Off-the-shelf Embedded Devices as Platforms for Security Research

Lucian Cojocar
l.cojocar@vu.nl

Kaveh Razavi
kaveh@cs.vu.nl

Herbert Bos
herbertb@cs.vu.nl

Vrije Universiteit Amsterdam

ABSTRACT

With increasing concerns about the security and trustworthiness of embedded devices, the importance of research on their firmware is growing. Unfortunately, researchers with new ideas for improving the security of these devices (e.g., fuzzing) or studying adversarial scenarios (e.g., malware) face massive hurdles when applying them to actual hardware. To conduct realistic experiments, we need real-world hardware that can be easily used for security research. Unfortunately, such devices are scarce and depend entirely on efforts by the hacker community. In this paper, we describe two new devices that we have opened up, a programmable logic controller (PLC) and a solid sate drive (SSD). These two types of devices have not been previously reverse engineered and they are both interesting cases given the recent developments on the security of embedded devices and the rise of Internet of Things. We discuss possible new directions with these two "real-world" research platforms. We further make the results of our efforts available to the security community in order to make it easier to get started in this research area.

1. INTRODUCTION

In the wake of Stuxnet [19] and a string of other incidents [13, 3, 24], the research community is focusing more and more on embedded devices [33, 34, 25, 10, 20, 16]. Furthermore, with the advent of the Internet of Things, the research into the security of closed-source embedded devices is becoming increasingly important.

Because embedded devices are only one part of the evaluation process, the amount of effort spent on opening them up for experimentation should ideally be smaller than that spent on designing the solution itself. Valuable research time should go to implementing or designing rather than hacking a device. Unfortunately, things are typically the other way around in practice. Researchers spend enormous effort on trying to make a device do what the researcher wants to test.

EuroSec'17, April 23 2017, Belgrade, Serbia

ISBN 978-1-4503-4935-2/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3065913.3065919

To repurpose a device for research, we need to achieve three main goals: code execution on the device, convenient firmware analysis, and a convenient debugging facility to help with the development of the research idea. There are a number of challenges that we should address to achieve these goals. First, we need to find convenient interfaces that provide us with control over code execution on the devices. While interfaces such as JTAG provide this mechanism, the mapping of the pins is not always known and sometimes difficult to reverse engineer. In these cases, reprogramming the flash chip on the device is an alternative for achieving code execution. Second, analysis of the firmware involves obtaining information about the instruction set, the load address, and so on. Debugging, finally, requires a direct communication channel with the device. Serial interfaces like UART are often included in embedded devices that provide this facility, but we still need to find the mapping of the pins.

We have addressed these challenges for two off-the-shelf embedded devices that have not been previously reverse engineered and are of significant interest to the security community: a programmable logic controller (PLC) and a solid state drive (SSD). In the case of the PLC, we modify the bootloader of the device, while on the SSD we find the JTAG mapping on the circuit board to achieve code execution. In both cases, we found an UART serial interface that in combination with our ability to execute code make it possible to start a `gdb` server on the device for debugging purposes.

There are many new applications that can make use of the capabilities that we provide on the PLC and the SSD. Remote attestation and intrusion detection on the PLC or provenance on the SSD are some examples of defensive applications, while studying backdoors and hunting for vulnerabilities are examples of offensive applications. We provide a more comprehensive list of applications of these capabilities on these two devices at the end of this paper. We further share the results of our research with the community in order to make it much faster to get started with these devices. In summary, we make the following contributions:

- To guide future efforts in this area, we systematically describe the challenges of achieving code execution and debugging on closed-source embedded devices (Section 2).
- We address these challenges for two embedded devices that are of interest to the security community and have not been previously studied: a PLC (Section 3) and an SSD (Section 4).
- We make our results accessible to the security com-

munity in order to make it easy to get started with research on these devices. We will further discuss interesting applications of the capabilities that we provide on these two devices (Section 5).

2. APPROACH

In this section, we describe the goals that we set for repurposing an off-the-shelf embedded device (OED), the challenges to achieve these goals and also a methodology for addressing the challenges.

2.1 Goals

G1: *Code execution.* The most important goal of the repurposing process is to get new code executed on the OED. Getting some code executed on the device is useful not only for prototyping a new attack scenario, but also for defenses. Moreover, combining code execution with a communication channel allows us to obtain a snapshot of the entire memory and perform subsequent firmware analysis on this snapshot.

G2: *Firmware analysis.* For prototyping software on embedded devices, we need to know the following: 1. the load address of the new code, 2. the instruction set used, and 3. (IO) address space information. We can base our analysis on a memory snapshot of the current code that runs on the OED. We can obtain a memory snapshot via debug access, or, using **G1**, via a communication channel. Analyzing a static memory snapshot rather than a firmware update (FU) has advantages: it captures the full state of the system at a certain point in time which means that we no longer need to worry about the format of FU and whether it was obfuscated or encrypted.

G3: *Debugging facility.* Once we can access the old code from the flash we can reprogram the flash with similar custom code. The trick with developing code for such embedded devices is to start small. In general, when developing new software, the first building block required is a way to debug and test the new code. For this purpose, the control of the exception vector is useful, as it allows us to control break points and inspect the cause of an unexpected exception.

2.2 Methodology

With the previous goals in mind, we divide the repurposing process in three phases: reconnaissance, code execution, and communication channel.

Phase 1: Reconnaissance. In this phase, we gather as much information as possible about the OED to be repurposed. Doing so requires reading the available user or service manuals for the targeted device. In addition, public information may be available in communities devoted to repairing such devices. Also, by taking off the device covers, it is possible to identify the main components and look for their data-sheets. Finally, a brief inspection of the available contents of flash, FU, or the update process is often helpful.

Phase 2: Code execution. The goal of this phase is to execute some valid code on the OED. First, we may start with the FU process and perform a FU modification attack [9]. Second, it may be possible to reprogram the device's persistent memory (usually flash). Third, we can potentially gain access via JTAG or other debug channel.

A good way to check whether we gained code execution is to use what is known as the *tight loop technique*: we replace one conditional jump with a `while(1);` equivalent and observe if the behaviour of the OED changes accordingly.

Phase 3: Communication channel. We can use anything observable from the outside world as a (one-way) communication channel. However, the vast majority of OEDs have a usable serial communication (universal asynchronous receiver/transmiter (UART)) connection. If we cannot find a dedicated communication channel, we can instead use a debug channel (e.g., JTAG) to emulate the communication.

2.3 Challenges

C1: *Accessing the flash.* For **G1** and sometimes for **G2** we want to get access to the storage device (usually a flash chip) that keeps the code that executes on the OED. For accessing the flash chip we have two options: 1) desolder some (or all) of the pins or, 2) keep the flash in-place. In-circuit programming of the flash chip is possible with an IC test clip that we can connect to a dedicated flash reading device. Powering up the flash while in-circuit may unintentionally power up other components too which may hamper the reading/writing process. One extra hurdle when changing the content of the flash is checksumming or crypto signatures. Bypassing correctly implemented signatures may be impossible, however, recomputing the checksum only requires more reverse engineering.

C2: *Watchdog timer interference.* While JTAG is convenient, most of the embedded devices have a watchdog timer that resets the device after a certain inactivity period. Therefore, to achieve **G1** or **G3**, the code has to explicitly write the reset value to the timer before it expires. In most of the cases, the debugging mode accessed via JTAG disables the receiving of interrupts on the targeted CPU and, in turn, the watchdog timer.

C3: *Physical signals pinout.* After the software reverse engineering process of finding some interesting IO memory address, to achieve **G3**, we need to find the physical pins that correspond to output signals. With the help of an oscilloscope, we can probe candidates for the generated signal while running the test code. The best candidates are pins that are routed to outside headers and test pads.

Duplex communication (**G3**) is harder than searching for the output signal. While identifying the send (TX) pin is easy, finding the input pin is often harder and requires trial and error. Fortunately, the RX signal is often routed on the PCB next to the TX signal (e.g., Figure 2).

Once we manage to gain code execution and have a communication channel with the OED, we can provide debugging facility via a simple library that we developed for providing a `gdb` server stub. At this point, we have successfully transformed the OED into a test platform for a new application.

3. SHOWCASE 1: PLC

Our target PLC is often found in SCADA systems. The model we considered is a Siemens `SIMATIC S7-1200`, CPU 1212C, `6ES7 212-1BE31 0XB0`, firmware version 3.1. We note that a memory disclosure vulnerability was found on another (older) version of a PLC from the same class of products[6]. The PLC has an Ethernet port, an input port, an output port, two external expansion ports and an MMC port.

3.1 Reconnaissance phase

Software. Upon installing the tools required to interact with the PLC, we had to comply to a set of software licenses which gave us hints about what software the PLC runs. While firmware updates are available for this device, the PLC ran the most recent version of the firmware available at the time and downgrading to an older version was not possible.

Hardware. We remove the plastic covers and found three stacked printed circuit boards (PCBs): a power board, an actuator board (containing relays) and the main logic board. We focus on the components from the main board. Among interesting identifiable chips, we find a 1Mbit, serial (SPI) flash, for which a data-sheet is available and also a parallel flash chip. The main board also contains two external expansion ports to which extension modules can be connected.

3.2 Code execution

To achieve **G1**, we first tried to interface with the device using JTAG. However, finding the physical mapping of signals was not straightforward. Instead, we looked into injecting our code in the FU or to reprogramming the flash contents.

Inject code via FU. The FU format is unknown and compressed with a custom algorithm. To test if an integrity check is in-place, we changed one byte from the FU and then uploaded it via the PLC's web-interface. Because the modified FU was rejected, we concluded that the firmware is checked on the device against a checksum or that the firmware is signed. At this stage, we looked for an alternative approach for gaining code execution.

Inject code via flash reprogramming. There are two flash chips present on the PLC: a small flash chip with a serial interface and a larger flash chip with a parallel interface. As the reconnaissance phase gave us information about the signal placement of the small serial flash, we now probe the clock signal of this flash with the help of an oscilloscope we probe. Doing so, we noticed that it is only in use for a couple of seconds after powering up the PLC. We further sniff the data traffic (**C1**) between the serial flash and the main SoC and encounter ARM opcodes. We conclude that this flash contains bootloading code and the larger flash contains the rest of the firmware.

For the SPI flash used by the PLC, flash write protection is enabled by asserting one pin. We avoid the power-up problem and the write protection by partially desoldering (**C1**) some of the pins. A *man-in-the-middle* like setup now allows us to quickly prototype and test a bootloader replacement. This setup enables either reprogramming the flash or in-circuit usage of the flash. The setup is capable of switching important signals (data clock, data in or chip select) of the SPI flash from the programmer to the PLC and vice-versa. Note that once the developed code is stable, there is no need for the switch.

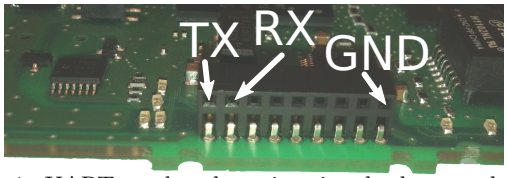

Figure 1: UART send and receive signals that can be used to interact with the PLC.

Working towards **G2**, we observed that a multi-stage bootloader is involved: the first stage of the bootloader is loaded from the SPI flash. The first stage checks the integrity of the main firmware and it further copies it to main memory (the second stage). The second stage will have to replace the exception vectors used in the first stage with new exception vectors. Now that we have access to the code that is executed in the early stage, we can inspect it and modify it to change the exception vectors of the main flash when it is being copied to memory (**G2** achieved).

3.3 Establishing a communication channel

By manual inspection of the original firmware, we now discover that an UART interface is present on this device. The default configuration for the UART is *polling mode*. In polling mode, a status flag that indicates the state of the input and output is checked until either there is space for sending data or input data is available. Polling (blocking) code is often simple to recognize (Listing 1).

Listing 1: Code for `putc` function over the UART interface

```
while (*0x1c00a62c & 0x40) ;
*0x1c00a624 = c;
```

We identify a similar code pattern as the one in Listing 1 by looking at the main firmware that we gained access to in the previous step. We now need to find which PCB tracks carry the UART signal. We reprogram the flash such that it outputs on the alleged UART port bytes from `0x0` to `0xff` in an infinite loop (**C3**). We probe some of the test-pads with our test program running and soon discover the signal that we are searching for. The discovered communication parameters are: even parity, two start bits and 38400 bps.

Because we could not find any JTAG footprint, to achieve **G3**, we (partially) implement the `gdb` protocol over serial communication. The `gdb` functionality enables dynamic analysis of the PLC's firmware which will significantly increase the number of applications of the targeted OED.

With the help of a multimeter, we discover that the send (TX) signal is routed to an external header (**C3**), while the RX signal is connected to the same external header. Figure 1 shows the partial pinout of the undocumented external header that is ready for communication.

Goal **G3** is only partially achieved at this point as the debug connection was dropped after a while, because of **C2**. Using **G2**, we locate the watchdog timer reset code by following simple heuristics, searching for slow operations (e.g.

serial communication, error logging) in the firmware and discover a write to an MMIO address with a magic (hardcoded) value. We now include this write in our `gdb` server. Whenever `gdb` is performing a blocking operation (i.e. sending a character or receiving a character) the firmware writes the same magic value to the same address. This solves the problem of spurious resets due to watchdog trigger on the PLC.

Goal **G3** is now achieved by having a stable debug connection via a custom `gdb` server that runs in the PLC from the SPI flash.

4. SHOWCASE 2: SSD

The SSD model is Crucial MX100 128 GB SATA 6Gb/s 2.5" Internal (product code: `CT128MX100SSD1`).

4.1 Reconnaissance phase

Software. Firmware updates are available for this device. At first glance, the FU seems to be cryptographically signed. Therefore, we do not try to further analyse the FU process. The SSD's user guide is not very helpful for this embedded device. However, the user guide mentions that the firmware is user upgradable and that the SSD implements hardware encryption.

Hardware. The SSD is built around the Marvell `88SS9189` controller. Apart from the main SoC, we can find NAND flash chips, one `MSP430` MCU and one serial flash (marked `25P16`). Even from online reviews [1], one can observe interesting PCB properties: an unpopulated 2x7 header, an unpopulated 1x4 header and a set of test points. The unpopulated headers give us enough confidence to select this device for repurposing.

4.2 Code execution phase

The way the traces are routed on the PCB hinted that the unpopulated 14-pin header may be a JTAG header. This is because the PCB traces are routed from the alleged JTAG header to the main SoC and a 14 pin header is a common way to connect the JTAG signals. After confirming that the unpopulated 14 pin header is indeed ARM JTAG, we easily reverse engineer the orientation of the ARM14 JTAG header by tracing the ground pins. We connect our JTAG debugger and detect two cores. We can immediately halt and resume execution on both cores. Memory is also accessible for read and write from this debug interface. This result is similar to work that has been done on a HDD by Domburg [27]. We make the `OpenOCD` configuration used to connect to the debug interface available as part of this paper.

We are now able to write in the SSD's memory our own data with `OpenOCD` commands and resume execution at any address. To test if our code is executed, we halted the core when it allegedly executed our test and observed that the program counter is bounded in our test loop. To avoid the watchdog (**C2**), the core that is not executing our test is halted during this phase.

Finding the JTAG debug port means we achieved **G1** but also **G3**. **G2** is achieved by dumping the memory via the debug interface.

4.3 Communication channel phase

Even though we could have used the already discovered debug as a **G3** communication channel (as described in Section 2.2), we choose to search for a dedicated communication channel. Emulating the communication channel through JTAG will be slow and we can easily run into race conditions. Having a dedicated communication channel also allows us to reuse existing software.

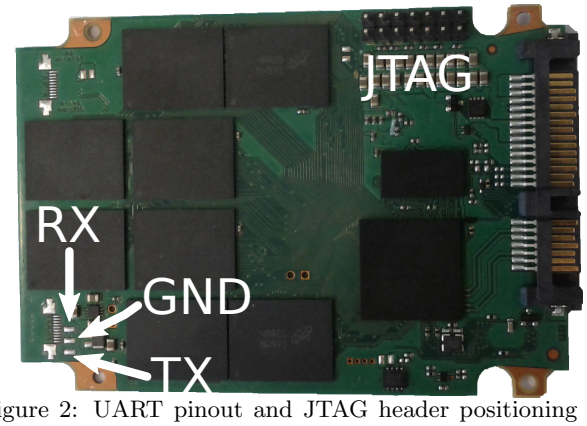

Figure 2: UART pinout and JTAG header positioning for the SSD.

Using hints provided by the error messages and the results of **G2**, we discover the IO address range that corresponds to an UART device. We face the same challenge as in the case of the PLC for finding the UART signal on the PCB and we solve it using the same approach. The communication parameters are: one stop bit, no parity, 115200 bps.

Once we discover the transmit signal mapped on an unpopulated footprint (**C3**), we have an obvious candidate for the receive signal. We note that while on the PLC the addresses of the input and output MMIO registers were the same, on the SSD these registers were mapped at different addresses.

The interesting pinout is shown in Figure 2. On the SSD we achieved code execution via JTAG and establish a communication channel via the UART port. The JTAG client can already start a `gdb` session on the SSD without any modification.

5. ENABLED SECURITY APPLICATIONS

We now list a number of examples of security projects that our open devices enable, and group them in attack and defense categories. In practice, some attack ideas (e.g. fuzzing for penetration testing) may also serve as defense. We emphasize that these are examples only and many other security-related research projects are possible. For this work, we consider non security oriented projects (e.g., reliability algorithms for industrial control systems or development of new storage scheduling algorithms) as out of scope.

5.1 Defenses

In the literature, we see many examples of offensive research on embedded devices [34, 15, 7, 36] but far fewer examples for defensive research. The main reason for this unbalance is the lack of open development platforms.

PLC. Most defenses for PLCs are based on intrusion detection [32, 17, 21] and require an external component that monitors the activity of the PLC. This paper enables deployment of these solutions on the PLC itself.

In addition, now that we can execute code on the PLC, we can add remote attestation [12, 20] and benchmark it. The software attestation layer will authenticate the PLC to the main system. The components of the attestation can be implemented in the PLC itself.

Finally, inspired by the Avatar framework [33, 18], more advanced analysis to discover vulnerabilities becomes possible when we can access memory on the device. Specifically, we can run the firmware through a symbolic execution engine and transfer memory requests from/to the real device. This is needed because the emulation environments for firmware typically do not emulate the peripherals that are attached in the real world. We can further exercise code paths that handle communications with peripherals.

SSD. Secure data deletion [23, 30] is still an open problem. Without the need for special devices or desoldering the flash chips, one can now prototype and investigate different data deletion techniques and evaluate the trade-off between security and performance.

Another open research area concerns data provenance. Provenance techniques track the usage of data. It can be used to defend against an internal adversary that wants to exfiltrate sensitive information from within an organization. The work proposed in ProvUSB [28] can be extended to storage devices through our work, while a TPM can be used to attest the validity of the SSD. The SSD can also provide provenance storage functionality.

As a final example, it is now possible to perform logging and auditing deployed at the SSD firmware. This solution will be transparent to the operating system.

5.2 Attacks

Besides defenses, a variety of new attacks are now possible.

PLC. Stuxnet [19] is the first malware that attacked the SCADA ecosystem and mainly attacked the application on the PLC. Since then, researchers proved that it is possible to implement a rootkit [5, 14, 22, 2] in the PLC's firmware.

Today's state-of-the-art fuzzers for testing PLCs are usually simple network protocol fuzzers [35, 29]. These fuzzers end up exploring the same code paths many times, thus wasting computation time. Evolutionary fuzzers, on the other hand, progress by looking at what code is reached by a new input. They then further refine this knowledge to reach unexplored paths and improve code coverage. Opening up the PLC benefits the building of a gray-box fuzzer. Better still, as the firmware may be shared across several devices, the findings will have impact on other PLCs. Moreover, by intercepting the exception handler, we can provide more information about the vulnerability (e.g. stack dump, crash location or even the instruction causing the crash) making exploit development and bug fixing easier.

SSD. With access to the mapping between physical blocks and the logical block view of the storage device, a subtle denial of service attack can be mounted on the SSD. The malware can bypass the wear leveling algorithm by reading and writing always from the same blocks, thus causing a premature malfunctioning of the device.

In addition, a very stealthy backdoor can be implemented by actively harvesting the information that is sent to the SSD and storing it information in a physical block that is hidden from normal SATA commands. The information can be exfiltrated later, for example by using a "magic" LBA read sequence that makes the hidden physical block visible. As shown by Zaddach et al. [34], the information can even be exfiltrated remotely.

6. RELATED WORK

We present related work in two categories: reverse engineering (repurposing) of embedded devices and existing dedicated open research platforms.

Reverse engineering (repurposing) OED. Zaddach et al. [34] reverse engineer and access the firmware of a hard drive. Their goal is to measure the security impact of a hard disk backdoor. They also implement a debugging channel. Cui et al. [9] and Costin et al. [8] are exploiting vulnerabilities in network printers' firmware. After the exploitation they achieve similar results to our work. We extend the current literature with two new devices as well as providing support for further research by open-sourcing our software and tools. Duflot et al. [11] reverse engineer the firmware of a network card to show the impact of an adversary that can control this device. Li et al. [20] implements software attestation at firmware level using a network card. Weinmann et al. [31] shows attacks enabled by changing the firmware of the GSM modem of mobile phones. They too needed a debugging channel to develop their exploit. Driessen et al. [10] reverse engineers satellite phones. By analyzing the firmware, they find weaknesses in the encryption algorithm used in communication.

Open research hardware platforms. While there are some options for an open PLC design (OpenPLC [4]), currently, they suffers from a low adoption rate. OpenSSD [26] platform is an open research platform that enables experimentation and benchmarking of flash storage related ideas. We note, that its resources are limited compared to real OEDs. NetFPGA [37] is another example of an open source platform designed for networking research.

Our work not only adds two new real-world, closed-source devices, but also systemically presents the challenges and a methodology for the repurposing process.

7. CONCLUSION

Recent attacks on embedded systems that run critical infrastructures are disconcerting and require immediate attention from the security community. Furthermore, with the growth of the Internet of Things, research on security of embedded devices is becoming ever more important. Unfortunately, getting started with real-world embedded systems is not straightforward due to lack of manuals and documentation. In this work, we opened up two devices that are security-sensitive and have not been investigated before: a PLC and an SSD. We managed to gain code execution on these two devices and built a convenient debugging facility for developing new ideas ranging from intrusion detection to fuzzing. To help the security community move forward with this type of research, we are open-sourcing all the tools and source code associated with these two devices: https://github.com/cojocar/embedded-reveng-research

8. ACKNOWLEDGMENTS

This research was supported by the NWO CYBSEC "OpenSesame" project (628.001.006) and by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 64457. We thank Roel Verdult for his help on choosing the right SSD.

9. REFERENCES

[1] Crucial MX100 SSD review. http://tweakers.net/.

[2] Abbasi, A. Ghost in the PLC: Stealth on-the-fly manipulation of programmable logic controllers'. BHEU'16.

[3] Advisory, C. S. CiscoWorks Internetwork Performance Monitor Remote Command Execution Vulnerability. CISCO-SA-20080313.

[4] Alves, T. R., Buratto, M., de Souza, F. M., and Rodrigues, T. V. Openplc: An open source alternative to automation. GHTC'14.

[5] Basnight, Z., Butts, J., Jr, J. L., and Dube, T. Firmware modification attacks on programmable logic controllers. Int. J. Crit. Infrastruct. Prot., 2013.

[6] Beresford, D. Exploiting Siemens Simatic S7 PLCs. *BH'11*.

[7] Breeuwsma, M., De Jongh, M., Klaver, C., Van Der Knijff, R., and Roeloffs, M. Forensic Data Recovery from Flash Memory. Small Scale Digit. Device Forensics J., 2007.

[8] Costin, A. Hacking Printers For Fun And Profit. EuSecWest10.

[9] Cui, A., Costello, M., and Stolfo, S. J. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. NDSS'13.

[10] Driessen, B., Hund, R., Willems, C., Paar, C., and Holz, T. Don't Trust Satellite Phones: A Security Analysis of Two Satphone Standards. S&P'12.

[11] Duflot, L., Perez, Y.-A., and Morin, B. What if you can't trust your network card? RAID'11.

[12] Francillon, A., Nguyen, Q., Rasmussen, K. B., and Tsudik, G. A Minimalist Approach to Remote Attestation. DATE'14.

[13] Garcia, F., de Koning Gans, G., Muijrers, R., van Rossum, P., Verdult, R., Schreur, R., and Jacobs, B. Dismantling MIFARE Classic. ESORICS'08.

[14] Garcia Jr, A. M. Firmware modification analysis in programmable logic controllers. DTIC'14.

[15] Guri, M., Solewicz, Y., Daidakulov, A., and Elovici, Y. DiskFiltration: Data Exfiltration from Speakerless Air-Gapped Computers via Covert Hard Drive Noise. ArXiv Prepr. ArXiv160803431.

[16] Halperin, D., Heydt-Benjamin, T. S., Ransford, B., Clark, S. S., Defend, B., Morgan, W., Fu, K., Kohno, T., and Maisel, W. H. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. S&P'08.

[17] Janicke, H., Nicholson, A., Webber, S., and Cau, A. Runtime-monitoring for industrial control systems. Electronics, vol. 4, no. 4, 2015.

[18] Kammerstetter, M., Platzer, C., and Kastner, W. Prospect: peripheral proxying supported embedded code testing. AsiaCCS'14.

[19] Langner, R. Stuxnet: Dissecting a Cyberwarfare Weapon. S&P'11.

[20] Li, Y., McCune, J. M., and Perrig, A. VIPER: verifying the integrity of PERipherals' firmware. CCS'11.

[21] Linda, O., Vollmer, T., and Manic, M. Neural network based intrusion detection system for critical infrastructures. IJCNN'09.

[22] Newman, T., Rad, T., ELCnetworks, L., Strauchs, J., and Strauchs, L. SCADA & PLC vulnerabilities in correctional facilities. Core Secur.'11.

[23] Reardon, J., Basin, D., and Capkun, S. Sok: Secure data deletion. S&P'13.

[24] Schneier, B. NSA Exploit of the Day, 2014.

[25] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.

[26] Song, Y. H., Jung, S., Lee, S.-W., and Kim, J.-S. Cosmos openSSD: A PCIe-based open source SSD platform. Flash Memory Summit '14.

[27] Sprite_tm. Hard disk hacking. http://spritesmods.com/?art=hddhack.

[28] Tian, D. J., Bates, A., Butler, K. R., and Rangaswami, R. ProvUSB: Block-level Provenance-Based Data Protection for USB Storage Devices. CCS'16.

[29] Voyiatzis, A. G., Katsigiannis, K., and Koubias, S. A Modbus/TCP fuzzer for testing internetworked industrial systems. ETFA'15.

[30] Wei, M. Y. C., Grupp, L. M., Spada, F. E., and Swanson, S. Reliably Erasing Data from Flash-Based Solid State Drives. In *FAST*, vol. 11, pp. 8–8.

[31] Weinmann, R.-P. Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks. USENIX Association. WOOT'12.

[32] Yang, D., Usynin, A., and Hines, J. W. Anomaly-based intrusion detection for SCADA systems. NPIC&HMIT'05).

[33] Zaddach, J., Bruno, L., Francillon, A., and Balzarotti, D. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. NDSS'14.

[34] Zaddach, J., Kurmus, A., Balzarotti, D., Blass, E.-O., Francillon, A., Goodspeed, T., Gupta, M., and Koltsidas, I. Implementation and Implications of a Stealth Hard-Drive Backdoor. ACSAC'13.

[35] Zhang, D., Wang, J., and Zhang, H. Peach Improvement on Profinet-DCP for Industrial Control System Vulnerability Detection. ECEE'15'.

[36] Zhang, L., Hao, S.-g., Zheng, J., Tan, Y.-a., Zhang, Q.-x., and Li, Y.-z. Descrambling data on solid-state disks by reverse-engineering the firmware. Digit. Investig.'15.

[37] Zilberman, N., Audzevich, Y., Covington, G. A., and Moore, A. W. NetFPGA SUME: Toward 100 Gbps as research commodity. IEEE Micro'14.