

USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

RVDebloater: Mode-based Adaptive Firmware Debloating for Robotic Vehicles

Mohsen Salehi and Karthik Pattabiraman, *The University of British Columbia*

https://www.usenix.org/conference/vehiclesec26/presentation/salehi

This paper is included in the Proceedings of the
4th USENIX Symposium on Vehicle Security and Privacy.

August 10–11, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-56-4

Open access to the Proceedings of the
4th USENIX Symposium on Vehicle Security and Privacy
is sponsored by USENIX.

ARTIFACT EVALUATED
USENIX
AVAILABLE

RVDebloater: Mode-based Adaptive Firmware Debloating for Robotic Vehicles

Mohsen Salehi
The University of British Columbia

Karthik Pattabiraman
The University of British Columbia

Abstract

Embedded firmware is becoming increasingly larger, thereby expanding its attack surface. Despite the increase in firmware size, many embedded devices, such as robotic vehicles (RVs), operate in distinct modes, each requiring only a small subset of the firmware code at runtime. Debloating is an approach to reduce attack surfaces by removing or restricting unneeded code. Unfortunately, existing techniques suffer from significant limitations, such as coarse granularity and irreversible code removal, limiting their applicability. We propose *RVDebloater*, a novel adaptive debloating technique for *mode-based embedded devices* such as RVs that automatically identifies unneeded firmware code for each mode using either static or dynamic analysis, and dynamically debloats the firmware for each mode at the function level at runtime. *RVDebloater* uses a software-based enforcement approach to ensure that only the required functionalities are accessible from each mode and supports diverse *mode-based embedded devices*.

We implemented *RVDebloater* using the LLVM compiler and evaluated its efficiency and effectiveness on six different RVs, including both simulated and real ones, with different real-world missions. We find that device requirements change throughout its lifetime for each mode, and that many critical firmware functions can be restricted in other modes, with an average of 85% of functions not being required. The results showed that none of the missions failed after debloating with *RVDebloater*, indicating that it neither incurred false positives nor false negatives. Further, *RVDebloater* prunes the firmware call graph by an average of 45% across different firmware. Finally, *RVDebloater* incurred an average performance overhead of 3.9% and memory overhead of 4% (approximately 0.25 MB) on real RVs, and also prevented three real attacks.

1 Introduction

Robotic vehicles (RVs) such as unmanned aerial vehicles (UAVs), unmanned ground vehicles (UGVs), and remotely operated underwater vehicles (ROVs), are becoming increasingly popular. A wide range of features (e.g., system calls and device drivers [43]) and hardware functionalities is being incorporated into these devices to support the requirements of different users, even though each user may only require a fraction of these features. This expansion enhances potentially buggy code, even in unused features, thereby increasing the devices' attack surface [17, 40, 50]. For instance, UAVs are vulnerable to various security vulnerabilities, such as buffer overflows [40, 45], which can allow attackers to hijack the control flow and manipulate critical programs like the flight control program, causing crashes.

One approach to reduce attack surfaces is minimizing or restricting unwanted features (e.g., security-critical system calls) through a technique known as *debloating* [31, 33, 37, 38, 41, 42, 45, 49]. Debloating *keeps* a *small subset* of firmware code necessary for specific workloads while *restricting* the execution of the remaining code [31, 36, 45, 52], or *removes* unwanted features such as peripherals (e.g., USB and UART [41, 42]). Although these techniques have gained popularity in desktop [31, 37] and server applications [33, 44, 52], embedded systems have not received sufficient attention [45].

Debloating techniques can be categorized into two main classes: *fixed* and *adaptive* techniques. *Fixed debloating techniques* such as SHARD [31], Minion [45], and FACE-CHANGE [37] analyze applications throughout the entire lifetime, without considering how applications' requirements change during different execution phases. For instance, critical firmware code or system calls like `execve` or `fork` are usually executed only during the initial moments of an application's lifetime. However, *fixed debloating techniques* often fail to disable such system calls or code, leaving these available to be used in exploits [36].

In contrast, *adaptive debloating techniques* [36, 52] consider the changing requirements of an application through different execution phases. For example, Ghavamnia et al. [36] and Rajagopalan et al. [52] proposed temporal system call specialization techniques for server applications, which restrict available system calls based on the application's execution phase. They use static analysis techniques to identify the required system calls for two distinct phases—*initialization*

and *serving*—and block unneeded system calls, such as `fork` and `execve`, after transitioning to the *serving* phase.

Unfortunately, adaptive debloating techniques [36,52] have two limitations. First, similar to fixed debloating techniques, they focus on disabling security-critical system calls such as `execve`. However, embedded devices' firmware, particularly those used in RVs (e.g., ArduPilot [9], PX4 [23]), contains critical functions, such as `disarm_motors`, which should only be executed in specific modes[1], e.g., `Turtle` mode (i.e., flipping the copter). If not, they can lead to serious safety violations.

Second, these techniques use a Linux mechanism known as Seccomp-BPF to restrict the set of available system calls for each phase [25]. However, the Seccomp-BPF mechanism for filtering system calls is irreversible, meaning that once a system call is filtered out, it cannot be reallowed. In contrast to server applications, which typically have a simple control flow and do not revert to the previous phase, RVs often experience frequent mode changes, including transitions back to earlier modes (e.g., `AUTO <-> LOITER`). Therefore, debloating techniques that cannot *dynamically* adjust access restrictions on firmware functions at runtime, such as those relying on Seccomp-BPF, are unsuitable for embedded devices.

Our key insight is that many embedded devices-such as RVs-operate in distinct modes (e.g., `LAND` or `TAKEOFF`) at runtime, with each mode requiring a specific set of functionalities, implemented as functions in their firmware. We call these devices mode-based embedded devices. With this insight, we propose *RVDebloater*[2], an adaptive debloating approach that automatically identifies the set of functionalities required by each mode (e.g., `TAKEOFF`) at the function-level and then monitors the device at runtime to ensure that only those firmware functions are allowed to execute when the device operates in the corresponding mode. For instance, when RVs are in `AUTO` mode, travelling from point A to B, they do not require the motor disarming function designed for `LAND` mode; misuse of this function can result in crashes. *Thus, we can restrict these functionalities (firmware functions) to their respective modes at runtime.* In this paper, we focus on RVs as a prominent class of embedded devices and evaluate *RVDebloater* on six RVs; however, *RVDebloater* can be generalized to other *mode-based embedded devices*, which we leave to future work.

RVDebloater has two main goals: 1) Specializing firmware code by *dynamically* adapting it to changes in the device's requirements at the mode level throughout its execution lifetime, and 2) Enforcing restrictions on firmware code at *function-level granularity*, providing finer specialization than system call-level [31,36,52] or hardware-level [41,42] (e.g., disabling unused peripherals) techniques.

RVDebloater addresses the above goals through two innovations. First, it uses either *static* or *dynamic analysis* to identify per-mode firmware code coverage, while considering changes in the device's requirements throughout its execution (i.e., the set of firmware functions required for each mode). Second, *RVDebloater* uses a reversible, software-based enforcement technique that statically instruments indirect control flow transfer instructions (e.g., indirect call sites and function return instruction) and the mode-switching function to redirect control flow, monitor function execution and mode transitions, and restrict access to functions belonging to other modes. Upon a mode switch, *RVDebloater* loads the required firmware functions for the new mode by applying per-mode configurations (goal 1), including *required* functions (goal 2) identified by either *static* (*RVDebloater-static*) or *dynamic analysis* (*RVDebloater-dynamic*). Henceforth, *RVDebloater* refers to both variants unless stated otherwise.

To the best of our knowledge, RVDebloater is the first fine-grained, adaptive debloating framework for mode-based embedded devices that automatically identifies and specializes the firmware code at the function-level for each mode.

Our main contributions are summarized as follows:

- Propose two automated analysis techniques (static and dynamic) to identify the required firmware code at the function-level, tailored to the operational requirements of RVs across different modes (e.g., `LAND`, `TAKEOFF`).
- Propose a runtime adaptive debloating technique that tracks mode switches and a novel reversible software-only enforcement technique to dynamically specialize the firmware code for each mode at runtime, thereby supporting frequent mode changes.
- Design *RVDebloater*, a fine-grained debloating framework to integrate the above techniques for mode-based embedded devices such as RVs. *RVDebloater* is implemented using the LLVM compiler [18], making it portable across different devices. Further, it requires no special hardware support (e.g., MPU or MMU), unlike prior compartmentalization and hardware-based isolation techniques [35,59], making it suitable for resource-constrained RVs.
- Evaluate *RVDebloater* on four simulated and two real RVs, all running one of two widely used open-source autopilot systems, ArduPilot or PX4, across a diverse range of scenarios and missions. We also evaluated it against three realistic code-reuse attacks on the RVs.

The results demonstrate that (1) *RVDebloater-static* reduces the attack surface by an average of 41% of the firmware functions. (2) *RVDebloater-dynamic* identifies all *required* functions for each mode by running RVs with only up to 10 missions, and achieves an average attack surface reduction of 85%. (3) Neither *RVDebloater-static* nor *RVDebloater-dynamic* incurred any false-positives (i.e., no mission fails due to a missing function), (4) The average performance overhead and power consumption overhead of *RVDebloater* on

[1] We use *modes* (e.g., `RTL`, `AUTO`) to refer to the *phases* of the RV.

[2] RVDebloater is available at: https://github.com/DependableSystemsLab/RVDebloater.git

real RVs are 3.9% and 0.47%, respectively. Further, the overall mission time was not increased due to *RVDebloater*. (5) The firmware instrumented by *RVDebloater* incurred an average memory overhead of about 4%, which is about 0.25 MB, and (6) *RVDebloater* successfully prevented all three attacks.

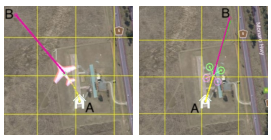
2 Background

2.1 Memory Corruptions

The majority of embedded devices' software, including firmware, is developed in low-level programming languages (i.e., C or C++) due to their efficiency and ability to provide full control over the underlying hardware. However, these languages are type-unsafe, requiring developers to carefully manage memory accesses and ensure their validity to prevent issues like dereferencing invalid pointers. The lack of memory safety in these languages, combined with developers' failure to meet these responsibilities, leads to memory corruption vulnerabilities (e.g., buffer overflow) that attackers can exploit to manipulate device behavior maliciously or even gain full control over the system's execution flow [57] (e.g., control flow hijacking attacks). Control flow hijacking attacks occur when a code pointer, such as a return address (backward edge) or a function pointer (forward edge), on the backward edge (e.g., return addresses) or forward edge (e.g., function pointers) becomes corrupted.

2.2 Mode-based Embedded Devices

Robotic vehicles (RVs) switch between different modes such as `AUTO` and `RTL` while performing missions like delivering a package from point A to B. Examples of simple missions for the ArduPilot firmware are shown on the right for ArduPlane and ArduCopter.



These modes are predefined in RVs' autopilot software such as ArduPilot [9] and PX4 [23]. Further, to manage their tasks, mode-based embedded devices often rely on real-time operating systems (RTOS) such as NuttX [3]. All these software components—the autopilot software, RTOS, and other essential modules like device drivers—are integrated into the device's firmware. As an example, Figure 1 presents mode switching in *ArduCopter* [4,10], a sub-platform of the ArduPilot software. The list of ArduPilot and PX4 modes is presented in the extended version of this paper available on arXiv [55].

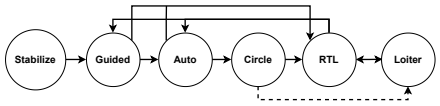


Figure 1: Mode switching process in ArduCopter.

Many embedded devices, including RVs, often lack hardware security features, such as Memory Management Unit (MMU), which results in all applications being mapped into a single memory space [45]. Therefore, these applications have unrestricted access to the entire shared memory in the privileged mode, enabling attackers to exploit a single memory corruption vulnerability to take over the entire system.

Furthermore, these devices may not support Memory Protection Units (MPUs) [53]. Even when an MPU is available, configuring and controlling memory access presents challenges due to its limitations. Unlike the MMU, which enforces access restrictions through page tables and virtual memory, an MPU operates with a limited number of hardware registers, typically supporting only 8 to 16 memory regions (e.g., 8 regions in ARMv7-M [35]). Each region is protected using read, write, and execute permissions, configurable for both privileged and unprivileged modes. Also, MPU regions must be a power of two in size, with a minimum size of 32 bytes, and their starting addresses must be a multiple of their size. This severely restricts MPUs' use in these devices.

3 Motivation and Threat Model

3.1 Debloating Techniques' Limitations

L1: Treating Applications as Monolithic Entities with Fixed Requirements. *Fixed techniques* in attack surface reduction [31, 38, 45, 48] specialize or remove functionality based on the entire application's lifetime requirements. However, *adaptive debloating techniques* [36,52] have shown that considering the changing requirements during the application's execution phases is more effective. In particular, many devices and programs, such as mode-based embedded devices and server applications, operate in distinct modes, with their requirements changing based on the execution phase.

L2: Removing Functionalities. *Fixed debloating techniques* [41,42] often remove functionalities, such as hardware features or kernel source code, that are not required at any point during the applications' lifetime. On the other hand, these techniques retain critical functionalities used for legitimate purposes during the application's operation, while some of these functionalities are only executed for a short period of time. Therefore, to prevent over-debloating and potential system crashes, these techniques often overestimate the functionalities required. Retaining such functionalities would inflate the attack surface.

L3: Using MPU. Some *Fixed debloating techniques* [35, 45] use MPUs to control memory access. However, this hardware may not be available on all devices, particularly embedded systems [53]. Moreover, due to MPU limitations (§2.2), these methods must rely on clustering techniques to group code for multiple processes. Thus, processes may gain access to unrequired code, thus leading to security risks.

L4: Considering System Calls. Existing debloating techniques, both *fixed* and *adaptive*, only focus on system calls to reduce attack surfaces. However, other functionalities implemented as firmware functions, such as those in RV autopilot software, may also pose security risks if accessible throughout the device's execution lifetime. For instance, Listing 1 and Listing 2 show two security-critical functions in ArduPilot for the `Land` and `Turtle` modes, respectively. The purpose of the function shown in Listing 1 is to land the RV when it is in specific states, such as upon reaching its destination. Also, the function in Listing 2 is executed to disarm the motors if the RV lands upside down, and enters the `Turtle` mode to flip the drone upright. However, executing `disarm_motors` during flight immediately kills the motors, causing a crash. Thus, these functions should not be accessible in other modes, such as `AUTO` or `GUIDED`, as attackers could exploit vulnerabilities to execute these functions.

```
1 void ModeLand::run () {
2   if (control_position){
3         gps_run();
4    }else {
5         nogps_run();
6    }
7 }
```

Listing 1: Landing function for ArduCopter's Land mode.

```
1 void ModeTurtle::disarm_motors(){
2    //... omit
3    motors->armed(false); // disarm
4     //... omit
5    hal.util->set_soft_armed(false);
6 }
```

Listing 2: ArduCopter's function executed in Turtle mode to disarm the vehicle motors if it lands upside down.

L5: Irreversible Mechanism. Although *adaptive debloating techniques* like SysPart [52] and TSCP [36] are more effective at reducing attack surfaces compared to *fixed techniques*, they employ an irreversible mechanism to disable system calls after transitioning to the *serving* phase. Specifically, these techniques leverage a Linux kernel feature called Seccomp-BPF, which is a one-way process, meaning the filtered system calls cannot be re-enabled. As a result, they are suitable for applications with simple control flow, such as *server programs*, but not for devices like *mode-based embedded devices* that frequently switch between modes, and transition back to earlier modes (Figure 1).

3.2 Our Approach: *RVDebloater*

RVDebloater proposes two solutions (i.e., S1 and S2) to address the limitations of existing debloating techniques.

S1: Addressing L1 and L4. We propose an adaptive debloating technique that specializes firmware code based on the device's requirements during its mode of execution (L1). This technique, detailed in §4.2, employs either a static or dynamic analysis to identify the required firmware code for each mode at the function-level (L4).

S2: Addressing L2, L3, and L5. We use a software-based technique that does not rely on hardware features to enforce firmware debloating based on the device's current execution mode throughout its execution lifetime (L5). Specifically, *RVDebloater* monitors mode switches and controls memory access by instrumenting the firmware's mode-switching functions and indirect control flow transfer instructions to restrict access to required firmware functions based on the device's current mode (L2 and L3).

3.3 Assumptions and Threat Model

Assumptions. We make the following two assumptions. First, the list of autopilot software modes (e.g., `TAKEOFF` and `AUTO`) and mode-switching functions responsible for changing modes at runtime are known to *RVDebloater* and provided as inputs to it. This information can be found in the firmware's source code or autopilot software's documentation (e.g., ArduPilot documentation [4, 10]). Although determining mode-switching function could be automated through dynamic analysis, we did not invest the effort to develop this capability, as it only needs to be performed once per firmware model (e.g., ArduPlane). It took us only a few minutes to identify this function in six different firmware models.

Second, similar to prior debloating techniques [31, 36, 45], we assume that *RVDebloater* has access to the firmware's source code. With that said, since *RVDebloater* operates on the LLVM IR [18], it can also work if the firmware is available in the LLVM IR format instead of source code.

Threat Model. We consider an adversary who is capable of performing remote attacks on the system without having root or physical access to it. Furthermore, we assume that the core system software is benign and developed by honest developers; however it may contain memory corruption vulnerabilities (e.g., buffer overflows). An adversary can exploit such memory corruption vulnerabilities, to hijack the firmware's control flow and call unneeded critical firmware functionalities via code-reuse attacks (e.g., through ROP or JOP gadgets). Hardware-level corruptions, physical attacks such as sensor spoofing, and side-channel attacks are outside the scope of this work. Our threat model is consistent with those of prior debloating techniques [31, 45].

4 Design: *RVDebloater*

This section presents the system design of *RVDebloater*, the first adaptive debloating technique for *mode-based embedded devices*. We start with an overview of *RVDebloater* and its key innovations (§4.1), before describing the different phases of the system in detail (§4.2, §4.3 and §4.4).

4.1 Overview of *RVDebloater*

Figure 2 shows *RVDebloater*'s workflow, which includes three main phases: (1) the analysis phase, (2) the guard insertion phase, and (3) the monitoring phase. Our approach aims to identify and dynamically specialize firmware code[3] running on mode-based embedded devices at the function-level granularity (i.e., firmware functions), depending on the device's execution mode. To accomplish this goal, we propose two key innovations. The first innovation is using either a static analysis or dynamic analysis to identify the firmware functions used in a specific mode (see S1 in §3.2). We leverage this method in the *analysis* phase.

The second main innovation of our technique is that, unlike previous debloating methods that rely on hardware components (MPU) [45], virtualization mechanisms (VMX) [31], or irreversible mechanisms (Seccomp-BPF) [36, 52], our approach uses a completely *software-based enforcement technique* (see S2 in §3.2). In particular, our *software-based enforcement technique* uses *static analysis* to instrument indirect control flow transfer instructions (e.g., indirect function call or function return instruction) and redirect the control flow to the *RVDebloater* monitor, which specializes firmware code for each mode using a mode-specific configuration file that includes the required functions and their addresses generated in the previous phase (i.e., *analysis* phase). This is achieved in two phases: (1) the *guard insertion* phase, which modifies the firmware code, and (2) the *monitoring* phase, which tracks the device's execution to check indirect control flow transfer instructions and specialize the firmware for each mode. As shown in Figure 2, the *analysis* and *guard insertion* phase are both performed offline, while the *monitoring* phase is performed online.

After generating the mode-based configuration file (*analysis* phase) and executing the instrumented firmware (*guard insertion* phase) on the device, the *RVDebloater* monitor is triggered when an indirect control flow transfer instruction is executed (*monitoring* phase). The monitor checks whether the jump and execution of the target address are allowed based on the configuration file. If the target is not present in the static output (*RVDebloater-static*) or in the dynamic output (*RVDebloater-dynamic*) (depending on which is used), the monitor can either activate safety mechanisms such as `Fail-Safe` mode or take other actions based on device maintainers' preferences (e.g., mechanisms they have developed). `Fail-Safe` mechanisms are commonly employed in various detection and recovery techniques [34, 45, 46, 51] to safely control the vehicle by returning it to the launch point or performing an emergency landing when an attack is detected or control is lost. ArduPilot provides 11 well-defined fail-safe mechanisms [6], and PX4 also includes a fail-safe landing [22]. Thus, we use fail-safe mechanisms in our evaluation.

[3] Embedded firmware consists of code from various software modules, such as libraries and applications, compiled into a single executable file.

We provide a detailed discussion of the process in the following sections (§4.2, §4.3 and §4.4). Note that, for simplicity, the examples are presented in the C language, even though *RVDebloater* operates on LLVM IR. Further, to describe the examples, we treat each line of C code (i.e., program statement) as a single instruction, even though, in practice, each of these would correspond to multiple LLVM instructions.

4.2 Analysis Phase

The first phase of the *RVDebloater* process is the offline analysis phase, which automatically identifies the *required functions* for each mode using either *static* or *dynamic* analysis. Since static analysis tends to overestimate the required functions for each mode, while dynamic analysis tends to underestimate them, *RVDebloater* leverages *both techniques*, giving device maintainers the flexibility to choose the most appropriate option based on their level of risk-tolerance.

Specifically, the offline analysis phase consists of either: (1) executing the device with different missions to dynamically identify the functions used in each mode at runtime (*RVDebloater-dynamic* §4.2.1), OR (2) generating a pruned call graph of the firmware to statically identify an overestimated set of required functions for each mode (*RVDebloater-static* §4.2.2). Subsequently, *RVDebloater-dynamic* uses the dynamic output to enforce firmware specialization based on the device's current mode, while *RVDebloater-static* uses the identified function lists for each mode from the static output.

4.2.1 *RVDebloater-dynamic*: Dynamic Analysis

To enforce debloating, *RVDebloater* leverages dynamic analysis to identify the required firmware functions for each mode (*RVDebloater-dynamic*). To do so, *RVDebloater* employs static instrumentation to generate a modified firmware version that records (1) mode switches, and (2) the firmware functions executed in each mode at runtime. Afterwards, *RVDebloater* runs the instrumented firmware on the device with various missions to determine the executed (*required*) firmware functions for each mode. The following outlines each step.

(1) Instrumentation. This step inserts trampolines[4] into the mode-switching function(s)[5] and firmware functions. *RVDebloater* provides an instrumentation component that automates this process. This component takes the firmware and mode-switching function(s) (§3.3) as input and outputs the instrumented firmware. Specifically, the instrumentation component statically analyzes the firmware to locate the given mode-switching function(s) and other firmware functions, then inserts the trampolines. To ensure that the mode has changed successfully, the component inserts a trampoline at

[4] A single instruction, like a jump or branch instruction, inserted into the program to redirect its original control flow.

[5] Although *RVDebloater* supports multiple mode-switching functions, in the tested firmware (e.g., ArduCopter, PXCopter), instrumenting a single mode-switching function was sufficient to monitor all mode changes.

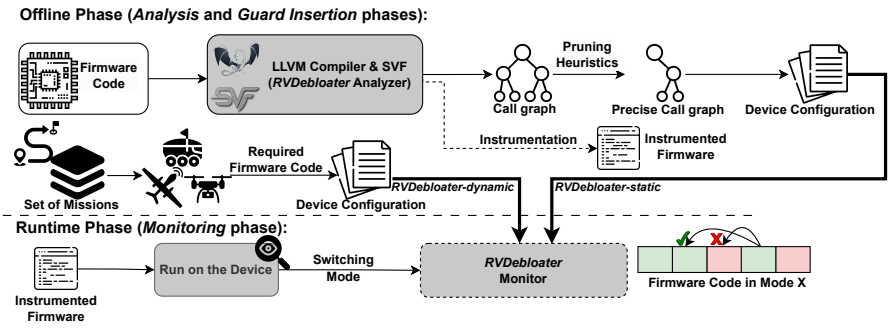


Figure 2: Overview of *RVDebloater*'s operation, including both the offline phase (top) and runtime phase (bottom).

the end of mode-switching function(s). If a mode switch fails, the firmware handles the error independently of *RVDebloater*.

```
1 bool Plane::set_mode_by_number(const Mode::
    Number new_mode_number, const ModeReason
    reason){
2     Mode *new_mode = plane.mode_from_mode_num(
    new_mode_number);
3     if (new_mode == nullptr) {
4         notify_no_such_mode(new_mode_number);
5         return false;
6     }
7     Trampoline
8     return set_mode(*new_mode, reason); }
```

Listing 3: Output of the instrumentation step for the mode-switching function in ArduPlane (ArduPilot).

Listing 3 shows an example of the mode-switching function in ArduPlane. This function first (line 2 in Listing 3) retrieves the name of the new mode (e.g., `CIRCLE` or `RTL`) by passing the input number (`new_mode_number`). Then, it checks the returned mode name to validate whether the new mode exists (lines 3 to 6). Finally, it calls a function to switch the current mode to the new mode (line 8). As seen in Listing 3, the component has inserted a trampoline at the end of the function (line 7) after the mode change has been successfully validated (lines 3 to 6). When the inserted trampoline is triggered at runtime, it passes the current and new mode to *RVDebloater*'s profiler, records the executed functions in the current mode, and changes the profiler's mode to the new mode.

Further, the component inserts trampolines at the beginning of all firmware functions (except the mode-switching function) to ensure that every function executed in the current mode triggers *RVDebloater*'s profiler. Algorithm 1 demonstrates the workflow of the instrumentation step. For the mode-switching function, the component inserts a trampoline at the end (lines 2 to 5). For all other functions, it adds trampolines to the beginning (lines 6 to 9).

(2) Dynamic Profiling. In the next step, *RVDebloater* runs the device with the instrumented firmware (e.g., ArduPlane or ArduCopter) across different missions and scenarios (e.g., with and without obstacles) to identify the firmware functions

Algorithm 1: Instrumentation for Profiling

```
1  instrumentationComponent (Functions)
       inputs : Firmware Functions and Mode-switching Function Set
                (modeFuncs)
       output : Instrumented Firmware
2      if Function ∈ modeFuncs then
3          if End of Function then
4              location := findLocation();
5              insertTrampoline(location);
6      else
7          if Function Entrance then
8              location := findLocation();
9              insertTrampoline(location);
10     return instrumentedFirmware;
```

for each mode at runtime by capturing functions with *RVDebloater*'s profiler. The instrumented firmware ensures that each function for the current mode is logged at most once when the device switches to a new mode. We assume that *RVDebloater* runs this step (dynamic profiling) in a benign environment, free from any security attacks.

4.2.2 *RVDebloater-static*: Static Analysis

RVDebloater also statically analyzes the firmware to construct its call graph. Since the firmware is written in C/C++, it often relies on indirect function calls via function pointers. Statically resolving the targets of these calls is both challenging and critical for accurately identifying the functions required by each mode. To address this, *RVDebloater* employs Andersen's points-to analysis [32], a static technique for resolving pointer targets in C/C++ programs. This analysis can have different sensitivities depending on how objects in memory are modeled: field sensitivity, path sensitivity, and context sensitivity. Higher levels of sensitivity yield more accurate results but also increase the analysis time and the implementation complexity. For instance, field sensitivity distinguishes and models each field of a structured variable (e.g., a `struct`) separately, while context sensitivity distinguishes between different call sites of a function, meaning that when the same

function is called from different places, each call gets its own context and is analyzed independently of the others. While a detailed discussion of these topics is beyond the scope of this paper, more details about these sensitivities can be found in previous work [36, 39]. Specifically, *RVDebloater* leverages SVF [56], which implements Andersen's analysis with field sensitivity but without path or context sensitivity.

As a result, SVF tends to over-approximate the call graph, as it associates indirect function calls with a large number of potential targets—only a small subset of which is actually executed, while the rest are spurious. Although this over-approximation increases the number of functions identified per mode, introducing unneeded functions that increase the attack surface and could be exploited by attackers to subvert the system, the approach is sound in determining which functions are not required (unreachable) for each mode. However, to improve precision, *RVDebloater* applies two pruning heuristics, inspired by the TSCP technique [36], to remove spurious targets caused by the lack of path and context sensitivity and generate a more accurate call graph. We modified these heuristics to support embedded firmware and refine call graphs further. The call graph is generated once per firmware model (e.g., ArduCopter), independent of the number of modes. The two heuristics are as follows. More details are §A in the Appendix.

(1) Signature-based Pruning Heuristic. SVF only prunes the spurious targets by checking the number of arguments, whereas *RVDebloater* uses two additional filtering methods: considering argument types and matching return types when resolving indirect call targets.

(2) Address-based Pruning Heuristic. In contrast to Andersen's algorithm, which assumes that all functions are reachable from the entry point, *RVDebloater* identifies and removes edges that point to functions whose addresses are not reachable from the entry point.

After refining the call graph, *RVDebloater* identifies the entry functions of each mode to determine their reachable (required) functions. To do so, it employs an instrumentation component similar to the one used in dynamic analysis (§4.2.1), which instruments both the firmware functions and the mode-switching function(s). *RVDebloater* then executes the device through a mission that includes all modes intended for monitoring and control at runtime by *RVDebloater-static*. During execution, when the device switches modes, *RVDebloater* tracks the function(s) invoked by the mode-switching function to initialize the parameters required for the new mode and marks them as the entry function(s) of that mode.

RVDebloater also performs mode-name pattern matching to confirm the validity of the identified entry function(s). In particular, we observed that these firmware models (e.g., ArduCopter or ArduPlane) have entry function(s) responsible for managing mode-specific parameters, and their names typically include the mode name itself. Therefore, *RVDebloater* uses the list of mode names provided as input (§3.3) to verify and confirm the correctness of these functions. We found that all identified entry functions were correct. Note that this pattern can be adapted by developers to support other firmware. However, in all the firmware models evaluated in this paper, this pattern was sufficient. Finally, *RVDebloater* uses the generated call graph to identify the reachable and unreachable functions for each mode, starting from its entry function(s).

Both *dynamic* and *static* analyses only need to be performed *once* for each firmware model (e.g., ArduPlane or ArduCopter) to identify and store the functions required for each mode at runtime. These identified functions are then used during the runtime (device execution) monitoring phase to enforce debloating by checking whether they are permitted in the current mode of the device.

4.3 Guard Insertion Phase

The next phase of the *RVDebloater* process is the *guard insertion* phase, which is performed offline and only once per firmware. The *RVDebloater monitoring* phase is designed to: (1) track mode switches, (2) load the appropriate list of identified functions for the new mode when a mode switch occurs, and (3) monitor indirect control flow transfer instructions and ensure they are allowed; otherwise, *RVDebloater* logs the violation and switches to `Fail-Safe` mode. The *RVDebloater* process has two main phases: **guard insertion** and **monitoring** (§4.4).

Similar to the previous instrumentation step for offline analysis phase, *RVDebloater* provides a component that automatically identifies mode-switching function(s) and inserts a trampoline at the end of each. However, in this instrumentation step, *RVDebloater* replaces *all* indirect control flow transfer instructions in the firmware code with calls to the *RVDebloater* monitor. This allows *RVDebloater* to observe all function calls and returns, and verify them against the list of allowed functions identified in the previous phase.

This component takes a list of mode-switching function(s) and the firmware code as input (§3.3), and outputs an instrumented firmware where trampolines are statically added to the mode-switching functions, and indirect control flow transfer instructions are replaced accordingly.

Algorithm 2 describes the workflow of this instrumentation component for monitoring. Similar to the prior instrumentation component (Algorithm 1), this component inserts trampolines at the end of mode-switching functions (lines 2 to 5). For other firmware functions, it iterates over their instructions, and replaces each indirect control flow transfer instruction (e.g., indirect call site or return instruction) with an instruction (i.e., `call "monitor_fn"`) that redirects the control to *RVDebloater*'s monitor (lines 6 to 15). *RVDebloater*'s monitor checks the call target before calling the intended target address directly. Section 4.4 provides details. Note that *RVDebloater* restricts enforcement to indirect call sites and return instructions; direct calls are inherently safe, as their targets are statically fixed in the binary, and cannot be redirected

Algorithm 2: Instrumentation for Monitoring

```
1  instrumentationComponent (Functions)
       inputs : Firmware Functions (Insts = inst_1 ... inst_n) and
                Mode-switching Function Set (modeFuncs)
       output : Instrumented Firmware
2      if Function ∈ modeFuncs then
3          if End of Function then
4              location := findLocation();
5              insertTrampoline(location);
6      else
7          foreach instruction inst_i ∈ Insts do
8              if inst_i.type is a indirect_cotrol_flow_instruction
                 then
9                  location := findLocation();
10                 replaceInst(location);
11             else if inst_i.type is a return_instruction then
12                 location := findLocation();
13                 replaceInst(location);
14             else
15                 continue;
16     return instrumentedFirmware;
```

through memory corruption attacks (§2.1).

4.4 Monitoring Phase

The final phase of the *RVDebloater* process is the *monitoring phase*, which operates after the *analysis* and *guard insertion* phases have determined the required functions for each mode (either static or dynamic output) and instrumented the firmware. The purpose of the monitoring phase is to *monitor* the device and *dynamically debloat* (specialize) the firmware functions for each mode based on the identified lists at runtime, such as during RV missions. To do so, *RVDebloater* uses the instrumented firmware from the previous phase (§4.3) and the list of functions identified for each mode, which is generated during the analysis phase (§4.2.2 or §4.2.1). The procedure of monitoring call sites and returns is illustrated in Figure 3. The figure consists of five steps (labelled (1) to (5)) and four phases: the process of switching modes in the RV, an access control table for each mode, an example of Mode A at runtime, and the *RVDebloater* monitor component.

Once a mode switch occurs, the mode-switching function jumps to *RVDebloater*'s monitor (step (1)) to update the access control tables for the new mode (step (2)). For example, as shown in Figure 3 (steps (1) and (2)), when the device mode changes from *Mode B* to *Mode A*, the access control table containing the required functions for *Mode A* is loaded from the generated configuration file.

As mentioned, *RVDebloater* rewrites call instructions to redirect them to its monitor (step (3) in Figure 3). When Function X calls Function Z, control is transferred to the *RVDebloater* monitor, which checks whether the call is permitted using the loaded access control table for the current mode (based on either the static or dynamic output).

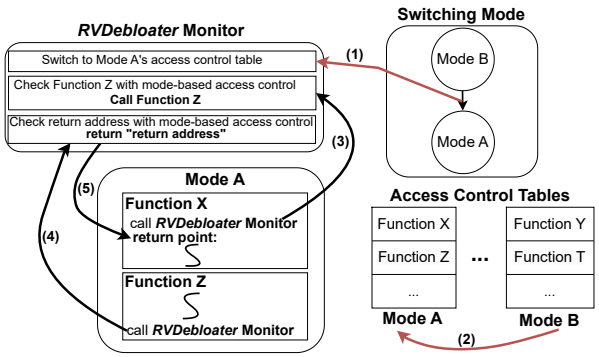


Figure 3: *RVDebloater* Runtime System.

If calling Function Z is permitted, the monitor saves the return address (e.g., the "return point" in Function X, as shown in Figure 3) and then invokes the function. Otherwise, if the function call is not authorized based on the static output for *RVDebloater-static* or the dynamic output for *RVDebloater-dynamic*, *RVDebloater* triggers an alert and switches the device to `Fail-Safe` mode, as the call is unauthorized according to the respective analysis.

After finishing the execution of Function Z, the return instruction jumps back to the monitor (step (4) in Figure 3), which verifies the stored return address. If the address matches the expected return address, control flow returns to Function X (step (5) in Figure 3). Otherwise, the monitor performs the same approach as for an unauthorized function call (i.e., switching to `Fail-Safe` mode).

5 Implementation

We implemented *RVDebloater*'s analysis phase using two LLVM module passes, which can be executed on firmware containing different functions [29]. Furthermore, we developed two functions that are linked to the firmware to support the instrumentation component and *RVDebloater*'s profiler. As mentioned in §4.2, this phase takes the mode-switching function(s) and firmware code as input and generates an instrumented firmware along with a list of required functions for each mode (configuration file). We also utilized SVF [27] to generate the firmware's call graph. However, we modified and added two LLVM passes to refine the generated call graph (i.e., signature-based and address-based pruning heuristics).

Further, the guard insertion phase is implemented as two LLVM module passes, and is executed on the firmware. This generates an instrumented firmware that tracks mode changes and controls function execution based on the identified required functions. Finally, the runtime monitoring phase monitors mode switching and indirect control flow transfer instructions to *dynamically* specialize the firmware code for each mode according to the configuration file. The implementation

details are discussed in §B in the Appendix.

6 Evaluation

This section describes the evaluation of different aspects of *RVDebloater* including the effectiveness and efficiency of the system. Effectiveness is the ability of *RVDebloater* to specialize the firmware code for each mode at runtime and reduce the attack surface, while the efficiency is the runtime and memory overheads of *RVDebloater*.

6.1 Experimental Setup

We evaluated *RVDebloater* on four simulated RV systems using either ArduPilot [9] or PX4 [23]—two popular open-source autopilot platforms—to test whether *RVDebloater* can work across different platforms without modification: (1) ArduPilot's quadcopter [5] (ArduCopter), (2) ArduPilot's quadplane [7] (ArduPlane), (3) ArduPilot's ground rover [8] (ArduRover), and (4) PX4's quadcopter [21] (PXCopter). We used Gazebo [15], QGroundControl [24], and APM SITL (Software-In-The-Loop) [9] platforms for simulations.

We also used two real RVs, shown to the right: **(Left)** Aion R1 ground rover (Aion rover) [2] and **(Right)** DIY drone (Pixhawk drone) [47].



Both are commodity RV systems based on the Pixhawk [19]. Pixhawk is a platform built on an ARM Cortex architecture that integrates ArduPilot or PX4 firmware, a flight management unit (FMU), memory, sensors, and I/O interfaces.

To generate the firmware call graph using the *RVDebloater*'s modified SVF, statically extract the required (reachable) firmware functions for each mode (*analysis* phase), and perform instrumentation during both the *analysis* and *guard insertion* phases, we used a desktop machine equipped with an Intel Xeon E-224 CPU (4 cores, 3.4 GHz) running Ubuntu 22.04 (64-bit). After identifying the required functions for each mode and generating the instrumented firmware, we tested *RVDebloater* using simulations and real devices.

Note that the Aion rover and the Pixhawk drone use the same platforms as ArduRover in ArduPilot and PXCopter in PX4, respectively; therefore, the evaluation results for ArduRover and the Aion rover, and for PXCopter and the Pixhawk drone, are similar. Thus, we do not report them separately, except when evaluating the efficiency in §6.3.

RV Missions: To evaluate the effectiveness of *RVDebloater*, we execute a diverse set of 40 missions (for each RV) with varying durations and distances, randomly selected to cover a range of scenarios. Each mission consists of a navigation path that begins at a starting location, passes through several intermediate waypoints, and ends at a specific destination. These missions emulate a range of real-world RV missions, including straight-line paths for last-mile delivery drones [28], polygonal paths for rovers in warehouse management [11], and both polygonal and circular paths for agriculture and surveillance [14]. The types of missions used for each RV are summarized in Table 1.

During each mission, the RV switches between different modes, such as `CIRCLE` and `LOITER` for ArduCopter and ArduRover, or `QHOVER` and `QLOITER` for ArduPlane. The main RVs' modes are outlined in the extended version of this paper available on arXiv [55]. Although our mission set includes other modes, this section presents the results for the main RV modes that we observed are used in real-world missions (Tables 2 and 3).

Table 1: Mission sets for each RV firmware type used to evaluate *RVDebloater*. In the "Missions" column, we use SL: straight line, MW: multiple waypoints, HFE: hover at fixed elevation, CP: circular paths, PP: polygonal path.

RV (Firmware) Type	Missions					
	Total	SL	MW	HFE	PP	CP
ArduCopter	40	10	12	5	5	8
ArduRover	40	12	13	-	7	8
ArduPlane	40	11	13	3	3	10
PXCopter	40	10	12	5	5	8

Metrics: We used four metrics to evaluate *RVDebloater*, namely (1) ***missed functions*** are those that *RVDebloater*'s dynamic profiler fails to identify during profiling missions using dynamic analysis. (2) ***false positive rate (FPR)*** is the percentage of mission failures caused by the absence of required functions in the configuration file (either static or dynamic results) for the current mode, despite the RV needing them for the mission's success. (3) ***false negative rate (FNR)*** is the percentage of functions that *RVDebloater* allows during missions, even though these functions are not required by the RV's current mode. (4) ***precision*** of the call graph, which refers to the reduction in the number of spurious call graph edges achieved by applying various pruning heuristics. This metric is similar to what prior work used [36, 52].

6.2 Effectiveness

Monitoring Phase. *RVDebloater* reduces the attack surface by restricting each mode's access to only the *required* firmware functions at runtime. In the following, we evaluate the effectiveness of *RVDebloater*'s profiler in identifying the required firmware functions for each mode, and measure the resulting attack surface reduction achieved by *RVDebloater*.

***RVDebloater-dynamic* Profiler.** To evaluate the *RVDebloater* dynamic analysis output, we measured the number of missed functions across varying numbers of profiling and testing missions on different RV types, and determined how many profiling missions are needed to identify all the *required* functions for each mode without missing any. To do so, we used the RV missions discussed in §6.1 to dynamically determine

the *required* firmware functions for each mode and RV. Specifically, we first randomly chose two out of the 20 test missions as profiling missions for *dynamic analysis* for each RV. We then used the remaining 18 missions to test the identified *required* functions for each mode at runtime. We further evaluated the dynamic analysis output by gradually increasing the number of profiling missions, adding 2 missions each time to the profiling set from the testing set (e.g., 4 profiling and 16 testing missions), all the way to 20 missions.

The results are shown in Figure 4. As can be seen, all RVs converged to *zero missed functions* after profiling within only 10 missions (i.e., 10 profiling and 10 testing missions). Although the *RVDebloater* profiler identified all the required functions for each mode using 10 profiling missions, we continued the dynamic analysis evaluation by incrementally adding two new missions at a time to the profiling mission set and testing on the remaining missions (e.g., 12 profiling and 8 testing missions) to ensure that no missed functions appeared after convergence. The results confirmed that the number of missed functions for each RV remained at zero.

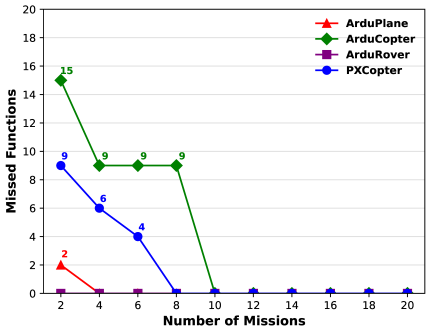


Figure 4: Number of missed functions across different missions for various platforms during dynamic analysis with *RVDebloater*. All platforms converge to 0 missed functions after 10 missions.

To further confirm this result, we evaluated the *RVDebloater* profiler on a totally different set of 20 different missions. The results, shown in Figure 4 (20 missions), demonstrate that *RVDebloater*'s dynamic profiler did not miss any required functions. Therefore, all *required* functions for each mode can be identified within 10 missions in all tested RVs.

Although the number of profiling missions used to identify all required functions for each mode at runtime is small, the *RVDebloater-dynamic* profiler may still miss some functions and produce false positives. To address this, *RVDebloater-static* can be used, which also includes the missing functions. However, the device maintainer may choose to rely on the dynamic analysis results (*RVDebloater-dynamic*) to achieve greater specialization compared to *RVDebloater-static*, which uses static analysis technique (§4.2 and §4.4).

Attack Surface Reduction. To evaluate the attack surface reduction, we used *RVDebloater*'s dynamic and static analysis to identify the required firmware functions for each mode. As we observed, after profiling 10 missions, the dynamic analysis converged to zero missing functions, and the number of functions remained unchanged with additional profiling. Hence, Tables 2 and 3 (*Dynamic Analysis* row) shows the number of firmware functions required at runtime for each mode, as identified by *RVDebloater*'s profiler with 10 missions. Furthermore, the *Static Analysis* row in Tables 2 and 3 presents the number of *reachable* functions for each mode, as determined by static analysis on the generated call graph. As discussed earlier (§4.2 and §4.4), *RVDebloater-static* and *RVDebloater-dynamic* use the outputs of static and dynamic analysis, respectively, to specialize the firmware code.

The results reveal that, on average, 85% and 41% of firmware functions are not required according to dynamic and static analysis, respectively. In particular, when *RVDebloater* uses static analysis outputs, it reduces the firmware functions by an average of 41% (Reduction *RVDebloater-static* in Tables 2 and 3), while achieving a 0% **FPR** and an average **FNR** of 44%, as this technique tends to overestimate the functions required for each mode (§4.2). Attempts to access debloated functions lead to *RVDebloater-static* activating `Fail-Safe` mode, and raising an alarm to the device maintainer (§4.4).

On the other hand, when the device maintainer uses *RVDebloater-dynamic*, which uses dynamic analysis—generated by profiling 10 missions—no missions fail, resulting in a 0% **FPR** and an 85% reduction in the attack surface (Reduction *RVDebloater-dynamic* in Tables 2 and 3). As in the case of *RVDebloater-static*, any attempt to access the debloated functions leads to *RVDebloater-dynamic* switching the RV to `Fail-Safe` mode, thereby restricting the attacker's access to only 15% of the firmware's functions with *RVDebloater-dynamic*. Furthermore, all functions allowed by *RVDebloater-dynamic* are actually required during execution (identified by profiling different missions), resulting in a 0% **FNR** for *RVDebloater-dynamic*.

From the results (Table 2 and 3), we can infer three main trends. First, each mode requires only a subset of the firmware functions rather than the entire firmware. For instance, when the RV is in `AUTO` mode, it requires approximately 2800 firmware functions, whereas in `RTL` mode, it requires a different set of around 3000 firmware functions (addressing L1, L2, and L4 in §3.1). Second, because RVs switch between different modes, including returning to earlier modes, they need an adaptive debloating technique that is reversible, in contrast to previous debloating techniques [36, 52] that use irreversible mechanisms to disable unwanted features (addressing L5 in §3.1). This is why we need a technique like *RVDebloater*. Third, *RVDebloater-dynamic* can debloat more functions, with only a few missions (fewer than 10), without introducing either *false negatives* or *false positives*. This is because embedded devices typically require a fixed set of functions (functionalities) for each mode, which can be

Table 2: Summary of firmware functions reduction (i.e., attack surface reduction through debloating) across different modes in ArduCopter, ArduPlane, and ArduRover (sub-platforms of ArduPilot). Reduction represents the percentage decrease in the number of firmware functions accessible to each mode when using *RVDebloater* with dynamic (*RVDebloater-dynamic*) or static analysis (*RVDebloater-static*), compared to the baseline firmware where all functions remain accessible.

Technique \ RV Type (modes)	ArduCopter						ArduPlane							ArduRover					
	AUTO	CIRCLE	STABILIZE	GUIDED	RTL	LOITER	MANUAL	AUTO	CIRCLE	GUIDED	QLOITER	RTL	QHOVER	AUTO	CIRCLE	MANUAL	RTL	GUIDED	LOITER
Dynamic Analysis	2869	3106	7054	3271	3153	2757	6891	2823	2756	2692	3120	2692	2700	2513	2435	6246	2433	2647	2255
Static Analysis	13520	13450	13447	13469	13470	13452	13235	13253	13232	13238	13252	13239	13238	12637	12558	12549	12566	12583	12550
w/o *RVDebloater*	22653	22653	22653	22653	22653	22653	22514	22514	22514	22514	22514	22514	22514	20990	20990	20990	20990	20990	20990
Reduction: ***RVDebloater-dynamic***	87.33%	86.28%	68.86%	85.56%	86.08%	87.83%	69.39%	87.46%	88.75%	88.04%	86.14%	88.04%	88%	88.02%	88.40%	70.24%	88.40%	87.39%	89.26%
RVDebloater-static	40.31%	40.62%	40.64%	40.54%	40.53%	40.61%	41.21%	41.13%	41.22%	41.20%	41.14%	41.19%	41.20%	39.80%	40.17%	40.21%	40.13%	40.05%	40.20%

Table 3: Summary of firmware functions reduction in PX-Copter across different modes (continuation of Table 2).

Technique	TAKEOFF	MISSION	LOITER	RTL
Dynamic Analysis	4320	2765	2660	2853
Static Analysis	12244	12238	12208	12227
w/o *RVDebloater*	20805	20805	20805	20805
Reduction: ***RVDebloater-dynamic***	79.23%	86.71%	87.21%	86.29%
RVDebloater-static	41.15%	41.18%	41.32%	41.23%

identified with just a few missions.

As discussed in §4.2, dynamic analysis may underestimate the required firmware functions — this is a known limitation of dynamic analysis techniques. Since the exact required functions per mode cannot be determined a priori, code coverage metrics cannot be used here, as they quantify execution depth rather than whether all functions required by each mode were captured. Nevertheless, for cases where rare or unanticipated conditions cause a function to be missed, *RVDebloater-dynamic* will trigger a transition to `Fail-Safe` mode, returning the device to its launch location or taking other actions based on the device maintainers' preferences (§4.1). Notably, in our evaluation across diverse missions (§6.1), *RVDebloater-dynamic* did not miss any required functions after converging to zero, and thereby never triggered `Fail-Safe` mode.

In contrast, *RVDebloater-static* eliminates fewer unneeded firmware functions for each mode, while employing a more careful and conservative debloating approach. Thus, the choice between these two approaches involves a trade-off: *static analysis* provides more cautious restrictions with lower protection, whereas *dynamic analysis* offers higher protection but has potential false positives.

Attack Evaluation. According to the list of most dangerous software weaknesses from CWE [1], out-of-bounds write is one of the most prevalent vulnerabilities. Thus, to evaluate *RVDebloater*'s effectiveness (*RVDebloater-dynamic*, *RVDebloater-static*) in restricting firmware access, we injected a buffer-overflow vulnerability—inspired by real-world vulnerabilities [13, 26]—into a function that processes user input, following prior work [45, 46]. An attacker who exploits this vulnerability can hijack control flow and perform a code-reuse attack. In our evaluation, we exploited this vulnerability to hijack the control flow and then performed three attacks. Each attack executed sensitive firmware functions, such as `disarm_motors` (§3.1), while the device was not in the correct mode. We discuss the three attacks in detail below.

(A1) Disarming RVs. ArduCopter provides a safety-critical function, `disarm_motors` (Listing 2), which disarms the motors during benign tasks such as flipping the device upright in `Turtle` mode (§3.1). However, after hijacking the control flow, our attack reused the `disarm_motors` function during a mission (e.g., while the device is in `GUIDED` mode), causing the device to crash. The attack succeeded because the firmware allowed unrestricted access to any function, including `disarm_motors`, regardless of the current mode.

When *RVDebloater* was deployed, it detected the attack and switched the device mode to `Fail-Safe`, allowing the device to land safely or return to the launch point (§4.1), because `disarm_motors` is restricted and is not accessible from other modes, such as `GUIDED`. The *disarming* functionality also exists in other platforms, such as ArduPlane and ArduRover, and is used during specific events like parachute release or crash detection. An attacker can perform the same attack to disarm the motors and cause physical damage. Again, *RVDebloater* detected such attacks because disarming should not be allowed in modes such as `QHOVER`.

(A2) Stopping Motors. ArduCopter and ArduPlane provide a function called `output_min`, which stops all motors and cuts engine power when specific events occur, such as when the device has landed. Similar to the previous attack, our stopping motors attack hijacks the control flow of the victim device (firmware) and calls `output_min` to stop the motors at an incorrect time (e.g., during mission while the device is in `GUIDED`), causing the device to crash. In our experiments, we confirmed that an attacker can easily disrupt a mission and damage the device by invoking this function. This attack was also detected and prevented by *RVDebloater* by switching the RV to the `Fail-Safe` mode. This is because `output_min` is not reachable from modes such as `GUIDED` or `QHOVER`.

(A3) Disarming Aion Rover. This attack case demonstrates the effectiveness of *RVDebloater* on a real-world device, the Aion Rover, discussed in §6.1. Similar to the previous attack cases, a buffer overflow vulnerability is intentionally inserted into the function that processes user input, inspired by previous work [46]. During the mission, this vulnerability is triggered, allowing the attacker to hijack the control flow and invoke a safety-critical function, `disarm`, from an incorrect mode (i.e., `MANUAL`). As a result, the motors are disarmed, and the rover stops abruptly, which may cause collisions with other nearby rovers or other objects.



(a) Boot Firmware (b) Runtime

Figure 5: Real-world attack (A3) on Aion Rover.

Again, when *RVDebloater* was deployed, it successfully detected this attack because the `disarm` function is not included in the list of required functions for modes such as `MANUAL`. As before, it switched the rover to the `Fail-Safe` mode upon detecting the attack. Figure 5 shows the setup for this attack on the Aion Rover in our lab.

Analysis Phase. As discussed in §4.2.2 and §B.1, *RVDebloater* uses a modified version of SVF along with two pruning heuristics—signature-based and address-based—to generate and refine the firmware call graph, identifying the *reachable* (*required*) functions for each mode (*RVDebloater-static*). To assess the effectiveness of these heuristics in reducing spurious edges in the call graph, we evaluated them on ArduCopter, ArduPlane, ArduRover, and PXCopter. As shown in Table 4, the number of edges in the original call graph generated by SVF ranges from 104K to 193K. By applying our pruning heuristics, we reduced the number of spurious edges in the call graph, achieving a precision of approximately 36% (66.8K), 45% (67.1K), 48% (60.1K), and 49% (98.5K) for ArduCopter, ArduPlane, ArduRover and PXCopter, respectively. Thus, our enhancement enables *RVDebloater-static* to more effectively restrict the functions accessible in each mode, ensuring that it only identifies the functions actually required by each mode.

Table 4: Precision after applying different heuristics. The third column shows the number of edges after applying the signature-based heuristic, and the fourth column shows the number after both heuristics.

RV (Firmware) Type	SVF	+ Signature-based	+ Addressed-based	Precision
ArduCopter	104K	87.4K	66.8K	36%
ArduRover	115.3K	80.3K	60.1K	48%
ArduPlane	122K	86.8K	67.1K	45%
PXCopter	193K	130.2K	98.5K	49%

The complexity of the input (i.e., firmware) affects the duration of the above process. On our platform, generating the call graph for each RV type (firmware) took approximately 3 to 4 hours on average, based on 10 executions of the SVF. However, as mentioned earlier, this step is performed offline, and is done only once per firmware.

6.3 Efficiency

Performance and Power Overhead. As discussed in §4.1, the only component of *RVDebloater* that incurs overhead on the RV is the monitoring component, which monitors the mode-switching function and checks indirect control flow transfer instructions to specialize the firmware code for each mode. Therefore, since both *RVDebloater-static* and *RVDebloater-dynamic* use this component, their results are similar. To evaluate the performance overhead, we used the real RVs mentioned in §5, i.e., the Aion rover and Pixhawk drone, as performance overheads in simulated RVs are dependent on the computing platform used for simulation. We used the scheduler in the RV autopilot platforms (ArduPilot and PX4) to track total CPU time and measured *RVDebloater*'s overhead by analyzing the additional CPU time incurred when *RVDebloater* was deployed.

We found that *RVDebloater* incurs modest performance overheads of 3.5% on the Aion rover and 4.3% on the Pixhawk drone, with an average of 3.9%. We also found that there was *no increase* in the overall mission time of the RVs with *RVDebloater* deployed for the missions discussed in §6.1.

Furthermore, we estimated the power consumption of *RVDebloater* running on those RVs, which are powered by a 5000 mAH battery. Since the processor typically accounts for 12% of the total power consumption [20] and *RVDebloater* adds an average overhead of 3.9%, we estimate that the RV's total power and energy consumption increases by approximately 0.47% due to *RVDebloater*, which is negligible.

Memory Overhead. To measure *RVDebloater*'s memory overhead, only the configuration files generated during the *analysis* phase and the instrumented firmware produced during the *guard insertion* phase are stored on the device. Thus, we focus on the overhead introduced by these two phases, as they are the only ones that incur overhead to memory usage on the device. As discussed in §B.2 and §B.3, *RVDebloater* builds the firmware using the LLVM compiler toolchain (i.e., gclang, gclang++, and clang). This process involves extracting the bitcode from the original firmware file, and then generating a new executable file from the instrumented bitcode for deployment on the device.

Table 5: Firmware size with and without using *RVDebloater* guard insertion phase.

Technique \ RV Type	ArduCopter	ArduPlane	ArduRover	PXCopter
w/o *RVDebloater*	6.2 MB	6.1 MB	5.7 MB	6.2 MB
RVDebloater	6.5 MB	6.4 MB	5.9 MB	6.4 MB
Overhead	**0.3 MB (4.8%)**	**0.3 MB (4.9%)**	**0.2 MB (3.5%)**	**0.2 MB (3.2%)**

We find that the instrumented firmware introduces an average memory overhead of 0.25 MB across different platform types, corresponding to approximately a 4% increase over the original firmware. A summary of firmware sizes is presented in Table 5. Further, the mode-based configuration files generated by *RVDebloater-static* and *RVDebloater-*

dynamic (§4.2) incur an average memory overhead of 0.65 MB and 0.17 MB, respectively. These files are not included in the firmware and thus do not contribute to the 4% size increase. Overall, the total memory overhead (i.e., instrumented firmware+configuration files) is negligible ($< 4\%$), as the memory available on the real RVs is over 20 MB.

Note that, like other debloating (specialization) techniques [31,36,45] that retain the firmware code while restricting access, *RVDebloater* also does not reduce the memory usage. As discussed earlier, mode-based embedded devices require the entire firmware, but only the functions relevant to the current mode should be accessible in it.

7 Related Work

Fixed debloating techniques. fall into two types based on whether they *retain* or *remove* unneeded features, such as security-critical system calls or hardware features.

1. Retaining Features. Abubakar et al. [31] proposed SHARD, which leverages context-aware hardening to specialize the Linux kernel at the application and system call levels. SHARD uses both static and dynamic analysis to identify the required kernel code for executing the system call invoked by an application. Afterwards, at runtime, SHARD ensures that only the identified kernel code is allowed to execute when the application calls the same system call. However, SHARD switches all kernel code pages for each system call invocation, making it unsuitable for resource-constrained embedded devices like RVs.

Minion [45] proposed a thread-level memory isolation technique for real-time microcontroller systems (MCS) that isolates the memory space for each process through offline static analysis of firmware and runtime memory access control using the Memory Protection Unit (MPU). However, Minion remains vulnerable to control flow hijacking attacks, as attackers can exploit memory corruption vulnerabilities in the shared memory between processes to hijack control flow and construct ROP or JOP chains. Further, since Minion uses the MPU to enforce memory isolation, it applies clustering techniques to group functions. However, this may allow processes to access functions that they do not require (§3.1).

Gu et al. proposed FACE-CHANGE [37], a kernel debloating technique that employs dynamic analysis to generate kernel profiles for each application, identifying the required kernel code at the application level. However, FACE-CHANGE's reliance on runtime profiling leads to incomplete identification of required kernel code (i.e., customized kernel view), limiting its analysis to the subset of the program executed during profiling. Furthermore, FACE-CHANGE suffers from coarse specialization granularity (i.e., single kernel view for the entire application), and hence attackers can still hijack the control flow of the sensitive programs (e.g., flight control program in RVs) and manipulate the devices' behavior.

2. Removing Features. Hu et al. proposed IRQDebloat [41], a debloating technique that leverages dynamic analysis to identify interrupt handlers and disable undesired hardware functionalities by rewriting their handlers with a no-op function in the target firmware. Hu et al. proposed Hacksaw [42], a kernel specialization technique that uses a list of hardware components attached to the target machine, called the device inventory, to remove kernel device drivers related to unnecessary hardware components.

Wu et al. proposed LightBlue [58], a profile-aware debloating technique that uses static analysis to identify the Bluetooth profiles (e.g., Advanced Audio Distribution Profile (A2DP)) utilized by an application (e.g., an Android app). Then, it removes unused code from the Bluetooth stack, including Bluetooth host code and firmware, thereby reducing the attack surface. However, LightBlue is limited to debloating of Bluetooth stacks and does not consider other features, such as firmware functionalities (e.g., `disarm_motors` in RVs).

Jahanshahi et al. proposed Minimalist [44], a semi-automated debloating technique for PHP web applications (e.g., WordPress and PhpMyAdmin) that uses static analysis to generate a call graph for these applications. Minimalist then removes unreachable and unused code based on the generated call graph and recorded access-log files, which include users' interactions with the web applications. However, Minimalist has two limitations as follows. First, users need to interact with the web server for each application to generate the access-log files, which is both time-consuming and potentially incomplete. Second, since Minimalist relies on user requests, it may remove essential parts of the code that are not captured in the log files, potentially causing system crashes.

All of these debloating techniques, which *remove unwanted features* such as hardware functionalities or kernel code, are irreversible, meaning that once unneeded features are removed, they cannot be restored. However, embedded devices may require different functionalities during various phases of execution. Therefore, these techniques are only suitable when certain features are guaranteed to remain unused during runtime; hence, our work is orthogonal to this line of work.

Adaptive Debloating Techniques. As mentioned in §1, *fixed debloating techniques* do not consider changes in application requirements at runtime, and as a result, they cannot disable critical features, such as the `fork` and `execve` system calls, which are necessary during the initial stages of an application. To address this issue, Ghavamnia et al. [36] and Rajagopalan et al. [52] proposed adaptive system call specialization techniques for server applications, which are discussed in detail,in Sections §1 and §3.1.

Unlike existing adaptive debloating techniques that rely on irreversible mechanisms to disable system calls, *RVDebloater* dynamically specializes firmware code at runtime for each mode. This is particularly important for mode-based embedded devices, such as RVs, where firmware functions may need to be re-enabled depending on the active mode (§6.2). Fur-

ther, while prior techniques focus on system call disabling, *RVDebloater* performs function-level specialization. This is important for embedded systems such as RVs.

Other Techniques. In addition to the two main categories of debloating techniques discussed above, several complementary security techniques have been proposed. (i) Compartmentalization techniques [35] partition a program into isolated compartments based on a specialization policy, such as filename. However, due to hardware constraints, MPUs typically provide limited number of configurable regions (e.g., four in ACES [35]), and as a result, compartments that should be isolated must be merged together, allowing them to access each other's code and data unintentionally [35]. (ii) Hardware-based isolation techniques [45, 59], some of which are discussed in fixed debloating techniques, require specific hardware features or hardware modifications for reducing the attack surface for control-flow hijacking. However, both (i) and (ii) rely on hardware support or modification that may not be available on resource-constrained mode-based embedded devices such as RVs (§2.2, §3.1). In contrast to these, *RVDebloater* uses a software-only enforcement mechanism to specialize the firmware based on *requirements of each operational mode*, without relying on hardware support. (iii) CFI techniques [30] monitor indirect control flow transfers according to a statically determined control flow graph, but do not consider the operational mode requirements of these devices. CFI is orthogonal to *RVDebloater*, and combining both techniques can provide stronger security guarantees, where *RVDebloater* reduces the required function set per mode while CFI further constrains transfers within that reduced set. (iv) Hotpatching techniques such as AutoPatch [54] automatically generate and apply patches to real-time embedded devices at runtime, without requiring a reboot. Since *RVDebloater* reduces the attack surface rather than patching known vulnerabilities, these two lines of work are orthogonal and can complement each other.

8 Conclusion and Future Work

We propose *RVDebloater*, the first adaptive debloating framework for *mode-based* embedded devices such as robotic vehicles (RVs). *RVDebloater* addresses limitations of existing debloating techniques by using static or dynamic analysis to *automatically* identify the *required* firmware functions for each mode and reduce the attack surface by restricting access to unneeded functions at runtime. Furthermore, *RVDebloater* uses a software-only technique to track mode-switches and specialize the firmware code at runtime, without relying on an MPU. *RVDebloater* is implemented using the LLVM compiler, making it portable across different embedded platforms.

We evaluated *RVDebloater* using real missions with various scenarios on six different RVs running either ArduPilot or PX4, two of the most widely used RV firmware. The results reveal that *RVDebloater* can reduce the RVs' attack surface by an average of 41% with static analysis and 85% with dynamic analysis. Furthermore, *RVDebloater* incurred a performance overhead of 3.9%, and negligible power and memory overheads on those RVs ($< 4\%$). Finally, *RVDebloater* successfully mitigated three attacks on the RVs.

Although *RVDebloater* has addressed limitations in existing debloating techniques, three possible directions remain for future work. First, although this paper focuses on RVs for evaluation, we believe *RVDebloater* can be applied to other types of *mode-based embedded devices*. Evaluating *RVDebloater* on such devices is a direction for future work.

Second, we required the mode-switching function(s) and list of modes to be provided to *RVDebloater* by device maintainers (§3.3). These functions can be easily identified from the firmware documentation or source code, as device maintainers are generally familiar with the supported modes since they are required for controlling RVs. However, automated techniques such as dynamic analysis can be used to monitor the device and identify which function is responsible for changing the mode at runtime. Further, our instrumentation component (§4.2.1) can assist such analysis by enabling the monitoring of firmware functions.

Third, although we evaluated *RVDebloater* using static or dynamic analysis separately, these techniques can be combined as a hybrid approach. *RVDebloater-hybrid* can first check using the dynamic output; if the target is not found, it can then check the static output. Since static analysis tends to overestimate, control flow integrity (CFI) techniques can then be applied to ensure that all control flow transfers adhere to the call graph.

Acknowledgments

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Department of National Defense (DND), Canada, and a Four Year Fellowship from UBC. We also thank the anonymous reviewers of USENIX VehicleSec 2026.

References

[1] 2024 cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html, Last Accessed, May 2026.

[2] Aion robotics. https://www.aionrobotics.com/, Last Accessed, May 2026.

[3] Apache nuttx. https://nuttx.apache.org/, Last Accessed, May 2026.

[4] Arducopter flight modes. https://ardupilot.org/copter/docs/flight-modes.html, Last Accessed, May 2026.

[5] Ardupilot copter. https://ardupilot.org/copter/index.html, Last Accessed, May 2026.

[6] Ardupilot fail-safe. https://ardupilot.org/copter/docs/failsafe-landing-page.html, Last Accessed, May 2026.

[7] Ardupilot plane. https://ardupilot.org/plane/index.html, Last Accessed, May 2026.

[8] Ardupilot rover. https://ardupilot.org/rover/index.html, Last Accessed, May 2026.

[9] Ardupilot software in the loop. https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html, Last Accessed, May 2026.

[10] Arduplane flight modes. https://ardupilot.org/plane/docs/flight-modes.html, Last Accessed, May 2026.

[11] Bot-in-time delivery. https://www.forbes.com/forbes/2009/0316/040_bot_time_saves_nine.html, Last Accessed, May 2026.

[12] Clang: a c language family frontend for llvm. https://clang.llvm.org/index.html, Last Accessed, May 2026.

[13] Cve-2022-28711. https://nvd.nist.gov/vuln/detail/CVE-2022-28711, Last Accessed, May 2026.

[14] Dji agras t50. https://ag.dji.com, Last Accessed, May 2026.

[15] Gazebo robot simulation. https://gazebosim.org/home, Last Accessed, May 2026.

[16] Gllvm. https://github.com/SRI-CSL/gllvm, Last Accessed, May 2026.

[17] Hijacking drones with a mavlink exploit. https://diydrones.com/profiles/blogs/hijacking-quadcopters-with-a-mavlink-exploit, Last Accessed, May 2026.

[18] The llvm compiler. https://llvm.org/, Last Accessed, May 2026.

[19] Pixhawk. https://ardupilot.org/copter/docs/common-pixhawk-overview.html, Last Accessed, May 2026.

[20] Pixhawk power consumption. https://diydrones.com/profiles/blogs/pixhawk-and-apm-power-consumption, Last Accessed, May 2026.

[21] Px4 airframes. https://docs.px4.io/v1.12/en/airframes/airframe_reference.html, Last Accessed, May 2026.

[22] Px4 fail-safe configuration. https://docs.px4.io/main/en/config/safety.html, Last Accessed, May 2026.

[23] Px4 open source autopilot. https://docs.px4.io/main/en/, Last Accessed, May 2026.

[24] Qgroundcontrol. https://qgroundcontrol.com/, Last Accessed, May 2026.

[25] Seccomp bpf (secure computing with filters). https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html, Last Accessed, May 2026.

[26] Stack buffer overflow. https://github.com/PX4/PX4-Autopilot/issues/5643, Last Accessed, May 2026.

[27] Svf. https://github.com/SVF-tools/SVF, Last Accessed, May 2026.

[28] Ups will use drones to deliver medical supplies in north carolina. https://www.theverge.com/2019/3/26/18282291/ups-drone-delivery-hospital-nc-matternet, Last Accessed, May 2026.

[29] Writing an llvm pass. https://llvm.org/docs/WritingAnLLVMPass.html, Last Accessed, May 2026.

[30] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[31] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. {SHARD}:{Fine-Grained} kernel specialization with {Context-Aware} hardening. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2435–2452, 2021.

[32] Lars Ole Andersen. Program analysis and specialization for the c programming language. 1994.

[33] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1697–1714, 2019.

[34] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Detecting attacks against robotic vehicles: A control invariant approach. In *Proceedings of the 2018 ACM*

SIGSAC Conference on Computer and Communications Security, pages 801–816, 2018.

[35] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. {ACES}: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 65–82, 2018.

[36] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1749–1766, 2020.

[37] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Face-change: Application-driven dynamic kernel view switching in a virtual machine. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 491–502. IEEE, 2014.

[38] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–394, 2018.

[39] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, 2001.

[40] Michael Hooper, Yifan Tian, Runxuan Zhou, Bin Cao, Adrian P Lauf, Lanier Watkins, William H Robinson, and Wlajimir Alexis. Securing commercial wifi-based uavs from common security attacks. In *MILCOM 2016-2016 IEEE Military Communications Conference*, pages 1213–1218. IEEE, 2016.

[41] Zhenghao Hu and Brendan Dolan-Gavitt. Irqdebloat: Reducing driver attack surface in embedded devices. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1608–1622. IEEE, 2022.

[42] Zhenghao Hu, Sangho Lee, and Marcus Peinado. Hacksaw: Hardware-centric kernel debloating via device inventory and dependency analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1994–2008, 2023.

[43] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. {KSplit}: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, 2022.

[44] Rasoul Jahanshahi, Babak Amin Azad, Nick Nikiforakis, and Manuel Egele. Minimalist: Semi-automated debloating of {PHP} web applications through static analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5557–5573, 2023.

[45] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *NDSS*, 2018.

[46] Ao Li, Jinwen Wang, and Ning Zhang. Software availability protection in cyber-physical systems. In *34nd USENIX Security Symposium (USENIX Security 25)*, 2025.

[47] Lorenz Meier, Petri Tanskanen, Friedrich Fraundorfer, and Marc Pollefeys. Pixhawk: A system for autonomous flight using onboard computer vision. In *2011 ieee international conference on robotics and automation*, pages 2992–2997. IEEE, 2011.

[48] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. {RAZOR}: A framework for post-deployment software debloating. In *28th USENIX security symposium (USENIX Security 19)*, pages 1733–1750, 2019.

[49] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through {Piece-Wise} compilation and loading. In *27th USENIX security symposium (USENIX Security 18)*, pages 869–886, 2018.

[50] Davide Quarta, Marcello Pogliani, Mario Polino, Federico Maggi, Andrea Maria Zanchettin, and Stefano Zanero. An experimental security analysis of an industrial robot controller. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 268–286. IEEE, 2017.

[51] Raul Quinonez, Jairo Giraldo, Luis Salazar, Erick Bauman, Alvaro Cardenas, and Zhiqiang Lin. {SAVIOR}: Securing autonomous vehicles with robust physical invariants. In *29th USENIX security symposium (USENIX Security 20)*, pages 895–912, 2020.

[52] Vidya Lakshmi Rajagopalan, Konstantinos Kleftogiorgos, Enes Göktas, Jun Xu, and Georgios Portokalidis. Syspart: Automated temporal system call filtering for binaries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1979–1993, 2023.

[53] Majid Salehi, Danny Hughes, and Bruno Crispo. {μSBS}: Static binary sanitization of bare-metal embedded devices for fault observability. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 381–395, 2020.

[54] Mohsen Salehi and Karthik Pattabiraman. Autopatch: Automated generation of hotpatches for real-time embedded devices. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 2370–2384, 2024.

[55] Mohsen Salehi and Karthik Pattabiraman. RVDebloater: Mode-based adaptive firmware debloating for robotic vehicles, 2026. arXiv preprint arXiv:2602.00270.

[56] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[57] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[58] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. {LIGHTBLUE}: Automatic {Profile-Aware} debloating of bluetooth stacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 339–356, 2021.

[59] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1219–1236, 2020.

A Pruning Heuristics

(1) Signature-based Pruning Heuristic. SVF resolves indirect call sites using a basic pruning heuristic that considers only the number of arguments. However, this approach does not effectively prune spurious targets, as it considers functions with the correct number of arguments but incorrect argument types as valid targets.

To improve precision, we additionally use two filtering methods. First, *RVDebloater* considers the argument types when resolving indirect call targets, discarding those that do not match. In particular, *RVDebloater* analyzes all indirect call sites and prunes call edges to functions whose argument types do not match the expected types at the call site. Second, *RVDebloater* enhances call graph accuracy by also considering return types. Specifically, functions with a `void` return type are identified as spurious targets for indirect call sites that expect a return value. It effectively prunes spurious targets and reduces the number of call edges in the generated call graph.

(2) Address-based Pruning Heuristic. To further prune spurious targets, *RVDebloater* applies an additional heuristic (address-based pruning heuristic) based on “taken" addresses in the reachable path to the entry point. Andersen’s algorithm assumes that *all functions in the program* are reachable from the entry point. However, a function may not have its address stored in a variable or may be unreachable through the entry point, making it an invalid target for an indirect call site.

Thus, *RVDebloater* identifies all functions whose addresses appear in paths reachable from the entry point. Then, it examines all indirect call sites and removes edges that point to functions that are not in this list—i.e., functions whose addresses are not in any reachable path from the entry point.

After refining the call graph, *RVDebloater* identifies the entry functions of each mode to determine their reachable (required) functions. To do so, it employs an instrumentation component similar to the one used in dynamic analysis (§4.2.1), which instruments both the firmware functions and the mode-switching function(s). *RVDebloater* then executes the device through a mission that includes all modes intended for monitoring and control at runtime by *RVDebloater-static*. During execution, when the device switches modes, *RVDebloater* tracks the function(s) invoked by the mode-switching function to initialize the parameters required for the new mode and marks them as the entry function(s) of that mode.

RVDebloater also performs mode-name pattern matching to confirm the validity of the identified entry function(s). In particular, we observed that these firmware models (e.g., ArduCopter or ArduPlane) have entry function(s) responsible for managing mode-specific parameters, and their names typically include the mode name itself. Therefore, *RVDebloater* uses the list of mode names provided as input (§3.3) to verify and confirm the correctness of these functions. We found that all identified entry functions were correct. Note that this pattern can be adapted by developers to support other firmware. However, we used the same pattern in all the firmware models evaluated in this paper, as it was sufficient. Finally, *RVDebloater* uses the generated call graph to identify the reachable and unreachable functions for each mode, starting from its entry function(s).

B Implementation Details

A summary of the *RVDebloater* implementation across its five main components, corresponding to the different phases (§5, §B.1, §B.2, and §B.3) is presented in Table 6.

Table 6: Lines of code in *RVDebloater* components.

Component	Lines of Code
Pruning Heuristics	220 C++
Instrumentation	1000 C++ (two LLVM passes)
***RVDebloater* Profiler**	500 C
Guard Insertion	1100 C++ (two LLVM passes)
***RVDebloater* Monitor**	400 C
Total	3220

B.1 Analysis Phase

As mentioned earlier, the instrumentation component of the analysis phase takes the firmware source code as input. In the first step, the firmware is compiled to LLVM bitcode (*bc* file) using the *gclang* and *gclang++* compilers [16]. Then, if *RVDebloater-static* is selected, it uses the modified version of SVF with two pruning heuristics—signature-based and address-based—to generate a pruned call graph. We modified the TSCP implementation [36] by adding new heuristics to refine the call graph. A Python script then analyzes this call graph to generate lists of reachable functions for each mode through static analysis.

If *RVDebloater-dynamic* is selected, the instrumentation component links the firmware bitcode with *RVDebloater*'s profiler functions (i.e., *mode_entry* and *log_fn*). This component then takes the mode-switching function as input and iterates through the firmware's instructions. Specifically, it inserts *mode_entry* at locations within the mode-switching function where the return value is `true`, indicating a successful mode change. Also, it inserts *log_fn* at the entry points of other firmware functions.

```
define dso_local zeroext i1 @AP_Arming_Plane(%
    class.AP_Arming_Plane*, i32, i1 zeroext) #0
    align 2 !dbg !97833 {
// Allocation Variables
call void @log_fn(arguments)
// ... omit
}
```

Listing 4: Inserting *log_fn* at the entry of functions.

In the example shown in Listing 4, the instrumentation component inserts *log_fn* at the beginning of the function, after the allocation instructions. Finally, *RVDebloater* automatically executes the instrumented firmware with different missions in a benign environment and tracks its execution using profiler functions to identify the required functions for each mode at runtime.

B.2 Guard Insertion Phase

In the next phase, a different instrumentation component takes the firmware and mode-switching function(s) as input. Once the firmware is compiled to LLVM bitcode using *gclang* and *gclang++* compilers and linked with *RVDebloater*'s monitor functions (i.e., *mode_entry_runtime* and *monitor_fn*), *RVDebloater* first identifies the mode-switching function(s). It then inserts *mode_entry_runtime*, which monitors mode changes and loads the required functions for the new mode from the configuration file (either static or dynamic output) at locations where the mode successfully changes. Next, the component iterates over the firmware functions and replaces indirect control flow transfer instructions (e.g., indirect call sites represented by *CallInst* in LLVM IR) and return instructions (*ReturnInst* in LLVM IR) with *monitor_fn*, which controls target execution, raises an alarm, and switches to `Fail-Safe` mode if needed.

```
1 define dso_local zeroext i1
    @Plane18set_mode_by_number(%class.Plane*, i8
    zeroext, i8 zeroext) #0 align 2 !dbg !566193 {
// ... omit
2 call void @mode_entry_runtime(arguments)
3 ret i1 %22, !dbg !566216
4 }
```

Listing 5: Inserting *mode_entry_runtime* before return instruction (Listing 3).

For instance, this component inserts *mode_entry_runtime* before the return instruction when its value is true, as shown in Listing 5. Finally, it stores the instrumented file.

B.3 Monitoring Phase

In the final phase, *RVDebloater* builds the instrumented firmware executable using the *Clang* compiler [12]. At runtime (during an RV mission), the *RVDebloater* monitor is triggered when a mode change occurs. The monitor component then loads the corresponding configuration file, which includes the required functions for the new mode identified by either static or dynamic analysis. Furthermore, the *RVDebloater* monitor is triggered whenever an indirect control flow transfer or a return instruction is executed. In such cases, the component checks permissions based on the loaded configuration for the current mode and logs the event, raising an alarm or switching to `Fail-Safe` mode if an unauthorized execution is detected.

USENIX VehicleSec '26 Artifact Appendix: RVDebloater: Mode-based Adaptive Firmware Debloating for Robotic Vehicles

A Artifact Appendix

A.1 Abstract

The RVDebloater artifacts provide a complete, reproducible evaluation framework for mode-based adaptive firmware debloating of robotic vehicles (RVs). The artifact includes a modified SVF pointer analysis framework, pre-generated call graphs for each RV platform (e.g., ArduCopter, ArduPlane, ArduRover, and PX4), a set of missions, profiling and monitoring tools and results, and all scripts necessary to run the RVDebloater static and dynamic analysis pipeline in a simulation environment (SITL). In addition, the artifact introduces the first approach to LLVM bitcode generation for RV firmware, enabling researchers to generate `.bc` files and apply custom LLVM passes and third-party analysis tools to real-world firmware such as ArduPilot and PX4.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Because our technique operates in a simulation environment, it poses no risk to evaluators. RVDebloater operates without requiring the disabling of any security mechanisms.

A.2.2 How to access

The artifact source code and all supporting data are hosted on GitHub and Zenodo: https://github.com/DependableSystemsLab/RVDebloater.git , https://doi.org/10.5281/zenodo.20513118

A.2.3 Hardware dependencies

The main contributions and results of RVDebloater—its effectiveness, including the identification of required functions via static and dynamic analysis and the reduction of the attack surface at runtime (Section 6.2 in paper)—can be evaluated without hardware access using a simulation environment (SITL). Performance overhead (efficiency) is assessed on real hardware platforms, specifically the Aion Rover R1 and the Pixhawk drone.

PC-based analysis and simulation (E1, E2). All artifact experiments (E1) and (E2) run entirely in simulation (SITL) and do not require any physical robotic vehicle hardware.

Minimum requirements for running the simulation-based experiments: Disk: 30 GB of free storage (after cloning, installing dependencies, and building firmware) OS: Ubuntu 22.04 LTS (64-bit). *Machine used to generate the paper's analysis results* (call graph generation, static reachability analysis, and instrumentation during analysis and guard-insertion phases): CPU: Intel Xeon E-2224 (4 cores, 3.4 GHz base), OS: Ubuntu 22.04.5 LTS (64-bit).

A.2.4 Software dependencies

Required OS: Ubuntu 22.04 LTS. The artifact has been developed and tested exclusively on Ubuntu 22.04 LTS. Other Linux distributions are not automatically supported and may fail during the build or simulation steps. Our artifact includes a prerequisite script at `scripts/pre-req.sh`, which installs and configures all required software dependencies.

A.2.5 Benchmarks

The artifact includes pre-defined mission files (in the `missions` directory) for each evaluated RV type (ArduCopter, ArduPlane, ArduRover, and PX4). These missions are used as workloads during simulation-based evaluation.

A.3 Set-up

For more details, please refer to the README in repository.

A.3.1 Installation

Step 1 — Create the project directory. Change `${HOME}/rvd-project` to a different path if needed, and update the `$RVD_PROJ` variable accordingly.

Step 2 — Check or remove an existing `go` installation. If `go` is already installed, verify the version: If the version is 1.17 or higher, skip to Step 3. Otherwise, remove the existing installation before proceeding.

Step 3 — Clone the repository in $RVD_PROJ.

```
git clone
    ↪ git@github.com:DependableSystemsLab/RVDebloater.git
```

A.3.2 Basic Test

The scripts are pre-configured to run RVDebloater in **Rover (profiling) mode**. To verify a successful installation, run `run-simulation.sh`. Once the simulation has loaded, you can control the RV (check README). A successful run will produce profiling output in the `profiles` directory and simulation logs confirming mission completion. The current setup uses an automatic script to execute the mission without further manual intervention. By default, run-simulation.sh (in RVDebloater/scripts/) runs the simulation automatically.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): RVDebloater effectively identifies required functions through combined static and dynamic analysis and reduces the firmware's attack surface at runtime. This is proven by experiments (E1) and (E2) described in Section 6.2, whose results are reported in the corresponding tables and figures.

(C2): RVDebloater supports multiple RV types and operates correctly in both profiling and monitoring modes. This is proven by experiments (E1) and (E2).

A.4.2 Experiments

(E1): *Effectiveness — Profiling Mode* [15 human-minutes + 1 compute-hour]: This experiment demonstrates that RVDebloater correctly identifies required functions using profiling-based analysis for each RV type.
Preparation: Edit `scripts/run-simulation.sh` to select the desired RV type.
Step 1 — Select RV type. Uncomment the appropriate mission line (keep only one uncommented).
Step 2 — Configure vehicle type. In the same file, uncomment the vehicle and frame configuration (keep only one uncommented). Also, uncomment one of the lines 168, 169, or 170 based on your vehicle type in `scripts/run-simulation.sh`.
Step 3 — Set analysis mode to profiling. Edit `scripts/dependent/instrument-ap.sh` and uncomment the profiling passes (ensure the monitoring passes are commented out). Also, based on your vehicle type, you need to uncomment the lines in `scripts/dependent/instrument-ap.sh`.
Sub-Step 3 — Set Vehicle Type. Edit `Instrumentation/AddTrack.cpp` and uncomment the vehicle type that is set in previous steps (ensure the other lines are commented out).
Also, uncomment the correspond lines based on your vehicle type in `Instrumentation/AddTrack.cpp` (*Keep exactly one of the lines above uncommented at a time*).
Copter: Lines 128-170, **Plane or Rover:** Lines 176-218.
Results: The experiment generates a file called `mode_profile` in the `profiles` directory. Run `FunctionsName_to_Mode.py` to produce per-mode files listing the required functions for each mode.
(E2): *Effectiveness — Monitoring Mode* [15 human-minutes + 1 compute-hour]: This experiment shows RVDebloater's runtime debloating in monitoring mode.
Preparation: Follow Steps 1 and 2 from (E1) to select the RV type. Then set the analysis mode to monitoring.
Step 3 — Set analysis mode to monitoring. Edit `scripts/dependent/instrument-ap.sh` and uncomment the monitoring passes.
Sub-Step 3 — Set Vehicle Type. Edit `Instrumentation/CallOverwrite.cpp` and uncomment the vehicle type that is set in previous steps (ensure the other lines are commented out)
Also, edit `Instrumentation/AddEnforce.cpp` and uncomment the vehicle type that is set in previous steps (ensure the other lines are commented out).
Copter: Lines 96-138, **Plane or Rover:** Lines 142-185.
Execution: Before running, you need to change few places in `run-simulation.sh` (check README).

A.5 Notes on Reusability

Beyond the contributions of RVDebloater discussed in the paper, we address a fundamental gap in the RV firmware ecosystem: the lack of an automated technique for generating bitcode files, which are essential for leveraging analysis frameworks such as LLVM. To overcome this limitation, we modified the infrastructure of **gllvm** and its underlying **Go** implementation, enabling the automated generation of LLVM bitcode (BC) during the firmware build process. Our artifact supports complex, real-world firmware systems such as ArduPilot and PX4. With this capability, developers can seamlessly apply LLVM-based analyses, including custom LLVM passes and third-party tools such as DataFlowSanitizer (DFSan). Moreover, because our changes extend the core infrastructure of `gllvm` and Go, the approach generalizes beyond ArduPilot and PX4 and can support a broader range of firmware.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20260101. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/vehiclesec2026/.