AutoCPS: Control Software Dataset Generation for Semantic Reverse Engineering

Haoda Wang
Information Sciences Institute
University of Southern California
haodawan@usc.edu

Christophe Hauser
Information Sciences Institute
University of Southern California
hauser@isi.edu

Luis Garcia
Information Sciences Institute
University of Southern California
lgarcia@isi.edu

***Abstract*—Binary analysis of closed-source, low-level, and embedded systems software has emerged at the heart of cyber-physical vulnerability assessment of third-party or legacy devices in safety-critical systems. In particular, recovering the semantics of the source algorithmic implementations enables analysts to understand the context of a particular binary program snippet. However, experimentation and evaluation of binary analysis techniques on real-world embedded cyber-physical systems are limited to domain-specific testbeds with a low number of use cases–insufficient to support emerging data-driven techniques. Moreover, the use cases rarely have the source mathematical expressions, algorithms, and compiled binaries. In this paper, we present AUTOCPS, a framework for generating a large corpus of control systems binaries along with their source algorithmic expressions and source code. AUTOCPS enables researchers to tune the control system binary data generation by varying different permutations of cyber-physical modules, e.g., the underlying control algorithm, while ensuring a semantically valid binary. We initially constrain AUTOCPS to the flight software domain and generate over 4000 semantically different control systems source representations, which are then used to generate hundreds of thousands of binaries. We describe current and future use cases of AUTOCPS towards cyber-physical vulnerability assessment of safety-critical systems.**

I. INTRODUCTION

The ubiquitous expanse of the internet of things increased the interaction of commodity software with the physical world. As a result, software-based vulnerabilities may allow attackers to cause physical damage in safety-critical applications, e.g., the power grid [1] or nuclear reactors [2]. Thus, a common goal for both adversaries and security analysts is to reverse engineer closed-source or legacy systems in the context of their subsuming cyber-physical system [3]. In particular, cyber-physical vulnerability assessment commonly focuses on understanding the cyber-physical impact of low-level, embedded firmware that directly interfaces with the sensors and actuators of a cyber-physical system.

Recent research [4], [3], [5], [6] has focused on the problem of *semantic reverse engineering* of closed-source, low-level, and embedded binary programs for cyber-physical systems. Semantic reverse engineering involves analyzing a software program and connecting the extracted information to a semantic model, e.g., a cyber-physical system model. Researchers increasingly apply procedural and data-driven approaches to recover or identify math expressions within the given binary, usually by employing either signature-based approaches from a set of labeled binary representations [7], [8], [9] or by performing semantic-pattern matching to a set of known algorithmic implementations [4], [3]. Recent efforts [5] have aimed to combine traditional program analyses with machine translation to extract math expressions without a reference set of known functions. However, all approaches are difficult to generalize and evaluate across other real-world, cyber-physical domains as the evaluation datasets are domain-specific and limited in the variety of samples.

Semantic reverse engineers commonly evaluate on a single repository that can be compiled to various hardware targets, e.g., the Ardupilot repository [10] for robotic vehicles consists of source code for various autonomous vehicle platforms along with various microcontroller permutations. However, there are often a scarce number of permutations of control algorithm implementations that are semantically different, i.e., control algorithms based on fundamentally different mathematical expressions rather than just having variance at the low-level instruction set due to the compiler. For instance, the Ardupilot framework [11] has controllers for rovers, submarines, blimps, helicopters, and antenna trackers and supports over 20 hardware targets. However, there are only six code repositories that are semantically different since the associated control algorithms have been fine-tuned for their domains. Moreover, CPS software repositories rarely contain the source algorithmic and mathematical expressions–which could serve as ground truth for recovering program semantics [5].

In this paper, we introduce AUTOCPS, an automatic *flight software* (FSW) dataset generation framework. The phrase "flight software" is most often defined as a subset of embedded real-time software that executes onboard a spacecraft (including ground-based systems such as landers and rovers as well) or aircraft and provides capabilities such as attitude control and mobility [12]. However, we extend this definition of flight software in this work to also include control software running on other cyber-physical systems such as UAVs and rovers.

AUTOCPS aims to generate a large corpus of unique, end-to-end controller software source code repository. Each unique source code repository is associated with a set of source mathematical expressions as well as a set of compiled versions of the source code. To enable AUTOCPS, we first survey several CPS source repositories commonly used for cyber-physical vulnerability assessment and semantic reverse engineering. We summarize the surveyed repositories into a common modularized structure. AUTOCPS works by first modularizing the summarized representations of flight software algorithmic expressions. We define a randomization space for each module based on the number of possible valid choices, e.g., choosing from a set of state estimation methods. Users can configure the randomization space for each module to customize the FSW source code generation. AUTOCPS then uses an autocoder to translate the module randomization space and generation configuration to generate a large corpus of random but valid FSW software repositories. Our results show that AUTOCPS can generate over 4K unique source implementations of valid FSW systems with tuples of source math expressions and, optionally, a set of compiled binary files. We discuss how AUTOCPS can interface with software-in-the-loop simulation frameworks to dynamically validate the behavior of the generated flight software, as well as the broad set of research directions enabled by AUTOCPS.

Contributions. We summarize our contributions as follows.

- We summarize a generalized and modular mathematical representation of flight software systems from a survey of popular, open-source repositories.
- We present AUTOCPS[1], a control systems binary generation framework to generate a large corpus of random, semantically-unique flight software systems. AUTOCPS's initial modularization has a randomization space of 4K unique source software repositories, which can subsequently be compiled to any hardware platform with any number of compilation permutations.
- We provide a mechanism to interface each generated FSW sample with software-in-the-loop simulators to enable dynamic modeling and visualization.

We open-source both the binary generation pipeline along with a sample large FSW repository dataset.

II. METHODOLOGY

In this section, we describe the primary research challenges we address over the development of AUTOCPS.

A. Challenges and Key Insights

Three primary challenges were identified during the development of AUTOCPS. Solving these challenges provided key insights into the design of common flight software frameworks.

For flight software generated by AUTOCPS to accurately represent different physical platforms, we needed to find a generalized flight software architecture. Flight software on platforms are often highly adapted to their use case, and thus implement a wide array of different functionalities [12]. Thus, to accurately model these systems, a set of core functionalities common to all flight software was ascertained, which guided the development of the AUTOCPS-generated software.

By modularizing the design of our flight software, AUTOCPS users can gain additional insight by analyzing specific modules before attempting to work on the full binary. However, this required AUTOCPS to split up the modules sensibly. In particular, this required grouping software tasks such as clock management, waypoint navigation, and attitude control into discrete sets of interdependent tasks.

The final challenge involved finding a method to randomly generate valid flight software modules. We define a "valid" flight software module to be a module which executes every task required of this module as determined by the problem above. Note that for the purposes of AUTOCPS the module may be valid without being able to correctly perform the task, as long as it exposes the respective function for the controller to call.

B. AUTOCPS *Design Goals*

Figure 1 depicts an overview of the AUTOCPS design. To address the above research challenges, we aim to achieve the following design goals:

- Provide a "summarization" of cyber-physical modules in flight software based on surveyed techniques in the real world, including valid permutations within each module.
- Design an autocoder that can generate a maximal set of random, unique, and valid set of flight software source code based on modular approach as well as a dataset configuration.
- For each flight software, generate a tuple that includes the set of mathematical expressions, the associated source code, as well as a set of compiled binaries.

III. MODULARIZING FLIGHT SOFTWARE DESIGN

We find that flight software for these vehicles differs primarily in their propulsion method and movement limits. For example, Jackson examines the design of missile flight control systems and concludes that such systems contain 4 basic elements: an inertial measurement unit (IMU), an autopilot, an actuator, and an airframe dynamics controller [13].

Ardupilot has a similar set of components. The IMU is now split into various sensors, including an Inertial Sensor, a Barometer, and a GPS. The autopilot is split into position control and attitude control, while the actuation and airframe dynamics are combined into a servo control library [10].

NASA-JPL also labels the control software onboard their Mars rovers as flight software [14]. The flight software components within the rovers are grouped into more fine-grained divisions labeled as modules. These modules similarly correspond to the components found in the previous vehicles. Just like in the Ardupilot software, the system's location is determined by a set of modules, including iit, imu, and ras. Attitude control is further divided into the seq, acs, nav, and aman modules, while position control is done by the drive module. Finally, the servo control is controlled by the mot module

Through our survey of these and other flight control software, we found that every flight control system consists of the following core functions:

- Some form of input to receive attitude and location data, such as sensors or IMU

[1]The source code for AUTOCPS can be found at https://github.com/usc-isi-bass/AutoCPS.

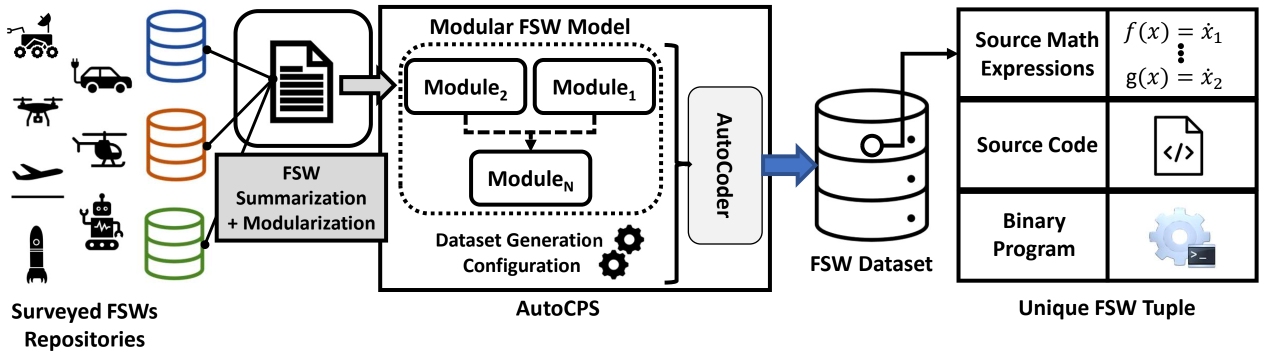

Fig. 1. Overview of AUTOCPS pipeline. We first surveyed common flight software designs and generalized them into a modular design that includes all core capabilities across multiple physical systems. Select components of this generalized FSW can be randomized within a set of restrictions by the autocoder, which also generates source code from a set of control equations.

- A set of waypoints, either user-defined, automatically generated, or input at runtime
- A method to convert the waypoint and attitude data into a modification of the system's control surfaces
- An interface to the control surfaces

While specific systems may contain code to manage certain domain-specific functions of the system, every system requires this set of functions to qualify as flight software. The following sections describe a generalized flight software design that integrates these core functions into a simple architecture.

A. Modularization of Flight Software

Flight software is often organized into either a module-based or layer-based architecture for ease of development. This design choice can be seen in commonly used flight software frameworks such as Ardupilot [10] and FPrime [15].

In a layered design, the flight software is divided into higher and higher levels of abstraction. Thus, one layer would be an interface with hardware, while the layer above may translate higher levels' movement commands into servo movements. Such a design can be seen in the Ardupilot flight software. Some tasks done by the flight software, such as auto-stabilization, may reach across multiple layers. While this type of design is simple to program, it is not amenable to code reuse. For example, the same vector propagation equation may be used by both the position and attitude control layers, which may result in code being duplicated for each layer.

In a modular design, this problem can be solved by having each of these modules reflect a set of closely related tasks done by the flight software. For example, the flight software could split off vector propagation into a module that interfaces with the position and attitude control modules. Such a design is found in the F Prime framework [15] and the Mars Exploration Rovers' flight software [14]. We similarly choose a modular design for our autocoded flight software. This is due to two primary factors:

- A modular design is a superset of the layer design since each layer can also be represented by a subset of modules. Therefore using modules allows us to emulate both types of designs.
- The modular design also allows for some modules to be offloaded to hardware. This is especially useful as FPGAs and ASICs are being adopted as coprocessors in flight controllers due to their power efficiency and computation speed.

B. Flight Software Design

We design our module architecture to achieve the four core functions outlined above while also dividing tasks such that each module's tasks are independent of the others. As shown in figure 2, modules may also depend on the outputs of other modules, reflecting the real-time multi-threaded design of real-life flight software implementations. Furthermore, additional modules can be easily added to our architecture to account for more unique use cases.

Two modules represent the navigational controls of the flight software. The `seq` module stores sequences of actions to be done. This can reflect an autopilot inputting new waypoints or a controller sending in new directional data to the system. The `autonav` module will contain optional automated additions to the input sequence. This represents automated tasks that will modify the waypoints of the system, such as auto-stabilization of a helicopter or plane, or a rover's obstacle avoidance logic.

Four modules control positioning hardware and process their data. The `imu` module receives data from the inertial measurement unit or the GPS and other related sensors. It then converts that data into direction and position vectors. These vectors will then be sent to the `kalman` module described below. Similarly, the `sensor` module receives data from sensors that don't directly provide direction and position information, such as a camera or an antenna. The `kalman` module wraps a Kalman filter to correct for errors in sensor measurements. The design of this module can also differ based on the model used. Once the sensor data has been received, the `ivp` module converts the data from the sensor's reference frame to the inertial reference frame of the system.

Three modules are used to convert the waypoints and position inputs into a set of movements done by the system. The `pos_ctrl` module unifies all waypoints from the controller (`seq`) as well as the input data from the `ivp` and `kalman` modules. It uses this data to select the next position to move to. The `att_ctrl` module takes the output of `pos_ctrl` and determines the next direction and heading for the physical system with this data. Finally, the `servo_ctrl` module contains code to interface with the control surfaces of the physical system, which requires knowledge of the design of the physical system. Communication to actual servos from this module is stubbed for our use case.

Finally, a set of auxiliary modules allow the system to operate within a testbed with no hardware support. The `clock` module provides a unified interface for other modules to fetch timing data from the system. Similarly, the `datatypes` module contains classes common to all modules, such as a Quaternion or a 3-dimensional vector, as well as corresponding helper methods. All autocoding work is done in the `autocode` module, which provides s-curve navigation and various other methods that modify the semantics of the flight software. Lastly, the `stub` module provides an interface for the generated flight software to communicate with the computer instead of the hardware. This module can be further modified to integrate into common software-in-the-loop simulators if testing is desired.

IV. Implementation and Corpus Description

AutoCPS consists of two core tools. The first is a generalized flight software that provides common functions used across all four supported vehicle types as well as a rudimentary testbed. The second is an autocoder library, which generates parameters used by the generalized flight software according to a set of constraints. Combined, the autocoder and flight software can generate over 4000 semantically different binaries. The source code for AutoCPS is available on GitHub [16].

A. Flight Software

AutoCPS generates a single-threaded, non-preemptive static binary that represents a random flight software. Though the industry standard is using a threaded real-time OS with a preemptive scheduler [15], this choice simplifies the autocoding process and does not significantly affect static analysis tools.

As outlined above, we divide the autocoded flight software into modules. These modules and their functions are listed in Table I. Each module acts as an independent library, with its own `.cpp` and `.h` files. These modules can be plugged in and out of the program as needed to simulate tasks offloaded to hardware. In particular, we add two modules that allow us to interface with the autocoder and user input. For ease of use, the `autocode` module contains functions generated by the autocoder, such as the code for s-curve navigation and some limit checking. Furthermore, the `stub` module allows us to abstract away hardware interfacing code without affecting the core modules within the flight software.

The design of our flight software provides approximations for four different vehicle types listed below and depicted in Figure 3. While these vehicles are the main focus of our dataset, it is trivial to extend the flight software to also represent other vehicle types.

- Ground vehicles, such as rovers and cars
- Fixed-wing aircraft, such as airplanes
- Rotary-wing aircraft, including helicopters and commodity UAVs
- Rockets

B. Autocoder Library

The autocoder is contained within a Python library that can randomize system parameters. This library can be easily integrated into other projects and provides fine tuning that would be cumbersome to do with a command line. We also provide a simple CLI interface for the autocoder with a much smaller space of possible generated code.

The library is divided into two major portions. One is used to generate parameters for the physical system, while the other handles the software characteristics. This split allows users to generate a similar software system for multiple different physical systems. For example, one might choose to test various generated s-curves on the same physical system.

The autocoder generates software functions, including maximum speed checks and S-curve fitting and navigation, in the `autocode.cpp` file. The physical and software parameters of the program are similarly generated in the `params.h` file. Preprocessor directives within each module control their behavior based on the settings within `params.h`.

C. Pipeline Design

To generate a new control system, the user begins by setting their required constraints in a new instance of our `PhysicalSystem` or `SoftwareSystem` class. Setting these constraints is optional, as the autocoder has sensible defaults for various physical attributes. For example, the mass and volume will be randomly generated, but upper bounds are set for volume and density. Other constraints may include one of the four supported types of physical systems, though a general `PhysicalSystem` type can also be used instead of the `Rover` or `Plane` types. After the constraints are set for the physical and software system, helper methods can set the uninitialized values randomly.

Once all the parameters are set, the `CodeGeneration` class is used to create a new `params.h` and `autocode.cpp` file according to these parameters. These generated source files are then moved into the directory containing the flight software.

To preserve as many different iterations of a binary as possible, the flight software binary is built using CMake. This allows us to vary compiler options such as optimization levels and target architectures, further increasing the space of possible binaries generated by AutoCPS. Each module is also built as a static library first before being incorporated into the final binary, which allows users to analyze specific modules rather than the full flight software if desired.

D. Randomization space

The autocoder has a few degrees of freedom to generate semantically different flight software:

- 4 different types of vehicles, including planes, rockets, helicopters, and rovers.

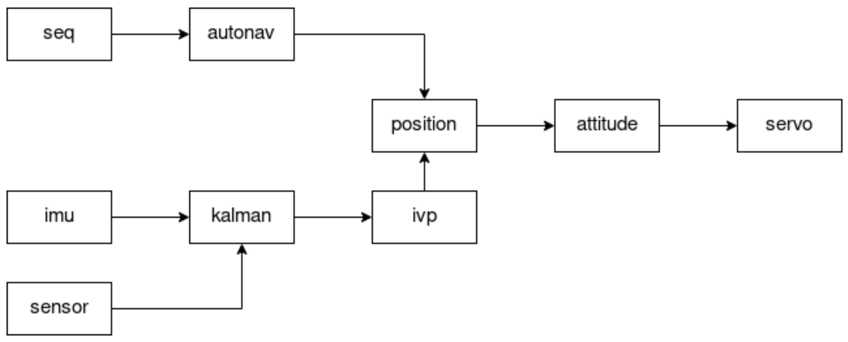

Fig. 2. AUTOCPS Module architecture. Arrows represent the minimum required data flow between modules, but additional data flows, such as from `autonav` to `attitude`, may be included as needed. Note that `clock`, `stub` and `autocode` modules are not shown.

- 4 choices between JPL and Hamilton quaternion conventions. In JPL quaternions, $ijk = 1$, while in Hamilton quaternions $ijk = -1$.
- 2^4 different ways to include or exclude 4 specific modules to simulate hardware offload. We selected the `autonav`, `kalman`, `imu` and `sensor` modules as offloadable, as their tasks may be done in hardware or ignored altogether. Without the `imu` or `sensor` module, the system will only receive input from `seq`, thus turning it into a drone.
- 4 different sigmoid curve functions, and 4 dummy curve-fitting methods. While these functions are currently extremely rudimentary, the relevant code can be easily extended to include more intricate functions.
- 4 different methods of calculating π, including a simple constant and through trigonometric functions.

This allows us to generate over 4000 semantically different versions of flight software with our default settings. Further modifications can also be made to other parameters of the software, such as the dimensions of the system as well as the positions and number of various sensors. While these changes will not significantly modify the semantics of the software, they may provide more data with different constants or code flow to work with.

Changing the target architecture or compiler options may cause the binary to have an entirely different control flow [17]. AUTOCPS can also compile the autocoded flight software with different sets of build options to further increase the search space of binaries it can generate. This is achieved by a set of CMake presets that can be covers a variety of compilers and optimization levels, and can be easily extended to other use cases.

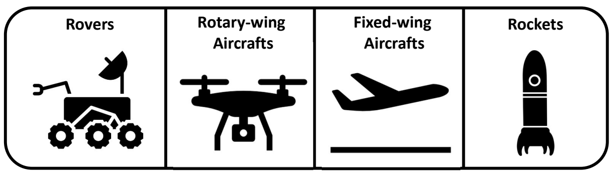

Fig. 3. Overview of domains extracted from survey.

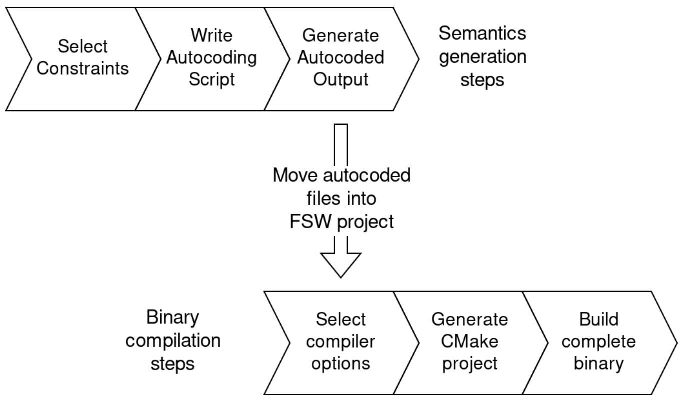

Fig. 4. Steps to generate a FSW binary using AUTOCPS. We can divide it into steps that modify the semantics of the program, and steps that modify the binary of the program.

Module Name	Description
`att_ctrl`	Attitude control logic
`autocode`	Autocoded parameters
`autonav`	Autopilot/auto-navigation code
`clock`	Timing and clock-related code
`datatypes`	Basic datatypes for other modules
`imu`	Interfacing code for inertial measurement units
`ivp`	Inertial vector propagation code
`kalman`	Kalman filtering code
`pos_control`	Position control logic
`sensor`	Interfacing code for general sensors
`seq`	Input flight software sequences
`servo_control`	Interfacing code for control surfaces
`stub`	Interfacing code between FSW and simulator/CLI

TABLE I. LIST OF MODULES IN FSW GENERATED BY AUTOCPS

V. RELATED WORK

In this section, we will describe works related to semantic program analysis that could benefit utilizing AUTOCPS. In particular, we will discuss their target evaluation datasets to highlight the contributions of AUTOCPS. The emerging body of semantic reverse engineering frameworks often evaluate on domain-specific datasets that cannot demonstrate the generalizability of the approaches.

Mismo [4] leveraged a contrived dataset of known control algorithms to perform pattern matching against ten control system binaries. RVFuzzer [18] aims to fuzz the inputs of robotic vehicles to discover *semantic bugs* within the controller code. They evaluate their approach on two different control programs for a single quadcopter model. Similarly, Choi et. al. use UAV binaries to generate sensor traces to discover *control invariants* to monitor the drone dynamics [6]. They reverse engineer the drone binary to integrate the control invariant monitor. They evaluated their approach on only eleven UAV binaries. ICSRef [3] is an automated reverse engineering framework for industrial control system binaries. They leveraged a commercial development environment to generate a large set of industrial control system binaries along with a knowledge base of semantic information stemming from the development environment metadata. Although this work provides a similar pipeline to automated control systems binary generation, the dataset is domain-specific, and they do not formalize the generation of various control algorithm modules that are semantically different. The most similar approach to AUTOCPS is PERFUME [5]–a framework to extract mathematical expressions from low-level binary representations without a reference dataset. They implemented a similar dataset generation pipeline by utilizing a random math expression generator [19], translating each expression to source code, and then compiling the source code to a binary file. Although this approach allows for end-to-end evaluation of binary representations to source mathematical expressions, the associated mathematical expressions are limited to simple arithmetic expressions that are not close to real-world, cyber-physical binaries.

VI. DISCUSSION

AUTOCPS provides a valuable tool for CPS reverse engineering projects by providing a corpus of randomly generated, semantically unique, and easy-to-understand FSW binaries that provide real-life examples of control equations. In this section, we will discuss the limitations of AUTOCPS, as well as future work to be done in this direction.

A. Limitations

The FSW generated by AUTOCPS is much simpler than its real-life counterparts, which often run on a real-time preemptive operating system [12]. However, AUTOCPS generates a flight software that simply runs using a non-preemptive loop. This simplifies the analysis of the program with methods such as symbolic execution while preserving the semantics of key FSW functions. However, this may result in different results when executed compared to an FSW running on a real-time operating system.

AUTOCPS may also be extended beyond the reverse engineering use case by being able to reflect a particular real-life physical system. Doing so with AUTOCPS will result in infinitely many different drones for each semantically similar FSW. Thus, simulations may need to be run upon the FSW to examine if the control equations accurately reflect the physical system. However, as noted in Section V, AUTOCPS's evaluation dataset is far more comprehensive and generalizable than prior evaluation datasets.

B. Future Work

AUTOCPS currently provides interfaces to work with four subtypes of physical systems. However, other physical systems, such as submarines and stationary landers, also exist and are only represented with a generic type in our autocoder. Future work can augment the set of AUTOCPS's modules to include these systems into our autocoder by adding specific control equations for each of these systems.

Given the modularity of our system, the dataset generation is easily amenable to new modules serving these use cases. However, future work can also focus on automating the extraction of source code modules, such as using machine learning based approaches. Moreover, data augmentation approaches can be explored to add more noise to the generation process.

For AUTOCPS to satisfy our definition of *valid*, we must define the exact tasks required of each FSW module within AUTOCPS. This set of tasks is defined in each module's header file and the function declarations within, which is then implemented by the autocoder or by the settings in `params.h`. In the future, we hope to restrict our definition of *valid* to only include control semantics that can achieve tasks provided to the FSW. This requires a way to validate the flight software. While a simple validation tool is available in the FSW in CLI form, this tool does not allow for external inputs such as obstacles or account for other forces such as wind acting on it. Thus, future work will include extending our `stub` module such that AUTOCPS can integrate into software-in-the-loop physics simulators.

VII. CONCLUSION

In this paper, we introduce a generalized design for flight software that can be easily analyzed by reverse engineering tools, while preserving the semantics of real-life control algorithms. Using this generalized design, we develop AUTOCPS, a tool to generate simplified flight software for binary analysis testing. AUTOCPShas a search space of over 4000 semantically different source files to generate from, all of which create valid variations of flight software. Further extending the search space is the number of compiler options and physical device characteristics available to the user, with nearly infinitely many binaries available for each variant of the original control equations. We describe how AUTOCPS can be interfaced with cyber-physical simulators and discuss immediate future research directions. AUTOCPS provides a state-of-the-art evaluation dataset for future cyber-physical security research directions that reach well beyond the scope of semantic reverse engineering.

VIII. ACKNOWLEDGEMENT

This research is based upon work supported by DARPA's ReMath program, Contract HR00112190020. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

References

[1] C. Vellaithurai, A. Srivastava, S. Zonouz, and R. Berthier, “Cpindex: Cyber-physical vulnerability assessment for power-grid infrastructures,” *IEEE Transactions on Smart Grid*, vol. 6, no. 2, pp. 566–575, 2014.

[2] J. P. Farwell and R. Rohozinski, “Stuxnet and the future of cyber war,” *Survival*, vol. 53, no. 1, pp. 23–40, 2011.

[3] A. Keliris and M. Maniatakos, “ICSREF: A framework for automated reverse engineering of industrial control systems binaries,” in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[4] P. Sun, L. Garcia, and S. Zonouz, “Tell me more than just assembly! reversing cyber-physical execution semantics of embedded iot controller software binaries,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 349–361.

[5] N. Weideman, V. K. Felkner, W.-C. Wu, J. May, C. Hauser, and L. Garcia, “Perfume: Programmatic extraction and refinement for usability of mathematical expression,” in *Proceedings of the 2021 Research on offensive and defensive techniques in the Context of Man At The End (MATE) Attacks*, 2021, pp. 59–69.

[6] H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Deng, “Detecting attacks against robotic vehicles: A control invariant approach,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 801–816.

[7] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, “Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned,” *arXiv preprint arXiv:2011.10749*, 2020.

[8] D. Xu, J. Ming, and D. Wu, “Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping,” in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 921–937.

[9] P. Sun, L. Garcia, G. Salles-Loustau, and S. Zonouz, “Hybrid firmware analysis for known mobile and iot security vulnerabilities,” in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 373–384.

[10] A. D. Team, “Code Overview (Copter),” https://ardupilot.org/dev/docs/apmcopter-code-overview.html, 2021.

[11] H. Bin and A. Justice, “The design of an unmanned aerial vehicle based on the ardupilot,” *Indian Journal of Science and Technology*, vol. 2, no. 4, pp. 12–15, 2009.

[12] D. Dvorak, “Nasa study on flight software complexity,” in *AIAA Infotech@ Aerospace Conference and AIAA Unmanned... Unlimited Conference*, 2009, p. 1882.

[13] P. B. Jackson, “Overview of missile flight control systems,” *Johns Hopkins APL technical digest*, vol. 29, no. 1, pp. 9–24, 2010.

[14] G. E. Reeves and J. F. Snyder, “An overview of the mars exploration rovers’ flight software,” in *2005 IEEE International Conference on Systems, Man and Cybernetics*, vol. 1. IEEE, 2005, pp. 1–7.

[15] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison, “F prime: an open-source framework for small-scale flight software systems,” *Small Satellite Conference*, 2018.

[16] “AutoCPS git repository,” https://github.com/usc-isi-bass/AutoCPS.

[17] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, *Unleashing the Hidden Power of Compiler Optimization on Binary Code Difference: An Empirical Study*. New York, NY, USA: Association for Computing Machinery, 2021, p. 142–157. [Online]. Available: https://doi.org/10.1145/3453483.3454035

[18] T. Kim, C. H. Kim, J. Rhee, F. Fei, Z. Tu, G. Walkup, X. Zhang, X. Deng, and D. Xu, “{RVFuzzer}: Finding input validation bugs in robotic vehicles through {Control-Guided} testing,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 425–442.

[19] G. Lample and F. Charton, “Deep learning for symbolic mathematics,” in *International Conference on Learning Representations*, 2019.