Scaling Software Security Analysis to Satellites: Automated Fuzz Testing and Its Unique Challenges

Johannes Willbold[1], Moritz Schloegel[2], Florian Göhler[1], Tobias Scharnowski[2], Nils Bars[2], Simon Wörner[2], Nico Schiller[2], Thorsten Holz[2]

[1] **Ruhr University Bochum**
first.lastname@ruhr-uni-bochum.de

[2] **CISPA Helmholtz Center for Information Security**
first.lastname@cispa.de

Abstract—**The security of space assets is becoming an increasingly important concern, as the number of satellite services offered from space grows at an accelerating rate. In recent years, the functionalities of satellites have become increasingly sophisticated, allowing them to seamlessly provide complex services such as space-based Internet and high-resolution Earth observation. A significant contribution to these advancements was made by the software systems that control spacecraft in the harsh space environment. However, the development of satellite software poses a significant challenge due to the absence of physical access to the spacecraft during its mission. Recent research conducted by Willbold et al. has highlighted software security concerns, revealing an alarming absence of modern security measures among many satellites. Their analysis uncovered various security vulnerabilities in satellite software that could potentially allow attackers to gain full control over the spacecraft. Despite these results, their analysis is limited by the fact that software is analyzed manually, making the approach hard to scale.**

In this paper, we propose to use an automated vulnerability analysis technique, *fuzz testing* (*fuzzing* for short), to scale the analysis *without* the need of a human expert. Fuzzing is a dynamic program analysis technique that has proven highly successful at locating bugs in application software, such as browsers, or the Linux kernel. Its effectiveness has seen widespread adoption among the industry, such as Google or Meta, and launched multiple research efforts to make it even more effective. In essence, fuzzing creates a large number of inputs for the system under test and executes them while monitoring the system behavior, i.e., execution paths and crashes. Advanced approaches use lightweight instrumentation to gain introspection capabilities, allowing them to track the program path executed by a specific input and thus to guide the exploration to unseen program behavior. Despite its success, applying fuzzing to spacecraft presents unique challenges that we introduce and thoroughly discuss in this paper. First, obtaining feedback from the target program proves challenging, necessitating the exploration of firmware rehosting techniques where the target firmware is executed in an emulated environment without a precise representation of all peripherals. Second, satellites often employ complex boot processes that ensure memory integrity, perform device checks and configurations, and execute various time-intensive tasks, thereby posing challenges to approaches like fuzzing that aim to execute a program as frequently as possible, i.e., thousands of times per second. Finally, fuzzers rely primarily on crashes to identify bugs in the software under test, which fails to account for unrecoverable configuration issues. Beyond discussing these issues, we analyze their practical impact on the software of three satellites, ESTCube-1, OPS-Sat, and Flying Laptop. By discussing the challenges associated with applying fuzzing to spacecrafts and exploring potential solutions, we aim to contribute to the advancement of security practices in the aerospace industry.

979-8-3503-0462-6/24/$31.00 ©2024 IEEE

Table of Contents

1. Introduction

MITRE's 2023 ranking of top 25 weaknesses [1] lists out-of-bounds writes as the number one dangerous software weakness. This vulnerability allows an attacker to corrupt data and often hijack the execution of a program. Other common types of memory corruption vulnerabilities, including use-after-frees, out-of-bounds reads, null pointer dereferences, or integer overflows, are also included in the ranking, emphasizing the crucial role that bugs related to memory safety still play in software. This threatens, in particular, software written in memory-unsafe programming languages such as C or C++, where the programmer is responsible for memory management, leading to numerous bugs in the categories mentioned above. Despite being susceptible to these critical security bugs, languages like C enjoy high popularity for writing compact software with high performance. Their characteristics make them a perfect fit for use in embedded devices or satellites, where space and performance constraints dominate the development considerations. At the same time, their security is an underrated concern, as the continued popularity of these bugs in the year 2023 demonstrates. Being known for decades, many programs still suffer from basic bugs, raising the question of how to address such problems. As software is notoriously hard to secure, multiple approaches to identifying bugs and mitigating potential vulnerabilities may be needed.

Ranging from simple measures such as code review or pair programming to sophisticated static and dynamic analysis techniques, a vast number of options exist to detect vulnerabilities and programming errors in software. While having multiple eyes on the code may increase its quality, it is unlikely to root out all bugs, especially complex ones that may escape the developers' attention. Static analysis considers all possible program behaviors but lacks insight into which code is executed at runtime. This often leads to many false positive reports, tiring developers and taking away their time from fixing actual bugs. Dynamic analysis, on the other hand, reports only actually observed errors but can never guarantee that a program is bug-free – it merely reports on encountered bugs. Arguably, finding and fixing observable bugs is a worthwhile low-hanging fruit, raising the bar for an attacker.

When looking at recent dynamic software testing techniques, we find that *fuzz testing*, often abbreviated as *fuzzing*, has proven highly effective and one of the strongest techniques to identify bugs in software. This technique generates different inputs for the software under test and observes its behavior, using crashes and similar abnormal behavior as an oracle for bugs. At the same time, it often uses lightweight program instrumentation to track the path exercised by a given input, allowing the fuzzer to keep track of the explored program behavior and make informed decisions regarding an input's interestingness (i.e., if it explored new functionality). One of its most significant advantages is that every bug report comes with a reproducer, the input that triggered this bug. Fuzzing has proven its worth in uncovering numerous bugs in userspace software but also in kernels, hypervisors, firmware, and even hardware. It has been successfully used to test more than $1,000$ popular open-source software programs [2] and is successfully used by large tech companies, such as Google and Meta; however, it has not yet found full adoption in other areas, such as testing embedded devices or remote assets.

When studying why fuzzing has found little application in testing satellite software, we find several challenges unique to spaceborne assets that require careful consideration of how and where to apply fuzz testing. First, modern fuzzers rely on so-called *coverage feedback*, i.e., lightweight instrumentation that tracks what parts of the program an input executes, which, in turn, relies on the presence of source code or introspection capabilities. For satellites and other firmware interacting with peripherals, a setup allowing such feedback and considering the interactions with peripherals is required. Second, the nature of satellite software makes it significantly different from typical programs run on commodity computers. It usually features elaborate memory health checks and similar operations that may take significant time after startup. As fuzzing thrives on a high number of executed inputs, any slow operation during initialization is an impediment that must be taken care of. Third, when finding a bug, the fuzzer's output is an input triggering this particular bug, for example, by crashing the software. Yet, a single crashing input without further explanation and expertise may pose a challenge for software developers to address if they are unfamiliar with the details. These challenges make it hard to apply fuzzing and require a careful setup to leverage its effectiveness.

In this work, we systematically analyze how satellites can be tested using fuzzing. We first provide an extensive introduction to the area of fuzzing, analyze the attack surface of satellites, and discuss the challenges unique to the scenario of fuzzing satellite firmware. We then present three different approaches to fuzz test these satellites. For each approach, we run a case study testing this approach for an actual satellite; we outline challenges, problems, and our experience with applying fuzz testing to such real satellites. Our results indicate that two of these three approaches are feasible in practice, making them an excellent choice for testing satellite firmware before the satellite is deployed.

Contributions. In summary, our key contributions are:

- We discuss *five key challenges* arising when fuzz testing satellites and highlight the differences between fuzzing satellite firmware and other embedded device applications.

- We present *three different approaches* to applying fuzzing to satellite firmware while analyzing how they address the previously defined challenges.

- We *experimentally evaluate* these approaches against real-world satellite firmware to analyze the results and the amount of manual effort required to realize them.

All bugs discovered during our security analysis of the real-world firmware have been responsibly disclosed to the respective operators.

2. Background

Fuzzing

Fuzzing executes a target application with numerous inputs with the overarching goal of triggering unexpected behavior, thus revealing bugs. The inputs for the target can be created by either mutating inputs known a priori or by generating them from scratch based on some domain-specific language, such as a grammar that describes the input's structure. These techniques are known as *mutational* and *generational* fuzzing, respectively.

Different types of feedback mechanisms have been developed for fuzzers to steer the fuzzing process. Depending on the information they receive on their target's execution, we distinguish between three categories of fuzzers: blind, feedback-driven, and heavyweight feedback-driven fuzzers. Although their boundaries are often blurred, the complexity and computational requirements increase with each category. Subsequently, we discuss each of these categories in detail.

Blind Fuzzers—A *blind fuzzer* is comparatively simple in its implementation and mostly follows a brute-force search without receiving in-depth feedback, which explains its name. It takes a set of seed files or some kind of input specification and uses them to generate inputs by mutating a seed file or leveraging the input format specification. The program under test is then executed with this input, but no information from within the program is received (other than whether the program has crashed). This technique's main drawback is the lack of feedback: It does not recognize inputs reaching deeper into the state space, such as nested conditions or complex protocol parsing. This is because the fuzzer does *not* receive any feedback on whether a particular input solved a nesting level and, consequently, whether it should continue mutating the input that allowed it to proceed. Consequently, the fuzzer likely discards these inputs, thus discarding the solved constraint and facing the same challenge again. In short, without knowing how individual inputs performed, the fuzzer can not decide on what input to focus on.

Feedback-driven Fuzzers—To overcome the limitations imposed by this blindness, more advanced fuzzers rely on *feedback* from the program under test. This effectively tackles the main challenge of deciding whether an input exercised some beneficial behavior in the target and should therefore be kept for future use. In contrast to blind fuzzers, feedback-driven fuzzers use lightweight instrumentation techniques to exfiltrate meta information related to a specific input provided to the target application. This feedback allows constructing a *fuzzing loop*, as visualized in Figure 1. In this loop, the fuzzer first generates input for the target application, then sends it to the application, executes it, receives feedback, and adapts consecutive inputs based on this feedback.

The most prominent examples of such fuzzers are AFL [3], its successor AFL++ [4], and its derivatives [5], [6], [7], which all use code coverage as their core feedback metric to determine whether a new code path has been covered in the target. This feedback is powerful, as it provides the fuzzer information on whether a constraint, such as a compare instruction, was solved without invalidating previously solved constraints.

Although coverage-based fuzzing already yields enhanced results, it still faces various challenges, such as complex input data constraints enforced by the target application. For example, passing comparisons that incorporate magic values (predefined constants) is challenging to solve for coverage-based fuzzers, as overcoming such checks is a binary operation, i.e., the provided inputs contain the correct value or not; hence, the fuzzer cannot narrow down the search space. Several techniques have been proposed to tackle issues related to more complex constraints [8], [9], [10], [11], [12], [13], [14].

Another more technical challenge in the fuzzing loop is the repetitive execution of the fuzzing target, as in a naive approach, every iteration of the fuzzing loop would restart the target. However, executing an application in modern operating systems entails the startup process of the executable, which is time-consuming because, for example, libraries have to be loaded. Performing this initialization procedure for every iteration is redundant, which is why multiple strategies exist to skip this process. One strategy uses a so-called *fork server*, in which the Linux-based `fork` functionality is used. In this approach, the target program is started once and halted when it reaches a predefined point during its execution, i.e.,, some function reading the fuzzing input from a file. Then, the application is forked, and only the newly created copy of the process consumes the fuzzing input and is allowed to proceed with its execution. In all consecutive loop iterations, a copy of the halted program is used. This technique ensures that each loop iteration skips the time-consuming start-up routine while still having a fresh global state. Another approach is called *persistent state fuzzing*. In this approach, the code part that is the primary fuzzing focus in the target is moved into a loop, i.e., a `for`-loop. Then, the loop requests a new fuzzing input in each loop iteration. This approach can yield higher performance as the *fork server* but is also more invasive regarding code modifications, and the global state of the application can potentially be altered within this loop, which persists across loop iterations. Finally, there are *snapshot-based* approaches that take a full snapshot of the application at some point in time, i.e., just before fuzzing input is read by the target, and restore to the snapshot at the end of the fuzzing loop iteration.

Further, since the main purpose of fuzzers is to identify bugs and vulnerabilities, it is crucial to have methods that identify error-inducing behavior in a target program. The most commonly used oracle to tell if a program misbehaves is to detect whether it crashes. This kind of *bug oracle* is commonly used as it is easy to implement. However, in many cases, an input can be problematic even if they do not cause the program to crash. For example, a memory corruption might not lead to a crash but to an undefined and perhaps unstable state.

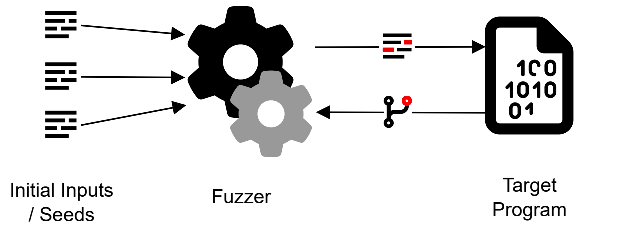

Figure 1. The *fuzzing loop* of a feedback-driven fuzzer. The fuzzer starts from a series of initial inputs, receives coverage feedback, mutates the input to check for changing coverage.

Heavyweight Feedback-driven Fuzzers— In contrast to relying solely on lightweight instrumentation, heavyweight feedback-driven fuzzers augment coverage-based techniques with feedback gathered through advanced program analysis techniques, e.g., taint tracking [11] or concolic/symbolic execution [13], [15], [16], [12]. Utilizing taint tracking allows us to reason about the influence of specific input bytes on instructions, such as branch conditionals. This additional semantic insight allows the fuzzer to concentrate its fuzzing efforts on specific parts of the application logic and thus avoid spending time on uninteresting components. Using symbolic execution allows modeling the target program with all its constraints. Based on this model, a concrete input exercising a specific path can be computed. This allows us to overcome complex constraints, such as checksums, which are virtually impossible to guess using the previously introduced lightweight techniques.

While heavyweight feedback-controlled fuzzers have proven to be an effective solution to the aforementioned problems, they suffer from a new set of problems. The main limitation is that they are relatively slow, do not scale to more complex targets, and require runtime environments that specify, for example, side effects of library functions [10]. Furthermore, complex constraints, e.g., cryptographic primitives such as hash functions or signatures, cannot be solved due to their non-deterministic polynomial time complexity.

Satellite Architecture

While the general architecture and components of satellites are well known in the community, we still discuss the components relevant to this paper to establish terminology and set the focus on the relevant components and interaction for this work. Satellites consist of a *satellite bus* and *satellite payloads*. The bus controls the power supply, attitude control, payloads, and other subsystems through the Command and Data Handling System (CDHS). The CDHS is the central command-and-control structure, as it deploys an On-Board Controller (OBC) that runs the Flight Software (FSW). This FSW, here also referred to as *firmware*, is the main focus of this work, as it executes security critical tasks, such as received Telecommands (TCs) from the operators and ground station, decoding and handling these TCs and generating Telemetry (TM) if necessary.

The components employed on the bus, such as the power or attitude control, can themselves contain processors for advanced configuration options. Thus, they might also handle some form of command-and-control traffic. Further, the OBC and potentially other components are connected to a wide range of *peripheral devices* such as sensors and actuators.

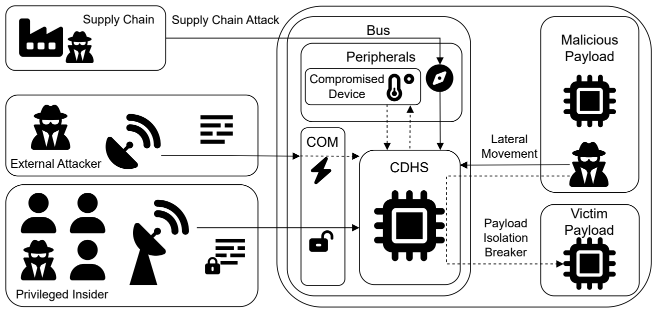

Figure 2. **Attack Surfaces**: The figure shows all three attack directions and their respective attacker types, with the CDHS in the center

The *satellite payload* deploys whatever equipment is necessary to carry out the satellite's mission, such as powerful radio equipment or Earth observation equipment. The payload can thereby exhibit an architecture similar to the bus, with its own infrastructure to handle commands through the Payload Data Handling System (PDHS). A satellite can have multiple payloads, of which each payload generates data and requests. These requests are handled by the bus systems to, e.g., receive information about the current state of the space vehicle. However, these requests might also interfere maliciously with the bus systems or even other payloads.

3. Attack Surface

Before discussing how satellites can be fuzzed, we analyze the *attack surface* to determine suitable targets for our testing efforts. An overview of the attack surfaces can be found in Figure 2.

The On-Board Software (OBSW) running on the CDHS exposes several surfaces that attackers can target from multiple angles. Specifically, we identify three main directions from where firmware security aspects of the OBSW become relevant. They account for TCs from the ground, commands from the satellite's payloads, and input from sensors and other peripherals that directly communicate with the bus. We further divide each direction into more fine-granular subdirections to accurately represent different attacker capabilities. They will be relevant for our fuzzing considerations, as each of these surfaces is potentially vulnerable and could be exploited using firmware vulnerabilities. Hence, our fuzzing approaches should first focus on testing these attacker-exposed surfaces.

Ground Station Telecommands

We consider TCs from a ground station the first attack surface since every TC has to be decoded, error corrected, potentially cryptographically verified, and handled through executing the corresponding TC handler in the FSW with the parameters contained in the TC. Since the corresponding data packets originate from outside the space vehicle, we can make no assumptions including that they are well-formatted, sanitized, or strictly compliant with a specific data format or message structure. Instead, they should be treated as untrusted input with the goal of exploiting vulnerabilities in protocol parsing, memory operations, or error handling within the space vehicle's software stack. Implementing robust input validation and sanitization is crucial for maintaining integrity and security. Given that a TC handler acts as an interpreter between the ground station's commands and the spacecraft's flight computer subsystems, any vulnerabilities at this stage could lead to unauthorized access, alteration of mission parameters, or even complete system failure.

Privileged Insider—Input validation and sanitization are even relevant for TCs coming from *privileged operators*. Ultimately, an operator consists of multiple individuals, one of which may have malicious intentions or one of which may have been hacked, giving an attacker their access and capabilities. This ability to be or act as an insider allows the attacker to send TCs that are cryptographically verified, i.e., signed and encrypted, and, thus, execute all TCs. In the case of *semi-privileged operators*, as discussed by Willbold et al. [17], an attacker may still execute a certain non-critical subset of TCs. Either way, insiders likely want to escalate their attack to achieve arbitrary code execution without relying on software updates, as they might be detected more easily. Hence, such attackers want to exploit vulnerable TCs, i.e., through memory corruption vulnerabilities, to gain remote code execution on the satellite.

External Attacker— In addition to insider threats, external actors can also send TCs to the space vehicle by bringing their own ground station, albeit they lack access to cryptographic key material. However, even when cryptographic protections are deployed, error correction and packet parsing happen prior to decryption and signature verification, according to standard protocols like CCSDS' Space Data Link Security (SDLS) protocol, such that only data of the network layer and above are encrypted. Hence, protocol parsing vulnerabilities in lower layers can be triggered without valid access keys. In addition, cryptographic forging attacks can create malicious input for the upper packet parsing layer without attackers possessing access keys.

Payload Command Handlers

In addition to threats from outside the space vehicle, *internal* attack surfaces must be considered from an OBSW firmware security perspective. Specifically, the payloads hosted by satellites often bring their own PDHS systems with custom firmware. Further, in many cases, these payloads are developed and operated by external organizations unrelated to the satellite operator's bus system [18]. Hence, these payloads should be regarded as untrusted external entities that could pose as threat against the bus system or other payloads on the satellite. To address this threat, the communication interfaces in the CDHS firmware interacting with payload components should be regarded as an attack surface, as they must adhere to the same rigorous input validation and sanitization as previously described for ground station TCs.

Lateral Movement—Within payload command handlers, we define the first subdirection for attacks against the CDHS itself from the payload. In this case, an attacker tries to laterally move from the compromised payload to the bus, escalating the attack.

Medial Movement—If an attack compromises the payload, it should be confined to this payload. Thus, in addition to preventing lateral movement, the bus must also ensure that a compromised payload may not infect other payloads. If attackers can freely interact with other payloads, this opens up the commands and data processing of other payloads as attack surfaces.

Malicious Components

Firmware security considerations must also include the attack surface exposed to components and peripherals on the satellites that are not directly connected to an attacker but still can send potentially malicious data to the CDHS. These include components such as power supply systems, attitude control, and peripherals, i.e., sensors and actuators.

Supply Chain Attacks—Peripherals acquired from third parties could have been prepared by an attacker prior to embedding them into the spacecraft in a supply chain attack. This would allow attackers to exploit vulnerabilities in the attack surfaces otherwise not considered relevant due to attackers being unable to exploit them.

Compromised Devices—While not immediately obvious, in cases where components bring their own memory and rudimentary computing setup, such as attitude control systems, attackers could reconfigure them to act maliciously when interacting with them from the CDHS. In these cases, an attacker that has compromised a subsystem or a part of the CDHS might be able to maliciously reconfigure another component. This compromised component might then be able to send malicious data to subsystems or components that are not directly exposed to the attacker. Crucially, this differs from the supply chain, as this attack is invoked during a mission, while a supply chain attack is performed ahead of a mission.

4. FUZZING CHALLENGES

In the following, we enumerate and discuss five key challenges that significantly impact fuzzing of satellites and are unique to satellite firmware fuzzing either in their nature or magnitude. Any approach that successfully performs fuzzing of spacecraft software must address these challenges. While some challenges exist for other embedded devices or areas of fuzzing as well as for space vehicles, we discuss why they have a unique impact on the fuzzing of satellites compared to other areas.

Complex Satellite Boot Process

Space vehicles often deploy a rigorous and complex boot process when first starting or restarting the OBSW. The startup procedure performs extensive error checking and correction on persistent flash memory and databases, checks the presence and health of peripherals, enables hot/cold spares or redundancies hardware if faults are present, polls all peripherals to populate the current database of sensor values, and fulfills numerous other tasks. While many devices have long boot processes, the one of satellites is particularly rigorous and lengthy due to the hostile environment. The main culprits are the extensive error checking and correction, as first documented by Scharnowski et al. [19].

Only Crashing Inputs

Most fuzzing approaches use crashes as the sole bug oracle (ref. Section 2), i.e., as a method to detect the presence of a bug. While this method is simple to implement and has a proven track record of identifying vulnerabilities, a crash is usually not the most problematic outcome on a satellite. Ultimately, watchdog devices can detect a crash and can restart a satellite. By contrast, changes in configurations of devices or modifications of flight plans might put the satellite in a state from which recovery is challenging. Thus, detecting these issues is vital to identifying critical vulnerabilities in space vehicles.

Computing Hardware with Limited-Performance

Satellites, especially small satellites such as CubeSats, are built with harsh space and power limitations in mind. This often leads to processors with low-performance and low-energy consumption for the CDHS to run the OBSW. However, this approach makes it hard to perform fuzzing on the actual hardware used on the satellites. While testing directly on the hardware brings the advantage of having the real-world setup, including the peripherals and other components, it comes with the drawback of being limited to the system's CPU. Further, interesting bug oracles, i.e., means to identify interesting fuzzing results, can often only be extracted using debuggers attached to, for example, a JTAG port [20]. However, this additionally slows down the processor, making fuzzing unfeasible in most cases.

Highly Specialized and Individual Setups

A solution, especially to the limited hardware performance, is *emulating* the OBSW. This allows the fuzzer to run on modern high-performance CPUs, while accepting the overhead introduced by the emulation setup. However, satellites feature highly specialized and target-specific setups, with no two satellites outside of constellations having the same setup. In addition, many components, such as attitude control, power supplies, specialized sensors, or payload systems, are unique or produced in low volumes, making the existence of off-the-shelf emulation setups unlikely. Further, redundancies and multiple processors distributed across bus, payload, and individual components complicate emulation approaches. These factors, the plethora of components, and their uniqueness make satellite emulation setups highly challenging, as they require intensive and costly target-specific efforts.

While other areas, such as the car or aviation industry, also feature highly complex setups, they differ in the fact that developer teams of satellites, especially small ones, consist of a handful of people. For cars or airplanes, there are hundreds or thousands of responsible engineers, which makes the development of accurate emulations possible, cost-effective, and, in many cases, even mandatory.

Performance of Existing Digital Twins

Many modern missions feature a digital twin setup to perform testing on the ground without having to purchase a second satellite for a flatsat model or to have a mobile testing setup for engineers on their personal machines. However, digital twins are often not a good fit to perform fuzz testing on, as the setups usually feature a simulation approach where target CPUs are simulated down to each processing cycle. While this provides insights for developers to debug issues in depth without needing real hardware, it also brings a significant overhead. Hence, these digital twin setups cannot provide the throughput and performance necessary for efficient fuzz testing.

5. CASE STUDIES

Considering the attack surface of satellites and the challenges inherent to fuzzing them, we now outline three different approaches towards fuzz testing them.

Subsystem Extraction

Developed in cooperation with *Airbus Defense and Space*, *Flying Laptop* is a small satellite launched in 2017 and is operated by the *Institute of Space Systems* at the *University of Stuttgart* [21]. The satellite is used for technology testing, scientific earth observation, and teaching. The total mass of 120 kg classifies the satellite as a medium-sized satellite and, thus, no longer as a small satellite.

The OBC is a *LEON3* processor, often found in space applications, and uses a 32-bit SPARC instruction set architecture. The OBC is connected with an S-Band antenna, which serves as the satellite's Communication Module (COM) using the Consultative Committee for Space Data Systems (CCSDS) protocol stack, with Space Packet Protocol (SPP) for the network layer. In addition, the CDHS, which includes the OBC, is also connected with a custom I/O board that acts as an internal connector.

The OBSW is logically subdivided into the *MkProm2* [22] bootloader that holds the initial RTEMS operating system state, compressed via LZSS and the Flight Software Framework (FSFW). The FSFW component is built on top of RTEMS and provides the base functionality to set up, initialize and connect system objects and tasks. In addition, FSFW contains reusable functionality to decode and route incoming CCSDS packets. Based on FSFW, the mission-specific logic is built, including custom packet parsing.

The fuzzing approach deployed for this system, as shown in Figure 3, is to extract the TC processing of the satellite firmware and place it into a Linux executable, which runs inside an emulated SPARC Linux environment. This extraction allows us to focus the fuzzer on the relevant TC processing, while also converting the problem of fuzzing embedded firmware and the prolonged boot process into a general-purpose fuzzing problem.

Execution Environment—On a technical level, we attempt to isolate the CCSDS stack and the TC handlers. Fully booting and running the system in an emulator entails allowing the firmware to interface with all its peripherals and implementing peripheral logic that provides appropriate responses. However, some components, especially the I/O board, are custom-made and would require a target-specific effort to implement emulation. To avoid this, we extracted parts of the firmware code containing relevant system features. In this approach, we targeted the CCSDS stack that parses and handles TCs. This is an interesting target, as it includes the mission-specific packet parsing built on top of FSFW. By isolating a custom subset of the code that implements the CCSDS stack, we aimed to create a Linux application that can function *without* accessing satellite hardware peripherals. We moved this isolated TC decoding, parsing, and handling into a Linux-based application that would act as a regular Linux-based application, which can be executed in the Linux user space using the QEMU user mode. QEMU is an open-source virtual machine and emulator. This allows execution of the application using one TC packet as input, which would be handled before the application terminates.

Fuzzing Setup—The isolated TC handling in the form of a Linux-based application allows us to utilize general-purpose fuzzing tools instead of firmware fuzzing tools. This is an advantage, as much work has been done on general-purpose fuzzers such as AFL++ (ref. Section 2). This also simplifies the fuzzing setup to the point where we only have to ensure that the TC input is provided in a way AFL++ picks up,

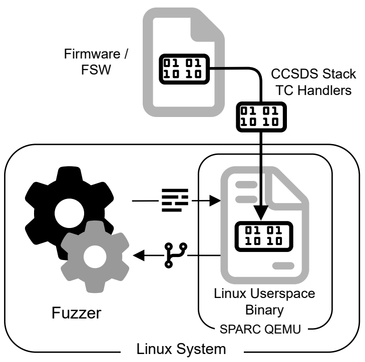

Figure 3. Fuzzing Setup for *Flying Laptop*.

i.e., using the Linux `read`-function. After that, the fuzzing setup is fully automated and performed by AFL++ as-is. This fuzzing setup is shown in Figure 3.

Results—It turned out that the FSFW needs a large set of global objects to be initialized. This made it necessary to initialize complex C++ object hierarchies manually, which requires extensive extraction of code and objects from the firmware into the Linux application. Additionally, some Linux-incompatible low-level code remained for primitives such as mutexes, output objects, and message queues. These instances of incompatible behavior need to be manually patched so that we can run them in a Linux application. It turned out that these modifications require extensive manual and target-specific work to run the CCSDS stack within a partial execution environment. As a result, finishing the work would have been beyond the scope of this research project, but it still serves as an interesting case study, as the insights to global state and code modifications are applicable for others as well.

Observations— As an advantage of partial emulation approaches, we may be able to focus the security analysis on a specific area of the target. Inherently, partial emulation requires no functional digital twin. As a major practical drawback of partial emulation, we find that, depending on the target, significant custom and target-specific manual work is required to produce a working execution environment. A second drawback pertains to the test coverage. By not executing large parts of the target under test, we also risk missing crucial functionality. Additionally, if potential security issues only become visible from interactions between different components, we may cut off execution before this interaction occurs, thus hiding potential security issues. In our experience, such interactions are often the cause for vulnerabilities, especially between components that originate from different authors or have been designed a long time apart.

Full System Emulation with Persistent Mode Fuzzing

OPS-SAT is a CubeSat-class spacecraft operated by the European Space Agency (ESA). Co-developed by the Graz University of Technology, it was launched in December 2019 and has since been serving as an open research experiment platform. Essentially, any person or group can develop an experiment for the platform. ESA will review the experiment and might decide to upload it onto the satellite. Since essentially any actor, including attackers, can develop code and submit it, this satellite highlights a scenario where untrusted code is to be executed. This aspect and *OPS-SAT* being open to research also lead to its inclusion in hacking competitions and live demonstrations, making it an excellent candidate for security testing.

In the following, we first outline the technical preparations before discussing our fuzzing approach. Here, we focus on the bus system of *OPS-SAT*, which handles security-critical command-and-control functionalities, such as decoding and executing TCs that control the power subsystem, attitude control, and payload systems. Specifically, the satellite deploys two separate command-and-control channels, one from a UHF antenna utilizing the *libcsp* protocol and one from and S-Band antenna utilizing a CCSDS protocol stack. The commands from the S-Band antenna can also be sent from the payload systems, thus this attack surface is also relevant from a malicious payload user perspective (ref. Section 3).

Our fuzzing setup, as shown in Figure 4, includes a full system emulation where the fuzzer generates TCs and lets the emulated firmware process them. All peripherals, such as sensors and actuators, only deliver constant values. The constant values are chosen to put the satellite in a valid operating state to receive and handle TCs. We did not patch the firmware itself; however, the emulator contains six target-specific patches that allow us to omit the implementation of two Memory-mapped I/O (MMIO) devices. Hence, the target-specific overhead is limited, especially compared to the approach for *Flying Laptop*.

Execution Environment—The execution environment to fuzz the *OPS-SAT* FSW is developed using a full-system emulation approach with the commonly used QEMU emulator. Since the OBC of the satellite is an *AT32UC3C* controller that uses the *AVR32* instruction set, we had to implement the entire instruction set from scratch, including disassembly, instructions, and processor peripheral devices. However, while such an effort is high, it only has to be done once for the target architecture. We published our implementation for future use[2].

Besides the instruction handlers, we implemented MMIO controlled processor devices, such as interrupt handlers or clocks, where a fully working version is required. In other cases, we only implemented the minimal required functionalities.

In cases where interactions were only required in the initialization phase of the FSW, i.e., to set up the device, we either created a minimal working version that always responds the same, or we performed an emulator-side target-specific patch, where we checked for a specific program counter and jumped to a specific address to skip certain code areas. The latter approach is highly target-specific and testing intensive, so we refrained from it as much as possible. Ultimately, we only had to use it for the SDRAM controller, which controls an external RAM module to expand memory size, and for the watchdog timer. Tracking the exact configurations for the SDRAM controller is unnecessary, as we can assume that it gets configured in a way that works on the target system, and there was no user input possible because all values were hard-coded. Further, exact interactions with the RAM are not simulated, but we assume that a working memory area inside the emulator gets treated as any other memory area; only the location inside the virtual memory space has to be configured.

[2]https://github.com/flogosec/qemu-avr32

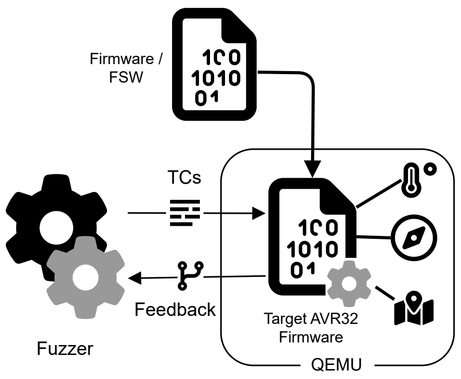

Figure 4. Fuzzing setup for *OPS-SAT*. The underlying devices have been specifically implemented in QEMU to support the platform.

Finally, we also implemented the necessary peripheral buses, such as SPI and I2C, which, again, was a one-time effort for the processor family. However, we did not implement target-specific peripherals, such as sensors, and instead either left them out entirely, which appears to the firmware as if the sensor is dead since it is not responding, or we created a minimal working version that always replies with the same message. Ultimately, the goal was to create a minimal working setup to see the minimum effort required to perform meaningful fuzzing for the target. Again, we ensure that the satellite is in a valid state to receive commands and process them, even in the absence of several devices. Theoretically, these missing devices prevent the firmware from reaching certain states that require specific inputs. However, building a perfect emulation is beyond the scope of this project.

In conclusion, this leaves us with a slightly target-specific full-system emulation, where only necessary components are fully implemented.

Fuzzing Setup—As a fuzzing approach, we utilize a feedback-driven persistent mode approach (ref. Section 2), see Figure 4. We retrieve coverage and crash information from the emulator, while not fully restarting the firmware after every execution. Instead, we perform the fuzzing in a loop, starting at a certain point in the system and *resetting* the execution to that point after executing the TC handling. This saves us from going through the time-consuming boot phase of the satellite (ref. Section 4). We do not provide fuzzing input to peripherals and only ensure that they put the satellite into a state where it can receive and handle commands, i.e., the antenna is deployed, sufficient power is available, and operating temperature is complied with.

Since the state of the satellite potentially changes after every TC, but fully rebooting it is too costly, we reboot it only every few thousand fuzzing iterations. This is a trade-off to prevent too much accumulated state, while still benefiting from fewer reboots.

Results—The performance of our fuzzing approach was evaluated by analyzing how many TC handler functions were executed. Additionally, we analyzed how many *basic blocks* inside each handler function were covered. Other functions that are called by the TC handler functions are not part of this metric. A *basic block* describes a series of processor operations that end with a jumping or branching instruction and start at a jump or branching target. Hence, they are the most basic form of describing chunks of code in a program. We refer to a basic block as being *covered* if it is executed in at least one program execution during the fuzzing. This ensures that the fuzzer generated input that could reach that specific basic block, which is a common metric to evaluate fuzzers. Measuring coverage is commonly done as a proxy for measuring bugs – the underlying insight is that a fuzzer cannot find a bug if it did not cover the respective code location [16].

The fuzzer-generated inputs were able to trigger the execution of all 59 commands in the firmware. The TC handling functions contain 1704 basic blocks, of which we covered 1300, or 76%. 32 of the 59 TC handlers were covered by more than 90%, while only 5 TC handlers achieved a coverage of less than 50%.

The fuzzer was able to identify one bug that can trigger a crash of the firmware. The code segment in question uses the `memcpy` function to copy data from the user data segment of a TC to another location in the memory. The size of the copied data is directly taken from the TC. However, there is no length check done by `memcpy` or by the calling code segment. Therefore, a *buffer overflow* is possible. If the length value in the TC is high enough, code pointers in the memory are overwritten and the firmware crashes when one of these pointers is loaded the next time. Because a TC is too short to overwrite the code pointers with useful data, the bug can result in a *denial of service* but not in the execution of arbitrary code.

Observations— Implementing and evaluating our approach for *OPS-SAT* yielded several interesting insights. **(1)** Skipping the boot phase by manipulating the program counter does reduce the overhead of the boot phase significantly, but **(2)** potentially leads not non-reproducible crashes that were triggered through an accumulated state that cannot be recreated easily. **(3)** Missing peripheral devices seem to have little impact on the satellite. We attribute this to the fault-resistant development goals and approaches that account for devices being broken after launch or over time in orbit. Interestingly, this makes it significantly easier to develop working emulations.

Full Firmware Rehosting

ESTCube-1 is the first satellite launched by Estonia in 2013. It was developed by students from the University of Tartu [23], [24], [25]. Though primarily serving as an educational project for the students, it also carried an experimental electric solar wind sail (E-sail) as a payload.

The CubeSat deploys a mesh-network approach for TCs, where each TC packet can be addressed using a dedicated field to either the Ground Station (GS), CDHS, Camera (CAM), or one of several other components on the satellite. Packets to the satellite from the GS are thereby first parsed by the CDHS before being dispatched to the target component, which may be the CDHS itself, which then unpacks the command and handles it, for example, through a command scheduler. Similarly, the CDHS also sends TM from the components to the GS. The scheduler system uses its own packet format to select one of the handler functions and provide arguments if needed.

Compared to the two previous fuzzing approaches, this approach uses a well-established fuzzing tool that automates access to peripherals while also rehosting the entire firmware, as shown in Figure 5. Since the fuzzer models peripherals, the manual target-specific overhead is severely reduced compared to the previous two case studies. The fuzzer is still a binary-only fuzzer and does not require source code. It receives feedback from the emulation setup used for the rehosting.

Execution Environment—Here, we fuzz test the CDHS with its command handler in a full-system rehosting approach. This allows us to use the firmware image as-is without manual modifications. We use the state-of-the-art rehosting fuzzer HOEDUR, which automatically infers an emulation implementation of peripherals during execution [26]. The initial configuration of the firmware is auto-generated. As a manual optimization to the generated configuration, we relaxed the handling of TC packets to feed inputs more efficiently. Generally the configuration of the firmware includes interrupts and memory mappings for ROM, RAM and MMIO regions. HOEDUR executes the firmware in QEMU, the Instruction Set Architecture (ISA) emulator that has already been used for *OPS-SAT* and *Flying Laptop*. However, HOEDUR modifies QEMU to allow for higher optimization through fast snapshots and precise execution control. Further, instead of requiring manual implementation of peripherals, accesses to peripherals are automatically analyzed through symbolic execution when they first occur. Hardware behavior is modeled by determining which part of the input is actually relevant to the firmware. The input structure learned by the fuzzer is further used to improve input mutations, which reduces the total space the fuzzer has to cover, resulting in improved code coverage.

Fuzzing Setup—We used the out-of-the-box fuzzing approach provided by HOEDUR. The firmware is loaded in the ISA Emulator and executed until the first peripheral access is found, then a snapshot is taken to skip this deterministic initialization in further executions. For each execution, the emulator restores the snapshot and forwards peripheral accesses to the fuzzer, see Figure 5. Each different peripheral is separated into an input stream, which keeps logically different values apart and helps the fuzzer to improve mutation efficiency. The fuzzer considers all possible values and, therefore, also exercises error handling code. After an execution, the fuzzer collects coverage and information about crashes, timeouts, and peripheral impact. This is then used to store inputs with new behavior and guide further input mutations.

Results— We evaluate the performance of the automated fuzzing approach in the same way as in the previous case study. We measure how many TC handlers have been reached in total as well as the code coverage in these handlers.

The fuzzer successfully crafts High-Level Data Link Control (HDLC) packets that are sent to the CDHS on the UART interface. These packets reach all of the 128 TC handlers on the CDHS. On average, the fuzzer achieves a basic block code coverage of 78% within the TC handler functions. More precisely, the handler functions consist of 981 basic blocks, 763 of which were covered.

The fuzzer identified several inputs that led to a firmware crash. We inspected these crashes and found that they can be classified into **(a)** dangerous TC handlers that directly write memory and **(b)** two additional bugs that were caused by programming errors. The TC handlers that directly modify memory are likely used for patching, recovery, or debugging purposes. Interestingly, this bug was discovered by marking certain memory areas as *read-only*, which triggered as the

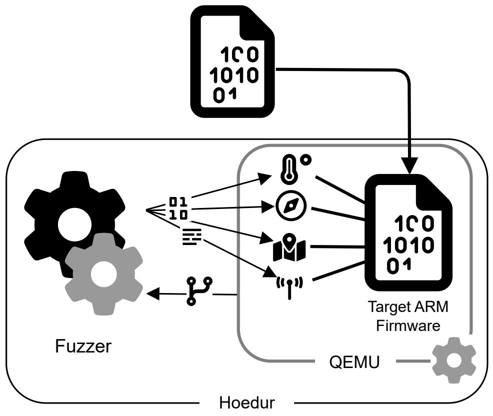

Figure 5. **ESTCube-1 Fuzzing Setup**: Peripheral and TC inputs are generated through the fuzzer, with no specific knowledge of underlying devices.

firmware attempted to write it. Specifically, we marked the memory area that contains the firmware image in memory as *read-only* to identify dangerous (and potentially undocumented) TCs that could overwrite firmware code regions and could thus allow an attacker to compromise the satellite.

The first additional bug can be triggered through an HDLC packet with an empty payload, which causes a scheduler packet to be a `nullptr`; when handling this packet later, the scheduler does not check for this, causing a *nullpointer dereference*. The second additional bug concerns TC handlers that do not verify the packet lengths. Thus, attackers can define a packet size field value that varies from the actual packet length. This, in turn, leads the TC handlers to read more memory from the location where the TC is stored, causing an *out-of-bounds read* that can potentially disclose secret information, similar to the well-known *OpenSSL heartbleed (CVE-2014-0160)* vulnerability. This bug, among others, was first presented by Willbold et al. in a case study of the satellite to explore the security situation of Low Earth Orbit (LEO) satellites [17].

Observations—This approach has low manual overhead and only requires access to the firmware image to get started. In addition, the fact that accesses to peripherals are automatically modeled avoids the need to manually implement an accurate peripheral handler as seen in the *OPS-SAT* approach (ref. Section 5). This also addresses the challenge regarding *highly specialized and individual setups* (ref. Section 4), as the exact setup is almost irrelevant due to the peripheral modeling approach. Further, the fuzzer deploys a *snapshotting* mechanism that takes the snapshot of the execution environment of the fuzzer upon first receiving input from a peripheral device after booting. This allows us to skip the overhead associated with prolonged booting phases, which we also identified as a typical challenge (ref Section 4). Finally, marking memory regions as *read-only* allowed us to identify dangerous TC handlers that pose a security risk, as attackers abuse them to introduce firmware patches, as already noted by Willbold et al. [17].

Takeaways

In summary, we analyzed firmware images of three satellites using different techniques. Our focus was in particular on the following four aspects:

(1) The amount of manual effort to track how feasible the approach is for other setups to overcome the challenge of *highly specialized and individual setups*.

(2) How the approach handles our challenge of *complex satellite boot processes*.

(3) The feasibility of performing the testing without hardware to address the challenge of *limited-performance computing hardware*.

(4) The ability to detect non-crashing inputs that trigger bugs, as defined in our challenge "*only crashing inputs*".

Our practical experience showed that using a full-system rehosting approach, as seen for *ESTCube-1* in our last case study, provides the most benefits. Addressing **(1)**, it required the least amount of manual effort, as it required only a small configuration file for the fuzzer, mainly auto-generated after a quick program analysis. Regarding **(2)**, the snapshotting mechanism to capture the execution state after booting allowed us to bypass the boot process without dealing with a tainted global state, as seen with the persistent state fuzzing approach for *OPS-SAT*.

Since all our approaches utilized emulation, they all bypassed the need for actual hardware to address **(3)**.

The only point all approaches fail to fully address is **(4)**. Fuzzing inherently needs a bug oracle that triggers on an observable event, such as a program crash or timeout. While very generic, these indicators may not detect all bugs, such as configuration issues. In other fields, first attempts at finding better bug oracles that are more tailored to a specific area have been made, for example, to find server-side web application bugs [27], [28]. To the best of our knowledge, no comparable approach has been undertaken to provide bug oracles for erroneous configurations or the realm of satellites. That said, HOEDUR mapping code pages as read-only to identify bugs in *ESTCube-1* overwriting code improves the detection capability of the regular crash oracle.

6. RELATED WORK

Firmware Fuzzing

Fuzzing has proven to be an effective technique to expose robustness and security issues in code. Firmware, however, as opposed to common software such as Linux command-line tools, is difficult to combine with off-the-shelf fuzzers. Thus, firmware fuzzing approaches have traditionally used *blind fuzzing* and tested firmware on the physical device [29], [30], [31], [32]. Later approaches have partially virtualized firmware fuzzing by combining an emulator with selective execution on the original hardware using hardware-in-the-loop [33], [34], [35]. However, none of these works looks at satellite or space systems firmware in particular.

Recently, research opened the possibility to fully virtualize firmware fuzzing via a technique called *rehosting* [36]. While firmware fuzzing approaches that require specialized hardware suffer from low execution speeds, incomplete coverage feedback, and the need for a manual hardware setup, *rehosting* executes firmware in a fully virtual and parallelized environment. This allows for gathering detailed execution feedback and scaling firmware fuzzing to server resources. A recent work, *Fuzzware* [37], has introduced hardware modeling via firmware binary analysis techniques. Building upon *Fuzzware*, *Hoedur* [26] introduced additional techniques to make rehosting-based fuzzing aware of its target, which further increases fuzzing effectiveness greatly. We use both Fuzzware and Hoedur in our setup for ESTCube-1.

Satellite Fuzzing

Previous work has explored some options to apply fuzzing to satellite firmware. Notably, Scharnowski et al. [19] performed three case studies using the ESTCube-1 approach presented in this paper by leveraging their rehosting-based fuzzer *Fuzzware* [37]. However, it was primarily an analysis of how well their fuzzer works for satellite applications, while this work explores the general challenges to fuzzing satellite applications and accounts for multiple approaches to do so.

Further, Gutierrez et al. [38] explored the possibility to perform fuzzing on their satellite *SUCHAI-1*. On a technical level, the paper discusses how the satellite's flight software can be fuzzed while discussing and evaluating several strategies. Again, the paper does not discuss challenges or compare larger-scale approaches like our work does.

7. Conclusion

In this work, we have discussed fuzzing and its applicability to satellites. After introducing fuzzing, we analyzed the attack surface of satellites, enumerated challenges that need to be addressed for efficient and effective satellite software fuzzing, and presented three concrete approaches backed by experimental case studies. Our results show that rehosting satellite software is the most promising approach towards fuzzing satellites and that fuzzing is a worthwhile security testing technique to improve the security of satellite software.

Acknowledgments

This work was funded by the German Federal Ministry of Education and Research (BMBF, project CPSec – 16KIS1564K). The work was partially supported by the MKW-NRW research training group SecHuman.

References

[1] The MITRE Corporation, "2023 CWE Top 25 Most Dangerous Software Weaknesses," https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2023.

[2] Google, "OSS-Fuzz: Continuous Fuzzing for Open Source Software." [Online]. Available: https://github.com/google/oss-fuzz

[3] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[4] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining Incremental Steps of Fuzzing Research," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[6] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[7] C. Lemieux and K. Sen, "Fairfuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage," in *International Conference on Software Engineering (ICSE)*, 2018.

[8] lafintel, "laf-intel - Circumventing Fuzzing Roadblocks with Compiler Transformations," https://lafintel.wordpress.com.

[9] Google Project Zero, "CompareCoverage," https://github.com/googleprojectzero/CompareCoverage, 2019.

[10] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "RedQueen: Fuzzing with Input-to-State Correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[11] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[12] S. Poeplau and A. Francillon, "Symbolic Execution with SymCC: Don't Interpret, Compile!" in *USENIX Security Symposium*, 2020.

[13] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *USENIX Security Symposium*, 2018.

[14] N. Bars, M. Schloegel, T. Scharnowski, N. Schiller, and T. Holz, "Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge," in *USENIX Security Symposium*, 2023.

[15] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[16] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[17] J. Willbold, M. Schloegel, M. Vögele, M. Gerhardt, T. Holz, and A. Abbasi, "Space Odyssey: An Experimental Software Security Analysis of Satellites," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[18] H. Burkhardt, "The DLR microlauncher and payload competition," https://www.dlr.de/rd/en/desktopdefault.aspx/tabid-15784/25586_read-65808/, 2022.

[19] T. Scharnowski, F. Buchmann, S. Wörner, and T. Holz, "A Case Study on Fuzzing Satellite Firmware," in *Workshop on the Security of Space and Satellite Systems (SpaceSec)*, 2023.

[20] M. Eisele, D. Ebert, C. Huth, and A. Zeller, "Fuzzing Embedded Systems Using Debug Interfaces," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2023.

[21] M. Pikelj and R. Heinrich. (2017) Successful Launch of German Technology Mini Satellite. [Online]. Available: https://www.airbus.com/en/newsroom/press-releases/2017-07-successful-launch-of-german-technology-mini-satellite

[22] Cobham Gaisler AB, "MKPROM2 User's Manual," https://www.gaisler.com/doc/mkprom.pdf, 2022.

[23] I. Sünter, “Design and Characterisation of Subsystems and Software for ESTCube-1 Nanosatellite,” Ph.D. dissertation, Tartu University, 2019.

[24] I. Sünter, A. Slavinskis, U. Kvell, A. Vahter, H. Kuuste, M. Noorma, J. Kutt, R. Vendt, K. Tarbe, M. Pajusalu *et al.*, “Firmware Updating Systems for Nanosatellites,” *IEEE Aerospace and Electronic Systems Magazine*, 2016.

[25] I. Sünter, “Software for the ESTCube-1 Command and Data Handling System,” Ph.D. dissertation, Tartu Ülikool, 2014. [Online]. Available: https://core.ac.uk/download/pdf/79106475.pdf

[26] T. Scharnowski, S. Woerner, F. Buchmann, N. Bars, M. Schloegel, and T. Holz, “Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs,” in *USENIX Security Symposium*, 2023.

[27] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, “Toss a Fault to your Witcher: Applying Grey-box Coverage-guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities,” in *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[28] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz, “Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities,” in *USENIX Security Symposium*, 2024.

[29] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, “IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing,” in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[30] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, “Snipuzz: Black-Box Fuzzing of IoT Firmware via Message Snippet Inference,” in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[31] C. Mulliner, N. Golde, and J.-P. Seifert, “SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale,” in *USENIX Security Symposium*, 2011.

[32] N. Schiller, M. Chlosta, M. Schloegel, N. Bars, T. Eisenhofer, T. Scharnowski, F. Domke, L. Schönherr, and T. Holz, “Drone Security and the Mysterious Case of DJI’s DroneID,” in *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[33] N. Corteggiani, G. Camurati, and A. Francillon, “Inception: System-Wide Security Testing of Real-World Embedded Systems Software,” in *USENIX Security Symposium*, 2018.

[34] M. Muench, A. Francillon, and D. Balzarotti, “Avatar2: A Multi-target Orchestration Platform,” in *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[35] M. Kammerstetter, C. Platzer, and W. Kastner, “Prospect: Peripheral Proxying Supported Embedded Code Testing,” in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[36] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory *et al.*, “SoK: Enabling Security Analyses of Embedded Systems via Rehosting,” in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2021.

[37] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, “Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing,” in *USENIX Security Symposium*, 2022.

[38] T. Gutierrez, A. Bergel, C. E. Gonzalez, C. J. Rojas, and M. A. Diaz, “Toward Applying Fuzz Testing Techniques on the SUCHAI Nanosatellites Flight Software,” in *IEEE Congreso Bienal de Argentina (ARGENCON)*, 2020.

Biography

Johannes Willbold *Johannes received his B.Sc. and M.Sc. from the Ruhr University Bochum in Germany in 2018 and 2020, respectively. He is currently a doctoral student in the systems security group, working on space and satellite systems security. His work focuses on firmware security aspects of space systems, with a recent research paper at the 44th IEEE Symposium on Security and Privacy (S&P) presenting a security analysis of LEO satellites. As subgroup chair, he is also working on transferring recent academic advances into the IEEE Standard for Space System Cybersecurity (S2CY).*

Moritz Schloegel *received his M.Sc. degree in IT security from Ruhr University Bochum. Currently, he is a systems security researcher and PhD candidate at CISPA Helmholtz Center for Information Security. He is working on automating the process of finding bugs, identifying their root cause, and assessing the severity. Beyond that, he works on scaling analysis techniques to new sectors, such as satellite security. He is contributing to the IEEE Standard for Space System Cybersecurity (S2CY) standard as a subgroup vice-chair.*

Florian Göhler *Florian received his B.Sc. from the TU Dortmund University in 2018. In 2022 he finished his M.Sc. at the Ruhr University Bochum. His Master’s thesis focused on the emulation and fuzzing of CubeSat firmware. Outside academia, Florian develops information security requirements for Germany’s public sector. Florian is also a member of the IEEE working group for the Standard for Space System Cybersecurity (S2CY).*

Tobias Scharnowski *is an embedded systems security researcher at CISPA Helmholtz Center for Information Security. He received his M.Sc. degree in IT Security from Ruhr University Bochum. In his academic research, Tobias develops scalable and efficient techniques to test (the security of) embedded systems via virtualization. Outside academia, he performed different software vulnerability assessments, including a hack of the DNP3 protocol, the technology underlying the US electric grid.*

Nils Bars *is a PhD student and systems security researcher at the CISPA Helmholtz Center for Information Security based in Saarbrücken. His research focuses on automated bug detection in software via testing techniques such as fuzzing. He received his B.Sc. degree in Applied Computer Science in 2016 from the Hamburg University of Applied Sciences and his M.Sc. in IT Security Networks and Systems in 2019 from the Ruhr University Bochum.*

Simon Wörner *received his B.Sc. from HTWG Konstanz - University of Applied Sciences in 2018 and his M.Sc. from Ruhr University Bochum in 2021. He is a security researcher and Ph.D. student at the CISPA Helmholtz Center for Information Security. His academic research is focused on scalable, automated and dynamic security testing of embedded systems firmware.*

Nico Schiller *is a security researcher and doctoral student at CISPA Helmholtz Center for Information Security. He received his B.Sc. degree in computer science in 2018 from the Bochum University of Applied Sciences and his M.Sc. in computer security in 2021 from the Ruhr University Bochum. His research focuses on the analysis and exploitation of consumer drones and he has a keen interest in fuzzing and wireless physical layer security.*

Thorsten Holz *received his M.Sc. degree in Computer Science from RWTH Aachen in 2005 and a Ph.D. in Computer Science from the University of Mannheim in 2009. He is currently a tenured faculty at the CISPA Helmholtz Center for Information Security. His research interests include technical aspects of secure systems, with a specific focus on systems security.*