IFFSET: In-Field Fuzzing of Industrial Control Systems using System Emulation

Dimitrios Tychalas
Electrical and Computer Engineering
NYU Tandon School of Engineering
Brooklyn, NY, USA
dimitris.tychalas@nyu.edu

Michail Maniatakos
Electrical and Computer Engineering
New York University Abu Dhabi
Abu Dhabi, UAE
michail.maniatakos@nyu.edu

***Abstract*—Industrial Control Systems (ICS) have evolved in the last decade, shifting from proprietary software/hardware to contemporary embedded architectures paired with open-source operating systems. In contrast to the IT world, where continuous updates and patches are expected, decommissioning always-on ICS for security assessment can incur prohibitive costs to their owner. Thus, a solution for routinely assessing the cybersecurity posture of diverse ICS without affecting their operation is essential. Therefore, in this paper we introduce IFFSET, a platform that leverages full system emulation of Linux-based ICS firmware and utilizes fuzzing for security evaluation. Our platform extracts the file system and kernel information from a live ICS device, building an image which is emulated on a desktop system through QEMU. We employ fuzzing as a security assessment tool to analyze ICS specific libraries and find potential security threatening conditions. We test our platform with commercial PLCs, showcasing potential threats with no interruption to the control process.**

***Index Terms*—Industrial Control, Emulation, Fuzzing**

I. Introduction

Industrial Control Systems (ICS) facilitate controlling, monitoring, and directing physical processes in industrial environments, enabling robust automation while minimizing human dependence. The pervasiveness of ICS in industrial environments has also spread to critical infrastructure, since facilities including electric plants, desalination factories and oil refineries are equipped with such devices. Consequently, ICS are being considered as functionally critical devices since disruption in their operation could lead to revenue loss, environmental disasters or even cost human lives [1].

Given their critical function in the environments they facilitate, ICS have been produced with robustness and durability as main design principles to ensure uninterrupted functionality. However, recent events and research efforts have highlighted potential threats from the cyber domain that could compromise entire facilities through these devices [2]. Stuxnet has been the most prominent one, a malware discovered in nuclear facilities in Iran, targeting uranium enrichment centrifuges [3].

The attraction these threats have brought to critical infrastructure ICS have highlighted the need for considering the cyber security posture of these devices. However, ICS have never been designed with cyber security threats in mind, offering minimal protection in this regard. In addition, a portion of currently internet connected PLCs are being deployed with general-purpose OS-based firmware, such as Linux (Table I), creating a bridge for Information Technology (IT) threats to enter the Operational Technology (OT) domain. With that in mind, a way to assess and evaluate the security capabilities of an ICS becomes essential. These devices however, cannot be freely taken out of commission, since their function is typically continuous and should be uninterrupted. Furthermore, ICS are expensive and diverse hardware. These factors create a necessity for a non-invasive and cross-platform means for a security evaluation of ICS *in-field*.

TABLE I: Statistics for ICS with open port 2455 (TCP/UDP). Collected from Shodan [4] on 4/4/2019.

	Total	Codesys	FTP	SSH	Linux	
No of devices	9880	3118	99	494	**4.x**	535
					3.x	383
					2.x	142

Towards that, emulation has been routinely used during production and testing of desktop or embedded systems as a convenient means of evaluation. While emulation can offer the adaptable non-invasive platform, the security evaluation itself is a different component. Fuzzing, is a method utilized for discovering corner cases which could potentially lead to a compromising situation. In contemporary literature, fuzzing has been explored as a potent means of evaluating embedded systems, showcasing strong results in uncovering threats.

In this paper we introduce IFFSET, a hardware agnostic In-Field Fuzzing through System Emulation Tool, for security assessment of Linux-based ICS leveraging emulation and fuzzing to thoroughly test a target platform for security. Our platform connects to a live ICS device through regular network connections, extracts the firmware in a non-intrusive way, and sets up an emulated instance making use of the QEMU framework [5]. Combining the file system and kernel from the target device, our platform achieves full system emulation which is then leveraged to perform fuzzing on components of the device firmware. We target devices compatible with the Codesys [6] framework, fuzzing platform specific libraries that handle the loading and execution of control logic. Our platform succeeds in uncovering system threatening conditions in commodity binaries and platform-specific libraries, while exhibiting an order of magnitude improvement in fuzzing performance compared to the actual device.

978-3-9819263-4-7/DATE20/©2020 EDAA

II. Problem Formulation

The assumed scenario is as follows: A potentially vulnerable device with Linux-based firmware, controls a critical function as part of an industrial environment. The device is air-gapped in terms of access outside its local network. The device cannot be unplugged, rebooted or have its system functionality inhibited in any way. There is no direct way of communicating with the device except through the local network, via SSH or FTP, given its isolation from outside network access. The device must be evaluated in terms of security to recognize potential existing vulnerabilities in order to decide whether it can be either left as is, be updated or be removed and replaced.

III. IFFSET

This section describes the main concept behind IFFSET, our emulation framework for Linux-based PLC firmware. Firmware in the context of this paper is the combination of a general purpose OS operating as the system software and a runtime environment that handles the binaries/applications which perform the control logic. The runtime is hosted in the OS as a single application and is contained in a simple `ELF` binary along with the required dynamically loaded libraries.

A. IFFSET Development

Firmware Acquisition: In this first step, IFFSET acquires the file system along with kernel information from the target device. On a live Linux-based machine, the existing files and folders are a superset of the ones required to enable system booting. IFFSET requires two files in the `proc` folder for the extraction and preparation stages, namely `Kconfing` and `proc_version`. These files provide the necessary information, kernel configuration and current version, which direct the correct source code selection. Temporary or boot-generated folders are omitted since the static information they contain will be perceived as boot-generated by the scripts handling the process, leading to a boot failure.

Extraction & Preparation: For the firmware extraction we have utilized the customized `binwalk`-based implementation introduced by Chen et al. in 2016 [7]. Preparing the necessary files for the emulation stage IFFSET follows two main procedures: a) Builds the kernel with information collected in the acquisition and extraction steps, and b) Modifies initialization scripts to successfully boot the emulated system. The extracted configuration file includes all necessary information to build the kernel used by the device itself, including the produced image type and target file system type. Controllers for storage, network, GPIO etc. must be interfaced to the device making use of existing drivers included in the kernel. Depending on the system component of the assessment, some devices are excluded from the configuration file but a few are essential for the evaluation stage, such as watchdog timers.

Finally some modifications are made in the scripts deployed during the `init` stage of the system boot. Incorrect invocation of the `rcS` script right after the root file system is mounted leads to a kernel panic and system crash. For this case, a pre-initialization script makes sure the necessary folders are created. Depending on the assessment configuration for the target device, peripheral-wise, some lines in the `rcS` file are also removed to avoid complications to the rest of the process in addition to configuration-irrelevant script files from the `rc.d` folder. The last components to be added are the fuzzer binaries that will be utilized in the evaluation, which are compiled for the correct architecture based on information extracted through `binwalk` and can be executed natively on the emulated instance.

Emulation: QEMU is being set to utilize a single CPU core from the host system to enable multiple instances to run simultaneously. Memory and storage configurations follow typical embedded systems hardware with 256 MBs of main memory and 1 GB QEMU image for storage.

Fuzzing: We have considered both components of the firmware for the fuzzing stage of our tool. We targeted commodity binaries from the OS `bin` folder and the runtime implementing binary along with the relevant dynamic libraries. For the typical binary fuzzing sessions, we identified the binary building tool through file metadata to be the OSELAS toolchain [1], a GNU-derived development tool-chain for Embedded Linux. OSELAS is a convenient tool for binary building but it is not a part of the typical and much more exposed GNU development process, and it is only updated once yearly. This introduces an unpredictability factor to the firmware binaries since uncovered issues in the building tool have the potential to compromise the built binary which adds to the necessity of thorough testing. We chose which binaries to test based on a combination of size, complexity, and frequency of use in a typical Linux-based system.

In addition to regular binaries, we also considered platform-specific binaries, in this case the binary that instantiates the Codesys 2.3 runtime environment. This runtime is the primary facilitator for the application of control logic in the deployed ICS. We chose Codesys enabled devices based on its popularity in the industrial sector, with over 250 PLC manufacturers supporting the Codesys framework for their products. This binary is built as an `ELF` and is part of a proprietary platform, so naturally it is closed source and information on its functionality is scarce. We attempted to directly fuzz this binary making use of dynamic instrumentation techniques available through our chosen fuzzer, the American Fuzzy Lop (AFL)[2], but our attempt yielded no useful results. Thus, we turned to reverse-engineering for better understanding of the execution flow and functionality of this binary. We employed `ghidra` to extract an approximate decompiled view of the binary and, since it is a well defined `ELF` file, begun with the `main` and followed every called function. While being a self-contained binary, its primary objective is to create tasks, instantiated as threads, which are the entities responsible for performing services such as network interfacing and input/output mediation. Following the creation of threads, the binary enters a sleep cycle, being suspended most of its uptime. Since AFL does not

[1]OSELAS.Toolchain() : https://www.pengutronix.de/

[2]American Fuzzy Lop: http://lcamtuf.coredump.cx/afl/

attach itself to cloned or forked threads of the fuzzed binary, the lack of results from our simplistic fuzzing was explained. We were thus left with a single means of fuzzing the runtime binary, isolating functions from dynamically linked libraries and calling them from an self-developed binary, i.e. a test harness. From an assortment of linked shared objects we chose network and I/O related ones, since their direct contact with environment makes them more interesting exploitation targets. One of the tested functions was part of the network introduced exploitation reported at [8].

For the fuzzing results the analysis takes into account the crashes and hangs caused by various tested inputs. Crashes are the most straightforward case of the two, since they are resolved events. When a binary crashes, the system software produces an error signal along with relevant data which can pinpoint to the cause of the crash, e.g. an unidentified bug. Hangs are a different case, interesting to explore but without much to offer in information for fixing the cause of it besides the given input. However, hangs are interesting for reliability as well as availability evaluation, offering the cause of suspension for the target program execution.

IFFSET Correctness: Since we perform security evaluation on an emulated instance and not the device itself, questions about whether the security evaluation is representative naturally arise. To prove that IFFSET results are appropriate, we performed experiments to test the correlation of threat detection between the emulated system and original device. In existing literature [9], correctness has been typically affirmed by developing a program with exploits and running fuzzing on the developed platform, as well as the device itself. Therefore, in this paper, we develop a C program with 12 exploits inserted that will lead to a crash inspired by the vulnerability insertion in [10]. These exploits include stack and heap based buffer overflows, null pointer exceptions and out-of-bound reads. We used binary instrumentation and a manually chosen input seed to accelerate the vulnerability discovery process. Figure 1 illustrates the results for this correctness check. The figure shows that all 12 exploits are detected by IFFSET and by performing fuzzing directly on the PLC. Moreover, IFFSET detects the crashes much faster (which will be elaborated further experimental results), since the computer running IFFSET performs much faster compared to the embedded PLC.

IFFSET Applicability: We have developed IFFSET to be generic, scalable, and hardware agnostic so it can be used to assess any Linux-based PLC regardless of CPU architecture. Leveraging the device tree, `Kconfig` file, and system file information IFFSET adapts to support multiple peripheral configurations for diverse CPU architectures that are supported by QEMU emulation. The firmware acquisition process targets files and folders staple to Linux-based systems and in preparation IFFSET adapts files based on extracted system information rather than a fixed model. To test the applicability of IFFSET, we have successfully deployed it on a Wago PLC (ARM), a Siemens PLC (x86), and a BeagleBone development board (ARM), all featuring Linux-based OS.

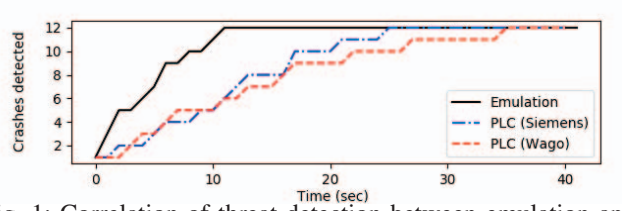

Fig. 1: Correlation of threat detection between emulation and PLC.

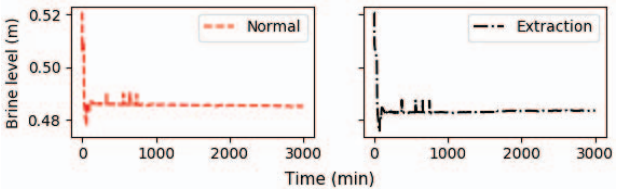

Fig. 2: Results of the desalination plant operation hosted on the PLC, during regular operation and the firmware extraction

IV. IFFSET Evaluation

For the experimental evaluation of IFFSET, we have targeted a Wago PFC-100 and a Siemens Simatic IOT2020. The computer hosting IFFSET is equipped with an i7 processor and 16GBs of RAM. As a sample process, we have integrated the PLC in a hardware-in-the-loop (HITL) simulation of a Multi-Stage-Flash desalination plant. We have chosen 20 of the most commonly used Linux utilities which receive a defined input in addition to the Codesys runtime binary/libraries. As mentioned in Section III, we chose the AFL, an open-source fuzzer which employs input mutation functionality and includes instrumentation capabilities. Our experiments focus on multiple aspects of the platform: Proof of no impact to the target device during firmware extraction, fuzzing results regarding performance, and discovered hang/crash instances. We performed 2 hour fuzzing sessions on both the system binaries and the extracted Codesys runtime libraries.

A. Non-Invasiveness

We first considered the potential impact our firmware extraction process might have on the target ICS and, by extension, the hosted control logic process. We performed firmware extraction on a live device operating with the desalination process, while monitoring a quantity directly controlled by the implemented control logic. Figure 2 illustrates the fluctuation of these values over time while the device is idle and when IFFSET is deployed, assuming that IFFSET is deployed at time = 0[3]. The figure clearly demonstrates that IFFSET is not intrusive and does not affect the process in way.

B. Fuzzing Results

Firmware Binaries: In this section, we target 20 of the most commonly used Linux binaries. Table II illustrates the results of IFFSET analysis, showcasing the number of hang and crash instances as well as executions performed in a 2 hour session.

The number of hangs is naturally larger than the number of crashes since the cause of a hung binary can also be system

[3]One second in real time corresponds to 5 minutes of simulation.

TABLE II: Hangs/crash results produced by fuzzing commodity binaries and platform specific library functions

Binary	Executions	Hangs	Crashes
bash	2181824	31	2
ls	2985317	26	1
mkdir	2458292	29	1
grep	2456325	35	1
cp	2799216	25	1
Function	**Description**	**Executions**	**Crashes**
SysSockRecv	TCP Receive	2418236	12
SysComRead	System comm.	2591513	3
GetLoginName	String Parser	2639122	15
kbus_kscock_write	I/O Write	2673199	4
kbus_ksock_read	I/O Read	2514036	3

related and the fuzzer does not wait indefinitely. A binary crash on the other hand, is only attributed to the binary itself, as a product of its own execution based on a malformed input. Regarding crashes, we have uncovered already known and patched bugs, such as [11], which have been corrected in current Linux versions but remain exploitable in the device we tested. bash exhibits the most uncovered crashes, a byproduct of its complexity compared to the rest of the tested binaries. This occurrence is also witnessed in the reported CVEs where bash has more than five times the amount of reported vulnerabilities in contrast to the other tested binaries.

Codesys runtime: Table II lists the fuzzing results on the Codesys runtime extracted functions from the utilized dynamic libraries. Out of 18 functions we tested, 14 produced crash instances, 5 of which are listed in Table II. We targeted a variety of utility functions for this experiment. SysSockRecv is the data receiving part of the TCP software stack of the runtime, the target for the aforementioned exploitation [8]. The two kbus functions are facilitators of input transfers between the control logic application and the GPIO. The results illustrate the necessity for a security evaluation of such platform-specific critical parts of the firmware as a whole. These functions are "front-facing", having direct contact with entities outside the system e.g. network, GPIOs etc. making them more eligible for exploitation. The fuzzing scheme for these functions is not perfect: we executed the runtime-loaded library functions but emulated their execution context information based on reverse-engineering. The results however are indeed worth noticing, presenting easily discovered potential vulnerabilities through a relatively simple methodology such as fuzzing.

C. Fuzzing Performance

Fig. 3 presents the fuzzing performance results for IFFSET when fuzzing with AFL. For comparison purposes, we have also performed fuzzing sessions using the Wago and the Siemens PLC; in other words, we have loaded the fuzzers on the PLC and fuzzed the binaries directly. Furthermore, given that our usage scenario assumes a desktop-grade computer performing the fuzzing, we also display the fuzzing performance when all CPU cores are utilized (4 cores/8 threads). As expected, the emulated platform far outperforms the original system. Our results demonstrate a 440% and 1520% speed-up on average among different benchmarks, for 1-core and 4-core emulated executions respectively when compared to

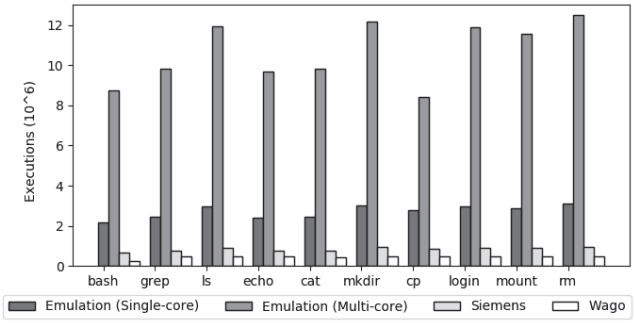

Fig. 3: Performance evaluation for cumulative executions during 2-hour fuzzing session with AFL.

the original systems. The result is not surprising since the emulated instance leverages desktop or server grade hardware which far outperform a low-power embedded device such as the ones powering ICS. Multiple instances of the same system can coexist in the same host, though, offering great scalability as seen with 4 concurrent instances with 4 utilized cores.

V. Conclusion

In this paper, we introduced a novel framework for evaluating the cyber security posture of ICS in-field. Our framework enables system emulation through QEMU and employs mature fuzzing tools on commonly used system binaries as well as platform specific libraries. We have tested our platform on commercial PLC, showcasing instances leading to binary suspensions or crashes as well as up to 15x performance gain.

Acknowledgment & Resources

This work was supported by the NYU Abu Dhabi Global PhD Fellowship program. Our platform can be found at https://github.com/momalab/IFFSET.

References

[1] H. Abdo, M. Kaouk, J.-M. Flaus, and F. Masse, "A safety/security risk analysis approach of industrial control systems: A cyber bowtie–combining new version of attack tree with bowtie analysis," *Computers & Security*, vol. 72, pp. 175–195, 2018.

[2] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno *et al.*, "Comprehensive experimental analyses of automotive attack surfaces." in *USENIX Security Symposium*, vol. 4. San Francisco, 2011.

[3] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[4] J. C. Matherly, "Shodan: The computer search engine," https://www.shodan.io/, 2009, [Online].

[5] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.

[6] D. H. Hanssen, *Programmable logic controllers: a practical approach to IEC 61131-3 using CODESYS*. John Wiley & Sons, 2015.

[7] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, 2016.

[8] SEC-Consult, "Critical CoDeSys Vulnerabilities in WAGO PFC 200 Series," https://sec-consult.com/en/blog/advisories/wago-pfc-200-series-critical-codesys-vulnerabilities/, 2012, [Online].

[9] 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 *NDSS*, 2018.

[10] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 110–121.

[11] V. Gite, "Understanding Bash fork() Bomb," https://www.cyberciti.biz/faq/understanding-bash-fork-bomb/, 2019, [Online].