RIVER: An eBPF-based Runtime Verification Platform for Cyber-Physical Systems

Dario Facchinetti
Università degli Studi di Bergamo
Bergamo, Italy
dario.facchinetti@unibg.it

Matthew Rossi
Università degli Studi di Bergamo
Bergamo, Italy
matthew.rossi@unibg.it

Zhenya Zhang
Kyushu University and
National Institute of Informatics
Fukuoka, Japan
zhang@ait.kyushu-u.ac.jp

Stefano Paraboschi
Università degli Studi di Bergamo
Bergamo, Italy
stefano.paraboschi@unibg.it

Paolo Arcaini
National Institute of Informatics
Tokyo, Japan
arcaini@nii.ac.jp

***Abstract*—Runtime verification has proven to be an effective approach for quality assurance of cyber-physical systems (CPSs). However, due to some technical difficulties, existing tools usually do not support analyses that require runtime inspection of the CPS internal states and do not allow to perform a prompt intervention. To fill this gap, we propose RIVER, a client-server platform that supports a wide range of novel runtime analyses, including injection of adversarial attacks and runtime enforcement. At the core of our approach there is the adoption of eBPF, a widely-used technology in security research that allows privileged programs to read and write the internal state of the CPS model. We showcase the functionalities of RIVER over two widely-adopted CPS models: an adaptive cruise control system and an abstract fuel control system.**
Tool available at: https://github.com/dariofad/river [1]
Screencast available at: https://youtu.be/g-3m0E2nNgM

***Index Terms*—Runtime Verification, Runtime Enforcement, Adversarial Attack, Cyber-Physical Systems, eBPF**

I. Introduction

Cyber-Physical Systems (CPSs) are systems in which computational components are tightly integrated with physical processes, enabling sensing, communication, and control to interact in real time with the physical world. They have been widely deployed in safety-critical domains, e.g., automotive systems, avionics, energy, and healthcare. The safety-critical nature of CPSs has stressed a strong demand for rigorous quality assurance; however, exhaustive verification is difficult due to the intrinsic complexity of CPSs, arising from non-linear dynamics, reliance on black-box components, and integration of machine learning models.

Given the complexity of CPSs, *runtime verification (RV)* [2], a lightweight formal approach, has demonstrated unique strengths in scalability and has been increasingly adopted as an alternative to exhaustive verification. Rather than proving that a system is entirely free of unsafe behaviors, RV monitors system executions at runtime and enforces safe actions when potentially dangerous situations are detected. Compared with exhaustive verification, it is lightweight, requires no access to system dynamics, yet it is effective to ensure system safety.

RV techniques for CPSs have been actively developed and supported by tools in the past decade. *Monitoring* [3]–[5] is the foundational technique that aims to decide whether a system execution satisfies a given specification. Based on monitoring results, *falsification* [6] is a testing technique that aims to find counterexample trajectories that violate the specification.

Motivation. Despite the success of existing RV techniques, the tools currently available still do not support some useful RV tasks proposed in the literature. For instance, 1) *adversarial attacks* [7] aim to assess the *robustness* of a CPS under adversarial perturbations or sensor noise, a task that has become increasingly challenging with the widespread adoption of neural network controllers; 2) *runtime enforcement* [8] aims to enforce safe actions to drive the system away from dangerous states. Both tasks require *intervention* into system executions at runtime; indeed, an attacker needs to inject carefully crafted noise with precise timing to remain stealthy, while an enforcer needs to intervene only when potential danger is detected. However, when a system simulation is running, the CPS state variables are stored in memory and updated according to the system dynamics, and they are not easily editable by external processes, blocking existing RV tools from supporting runtime intervention. A potential solution to this problem is the use of Just-In-Time (JIT) compilation, which allows the execution of a modified version of the application including the required or missing runtime checks. This approach was proposed for example by Pin [9], DynamoRIO [10], and Valgrind [11]. However, in practice, it is not always possible to implement this process for various reasons, such as: (i) the use of black-box components by CPS, (ii) the complexity of embedding high-level runtime interventions with JIT compilation, and (iii) the absence of an ad-hoc Virtual Machine supporting the runtime injection of code.

P. Arcaini is supported by the ASPIRE grant No. JPMJAP2301, JST. Z. Zhang is supported by JST BOOST Grant No. JPMJBY24D7 and JSPS Grant No. JP25K21179. M. Rossi and S. Paraboschi are supported by MASE Mission Innovation 2.0 - D.M. 386/2023 under project EVFTEG grant No. F53C25001680001.

TABLE I: Comparison with existing CPS RV tools

	falsification	monitoring	diagnostics	attack	enforcement
Breach [13]	✓	✓			
S-TaLiRo [14]	✓	✓			
RTAMT [15]		✓			
CPSDebug [16]		✓	✓		
CauMon [17]		✓	✓		
RIVER (ours)	✓	✓	✓	✓	✓

Our Contribution. To bridge this gap, we propose *Runtime Intervention and VERification platform* (RIVER), a tool that allows intervention into system executions at runtime, thereby enabling investigations of RV techniques such as injection of adversarial attacks and runtime enforcement. Notably, the goal of RIVER is not to devise novel RV techniques, but rather provide a platform to support the deployment of those techniques, so they can be sufficiently assessed. At the core of RIVER there is the adoption of *eBPF* [12], a widely-used technology in the field of security, which permits to directly access and modify the internal state of a monitored program. In our context, we apply eBPF to preempt system executions and read and write the state variables in the memory according to user-provided algorithms. This unlocks the potential to design novel RV techniques that can perform more detailed analyses of the monitored CPS, without the need to recompile CPS models. Indeed, read access allows to *monitor* internal states, while write access allows to perturb or correct CPS trajectories, supporting *adversarial attacks* or *runtime enforcement*.

RIVER adopts a client-server architecture: a CPS model is uploaded to the server and started upon receiving a request from the client. While the model is executing, the client can run an RV program (that implements a user-provided technique) to decide *when* and *how* to perturb or enforce the running execution according to a user-defined strategy, and instruct the server to do so. On the server side, the CPS is continuously monitored leveraging eBPF, and changes to the system state are applied as detailed by the RV program. We showcase the effectiveness of RIVER with two commonly-adopted CPS models: an *adaptive cruise control* system and an *abstract fuel control* system. Our tool is available online at https://github.com/dariofad/river.

Related Work. RV has been extensively investigated as a lightweight quality assurance approach with established tool supports. Table I reports existing RV tools; these tools can either send to models pre-defined inputs or read observable output states, but none of them support analyses that require runtime state interventions, such as adversarial attack and enforcement, as supported by RIVER.

II. Background: Runtime Verification of CPS

Runtime verification (RV) [2] has demonstrated feasibility and scalability for a broad range of applications. Compared to exhaustive verification, it can accommodate black-box systems and complex specifications expressed by temporal logic:

- *Black-box systems.* As RV techniques aim to ensure safety at the level of system behaviors, it does not require access to system dynamics. Formally, a black-box system $\mathcal{M}$ can be represented as a function that maps a *time-variant* (input) signal $\mathbf{u}\colon [0,T] \to \mathbb{R}^M$ to an (output) signal $\mathcal{M}(\mathbf{u})\colon [0,T] \to \mathbb{R}^N$, where $T \in \mathbb{R}_+$ is the *time horizon*, $M, N \in \mathbb{N}_+$ are the dimensions of $\mathbf{u}$ and $\mathcal{M}(\mathbf{u})$. In other words, RV techniques work for any $\mathcal{M}$ as long as its behaviors are observable via output signals.
- *Temporal logic specifications.* Instead of simple properties such as reachability, RV can handle more complex properties expressed by *signal temporal logic (STL)* [18]. The expressivity of STL is similar to *linear temporal logic (LTL)*, but it can predicate over signals in continuous temporal and spatial domains. STL has quantitative semantics [18], i.e., given a signal $\mathbf{w}$ and an STL formula φ, the semantics returns a quantity $\mathsf{rob}(\mathbf{w}, \varphi) \in \mathbb{R}$ called *robustness*, which indicates not only *whether or not*, but also *how much* $\mathbf{w}$ satisfies φ.

Existing Work. RV is a well-established field. In particular, the following techniques have been extensively investigated and applied in industrial contexts.

- *Monitoring* aims to decide whether a signal $\mathbf{w}$ satisfies an STL specification φ. Depending on whether a signal $\mathbf{w}$ is completed or still growing, it can be grouped as either *offline* or *online* monitoring. Moreover, *signal diagnostics* [5] is an emerging monitoring paradigm that aims to identify the causality of violation, thereby delivering more information than classic monitoring.
- *Falsification*, as a downstream application of monitoring, aims to synthesize an input signal $\mathbf{u}$ such that $\mathsf{rob}(\mathcal{M}(\mathbf{u}), \varphi) < 0$. Existing falsification techniques [6] mostly rely on offline monitoring to cast the problem into a numerical optimization that minimizes $\mathsf{rob}(\mathcal{M}(\mathbf{u}), \varphi)$.

III. The Architecture and Features of RIVER

RIVER allows the supervised execution of a CPS model, introducing the ability to read and write any input or output signal, as well as the internal states. No model-level modification is required, leading to considerable advantages when a system cannot be halted, or when it cannot be re-compiled due to the lack of source code (or part of it). In the following, we overview RIVER's architecture and functionalities.

Workflow. As shown in Fig. 1, RIVER exposes its functionalities as a web server, allowing a client to connect and request for a specific task. The architecture separates the system execution from the strategic intervention (e.g., adversarial attack or enforcement) and provides the flexibility to operate on remote systems. Table II lists the tasks supported by RIVER.

To use RIVER, users first need to upload their CPS system models to the server and prepare their configurations and algorithms for the desired tasks. In the current setting, the algorithms for monitoring and diagnostics are built-in and run on the server side, whereas the algorithms for other tasks, including attack, enforcement, and falsification, should be provided by users and run on the client side. To initiate a task, users need to ping the server by sending a starting request

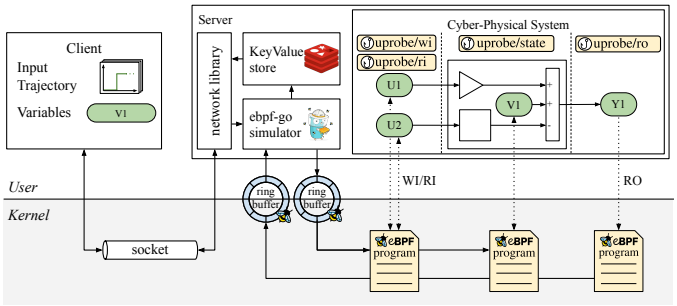

Fig. 1: Overview of RIVER. The client details the input signals and internal variables, while the server supervises the CPS. Dedicated eBPF programs preempt the CPS execution logging the trajectory, and altering the runtime behavior with the injection of values. Runtime information is periodically collected by RIVER and saved to a KeyValue store.

TABLE II: RIVER tasks and usage

Task	Description	Input	Output
Monitoring	Monitor behavior	• CPS • Loop function • Addresses of signals and variables	• Trajectory
Diagnostics	Detect faults	• CPS • Loop function • Addresses of signals and variables	• Trajectory
Falsification	Find violations	• CPS • Loop function • Addresses of signals • Input trajectory • Search algorithm	• Trajectory • Robustness coefficient
Adversarial attack	Adversarial tampering	• CPS • Loop function • Addresses of signals and variables • Runtime injections	• Trajectory
Enforcement	Adaptive trajectory control	• CPS • Loop function • Addresses of signals and variables • Runtime injections	• Trajectory

specifying the model, the specification, and the task; upon receiving the request, the server keeps listening to a port for further input (e.g., runtime injections).

A. Monitoring, Diagnostics, and Falsification

These three tasks share the commonality that they only rely on reading the state variables in running trajectories of the CPS model, without requiring to modify its internal state. This is well supported by eBPF: as shown in Fig. 1, RIVER uses dedicated eBPF programs to read the input and output signals, which are then copied to the ring buffer maps, and finally saved to the KeyValue store for persistency.

To perform these tasks, users need to send an STL specification to the server, as well as the state variables concerned with the specification. These variables correspond to the memory locations the eBPF needs to read. After the server receives the request, it immediately starts supervising the system execution, using eBPF to monitor its trajectory and state.

Monitoring. This task includes both offline monitoring and online monitoring. For the former, we adopt the Donzé et al. algorithm [3] that can efficiently compute the quantitative robustness of a given trajectory against an STL specification. For the latter, we adopt the robust online monitoring algorithm proposed by Deshmukh et al. [4], which decides whether a trajectory satisfies a specification by computing the reachable set of the robustness. As monitoring is built-in in RIVER, these algorithms are prepared in the server and can be called based on users' requested tasks.

Diagnostics. This provides a variant of monitoring that delivers causal information about specification violation. At runtime, it applies the causation-based algorithm by Zhang et al. [5].

Falsification. As it aims to find an execution that violates the given specification, it requires the client to send different input signals to the server iteratively. The effectiveness of falsification depends on the underlying algorithm in charge of suggesting new input signals in each iteration. A naive baseline could be random sampling, but more advanced techniques suggest new signals based on the feedback from previous simulations (i.e., their robustness). It is also worth mentioning that there are several options for proposing new signals (e.g., metaheuristic optimization, Monte-Carlo tree search).

B. Adversarial Attack and Runtime Enforcement

Similarly to monitoring, users need to provide a specification and the state variables concerned with the specification; moreover, these tasks also require additional inputs from users, including: 1) the state variables to be changed; 2) the algorithm to trigger the changes (e.g., the value to be applied, the injection time). Typically, during system execution on the server side, the algorithm running on the client side keeps receiving monitoring results from the server and establishes the timing and actions to perform the intervention. When a change is requested, the client sends a message to the server listing the requested actions and the timing. On the server side, upon receiving the message, eBPF applies the interventions only if they arrive *on time* (i.e., before the requested action time). There is virtually no limit to the number of client-server interactions, hence multiple injections can be performed.

Adversarial Attack. Specifically, this task perturbs specific state variables during system execution, allowing to simulate various attacking scenarios, such as environmental noises, intentional attacks, and sensor malfunctions. It is also used to assess the robustness of the system against small perturbations or to expose potential vulnerabilities. In RIVER, this is enabled by the ability of eBPF to mask variables in memory, as well as to detect state updates at runtime. Sect. IV-A1 presents an example where system sensors are tempered with injected signals, leading the controller to unsafe decisions.

Runtime Enforcement. For runtime enforcement, the selection of the variables to perturb demands the user to be aware of the logic of the CPS controller. In this scenario, the variables should be either externally controllable (e.g., external input signals, control actions) or instantly changeable (e.g., gear). The goal of enforcement is to intervene in the execution to bring the system back to conformance to the specification whenever it is close to a violation. Through RIVER, users can

assess enforcement strategies developed by themselves, such as rule-based strategies and optimization-based strategies. We provide an example of enforcement in Sect. IV-A2.

We highlight that adversarial attack and enforcement tasks are not supported by existing tools without changing the implementation of the CPS model. Using eBPF, RIVER can not only read, but also write any variable in the internal state of the CPS. The effect of these modifications can naturally persist for subsequent CPS loop cycles.

C. Implementation Details

Due to the strict constraint for real-time synchronization, in particular for tasks such as adversarial attack and enforcement, we take multiple measures to ensure the real-time property of RIVER; for example, RIVER relies on special map types named *ring buffers* to move data to the KeyValue and to perform runtime injections. Moreover, to support CPS state updates, it requires high precision for floating point arithmetic. To overcome this limitation, RIVER implements within eBPF static functions the IEEE 754 standard.[1]

IV. Evaluation

We showcase the potential of RIVER as a flexible platform that enables users to integrate and evaluate their own algorithms on CPS models, demonstrating how runtime intervention can be performed on two widely-adopted models (originally in Simulink but translated to C for demonstration of RIVER) in the CPS community, namely [19] an *adaptive cruise control* system and an *abstract fuel control* system. We recall that RIVER enables runtime intervention independently of the sophistication of the selected enforcement algorithm.

A. Examples of RIVER Usages

1) Adaptive Cruise Control (ACC): The first model we consider is an *adaptive cruise control (ACC)* system. As shown in Fig. 2, it models an autonomous driving car (called the *ego* car) with a *model predictive controller*, whose objective is to maintain a safe distance from a lead car. The lead car is considered part of the environment; hence, its motion follows a predefined route that is not controllable by the ego car. The ego car controller receives as input the velocity and position of the lead car, then computes the relative distance based on its current position, and outputs the desired acceleration a_{ego} and speed v_{ego}. A system specification imposes a constraint on the safety distance as follows:

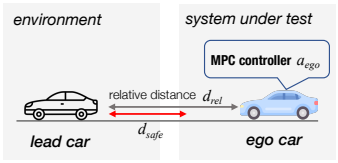

Fig. 2: Adaptive cruise control

$$\varphi_{\mathsf{ACC}} \equiv \square_{[0,450]}(d_{rel} - 1.4 \cdot v_{ego} \geq d_{safe})$$

where 1.4 s is the sampling period and d_{safe} is set to 4 meters.

Adversarial attack (Sensor tamper). In this study, the objective is to assess the model under sensor problems induced by cyber attacks or sensor degradation. To do this, we fool the controller by masking the real state values (which are normally obtained by the controller from sensors) and feeding faked values to the controller. This requires runtime intervention of normal system executions and is supported RIVER through eBPF. We proceed as follows. We inject into the controller a fake value of the relative distance d_{rel} between the ego car and the lead car, larger than the real one; in this way, the ego car may decide to accelerate, which could lead to a dangerous situation (e.g., violate the safety distance).

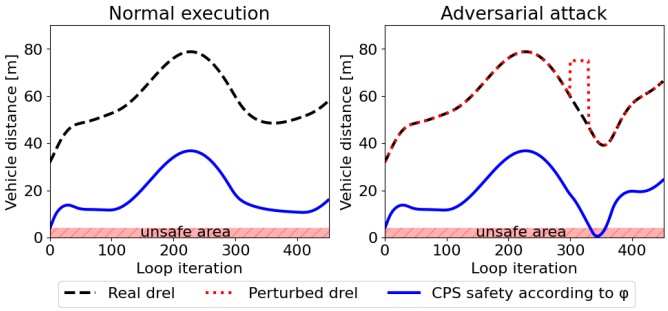

Fig. 3: Adaptive Cruise Control – Adversarial attack

Fig. 3 visualizes the data collected by RIVER in two cases. The plot on the left shows the extracted internal state variable d_{rel} (dashed-black line) without any attack; it also shows how the value of the projected distance (i.e., $d_{rel} - 1.4 \cdot v_{ego}$) changes (blue line). The unsafe area marks the zone in which the projected distance violates the specification.

The plot on the right instead shows the effect of the attack. In detail, from iteration 300, the real value of d_{rel} (dashed-black line) is masked with a synthetic value (dotted-red) generated with a step function. As a consequence, the ego car accelerates and the real distance d_{rel} decreases, causing the violation of the safety specification (i.e., φ_{ACC} becomes false, as the continuous-blue line crosses the red unsafe area).

2) Abstract Fuel Control (AFC): The considered benchmark originates from a powertrain control system introduced by *Toyota* and has become a standard case study in the CPS testing community [19]. The model captures air-path dynamics, including the throttle and intake manifold, together with a controller responsible for regulating the *air-to-fuel ratio*, a key factor affecting both system safety and performance. The system is driven by two external input signals: *engine speed* and *pedal angle*, as depicted in Fig. 4. Following the configuration in [19], we restrict the engine speed to the interval $[900, 1100]$ and the pedal angle to $[8.8, 61.1]$, ensuring nominal conditions.

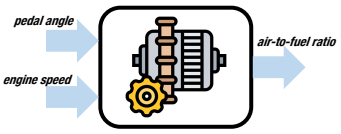

Fig. 4: Abstract fuel control

Runtime enforcement. A runtime enforcement strategy has the effect of changing the behavior of the monitored CPS. In this case study, we assume that we are aware (e.g., by domain knowledge or system inspection) that the system is

[1]The universal technical standard for floating point arithmetic.

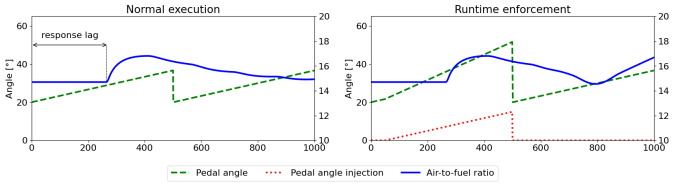

Fig. 5: Abstract Fuel Control – Runtime enforcement

not responsive enough to accelerations as the pedal is not well calibrated, and hence we change the pedal signal at runtime. Fig. 5 visualizes the effect of the enforcement. The plot on the left shows that, although the pedal is pushed twice (the two dashed-green ramps, whose scale is reported on the left y-axis), the system does not properly respond to the second acceleration, as the air-to-fuel ratio (blue line, whose scale is reported on the right y-axis) decreases after the first ramp.

The plot on the right, instead, shows how the runtime enforcement applied by RIVER helps in mitigating this problem. Specifically, during enforcement, we add an additional pressure to the pedal (dotted-red line in the figure), increasing the total pedal angle (dashed-green line). In this way, the value of the air-to-fuel increases also during the second ramp (specifically, after timestamp 800).

B. Performance Overhead

As RIVER preempts the execution of the CPS to perform the requested read and write operations, it introduces a slowdown. To verify that the overhead does not interfere with the CPS application, we performed the following analysis: 1) we first enabled the collection of eBPF performance statistics using in-kernel high precision timers with `sysctl`, then 2) we re-executed the use cases presented in Sect. IV, and finally 3) we extracted the relevant statistics using `bpftool`.

Regarding the experiment on ACC (see Sect. IV-A1), we measured 0.199 μs to preempt model execution, 1.177 μs to inject the masking signal, and 1.634 μs to read the output signals; a total of 3.010 μs per cycle. Given that the model is evaluated with a frequency of 1000 Hz and a period of 1 ms, such delay is practically negligible (i.e., more than two orders of magnitude difference). With regard to the AFC experiment (see Sect. IV-A2), we measured 0.295 μs to preempt model execution, 7.924 μs to inject a state perturbation, and 0.962 μs to read the output signals; a total of 9.181 μs per cycle. In this case, the model evaluation frequency is 200 Hz and the period 5 ms; again, a negligible overhead.

V. Lesson Learned

eBPF is a technology originally meant to be used in the security and deep instrumentation domains, where program intervention is performed at well defined interfaces. However, in the CPS domain this may not always be true, due to the potential presence of black box components. Since in our work we used well-established Simulink models, we solved the issue by leveraging the subsystem interfaces shown in Simulink.

Another aspect that is important to consider in a given application scenario is local proximity, which affects the communication delay between client and server. We currently relied on network sockets to implement the communication channel. However, the adoption of low latency Remote Procedure Calls can further reduce the delay.

VI. Conclusion and Future Work

Runtime verification is a popular approach for the analysis of CPS models, with different tools developed in the last decade. However, tasks such as adversarial attacks and runtime enforcement do not have adequate support, due to technical challenges in inspecting the internal state of the CPS model at runtime. To overcome these limitations, we proposed RIVER, a framework capable of reading and writing the internal state of the model thanks to the eBPF technology.

Future works include experimenting with RIVER developing new RV techniques that exploit its features. Moreover, we plan to integrate RIVER with open-source data visualization and monitoring platforms such as Grafana, improving observability through the creation of a dashboard.

References

[1] D. Facchinetti, "RIVER Artifact," 2026, doi: 10.5281/zenodo.19486626.
[2] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, *Introduction to Runtime Verification*. Springer International Publishing, 2018.
[3] A. Donzé, T. Ferrère, and O. Maler, "Efficient robust monitoring for STL," in *CAV*, 2013.
[4] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," *Formal Methods in System Design*, 2017.
[5] Z. Zhang, J. An, P. Arcaini, and I. Hasuo, "Online causation monitoring of signal temporal logic," in *CAV*, 2023.
[6] A. Corso, R. Moss, M. Koren, R. Lee, and M. Kochenderfer, "A survey of algorithms for black-box safety validation of cyber-physical systems," *Journal of Artificial Intelligence Research*, 2022.
[7] A. Chandratre, T. Hernandez Acosta, T. Khandait, G. Pedrielli, and G. Fainekos, "Stealthy attacks formalized as STL formulas for falsification of CPS security," in *HSCC*, 2023.
[8] Y. Sun, C. M. Poskitt, X. Zhang, and J. Sun, "REDriver: Runtime enforcement for autonomous vehicles," in *ICSE*, 2024.
[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[10] DynamoRIO Project, "DynamoRIO dynamic instrumentation tool platform," https://dynamorio.org/.
[11] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN*, 2007.
[12] L. Rice, *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*. O'Reilly Media, Inc., 2023.
[13] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *CAV*, 2010.
[14] Y. Annapureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-taliro: a tool for temporal logic falsification for hybrid systems," in *TACAS*, 2011.
[15] T. Yamaguchi, B. Hoxha, and D. Ničković, "RTAMT – Runtime Robustness Monitors with Application to CPS and Robotics," *STTT*, 2024.
[16] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Ničković, "CPSDebug: Automatic failure explanation in CPS models," *STTT*, 2021.
[17] Z. Zhang, J. An, P. Arcaini, and I. Hasuo, "CauMon: An informative online monitor for signal temporal logic," in *FM*, 2024.
[18] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *FORMATS*, 2010.
[19] T. Khandait, D. Lyu, P. Arcaini, G. Fainekos, F. Formica, S. Gon, A. Hekal, A. Kundu, C. Menghi, G. Pedrielli, R. Ray, Q. Thibeault, M. Waga, and Z. Zhang, "ARCH-COMP25 Category Report: Falsification," in *ARCH Workshop*, 2025.