Program $\texttt{ctrl}_{aoa}$ is not robustly safe if either of the AOA sensors is compromised, since aoa_s can have false readings. Therefore, the system is not robustly safe for attacks on AOA sensors. Boeing's proposed fix to MCAS includes a requirement that the controller should compare inputs from both AOA sensors [44], which can be modeled by adding the following at the end of $\texttt{ctrl}_{aoa}$:

$$(?s_L = s_R)\ \cup\ (?\neg(s_L = s_R);\ aoa_s := 0)$$

We can prove that the system with this fix is robustly safe. Let $\texttt{ctrl}'_{aoa}$ be $\texttt{ctrl}_{aoa}$ with this simple fix. Then we know $\texttt{ctrl}'_{aoa} \approx_{\{aoa_s\}} \text{ATTACKED}(\texttt{ctrl}'_{aoa}, \{s_L\})$

```
1  Definitions.
2   ...
3   HP accel     ≡ ?ψ; busV := A
4   HP brake     ≡ busV := −B
5   HP ctrl_v    ≡ v_s := v_p; d_s := d_p;
6                  if (c) then accel else brake
7   HP ctrl_t    ≡ temp_s := temp_p;
8                  ( (?temp_s > T; busV := -1)
9                  ∪ (?temp_s < T; busV := 1)
10                 ∪ (?temp_s = T) )
11  HP ctrl_bus  ≡ ( (?busV = a; ctrl_t; busH := 1)
12                 ∪ busH := 0 )
13  HP ctrl_r    ≡ ( (?busH = 0; a := busV)
14                 ∪ (?busH = 1; thermo := busV) )
15  HP ctrl      ≡ choices; ctrl_v; ctrl_bus; ctrl_r; t := 0
16  HP plant     ≡ d_p' = −v_p, v_p' = a, temp' = thermo, t' = 1
17                   & (v_p ≥ 0 ∧ t ≤ ε)
18  ProgramVariables.
19   R busV.   /* value on the bus */
20   R busH.   /* header indicating the type of information */
21   ...
22  Problem.
23   φ_pre → [(ctrl; plant)*]φ_post
```

Fig. 11: $\mathcal{HP}$ model (in canonical form) of an autonomous vehicle with an internal bus

and $\texttt{ctrl}'_{aoa} \approx_{\{aoa_s\}} \text{ATTACKED}(\texttt{ctrl}'_{aoa}, \{s_R\})$. Program $(\texttt{ctrl;plant})^*$ with $\texttt{ctrl}'_{aoa}$ is robustly safe by the decomposition approach. The proof is included in Appendix B.

C. Case Study: An Autonomous Vehicle with an Internal Bus

Figure 8 shows the self-composition approach with a model of an autonomous vehicle with interior temperature control. However, the model doesn't account for any internal communication mechanisms. In modern vehicles, Electronic Control Units (ECUs) oversee a broad range of functionality, including the drivetrain, lighting, and entertainment. They often communicate through an *internal bus* [45].

In this case study, we explore how to use the self-composition approach to analyze robustness of high-integrity state in a model of an autonomous vehicle with an internal bus that communicates both low-integrity messages (sensed temperature) and a high-integrity messages (sensed velocity). We are interested in whether the high-integrity state (i.e., velocity) is robust when the temperature sensor is compromised.

a) Modeling a Vehicle with an Internal Bus: Figure 11 shows a model (in canonical form) of such a system (elided contents are the same as in the model previously presented in Figure 8). We model the bus using two variables: a value variable ($busV$), which indicates the current value that sits on the bus, and a header variable ($busH$), which indicates the type of information that sits on the bus: $busH = 0$ for acceleration, $busH = 1$ for temperature. Exactly one message is communicated via the bus at each control loop iteration. Acceleration messages have higher priority over thermostat messages. Program ($\texttt{ctrl}_v$) first sets the bus value to the next acceleration value. Program $\texttt{ctrl}_{bus}$ then checks if the value has changed from the existing acceleration value. If not, it activates temperature control ($\texttt{ctrl}_t$) to set the bus value to desired thermostat value (line 11). Otherwise, $busH$ is sent to 0 to indicate that a new acceleration value has arrived (line 12). Program $\texttt{ctrl}_r$ reads a value off the bus and sets corresponding values based on the header (lines 13–14).

b) Robust High-Integrity State: We are interested in whether the vehicle's high-integrity state—v_p, the velocity of the vehicle—is robust when its low-integrity sensor ($temp_s$) is compromised. Specifically, we wonder whether $P \approx_{\{v_p\}} \text{ATTACKED}(P, \{temp_s\})$, where P is the model shown in Figure 11. We can prove this using the self-composition approach. Figure 12 shows $\text{IC}(P, \{temp_s\}, \xi)$, where ξ renames variables in $\text{BV}(P)$ with a subscript 1 (we elide the descriptions of program variables introduced in Figure 11). By choosing the equivalence formula as $v_p = v_{p_1} \wedge d_p = d_{p_1} \wedge a = a_1$ (line 3), we are able to prove the desired property at line 40. Proving this property means for this vehicle, its high-integrity variable v_p, d_p, and a are robust when its temperature sensor is compromised. We have proven the model in Figure 12 using KeYmaera X.

Note that the decomposition approach and self-composition approach may work well in different settings. The decomposition approach is easy to apply and works well when the sensor attack affects a small portion of the system, as in our first two case studies; by constrast, the self-composition approach can handle cases where the effect of the attack may be complicated—as in our third case study—but requires more effort to use. It is possible to combine the two techniques to prove robustness properties of complicated cases. For example, if we can identify that only a single component of a large system is affected by an attack, the self-composition approach can be used to prove robustness of this component, while the decomposition approach delivers the robustness proof of the whole system.

VII. Related Work

Formal analysis of sensor attacks Lanotte et al. [16], [17] propose formal approaches to model and analyze sensor attacks with a process calculus. The threat model allows attacks that manipulate sensor readings or control commands to compromise state. Their model of physics is discrete and it focuses on timing aspects of attacks on sensors and actuators. In comparison, we analyze relational properties in systems whose dynamics are modeled with differential equations and we introduce techniques to establish proofs of these properties.

Bernardeschi et al. [46] introduce a framework to analyze the effects of attacks on sensors and actuators. Controllers of systems are specified using the formalism PVS [47]. The physical parts are assumed to be described by other modeling tools. Their threat model is similar to ours: the effect of an attack is a set of assignments to the variables defined in the controller. Simulation is used to analyze effects of attacks. By contrast, we focus on formal analysis for the whole system and propose concrete proof techniques for relational properties.

Analyzing relational properties of cyber-physical systems Akella et al. [48] use trace-based analysis and apply model

```
Definitions.
 ...
B eq_η          ≡ v_p = v_p1 ∧ d_p = d_p1 ∧ a = a_1
B ψ             ≡ 2Bd_s > v_s^2 + (A + B)(Aε^2 + 2v_sε)
HP choices      ≡ c:= *
HP accel        ≡ ?ψ; busV := A
HP brake        ≡ busV := −B
HP ctrl_v       ≡ v_s:= v_p; d_s:= d_p;
                  if (c) then accel else brake
HP ctrl_t       ≡ temp_s:= temp_p;
                  (  (?temp_s > T; busV:= -1)
                  ∪ (?temp_s < T; busV:= 1)
                  ∪ (?temp_s = T) )
HP ctrl_bus     ≡ (  (?busV = a; ctrl_t; busH:= 1)
                  ∪  busH:= 0 )
HP ctrl_r       ≡ (  (?busH = 0; a:= busV)
                  ∪ (?busH = 1; thermo:= busV) )
HP ctrl         ≡ ctrl_v; ctrl_bus; ctrl_r; t:= 0
B ψ_1           ≡ 2Bd_s1 > v_s1^2 + (A + B)(Aε^2 + 2v_s1ε)
HP choices_1    ≡ c_1:= c
HP accel_1      ≡ ?ψ_1; busV_1 := A
HP brake_1      ≡ busV_1 := −B
HP ctrl_v1      ≡ v_s1:= v_p1; d_s1:= d_p1;
                  if (c_1) then accel_1 else brake_1
HP ctrl_t1      ≡ temp_s1:= *;
                  (  (?temp_s1 > T; busV_1:= -1)
                  ∪ (?temp_s1 < T; busV_1:= 1)
                  ∪ (?temp_s1 = T) )
HP ctrl_bus1    ≡ (  (?busV_1 = a_1; ctrl_t1; busH_1:= 1)
                  ∪  busH_1:= 0 )
HP ctrl_r1      ≡ (  (?busH_1 = 0; a_1:= busV_1)
                  ∪ (?busH_1 = 1; thermo_1:= busV_1) )
HP ctrl_1       ≡ ctrl_v1; ctrl_bus1; ctrl_r1; t_1:= 0
HP ctrl'        ≡ choices; ctrl; choices_1; ctrl_1
HP plant'       ≡ d_p' = −v_p, v_p' = a, temp_p' = thermo, t' = 1
                  d_p1' = −v_p1, v_p1' = a_1, temp_p1' = thermo_1, t_1' = 1
                    & (v_p ≥ 0 ∧ v_p1 ≥ 0 ∧ t ≤ ε ∧ t_1 ≤ ε)
 ...
Problem.
eq_η → [(ctrl'; plant')*]eq_η
```

Fig. 12: Interleaved composition of the hybrid program (in canonical form) modeling an autonomous vehicle with an internal bus shown in Figure 11

checking to verify information-flow properties for discrete models based on process algebra. Prabhakar et al. [49] introduce a type system that enforces noninterference for a hybrid system modeled as a programming language. Nguyen et al. [50] propose a static analysis that checks noninterference for hybrid automata. Liu et al. [51] introduce an integrated architecture to provide provable security and safety assurance for cyber-physical systems. They focus on integrated co-development: language-based information-flow control using Jif [52] and a verified hardware platform for information-flow control. Their focus is not on sensor attacks.

Bohrer et al. [53] verify nondeducibility in hybrid programs, a noninterference-like guarantee. To do this, they introduce a very expressive modal logic that can explicitly express that formulas hold in a given world (i.e., state). By contrast, we use an existing logic (that has good tool support) to express and reason about a specific threat model.

Closely related to our work is that of Kolčák et al. [54] which introduces a relational extension of dL. A key contribution of their work is a new proof rule to combine two dynamics, allowing existing inference rules of dL to be applied in a relational setting. Similar to their work, our self-composition technique expresses relational properties by leveraging a composition of two programs whose variables are disjoint. Unlike their work, our self-composition technique aims to prove relational properties that require some of the nondeterministic choices to be resolved in the same way in both executions. For instance, our example shown in Figure 8 is not directly expressible in their setting. We believe that the work by Kolčák et al. [54] is orthogonal to ours, and the two can be combined to express and prove more complicated relational properties.

Security analysis for CPSs Much work have focused on the security of cyber-physical systems (CPS), but primarily from a systems security perspective rather than using formal methods. Various attacks (and mitigations of these attacks) have been identified, including false data injection [55], replay attacks [56], relay attacks [57], spying [58], and hijacking [59]. Our work focuses on formal methods for CPS security, ruling out entire classes of attacks.

Mitigating sensor attacks Some work propose attack-resilient state estimation to defend against adversarial sensor attacks in cyber-physical systems [60], [61]. These methods model systems with bounded sensor noises as an optimization problem to locate potentially malicious sensors. Our work has a different formal model of sensor attacks and focuses on formal guarantees of robustness of systems under sensor attacks.

VIII. Conclusion

We have introduced a formal framework for modeling and analyzing sensor attacks on cyber-physical systems. We formalize two relational properties that relate executions in the original system and a system where some sensors have been compromised. The relational properties express the robustness of safety properties and the robustness of high-integrity state.

Both relational properties can be expressed in terms of an equivalence relation between programs, and we presented two approaches to reason about this equivalence relation, one based on decomposition and the other based on using a single program to represent executions of the original system and the attacked system. We have shown both of these approaches sound, and used them on three case studies of non-trivial cyber-physical systems.

This work focuses on sensors, but our approach can also be used to model and analyze attacks on actuators.

References

[1] R. Alur, "Formal verification of hybrid systems," in *ACM International Conference on Embedded Software*, 2011, pp. 273–278.

[2] D. Bresolin, L. Geretti, R. Muradore, P. Fiorini, and T. Villa, "Formal verification applied to robotic surgery," in *Coordination Control of Distributed Systems*, 2015, pp. 347–355.

[3] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, E. Zawadzki, and A. Platzer, "A formally verified hybrid system for the next-generation airborne collision avoidance system," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 21–36.

[4] S. Mitsch, K. Ghorbal, D. Vogelbacher, and A. Platzer, "Formal verification of obstacle avoidance and navigation of ground robots," *The International Journal of Robotics Research*, vol. 36, no. 12, pp. 1312–1340, 2017.

[5] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial sensor attack on LiDAR-based perception in autonomous driving," in *ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2267–2281.

[6] "'Spoofers' use fake GPS signals to knock a yacht off course," MIT Technology Review, 2013.

[7] D. Davidson, H. Wu, R. Jellinek, V. Singh, and T. Ristenpart, "Controlling UAVs with sensor input spoofing attacks," in *USENIX Workshop on Offensive Technologies*, 2016.

[8] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *USENIX Security Symposium*, 2015, pp. 881–896.

[9] N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?" *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016.

[10] A. Platzer, "The complete proof theory of hybrid systems," in *IEEE/ACM Symposium on Logic in Computer Science*, 2012, pp. 541–550.

[11] R. Alur, *Principles of cyber-physical systems*. MIT Press, 2015.

[12] K. G. Larsen, "Verification and performance analysis for embedded systems," in *IEEE International Symposium on Theoretical Aspects of Software Engineering*, 2009, pp. 3–4.

[13] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*. MIT press, 2016.

[14] P. Tabuada, *Verification and control of hybrid systems: a symbolic approach*. Springer, 2009.

[15] A. Tiwari, "Logic in software, dynamical and biological systems," in *IEEE Symposium on Logic in Computer Science*, 2011, pp. 9–10.

[16] R. Lanotte, M. Merro, R. Muradore, and L. Viganò, "A formal approach to cyber-physical attacks," in *IEEE Computer Security Foundations Symposium*, 2017, pp. 436–450.

[17] R. Lanotte, M. Merro, A. Munteanu, and L. Viganò, "A formal approach to physics-based attacks in cyber-physical systems," *ACM Transactions on Privacy and Security*, vol. 23, no. 1, pp. 1–41, 2020.

[18] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[19] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[20] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[21] A. Platzer, "Differential dynamic logic for hybrid systems," *Journal of Automated Reasoning*, vol. 41, no. 2, pp. 143–189, 2008.

[22] A. Platzer, *Logical foundations of cyber-physical systems*. Springer, 2018, vol. 662.

[23] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer, "KeYmaera X: An axiomatic tactical theorem prover for hybrid systems," in *International Conference on Automated Deduction*, 2015, pp. 527–538.

[24] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Proceedings. 17th IEEE Computer Security Foundations Workshop*, 2004, pp. 100–114.

[25] D. Kozen, "Kleene algebra with tests," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 3, pp. 427–443, 1997.

[26] A. Platzer, "A complete uniform substitution calculus for differential dynamic logic," *Journal of Automated Reasoning*, vol. 59, no. 2, pp. 219–265, 2017.

[27] D. Harel, D. Kozen, and J. Tiuryn, *Dynamic Logic*. MIT Press, 2000.

[28] D. Sangiorgi, *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[29] L. Pick, G. Fedyukovich, and A. Gupta, "Exploiting synchrony and symmetry in relational verification," in *International Conference on Computer Aided Verification*, 2018, pp. 164–182.

[30] R. Shemer, A. Gurfinkel, S. Shoham, and Y. Vizel, "Property directed self composition," in *International Conference on Computer Aided Verification*, 2019, pp. 161–179.

[31] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *International Static Analysis Symposium*, 2005, pp. 352–367.

[32] A. Müller, S. Mitsch, W. Retschitzegger, W. Schwinger, and A. Platzer, "A component-based approach to hybrid systems safety verification," in *International Conference on Integrated Formal Methods*, 2016, pp. 441–456.

[33] A. Platzer, "A temporal dynamic logic for verifying hybrid system invariants," in *International Symposium on Logical Foundations of Computer Science*, 2007, pp. 457–471.

[34] J.-B. Jeannin and A. Platzer, "dTL^2: differential temporal dynamic logic with nested temporalities for hybrid systems," in *International Joint Conference on Automated Reasoning*, 2014, pp. 292–306.

[35] U. Kiencke and L. Nielsen, *Automotive Control Systems: For Engine, Driveline and Vehicle*, 1st ed. Berlin, Heidelberg: Springer-Verlag, 2000.

[36] M. Tanelli, A. Astolfi, and S. M. Savaresi, "Robust nonlinear output feedback control for brake by wire control systems," *Automatica*, vol. 44, no. 4, pp. 1078–1087, 2008.

[37] P. Bhivate, "Modelling & development of antilock braking system," Ph.D. dissertation, 2011.

[38] S. Solyom, A. Rantzer, and J. Lüdemann, "Synthesis of a model-based tire slip controller," *Vehicle System Dynamics*, vol. 41, no. 6, pp. 475–499, 2004.

[39] Y. Shoukry, P. Martin, P. Tabuada, and M. Srivastava, "Non-invasive spoofing attacks for anti-lock braking systems," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2013, pp. 55–72.

[40] "Maneuvering characteristics augmentation system - wikipedia," https://en.wikipedia.org/wiki/Maneuvering_Characteristics_Augmentation_System, accessed: 2021-1-10.

[41] P. Heidlauf, A. Collins, M. Bolender, and S. Bak, "Verification challenges in F-16 ground collision avoidance and other automated maneuvers." in *5th International Workshop on Applied Verification for Continuous and Hybrid Systems*, 2018, pp. 208–217.

[42] "Boeing 737 Max: Is automation to blame?" https://www.eetasia.com/automation-and-boeings-b737-max-crash/, accessed: 2021-1-10.

[43] A. Marcos and G. Balas, "Linear parameter varying modeling of the Boeing 747-100/200 longitudinal motion," in *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2001, p. 4347.

[44] "Boeing: The 737 MAX MCAS software enhancement," https://www.boeing.com/commercial/737max/737-max-software-updates.page, accessed: 2021-1-10.

[45] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces." in *USENIX Security Symposium*, 2011, pp. 447–462.

[46] C. Bernardeschi, A. Domenici, and M. Palmieri, "Formalization and co-simulation of attacks on cyber-physical systems," *Journal of Computer Virology and Hacking Techniques*, pp. 1–15, 2020.

[47] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *International Conference on Automated Deduction*, 1992, pp. 748–752.

[48] R. Akella, "Verification of information flow security in cyber-physical systems," 2013.

[49] P. Prabhakar and B. Köpf, "Verifying information flow properties of hybrid systems," in *ACM international conference on High confidence networked systems*, 2013, pp. 77–84.

[50] L. V. Nguyen, G. Mohan, J. Weimer, O. Sokolsky, I. Lee, and R. Alur, "Detecting security leaks in hybrid systems with information flow analysis," in *ACM-IEEE International Conference on Formal Methods and Models for System Design*, 2019, p. 14.

[51] J. Liu, J. Corbett-Davies, A. Ferraiuolo, A. Ivanov, M. Luo, G. E. Suh, A. C. Myers, and M. Campbell, "Secure autonomous cyber-physical systems through verifiable information flow control," in *Proceedings of the 2018 Workshop on Cyber-Physical Systems Security and Privacy*, 2018, pp. 48–59.

[52] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 228–241.

[53] B. Bohrer and A. Platzer, "A hybrid, dynamic logic for hybrid-dynamic information flow," in *ACM/IEEE Symposium on Logic in Computer Science*, 2018, pp. 115–124.

[54] J. Kolčák, J. Dubut, I. Hasuo, S.-y. Katsumata, D. Sprunger, and A. Yamada, "Relational differential dynamic logic," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 191–208.

[55] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *IEEE Symposium on Security and Privacy*, 2010, pp. 447–462.

[56] C. Li, A. Raghunathan, and N. K. Jha, "Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system," in *IEEE International Conference on e-Health Networking, Applications and Services*, 2011, pp. 150–156.

[57] A. Francillon, B. Danev, and S. Capkun, "Relay attacks on passive keyless entry and start systems in modern cars," in *Network and Distributed System Security Symposium*, 2011.

[58] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces." in *USENIX Security Symposium*, 2011, pp. 447–462.

[59] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[60] M. Pajic, J. Weimer, N. Bezzo, P. Tabuada, O. Sokolsky, I. Lee, and G. J. Pappas, "Robustness of attack-resilient state estimators," in *2014 ACM/IEEE International Conference on Cyber-Physical Systems*, 2014, pp. 163–174.

[61] M. Pajic, I. Lee, and G. J. Pappas, "Attack-resilient state estimation for noisy dynamical systems," *IEEE Transactions on Control of Network Systems*, vol. 4, no. 1, pp. 82–92, 2016.

APPENDIX A
DEFINITIONS

We present the formal definitions of bound variables, free variables, and variable sets here. These definitions are exactly as given in [22], [26], and included for the reader's convenience.

Definition 11 (Bound variables)**.** *The set* $\mathrm{BV}(\phi)$ *of bound variables of* dL *formula* ϕ *is defined inductively as:*

$$\begin{aligned}
\mathrm{BV}(\theta_1 \sim \theta_2) &= \emptyset \qquad \sim \in \{<, \leq, =, >, \geq\} \\
\mathrm{BV}(\neg\phi) &= \mathrm{BV}(\phi) \\
\mathrm{BV}(\phi \vee \psi) = \mathrm{BV}(\phi \wedge \psi) &= \mathrm{BV}(\phi) \cup \mathrm{BV}(\psi) \\
\mathrm{BV}(\phi \to \psi) &= \mathrm{BV}(\phi) \cup \mathrm{BV}(\psi) \\
\mathrm{BV}(\forall x.\ \phi) = \mathrm{BV}(\exists x.\ \phi) &= \{x\} \cup \mathrm{BV}(\phi) \\
\mathrm{BV}([\alpha]\phi) &= \mathrm{BV}(\alpha) \cup \mathrm{BV}(\phi)
\end{aligned}$$

The set $\mathrm{BV}(P)$ *of bound variables of hybrid program* P*, i.e., those may potentially be written to, is defined inductively as:*

$$\begin{aligned}
\mathrm{BV}(x := \theta) = \mathrm{BV}(x := *) &= \{x\} \\
\mathrm{BV}(?\phi) &= \emptyset \\
\mathrm{BV}(x' = \theta \,\&\, \phi) &= \{x, x'\} \\
\mathrm{BV}(\alpha; \beta) = \mathrm{BV}(\alpha \cup \beta) &= \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta) \\
\mathrm{BV}(\alpha^*) &= \mathrm{BV}(\alpha)
\end{aligned}$$

Definition 12 (Must-bound variables)**.** *The set* $\mathrm{MBV}(P) \subseteq \mathrm{BV}(P)$ *of most bound variables of hybrid program* P*, i.e., all those that must be written to on all paths of* P*, is defined inductively as:*

$$\begin{aligned}
\mathrm{MBV}(x := \theta) = \mathrm{MBV}(x := *) &= \{x\} \\
\mathrm{MBV}(?\phi) &= \emptyset \\
\mathrm{MBV}(x' = \theta \,\&\, \phi) &= \{x, x'\} \\
\mathrm{MBV}(\alpha \cup \beta) &= \mathrm{MBV}(\alpha) \cap \mathrm{MBV}(\beta) \\
\mathrm{MBV}(\alpha; \beta) &= \mathrm{MBV}(\alpha) \cup \mathrm{MBV}(\beta) \\
\mathrm{MBV}(\alpha^*) &= \emptyset
\end{aligned}$$

Definition 13 (Free variables)**.** *The set* $\mathrm{FV}(\theta)$ *of variables of term* θ *is defined inductively as:*

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(c) &= \emptyset \\
\mathrm{FV}(\theta_1 \oplus \theta_2) &= \mathrm{FV}(\theta_1) \cup \mathrm{FV}(\theta_2) \quad \oplus \in \{+, \times\}
\end{aligned}$$

The set $\mathrm{FV}(\phi)$ *of free variables of* dL *formula* ϕ *is defined inductively as:*

$$\begin{aligned}
\mathrm{FV}(\theta_1 \sim \theta_2) &= \mathrm{FV}(\theta_1) \cup \mathrm{FV}(\theta_2) \\
\mathrm{FV}(\neg\phi) &= \mathrm{FV}(\phi) \\
\mathrm{FV}(\phi \vee \psi) = \mathrm{FV}(\phi \wedge \psi) &= \mathrm{FV}(\phi) \cup \mathrm{FV}(\psi) \\
\mathrm{FV}(\phi \to \psi) &= \mathrm{FV}(\phi) \cup \mathrm{FV}(\psi) \\
\mathrm{FV}(\forall x.\ \phi) = \mathrm{FV}(\exists x.\ \phi) &= \mathrm{FV}(\phi) \setminus \{x\} \\
\mathrm{FV}([\alpha]\phi) &= \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))
\end{aligned}$$

The set $\mathrm{FV}(P)$ *of bound variables of hybrid program* P *is defined inductively as:*

$$\begin{aligned}
\mathrm{FV}(x := \theta) &= \mathrm{FV}(\theta) \\
\mathrm{FV}(x := *) &= \emptyset \\
\mathrm{FV}(?\phi) &= \mathrm{FV}(\phi) \\
\mathrm{FV}(x' = \theta \,\&\, \phi) &= \{x\} \cup \mathrm{FV}(\theta) \cup \mathrm{FV}(\phi) \\
\mathrm{FV}(\alpha \cup \beta) &= \mathrm{FV}(\alpha) \cup \mathrm{FV}(\beta) \\
\mathrm{FV}(\alpha; \beta) &= \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\beta) \setminus \mathrm{MBV}(\alpha)) \\
\mathrm{FV}(\alpha^*) &= \mathrm{FV}(\alpha)
\end{aligned}$$

Definition 14 (Variable sets)**.** *The set* $\textsc{Var}(P)$*, variables of hybrid program* P *is* $\mathrm{BV}(P) \cup \mathrm{FV}(P)$*. The set* $\textsc{Var}(\phi)$*, variables of* dL *formula* ϕ *is* $\mathrm{BV}(\phi) \cup \mathrm{FV}(\phi)$*.*

APPENDIX B
PROOFS

Proof of Theorem 1. $P \approx_{\mathrm{FV}(\phi_{pre} \wedge \phi_{post})} \textsc{attacked}(P, S_A)$ means for any execution σ^q of $\textsc{attacked}(P, S_A)$, there exists an execution σ^p of P that agrees on $\mathrm{FV}(\phi_{pre} \wedge \phi_{post})$ at the starting state and the end of every control iteration. That means if the starting state of σ^q satisfies ϕ_{pre}, the starting state of σ^p satisfies ϕ_{pre} (Lemma 3 from [26]). Meanwhile, since $\phi_{pre} \to [P]\phi_{post}$, the last state of σ^p satisfies ϕ_{post}. And last states of σ^q and σ^p agree on free variables used in ϕ_{post}, so the last state of σ^q satisfies ϕ_{post} (Lemma 3 from [26]). $\phi_{pre} \to [\textsc{attacked}(P, S_A)]\phi_{post}$ holds. □

Proof of Property 1 to 3 of Theorem 2. By the definition of $\mathcal{H}$-equivalence. □

Lemma 1. *$\mathcal{H}$-equivalence of states is transitive, reflective, and symmetric.*

Proof. By the definition of $\approx_{\mathcal{H}}$. □

Lemma 2. *For program P, state ω, ω', ν, and set $\mathcal{H}$ such that $(\omega,\nu) \in [\![P]\!]$, $\omega \approx_{\mathcal{H}} \omega'$, and $\mathrm{FV}(P) \subseteq \mathcal{H}$, then there exists ν' such that $(\omega',\nu') \in [\![P]\!]$ and $\nu \approx_{\mathcal{H}} \nu'$.*

Proof. By the definition of $\approx_{\mathcal{H}}$ and lemma 4 from [26]. □

Proof of Property 4 of Theorem 2. We prove that for any execution of $A;C$, there exists an execution of $B;D$ that agrees with it on $\mathcal{H}$. The other direction can be proven similarly. For any execution σ^{ac} of $A;C$, let ω_{ac_f} and ω_{ac_l} be its first and last state respectively. Then there exists a state ω_{ac_m} such that $(\omega_{ac_f},\omega_{ac_m}) \in [\![A]\!]$ and $(\omega_{ac_m},\omega_{ac_l}) \in [\![C]\!]$. Since $A \approx_{\mathcal{H}} B$, there exist state ω_{b_f}, ω_{b_l} such that $(\omega_{b_f},\omega_{b_l}) \in [\![B]\!]$, $\omega_{b_f} \approx_{\mathcal{H}} \omega_{ac_f}$, and $\omega_{b_l} \approx_{\mathcal{H}} \omega_{ac_m}$. Likewise, since $C \approx_{\mathcal{H}} D$, there exists state ω_{d_f}, ω_{d_l} such that $(\omega_{d_f},\omega_{d_l}) \in [\![D]\!]$, $\omega_{d_f} \approx_{\mathcal{H}} \omega_{ac_m}$, and $\omega_{d_l} \approx_{\mathcal{H}} \omega_{ac_l}$. By transitivity (Lemma 1), we get $\omega_{b_l} \approx_{\mathcal{H}} \omega_{d_f}$. Since $\mathrm{FV}(D) \subseteq \mathcal{H}$, by Lemma 2, there exist state $\omega_{d'_l}$ such that $(\omega_{b_l},\omega_{d'_l}) \in [\![D]\!]$ and $\omega_{d_l} \approx_{\mathcal{H}} \omega_{d'_l}$. Since $\omega_{ac_l} \approx_{\mathcal{H}} \omega_{d'_l}$ and $\omega_{ac_f} \approx_{\mathcal{H}} \omega_{b_f}$ (by transitivity), for the execution of $A;C$ from ω_{ac_f} to ω_{ac_l}, we have $(\omega_{b_f},\omega_{d'_l}) \in [\![B;D]\!]$, $\omega_{ac_f} \approx_{\mathcal{H}} \omega_{b_f}$, and $\omega_{ac_l} \approx_{\mathcal{H}} \omega_{d'_l}$. $A;C \approx_{\mathcal{H}} B;D$ holds. □

Proof of Property 5 of Theorem 2. By induction on the number of iterations of α^* and β^*. Base case is trivial. For the induction case, assume $\alpha^k \approx_{\mathcal{H}} \beta^k$ is true, we can prove $\alpha^k;\alpha \approx_{\mathcal{H}} \beta^k;\beta$ using Property 4 by letting A be α^k, B be β^k, C be α, and D be β. Thus, $\alpha^* \approx_{\mathcal{H}} \beta^*$ holds. □

Proof of robust safety of the ABS model. Let P be the hybrid program modeling ABS with duplicated sensors. Assume sensor ω_1 is compromised. Let A be the `voting` program, B be $\textsc{attacked}(A,\{\omega_1\})$, and C be program P with `voting` excluded (i.e., $P = (A;C)^*$ and $\textsc{attacked}(P,\{\omega_1\}) = (B;C)^*$). Here, $\mathrm{FV}(A) = \mathrm{FV}(B) = \{\omega_p\}$, $\mathrm{FV}(A;C) = \mathrm{FV}(B;C)$, and $\mathrm{FV}(C) = \{\omega_s\} \cup \mathrm{FV}(A;C)$.

By the definition of $\approx_{\mathcal{H}}$, $A \approx_{\{\omega_s,\omega_p\}} B$ holds, which means

$$A \approx_{\mathrm{FV}(C)} B \qquad \text{(Property 3)}$$

With $C \approx_{\mathrm{FV}(C)} C$ (Property 1), we get

$$(A;C) \approx_{\mathrm{FV}(C)} (B;C) \qquad \text{(Property 4)}$$

which leads to

$$(A;C)^* \approx_{\mathrm{FV}(C)} (B;C)^* \qquad \text{(Property 5)}$$

Property 2 also applies to programs with loop, and $\{\omega_p,v_p\} \subseteq \mathrm{FV}(C)$, thus

$$(A;C)^* \approx_{\{\omega_p,v_p\}} (B;C)^* \qquad \text{(Property 2)}$$

Since $\mathrm{FV}(\phi_{pre} \wedge \phi_{post}) = \{\omega_p,v_p\}$, we have $\textsc{robust}(P,\phi_{pre},\phi_{post},\{\omega_{s_1}\})$ (Theorem 1).

Similarly, we can prove $\textsc{robust}(P,\phi_{pre},\phi_{post},\{\omega_{s_2}\})$ and $\textsc{robust}(P,\phi_{pre},\phi_{post},\{\omega_{s_3}\})$. □

Proof of robust safety of Boeing 737-MAX model. Let A be program $\mathtt{ctrl}'_{aoa}$, B be program $\textsc{attacked}(A,\{s_L\})$, C be program `MCAS(aoa); plant` in Figure 10. Here, $\mathrm{FV}(A) = \mathrm{FV}(B) = \{aoa_p\}$, let `fv` be the set of free variables of program $A;C$, then $\mathrm{FV}(B;C)$=`fv`, and $\mathrm{FV}(C)$ would be $\{aoa_s\} \cup$ `fv`. We can prove $\textsc{robust}(A;C,\phi_{pre},\phi_{post},\{s_L\})$ with the following steps:

By definition of $\approx_{\mathcal{H}}$, we prove $A \approx_{\{aoa_s,aoa_p\}} B$, which means

$$A \approx_{\{aoa_s,aoa_p\}\cup \mathtt{fv}} B \qquad \text{(Property 3)}$$

With $C \approx_{\{aoa_s,aoa_p\}\cup \mathtt{fv}} C$ (Property 1), we know

$$A;C \approx_{\{aoa_s,aoa_p\}\cup \mathtt{fv}} B;C \qquad \text{(Property 4)}$$

Since $\mathrm{FV}(A;C) \cup \mathrm{FV}(B;C) \subseteq \{aoa_s,aoa_p\} \cup$ `fv`, we know

$$(A;C)^* \approx_{\{aoa_s,aoa_p\}\cup \mathtt{fv}} (B;C)^* \qquad \text{(Property 5)}$$

Property 2 applies to programs with loop as well, so

$$(A;C)^* \approx_{\mathtt{fv}} (B;C)^* \qquad \text{(Property 2)}$$

Since formula ϕ_{pre} and ϕ_{post} typically refer to free variables in `fv`, we get $\textsc{robust}(P,\phi_{pre},\phi_{post},\{s_L\})$ holds. (Theorem 1). Similarly, we can prove $\textsc{robust}(P,\phi_{pre},\phi_{post},\{s_R\})$. □

Appendix C
Limitations of the Self-Composition Approach

One limitation of our self-composition approach is that it applies only for hybrid programs that have *total* semantics for all low-integrity inputs. It means if a program has a *valid* execution on an input state ω (i.e., exist state ν such that $(\omega,\nu) \in [\![P]\!]$), then for any state ω' that differs with ω only in low-integrity inputs, there exists ν' that $(\omega,\nu) \in [\![\textsc{attacked}(P,S_A)]\!]$.

A program may have *partial* (not total) semantics on low-integrity inputs for two reasons: (1) some low-integrity inputs fail test conditions in all execution paths, for example, if a is a low-integrity variable, then $?a > 0$ is a program whose semantics are partial on low-integrity inputs; (2) the program's evolution constraint depends on low-integrity inputs. For example, if a is a low-integrity variable, $(x' = \theta \& a > 0)$ is a program whose semantics are partial on low-integrity inputs.

Fortunately, there is a relatively simple way to check that hybrid programs meet this requirement. First, given a set of low-integrity sensor variables, a straightforward program analysis can identify all variables that might depend on a low-integrity sensor variables; call these the low-integrity variables. Second, check that all evolution constraints do not include any low-integrity variables. Third, check that any test $?\phi_i$ that includes a low-integrity variable occurs as part of a construct $?\phi_1;\alpha_1 \cup \cdots \cup ?\phi_n;\alpha_n$ such that $\phi_1 \vee \cdots \vee \phi_n$ is valid (i.e., the tests are exhaustive and so at least one of the branches of the nondeterministic choice will be true).

Well-designed hybrid program models should have total semantics on low-integrity inputs, except in specific situations that rarely depend on low-integrity sensor variables. Models that do not have total semantics on low-integrity inputs typically do not correspond to actually implementable control strategies, and are therefore only vacuously safe.