A Case Study on Fuzzing Satellite Firmware

Tobias Scharnowski
Ruhr-Universität Bochum
tobias.scharnowski@rub.de

Felix Buchmann
Ruhr-Universität Bochum
felix.buchmann@rub.de

Simon Wörner
CISPA Helmholtz Center for Information Security
simon.woerner@cispa.de

Thorsten Holz
CISPA Helmholtz Center for Information Security
holz@cispa.de

***Abstract*—Satellites perform critical functions of our modern digital infrastructure, such as providing communications, navigation, and earth observation services. Maintaining a satellite requires remote access, so securing that access is an essential aspect of developing and operating a satellite. While satellites have traditionally not been subjected to regular attacks, this might not hold in the future. Hence, securing satellite firmware—the software that controls the space segment of satellite missions—becomes increasingly relevant.**

In this work, we perform a case study of applying recent embedded firmware analysis techniques to satellite payload data handling systems. We explore whether FUZZWARE, a state-of-the-art firmware fuzz testing system, can be used to these firmware images. During this case study, we also describe and apply the process of manually optimizing FUZZWARE configurations for firmware targets, and measure the impact of different optimizations. Finally, we identify challenging aspects of fuzz testing satellite firmware and directions for future work to optimize fuzz testing performance in a fully automated manner. As part of our case study, we identified and responsibly disclosed 6 bugs in 3 satellite firmware images.

I. INTRODUCTION

Satellites are at the foundation of our modern digital society. They provide critical space-born capabilities, such as telecommunication, navigation, and earth observation services. Due to their safety- and security-critical functions, keeping satellites operational and under control is essential to satellite missions. In addition to reliable hardware, a successful satellite mission also requires reliable firmware, which is the software that controls the space segment operations using a microcontroller aboard the satellite. Traditionally, malicious external actors have not been much of a practical concern to the space community [13]. Communicating with a satellite requires specialized equipment, and the intellectual property surrounding satellites has mostly remained proprietary [9], both regarding hardware and software. This made it hard for an outside party to interact with satellites, let alone attack them.

Now, the environment which used to make attacks on satellites highly improbable is changing [19]: Ground stations may now be rented [1], [10] or custom-built affordably [16]. This allows parties with limited funding to communicate with satellites. Similarly, satellites progressively rely on Commercial off-the-shelf (COTS) components [9] and open-source software libraries. In combination with a growing community, satellite systems become more accessible, and information about satellite systems becomes available. As a result, previously unrealistic attacker models become increasingly likely. With these ongoing changes, efforts to ensure a secure and protected access to satellites grow, as attackers may attempt to exploit vulnerabilities in satellite firmware [3].

These developments motivated us to test whether recent advances in the security analysis of embedded systems can be applied to the field of satellite firmware. More specifically, we attempt to apply the state-of-the-art firmware fuzzer FUZZWARE [14] to three different Payload Data Handling System (PDHS) firmware images. In our case, the PDHS controls the satellite's scientific payload equipment. In addition, we explore how involving human expertise and utilizing manual configuration options of FUZZWARE can improve the fuzzing process. We analyzed the fuzzing results and identified a total of 6 bugs, which we responsibly disclosed. Finally, we identify challenging aspects of fuzzing satellite firmware. We also discuss directions for future research that may improve fuzzing of satellite firmware in a fully automated setting.

In summary, we make the following contributions:

1) We apply FUZZWARE to fuzz testing of three different satellite PDHS firmware images.
2) We perform a case study of manually optimizing fuzzing configurations and measure their impact on different performance metrics.
3) From our experimental results we derive challenging areas of current solutions for fuzzing satellite firmware. We identify areas of possible future work to optimize the results of fully automated fuzzing of satellite firmware images.

II. REHOSTING-BASED FIRMWARE FUZZING

Embedded firmware is the software that implements the tasks that an embedded system fulfills. Embedded firmware is often purpose-built and closely tied to the embedded chip that it runs on. Firmware code communicates with its surrounding peripherals via a set of low-level communication methods, including memory-mapped input and output (MMIO), interrupts, and direct memory access (DMA).

Fuzz testing (*fuzzing*), a well-established program security testing technique, identifies flaws in a target program by feeding it large amounts of different inputs and observing its behavior. However, running firmware on its embedded device directly commonly does not provide the execution throughput required to apply fuzzing to firmware effectively. A recent line

Workshop on Security of Space and Satellite Systems (SpaceSec) 2023
27 February 2023, San Diego, CA, USA
ISBN 1-891562-89-4
https://dx.doi.org/10.14722/spacesec.2023.230707
www.ndss-symposium.org

of research [4], [14], [20] eliminates the throughput constraint by executing firmware outside their original hardware environment in a process known as *rehosting*. This is achieved by modeling the original hardware behavior and providing the resulting models to an emulator. As the resulting emulator is then able to achieve high firmware execution speeds, these works enable efficient fuzzing of embedded firmware.

III. Fuzzing Setup and Optimization

To evaluate whether current rehosting-based embedded firmware fuzzing approaches apply to satellite firmware, we tested FUZZWARE on three CubeSat firmware images: SAT-A, SAT-B, and SAT-C[1]. The corresponding satellites are in orbit at the time of writing, with SAT-B being inactive. The firmware images are built for variants of STM32F4 chips implementing the ARM Cortex-M architecture. The three firmware images are built on RODOS [11], an embedded real-time operating system. To prepare fuzzing, we aim to create a basic configuration that allows us to test our targets in FUZZWARE, and then increase fuzzing performance incrementally by applying custom, target-specific configurations. This results in multiple configuration iterations for each firmware target. An overview of the resulting configuration iterations is shown in Table I.

a) Configuration Iteration 1: To prepare the firmware images for testing in FUZZWARE, we first set up a basic configuration that includes memory mappings and interrupt triggers. To support fuzzing of SAT-A, we added target-specific base configurations. We increased execution limits to account for its computation-heavy boot process, configured a custom MMIO model which provides a correct value for the board's flash size, and schedule an additional timer interrupt to allow the firmware to finish waiting during the early boot stages. For SAT-B and SAT-C, no additional configurations were required.

b) Configuration Iteration 2: For the second configuration iteration, we manually applied a set of best-practice configuration optimizations, as suggested in FUZZWARE's project documentation [18]. First, we configure exit points in functions that indicate an unrecoverable error. We also direct the emulator to immediately return from certain functions that either output log data, or significantly delay the execution, such as busy loops. These adjustments allow the emulation to skip parts of the firmware execution that either do not contribute to or even hinder fuzzing progress. Second, we aim to further increase FUZZWARE's test case throughput by reducing the early boot stage code which the emulator has to execute for each input. We achieve this by inserting custom models for MMIO accesses which avoid consuming fuzzing input early.

c) Configuration Iteration 3: While not mentioned in the original paper, FUZZWARE provides a custom *boot* configuration option. This option allows FUZZWARE to snapshot firmware execution after the boot process. This avoids booting the firmware again for each newly tested input. With this option, the user may specify the locations in the firmware that need to be visited (and to be avoided) for firmware execution to constitute a valid boot sequence. Based on this description, FUZZWARE initially monitors the fuzzing output for inputs that trigger a valid boot sequence. Once found, it uses this input to create a snapshot of the fully booted firmware, and continues fuzzing from this snapshot. This allows the emulator to skip executing the remaining parts of the boot process for each input. Note that while requiring more insight into the firmware target, the boot configuration essentially supersedes parts of the second configuration iteration where manual MMIO access models are used to reduce the amount of early boot stage code execution. For this configuration iteration, we set the boot target location to the idle tasks of the respective firmware images. The idle task first being executed indicates that the firmware is fully booted and ready to process external input.

[1]The satellite firmware is under NDA and hence we do not disclosure the affected satellite.

TABLE I. OVERVIEW OF FIRMWARE FUZZING CONFIGURATION ITERATIONS. The table shows the number of manually assigned models (**#M**), interrupt triggers (**#IT**), whether custom execution limits have been assigned (**Lim**), number of exit hooks (**#Ex**), number of functions to skip during emulation (**#Sk**), and the facts whether a custom boot configuration (**BC**) was used, and comparison coverage was enabled (**CC**).

Configuration	#M	#IT	Lim	#Ex	#Sk	BC	CC
SAT-A-1	1	2	X	0	0		
SAT-B-1	0	1		0	0		
SAT-C-1	0	1		0	0		
SAT-A-2	7	2	X	1	7		
SAT-B-2	2	2		1	7		
SAT-C-2	2	2		1	7		
SAT-A-3	7	2	X	1	7	X	
SAT-B-3	2	2		1	7	X	
SAT-C-3	2	2		1	7	X	
SAT-A-4	7	2	X	1	7	X	X
SAT-B-4	2	2		1	7	X	X
SAT-C-4	2	2		1	7	X	X

d) Configuration Iteration 4: For the final configuration iteration, we take a different approach to adapting the fuzzing process. Instead of utilizing configuration options, we extend FUZZWARE's emulator functionality to include a previously unsupported fuzzing feature. To this end, we added patches to FUZZWARE's Unicorn Engine fork that implement the comparison coverage (*compcov*) feature from AFL++ [2], [5]. Comparison coverage alleviates a common fuzzing roadblock where a fuzzer fails to guess magic values. While 8-bit values (256 options) are reasonable for random mutations to guess, this is not the case for specific 32-bit values (four billion options). Compcov thus splits comparisons into a set of smaller 8-bit ones, which the fuzzer may discover incrementally.

IV. Experimental Results

After creating different configuration iterations, we performed fuzzing experiments based on these configurations. For each configuration iteration of each firmware image, we ran FUZZWARE for 24 hours on Intel Xeon Gold 5320 CPUs @ 2.20GHz. For each run, we assign four CPU cores. We repeat the experiment five times to account for the non-deterministic nature of fuzz testing, as recommended by Klees et al. [8].

A. Test Coverage

From the results, we collected coverage metrics and the fuzzer's test case throughput for every configuration. The results are shown in Table II. For reference, the total number of

TABLE II. FUZZING EXPERIMENT COVERAGE RESULTS

Configuration	#Covered Basic Blocks				Executions
	min	median	max	total	per second
SAT-A-1	3899	4143	4392	4450	15
SAT-A-2	4225	4225	4362	4447	20
SAT-A-3	4326	4636	4636	4680	50
SAT-A-4	4295	4295	5140	5194	40
SAT-B-1	1237	1237	1237	1237	8
SAT-B-2	3959	4096	4096	4193	165
SAT-B-3	4097	4210	4300	4343	226
SAT-B-4	4190	4303	4365	4443	284
SAT-C-1	1438	1438	1438	1438	60
SAT-C-2	3430	3682	3982	4310	88
SAT-C-3	3527	3957	4384	5320	238
SAT-C-4	3609	3793	3994	4889	245

basic blocks contained within the firmware samples are 14,647 for SAT-A, 12,971 for SAT-B, and 12,561 for SAT-C.[2]

As shown in Table II, assigning an additional set of manual configurations increases the test case throughput. Depending on the target, the throughput is increased by up to 20 times (SAT-B-2). Assigning a boot configuration further increases the test case throughput between 37% and 270%. The impact of using compcov varies by the target.

Regarding code coverage, it should be noted that skipping the execution of functions (configuration iterations 1 vs. 2) naturally decreases the possible code coverage. Despite this, the code coverage of configuration iteration 2 either stays competitive (SAT-A) or greatly exceeds the code coverage of the basic configuration (SAT-B and SAT-C). A closer inspection of the coverage increase for SAT-B and SAT-C shows that in the two cases, output logging constituted a significant roadblock for fuzzing in the base configuration. As these functions (and, therefore, these roadblocks) have been removed in configuration iteration 2, the code coverage has increased. Introducing a boot configuration (configuration iteration 3) consistently increases the total code coverage achieved. For compcov, fuzzing also sees improvements in the overall code coverage for two targets (SAT-A and SAT-B), but this seems to be a rather target-specific effect. For example, the median code coverage of SAT-A drops from 4,636 to 4,295 for the compcov emulator build (configuration iteration 4). While compcov provides additional feedback to the fuzzer, it may also divert its attention. If the low-level driver logic of a particular firmware makes heavy use of comparisons while not contributing to meaningful code exploration, compcov may result in a net loss of fuzzing progress.

B. Bug Case Studies

During our analysis of the fuzzing results, we identified 6 bugs and responsibly disclosed these issues to their respective maintainers. Two of these bugs occurred in the open-source operating system RODOS [11]. Identifying implementation issues in an operating system is interesting as multiple satellite firmware images may share its logic as a common attack surface. In addition, a space-proven operating system such as RODOS likely has undergone rigorous testing. This gives us insight into the type of issue which may evade existing testing procedures. We describe these two issues in the following.

RODOS implements `sbrk`, a function that is commonly used to extend the heap from system memory. To avoid a collision of heap memory with the stack space, the implementation performs a bounds check on the remaining available system memory. For this, the end of the stack is calculated from the stack base and a stack size constant. The stack size constant is defined in a hardware-specific linker script. Due to the way linker scripts work, access to the variable, which is defined in the linker script, must be performed via the address of the operator (which is a behavior that the linker manual mentions to be *not intuitive* [6]). In the given implementation, however, the required operator is not present. This results in the variable's value mistakenly being evaluated and dereferenced as a pointer. Generally, the stack size is not a large value, `0x1C00` bytes in this case, leading to a memory read in the lower address space. The hardware for which the analyzed firmware images are built uses these low parts of the address space to alias either flash or RAM. As a result, the accidental memory access constitutes a valid memory read, and does not cause a hard fault on the physical hardware. Instead, a wrong, too small size value is used to perform the stack size check. As a consequence of using this too small size value, the firmware logic does not correctly identify situations where heap memory collides with the stack space. A potential attacker who is able to cause memory allocations (e. g., by sending telemetry commands which require the satellite to perform bookkeeping or maintain state) may be able to first grow the heap into the stack space. Subsequently, this attacker could control contents of the system stack, and hijack program control flow on the satellite firmware. While this issue does not become visible on physical hardware, we used FUZZWARE to enforce stricter memory access controls within the emulator. This allowed us to identify the faulty memory access as a firmware crash, rather than leaving it unnoticed.

RODOS uses the priority ceiling protocol [15] to lock access to certain shared resources. This means that inside the critical section, the scheduling priority of the active thread is set to the maximum value and no other thread can be scheduled. One resource using this protocol is a synchronized FIFO for sending messages from the radio uplink thread to the telemetry thread. While reading data from the FIFO, the priority ceiling is activated. If no data is available, the reader will be suspended and yields back to the scheduler. Before this, a handle to the reader is saved in the FIFO object. At this time, the reader has entered a critical section. The logic of the reader thus assumes that the handle may not be modified by another thread. However, this assumption is broken as the reader is now suspended, such that it can no longer be scheduled. Instead, the writer will be scheduled and be able to access the shared FIFO object which is supposed to be protected by the ceiling protocol. Every time the writer puts data into the FIFO, it wakes up the reader with the handle from the FIFO object. The writer checks the handle not to be null before using it. However, as the check which the writer assumes to be guarded by priority ceiling is actually not guarded, a race condition occurs on this check. Now, the context switch may occur in

[2]These numbers may include code that is not actually reachable within the respective firmware images. Therefore, the numbers should be taken as a rough (over-)estimate.

the writer after checking the handle but before using it. Next, the reader sets the handle to null because it no longer blocks on the FIFO. Once the writer gets scheduled again, it will use the now invalid handle and cause a null pointer to dereference. While the root cause of this bug is improper usage of the priority ceiling protocol, it manifests in a race condition.

Triaging these types of bugs is cumbersome and requires a reproducible environment. This is difficult to achieve with physical hardware. Controlling the exact timings is hard, even in an emulator. This shows another advantage of a rehosting-based testing approach which is not affected by external timings. Given the same input, the result will always be the same, even with very tight timing constraints.

V. Discussion

In our experiments, we have seen that rehosting-based firmware fuzzing applies to satellite firmware, and can uncover bugs in the firmware of active satellites and in space-proven operating system code. We have also seen that the overall results of the fuzzing campaign may be improved significantly by using additional configurations. To improve fuzzing effectiveness, we manually assigned target-specific configurations. Based on the target under test, specific configurations proved more effective than others. At the same time, we have also seen some configuration options work well for one target while even reducing the quality of results for another target (e. g., enabling compcov for SAT-C).

A. Factors Contributing to Implementation Issues

Reflecting on our analysis of the satellite firmware crashes triggered by FUZZWARE, we identify different contributing factors to why different bugs existed and why rehosting-based firmware fuzzing is a good fit for finding such issues.

a) Re-use of legacy libraries and patching: We found different indications of code growing over a long period of time and over multiple iterations. Different issues seem to have occurred as an existing library was used for a new code base, and adapted to the specific mission. A change which was meant to fix a synchronization issue in a fully operational satellite introduced an issue in the boot process. The original author may have been aware of undocumented assumptions and possible side effects at the time the code was written, but this knowledge seemingly did not propagate to the later mission. As a historical reference, these occurrences seem to stem from a common broad root cause with the Ariane 5 incident, where issues with the use of a legacy library function caused a launch failure [7].

b) Run-time introspection and crash detection: On STM32F4 boards, the region around address zero is mapped. Accesses to the NULL page do not cause a visible fault when running on the physical chip [17]. Consequently, testing on the physical hardware will not uncover issues related the use of many corrupted or uninitialized pointers. An emulator, on the other hand, provides more introspection and the ability to test for faulty firmware states without modifying firmware source code. Issues pertaining to the identification of firmware memory corruptions and crashes during on-device testing have also been previously reported by Muench et al. [12].

c) Control over specific timings: When performing tests on a physical chip, especially tests involving outside inputs (which are possibly transmitted over the air), exact timings are hard to control and likely do not reproduce. This means that race conditions with a tight timing window become hard to trigger, and even harder to reproduce reliably. This is different for recent rehosting solutions, as they are built to provide the fuzzer with more control over event timings. They are also built to exactly reproduce firmware execution when feeding the same input multiple times. This makes it feasible to trigger, reproduce, and analyze issues involving tight race conditions, such as the one we have detailed in Section IV-B.

B. Challenges in Fuzzing the Satellite Firmware Target Set

Regarding the process of fuzz testing the satellite firmware images themselves, we found a set of challenging aspects.

a) Elaborate boot process: We found different satellite firmware images to perform an elaborate boot process. During fuzzing, this leads to the repeated execution of large amounts of boot-related code, and thus overall slow test case throughput speeds in default configurations. These issues could be remediated via different types of manual configuration options. However, a fully automated setup of the current version of FUZZWARE has hit a road block resulting in low overall firmware coverage in two out of three cases.

b) Varying effectiveness of specific configuration types: While particularly effective in one case, certain configuration options have proven to have less impact or even impede the fuzzing progress in other cases. Blindly applying certain types of configurations without reasoning about their effectiveness for a given target under test may impede testing outcomes, rather than improving them.

c) Interrupt timing requirements and DMA: While focusing on MMIO, current rehosting solutions pay much less attention to automatically analyzing timing requirements of the target firmware under test. As described in Section III, we manually added an interrupt configuration for SAT-A to allow the firmware to boot correctly. In a default configuration, the boot process may not have been passed as effectively. The same concept applies for DMA, a hardware communication mechanism which is currently not automatically handled by many recent rehosting systems.

C. Potential Future Research Directions

Summarizing our practical experience of this case study, we see significant value in future work around increasing the degree of automation of generating a robust and target-aware fuzzing configuration. Requiring a human expert to adjust configurations for fuzzing adds an expensive step to the firmware testing process, and makes it less feasible to utilize fuzz testing during continuous integration efforts of firmware development. Improvements around this automation could include the automated handling of certain hardware features (such as DMA). The automation may also include more efficiently dealing with the firmware boot process (especially an elaborate boot process as can be found in many satellite firmware images), or automatically applying specialized configurations and adjusting them based on their merit for a given firmware target under test.

VI. Conclusion

In this paper, we showed in several case studies that recent rehosting-based firmware fuzzing techniques apply to testing satellite firmware. We found that fuzzing results may be heavily improved via target-specific configurations, but the effectiveness of a type of configuration still depends on the target under test. From these results, we identified challenges in fuzzing satellite firmware and possible future work to improve fuzzing results in a fully automated setting.

VII. Acknowledgements

This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and by the German Federal Ministry of Education and Research (BMBF, project CPSec – 16KIS1564K).

References

[1] Amazon Web Services. (2023) AWS Ground Station. [Online]. Available: https://aws.amazon.com/ground-station

[2] 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.

[3] G. Falco and N. Boschetti, "A security risk taxonomy for commercial space missions," in *ASCEND 2021*, 2021, p. 4241.

[4] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *USENIX Security Symposium*, 2020.

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

[6] GNU Project. (2023) ld Source Code Reference. [Online]. Available: https://sourceware.org/binutils/docs/ld/Source-Code-Reference.html

[7] J. L. Lyons et al., "Ariane 5 flight 501 - failure report by the inquiry board," 1996.

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

[9] M. Manulis, C. P. Bridges, R. Harrison, V. Sekar, and A. Davis, "Cyber security in new space," *International Journal of Information Security*, vol. 20, no. 3, pp. 287–311, 2021.

[10] Microsoft Azure. (2022) Azure Orbital - Satellite Ground Station and Scheduling Services for Fast Downlinking of Data. [Online]. Available: https://azure.microsoft.com/en-us/services/orbital

[11] S. Montenegro and F. Dannemann, "Rodos - real time kernel design for dependability," *DASIA 2009 - DAta Systems in Aerospace*, vol. 669, p. 66, 2009.

[12] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[13] J. Pavur and I. Martinovic, "Building a Launchpad for Satellite Cybersecurity Research: Lessons from 60 Years of Spaceflight," *Journal of Cybersecurity*, 2022.

[14] 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.

[15] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, 1990.

[16] V. Singh, A. Prabhakara, D. Zhang, O. Yağan, and S. Kumar, "A Community-driven Approach to Democratize Access to Satellite Ground Stations," *GetMobile: Mobile Computing and Communications*, 2022.

[17] STMicroelectronics. (2023) STM32F427-437 documentation. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f427-437.html#documentation

[18] Tobias Scharnowski. (2023) Fuzzware source code repository. [Online]. Available: https://github.com/fuzzware-fuzzer/fuzzware

[19] 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 & Privacy*, 2023.

[20] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic Firmware Emulation through Invalidity-guided Knowledge Inference," in *USENIX Security Symposium*, 2021.