ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für
Technische Informatik und
Kommunikationsnetze

Intrusion Prevention for flexible Protocol Stacks.

Stefan Kronig

Master's Thesis MA-2013-04
March until September 2013
Advisors: Ariane Keller, Dr. Markus Happe, Dr. Stephan Neuhaus
Supervisor: Prof. Dr. Bernhard Plattner

Acknowledgements

Thanks to my project advisors, Ariane Keller, Dr. Markus Happe and Dr. Stephan Neuhaus for their help and great support during the last half year. Thanks to the TIK Communication Systems Group, directed by Prof. Dr. Bernhard Plattner, for providing me the possibility to carry out this Master's Thesis in their group. This thesis was a great opportunity to put into practise many aspects of the (mostly theoretical) knowledge I gained during my studies at ETH.

Special thanks go to my parents who have always given me good advice and the needed support during my years of study.

Abstract

In the context of the Future Internet, recent research at ETH have developed a networking platform for flexible protocol stacks with hardware- and software support. This platform is called EmbedNet and runs on a FPGA-based system on chip. Functional blocks for the networking protocol stack can be run in hardware or in software, whichever is more optimal for the current communication scenario.

As today, also in the context of the Future Internet, security will play a crucial role. In this thesis, we pay regard to the security aspect in flexible protocol stacks, by implementing an intrusion prevention functional block for EmbedNet. The presented functional block is available as a Kernel Module written in C, as well as a hardware implementation written in VHDL. The hardware version runs directly inside the FPGA and can process packets in the full line rate of of 1 GBit/s.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context of this Thesis

This thesis is in the context of the Engineering Proprioception in Computing Systems (EPiCS) Project. The goal of EPiCS is to make computing systems self-aware, in order that they can adapt to exterior influences.

In this thesis we work on the networking aspect of EPiCS, which is called EmbedNet. EmbedNet is based on Autonomic Network Architecture (ANA), a networking core intended for future Internet architectures. Instead of a static protocol stack, networking functionality is organised into specific Functional Blocks, they can be dynamically assembled to a protocol stack at run-time.

The EPiCS project maintains its own Operating System (OS), called ReconOS. ReconOS is a Linux-based OS extension which provides an abstraction of hardware resources to so-called hardware threads, which can be handled like threads or processes in the software.

The goal of this thesis is to implement an intrusion prevention block. This will be a Functional Block (FB) for EmbedNet and will be implemented in hardware and software. For the hardware version we will use ReconOS' features in order to communicate between the hardware block and the OS.

1.2 Motivation

In this thesis, we will present an Intrusion Prevention System (IPS) for flexible protocol stacks. Intrusion detection and prevention plays an important role in today's computer network architectures. Large organisations rely on Intrusion Detection and Prevention Systems (IDPSs) to protect their IT assets. Network-based IDPS analyse all network traffic which enter and leaves the the organisation's network and can detect malicious activities, independently if end nodes are separately secured or not.

New threats are arising daily in the Internet, and this is unlikely to change in the future. So when thinking about concepts and technologies for the future Internet, it is advised to think about security, too. Intrusion detection and prevention is one possible approach to satisfy this need for security, also in future Internet technologies, like flexible protocol stacks.

In recent years, a new trend towards hardware-accelerated computing can be observed, like General Purpose computing on Graphics Processing Units (GPGPU), Field Programmable Gate Arrays (FPGAs), etc. Intrusion prevention systems must handle high amount of data at the full data rate of a organisation's Internet link.

Hardware-accelerated solutions can provide more computing power and, thus, increase the overall performance of an IPS.

This thesis will present a prototype for a Network-based IPS which runs on EmbedNet, a hardware platform implementing flexible protocol stacks. This IPS will be implemented as a hardware block, was well as in software.

In order to implement a new kind of intrusion prevention, some attack had to be chosen which the IPS shall be able to detect in order to show that it works correctly. We chose to consider attacks on UTF-8 encoding. Unicode is the default encoding in HTML and XML [1][p. 1], thus its importance in today's Internet is indisputable. However, although Unicode's roots reach back to the 1990's, also in 2013 we find examples of software tools which have troubles with Unicode support[1] [2]. Or, if we look some years back, examples of vulnerabilities in widespread software which were vulnerable to UTF-8 non-shortest form attacks: Microsoft Internet Information Services (IIS) allowed an attacker to bypass authentication mechanisms by sending a non-shortest form / character in 2009 [2], or a standard function of PHP, which was vulnerable to non-shortest form attacks in the end of 2010 [3].

These examples indicate that Unicode encoding problems are still an issue in modern software and worth to take care about in a future IPS.

1.3 Software and Concepts used in this Thesis

1.3.1 Intrusion Prevention Systems

Intrusion Prevention Systems (IPSs), also known as Intrusion Detection and Prevention Systems (IDPS) are systems which monitor network traffic for malicious activities and also attempt to stop the detected activities. Network-based IDPS are able to analyse all network traffic. They do not rely on any end-node security applications, but can detect malicious activities in the network itself, by analysing traffic patterns or the the payload of the traffic (deep packet inspection).

Today's IPS are commonly realised in software and optimised for the static Transmission Control Protocols (TCPs)/Internet Protocols (IPs) protocol stack. In this thesis, we will lay the foundation for a network-based IPS running in hardware and software. Further, the IPS is not intended for legacy protocol stacks, but for a novel, flexible protocol stack, as it may be used in the Future Internet.

1.3.2 Flexible Protocol Stacks

This inflexibility, although working quite fine for the current Internet as we know it, is not well suited for all needs of communication: For instance, it reaches its functional limit for mobile and high-delay networks, and is not suited at all for delay-tolerant networking.

The basic idea of flexible protocol stacks is to emerge from this paradigm of static layers and create a new, modular network architecture. Network functionality is organised in Functional Blocks (FBs) (also referred to as network blocks or network elements in the literature) which can be freely connected to build a protocol stack according to the current communication needs. Figure 1.1 illustrates this. Such a network block can provide anything from simple operations like packet classification or checksum calculations, up to a complete networking stack. Examples for implementations of flexible protocol stacks are [4]/[5] or [6].

[1] Fedora problem with the release name "Schrödinger's Cat": `https://bugzilla.redhat.com/show_bug.cgi?id=922433`

[2] Vulnerability in the VFAT file system implementation of the Linux Kernel, when converting from UTF-8: `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1773`

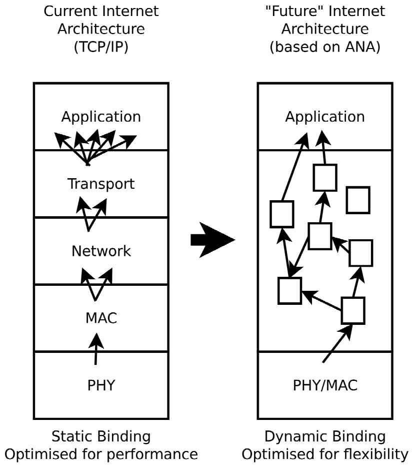


Figure 1.1: The Ideas of traditional, static protocol stacks (TCP/IP) on the left, compared to flexible protocol stacks (ANA/LANA) on the right. Figure taken from [4][p. 7].

This thesis will be implemented as a FB for a flexible protocol stack implementation.

1.3.3 The EPiCS Project

This thesis is in the context of the Engineering Proprioception in Computing Systems (EPiCS) Project [7]. EPiCS is an international multi-disciplinary research project, its goal is to make computing systems more self-aware (hence proprioceptive). Proprioceptive computing systems collect information about their state and environments and are able react to them by adapting their behaviour.

EPiCS maintains the ReconOS software, which also includes LANA, an implementation of dynamic protocol stacks developed at ETH Zurich. ReconOS and LANA will be explained in the following sections.

1.3.4 ReconOS

ReconOS [8] is a Linux-based Operating System (OS) which extends the threading programming model, known from software, to reconfigurable hardware. It is intended for the use on Systems on Chip (SoCs). The basic idea behind is to outsource resource intensive tasks to specialised hardware resources, while still maintaining the flexibility and simplicity of software threads.

The ReconOS API simplifies the access to hardware resources from the software, by providing a variety of POSIX-like functions to communicate with these so-called hardware threads, as if they were software threads: Semaphores, mutexes, shared memory, mailboxes etc. Within the hardware, the OS synchronisation state machine controls the data exchange with the software. For instance, the software may copy data to be processed into a specified shared memory location. It then passes a message to the hardware using the *mbox_put* call. The hardware state machine retrieves this message using (a hardware version of) the *mbox_get*-call and can react to it, e.g. by starting the processing of the data. In the same way, the hardware can pass a message back to the software as soon as the computation is finished. Figure 1.2 illustrates this.

In the context of this thesis, the IPS will run as a ReconOS hardware thread.

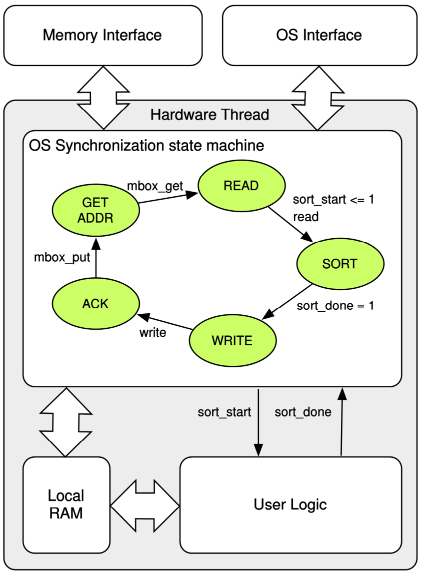


Figure 1.2: ReconOS abstracts communication to the hardware, s.t. communication with hardware threads is possible like with software threads. Figure taken from [9].

We will use mailboxes to communicate with the hardware, in order to exchange configuration messages (cf. section 3.3.1).

1.3.5 LANA

Our work is focused on the networking part of EPiCS. The networking stack included in EPiCS is Lightweight ANA (LANA) [4], an optimised version of the Autonomic Network Architecture (ANA) [5] networking stack. The EPiCS project includes LANA as its networking part.

LANA is a flexible networking core, intended for future Internet architectures. Instead of a static protocol stack, networking functionality is organised into specific FBs. Different FBs can be dynamically loaded and assembled at run-time, to build a protocol stack according to the current need of communication: One may imagine a couple of battery driven devices communicating with each other: Since they are proprioceptive, they may detect that the battery level is dropping too fast and dynamically adapt the protocol stack to a less reliable, but more energy-effective one.

1.3.6 EmbedNet

By combining the two ideas of hardware threads and flexible protocol stacks, a dynamic protocol stack can be built with support for hardware resources. EmbedNet [10] implements this idea by providing a SoC implementation of flexible protocol stacks, based on LANA. Parts of the protocol stack can be moved between software and hardware at run time. By doing so, EmbedNet enables developers to implement LANA FBs with high resource demand in hardware, and others in software.

EmbedNet provides separate data channel to transmit packets within the hardware: A Network on Chip (NoC) [11]. Two special hardware FBs, the Hardware to Software FB (h2s) and Software to Hardware FB (s2h), are used to transmit packets across the hardware-software boundary. These FBs are connected to the NoC to transmit packets within the hardware, as well as they use the the already presented ReconOS mechanisms like shared memory or message boxes to copy packet data between hardware and software. In order to send and receive packets from and to the NoC, each hardware FB is equipped with a so-called packet encoder which can send data to, resp. packet decoder which receives packets from other hardware FBs via the NoC [11].

Not only the h2s and s2h can communicate with the software, but all FBs can, as they are all ReconOS hardware threads. The difference is that h2s and s2h are meant to transfer packet data to the software, all other FBs shall transfer their packet data via the NoC.

In the context of this thesis, we will use the packet decoder and packet encoder features of EmbedNet to exchange data packets with other FBs (cf. section 3.3.1).

1.4 Introduction to Unicode

The attack presented in this thesis is based on Unicode and the way characters are stored in UTF-8. This section shall give an overview of the most relevant principles used in Unicode.

1.4.1 Unicode Characters

Unicode [1] is an international standard to represent characters. It aims to unify all of the world's writing systems and defines ways to encode multilingual text, in order to exchange text data internationally.

The standard, defines a mapping of a numeric value (so-called code point) and name for each of its characters in the Universal Character Set (UCS). It contains 1'114'112 ($\approx 2^{20}$) code points, the most common characters of all languages still used today are encoded in the first $65'536 = 2^{16}$ code points. Unicode code points are usually referred to using the *U+<hexcode>* notation, e.g. *U+0053* is the code point with the number $53_{hex} = 83_{decimal}$, the latin capital letter "S"[3].

Unicode is compatible to the ISO/IEC 10646 standard.

1.4.2 UTF-8 Encoding

Besides the mapping of characters to code points just introduced, there is a second mapping of the used code points into binary data, the so-called encoding. Unicode characters can be encoded in different forms: UTF-32, UTF-16 and UTF-8. We will only explain UTF-8, as the attack we work with is based on this encoding.

In UTF-8, characters are represented using a variable length of 8, 16, 24 or 32 bits (1, 2, 3 or 4 bytes). Early code points require less data to be stored, so 8 or 16 bits provide enough space to store them, whereas later code points need more bytes of space to be stored. Table 1.1 presents the convention how data is to be stored within the individual bytes of a UTF-8 character. The bits marked with *1* and *0* are given by the convention, while the bits market with x shall be filled with the (binary) code point number. The "max." column in the table lists the maximum length of the code point which can be encoded. The theoretic maximum of characters which can be encoded using this method is $2^{21} \approx 2$ million.

Using variable bit lengths sounds complicated at first glance, but brings the major advantage of backwards compatibility: The first $2^7 = 128$ code points (U+0000 to U+007F) were chosen s.t. they are identical to the widely used American Standard Code for Information Interchange (ASCII) character set. With their length of 7 bits, these code points are still short enough to be stored as 1-byte character, so their binary encoding in UTF-8 corresponds exactly to their encoding in ASCII.

[3] Unicode character search available at `http://www.unicode.org/charts/`.

char. length	binary representation				max.
1 byte				0xxx xxxx	7 bits
2 bytes			110x xxxx	10xx xxxx	11 bits
3 bytes		1110 xxxx	10xx xxxx	10xx xxxx	16 bits
4 bytes	1111 0xxx	10xx xxxx	10xx xxxx	10xx xxxx	21 bits

Table 1.1: Convention to encode Unicode code points in UTF-8: Bits marked with x shall be filled with the code point number.

1.5 FPGA Board and Xilinx Software used

As mentioned, ReconOS and EmbedNet are made for SoCs. In this thesis we use a SoC implemented on a Field Programmable Gate Array (FPGA). The used hardware is a Xilinx ML605[4] Evaluation board. This board is equiped with a Xilinx Virtex-6[5] XC6VLX240T FPGA. The board also provides all peripherals needed, such as a programming interface and Serial Console interface, external RAM for the OS, an Ethernet interface and a Compact Flash (CF) card reader to store Linux' root filesystem. The board has no hard-wired CPU, therefore we make use of a MicroBlaze[6] soft-core CPU.

Using an FPGA (and not an ASIC) provides the desired flexibility to reconfigure parts or all of the hardware resources needed for the current communication scenario. At present, the desired hardware configuration has to be chosen at synthesis time, but there are ambitions to make EmbedNet hardware FBs reconfigurable at run-time, which is possible with this FPGA.

To synthesise and simulate the hardware, we use the tools provided by Xilinx in the ISE Design Suite[7] version 13.3.

1.6 Related Work

In 2012, Yang wrote an AES hardware encryption FB [12] for EmbedNet. The FB was was written as a Semester Thesis at ETH Zurich. In this thesis, we write a hardware FB as well as a software FB. The AES FB is included in the current version (September 2013) of the ReconOS git repository and uses the same same interfaces to control with the outside world, as our hardware IPS FB. In the Get Started (Section A.2), we will explain how to replace the AES FB with our IPS FB.

[4] http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm

[5] http://www.xilinx.com/onlinestore/silicon/online_store_v6.htm

[6] http://www.xilinx.com/tools/microblaze.htm

[7] http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm

Chapter 2

Unicode Attacks

This chapter describes the different possibilities of attacks which are possible through Unicode and UTF-8. We will first introduce the system to be protected. Then, we provide an attack classification (Section 2.2) and choose an attack to be implemented in the IPS (Section 2.4). The chapter is concluded by providing an attacker model (Section 2.5).

2.1 Description of the System to be Protected

In Order to demonstrate and test the functionality of the IPS, we need some Software which is vulnerable to the Attacks described in section 2.4. As we use the (L)ANA Protocol stack and not a standard TCP/IP protocol stack, it is not possible to use an already existing software, it must be written by ourselves. Thus, the Software should be as simple as possible. Neverthless, the software should be somehow reusable and for demonstration purpose, the used protocols should be well-known to users.

We found that a simple web server is the ideal trade-off between these two goals. The web server should only deliver static webpages, it has no support for scripting (like CGI[1], PHP[2], or similar). It accepts so-called "Simple-Requests" as defined in HTML 1.0 [13] sends the requested file back to the user.

The web Server is protected against directory traversal attacks, using the following (naïve) approach: It searches the input string for `0x2E 0x2E 0x2F`, which is the hexadecimal representation of `../` in ASCII or the shortest form in UTF-8. It drops any requests which contain it. However, it will not search for other (UTF-8-)representations of the `../` String.

In a more real-world like scenario, one could imagine a web application which protects itself against SQL injection attacks[3] by searching all input strings for `0x27`, the shortest form representation of a single quote, but not for other representations of it.

2.2 Attack Categorisation

In the last few years, many legacy software systems were updated to support Unicode. E.g. UTF-8 has become the standard encoding for HTML files[1][p. 1], or internationalised domain names[4] were made possible. This obviously brings the

[1] Common Gateway Interface

[2] PHP Hypertext Preprocessor, `http://www.php.net`

[3] `https://www.owasp.org/index.php/SQL_Injection`

[4] `http://www.icann.org/en/resources/idn`, e.g. domains with umlauts in German.

advantages which Unicode was invented for, i.e. better and simpler internationalisation. But with this transition, also new attacks became possible by abusing UTF-8 characters.

Thus, attackers do not focus on breaking or exploiting vulnerabilities directly in UTF-8, but hiding an attack to the underlying system using Unicode encodings, by abusing faulty implementations of UTF-8 interpreters or converters. The Unicode Consortium provides a good overview of attacks and best practises when implementing Unicode and UTF-8 in [14]. When not stated otherwise, the attacks described in this chapter were found in this report [14].

2.2.1 Homograph Attack

Description and examples: The attacker sends a message to the victim, using characters which look equivalent to characters well known by the victim. Although this attack is easily detectable by a computer, a human cannot tell the difference between the original and faked message. The more User Interfaces (UIs) support Unicode, the more attack vectors are present which can be exploited. Goals of the attacker can be:

- Spamming, getting through Spam filters undetected. This method is very similar to Image Spam, i.e. Spam messages where the actual text is put into image files, in order to hide it from spam filters [15].
- Try to fool the user by presenting text which looks very similar to a well-known, trusted website or Company. Examples are links to www.paypa1.com, which uses the digit 1 instead of a lower-case L, or the following link to the website of a famous Swiss bank, www.ubs.com[5]. Latter uses a Cyrillic Dze-Letter instead of the Latin s and is not distinguishable for a human (at least not in font used in this document). This kind of attack is known as homograph attack [16].
- Circumvent security measures, e.g. by presenting the victim a forged certificate, with a name which looks like a trusted company or organisation.

Possible countermeasures: In the Internationalised Domain Names (IDN) system, characters are converted to so-called compatibility-equivalent characters [17], i.e. the most simple form a character can have.

An IPS could block all requests to URLs containing non-Latin characters, but it is highly arguable if this measure is wise or not. Especially in non English-speaking countries, such requests might well be legitimate traffic and users will feel confused or hindered if they cannot access certain certain web sites that they expect to work.

Possible attacks and impact on our system: Since our system does not use any domain names (there is no DNS support yet for LANA), we are currently not vulnerable to any attack which relies on domain names. Of course, an attacker could still send forged Links to "our" users with links to any website of our server, but when the users access the webserver, the requested file or resource will most probably just not be found, which makes this attack uninteresting for an attacker. Thus we weight the likelihood and the possible impact on our system as low.

As already stated in the motivation (section 1.2), we will present a network-based IPS in this thesis. Even if our system used domain names, a network-based IPS

[5] If you are reading this document in a PDF reader, you can try clicking on the URL and see if your browser is smart enough to inform you about the possibly fraud website.

running inside our infrastructure cannot do anything against this attack, because the attack happens on the client side. The only way to protect would be to run an IPS on “our” client’s machines, too. We do not assume this scenario for this thesis, so no countermeasures are planned either.

2.2.2 Injection of Control Characters

Description and examples: Instead of fooling the user, the attacker can also try to fool the system by encoding control characters using non-ASCII encoding. Examples for such control characters are: `.` and `/` for paths, allowing directory traversal attacks, `<` `>` for HTML (XSS attacks), and `"` `'` `;` etc. which allow different kinds of injection attacks.

There are different methods to inject control characters.

- All legacy control characters are within the ASCII character set, i.e. the first 127 code points of the UCS. So the preferred attack methods are to send these characters using UTF-16 or UTF-32 encoding, which are not backwards compatible to ASCII, or using UTF-8 with a non-shortest form representation instead of the shortest form (the non-shortest form representation will be explained in section 2.4).

  Assume a system where the user input is passed to a program or a function A, which does input sanitising and then forwards the input to program B. A does not support UTF-8 (e.g. because it is a legacy tool which only supports ASCII), but B does. If this is the case, A will check the input, but will not detect the control characters and forward the message to B. B, thus, will interpret the full Unicode input string, including the hidden control characters. By exploiting this behaviour, the attacker could successfully bypass input sanitising.

- An attacker could send invalid Unicode characters, e.g. a byte where the first 3 bits are set to ’110’ (which indicates a 2-byte character, cf. table 1.1), but not followed by a second byte beginning with ’10’. Depending on how the UTF-8 interpreter interprets this combination of invalid byte and the following, possibly valid, second byte, the attacker may hide control characters or control sequences.

  Assume the same two processes A and B, as described before: Process A, as part of its input sanitising, searches for the string “attack”. Process B just ignores (i.e. deletes) unknown or invalid characters. So, the attacker could send a string of the form “at◇tack”, where the diamond stands for the invalid character, and this string will pass to B, unnoticed by process A. Later, the diamond character is removed by process B and it becomes “attack” again.

  Similarly, the attacker can “hide” a control character in the second byte of the mentioned invalid 2-byte character: If the interpreter only replaces the first byte and ignores the others. The inverse case is also possible, the input byte can be deleted if the UTF-8 interpreter replaces all 2 bytes of the character.

Possible countermeasures: The recommended best practises according to [14] are that a converter shall substitute invalid UTF-8 bytes by the replacement character (U+FFFD), but not delete or substitute the successor byte of them. The interpretation of UTF-8 characters encoded in the non-shortest form was allowed in early Unicode versions, but is forbidden since unicode version 3.0. So, non-shortest forms must not be interpreted as valid UTF-8.

From the IPS perspective, it would make perfectly sense to block all traffic which contains requests with invalid UTF-8 characters or non-shortest forms. Such requests are either an erroneous implementation or an attack; no matter which applies, it is extremely unlikely to be legitimate traffic.

Possible attacks and impact on our system: The attacker can forge requests which contains a path containing the parent directory `../` string, thus try to perform a directory traversal attack. When he succeeds, he is able to read all the files on our system. He can thus search all files on the webserver which are readable for the www user. Such files may contain information, which is a severe privacy breach, or configuration information, which the attacker can search for weaknesses (e.g. try to crack weak passwords). Since our webserver does neither support any scripting nor accept form data sent to it, injection attacks and Cross site scripting (XSS) attacks are not a concern.

We rate the likelihood that someone attacks the system (at least tries it) using this method as high, because this attack is quite simple and the expected reward for the attacker is high.

The impact is classified medium or high, depending on what data (besides the actual public web site) the web server hosts. Even if the web server does not hold any sensitive data, a successful directory traversal attack is a privacy breach which should not happen under normal circumstances, thus we classify it at least medium.

Character	Shortest form	Non-shortest forms

U+002E “full stop”
a.k.a. dot (`.`)
	2E	`C0 AE`
		`E0 80 AE`
		`F0 80 80 AE`

U+002F “solidus”
a.k.a. slash (`/`)
	2F	`C0 AF`
		`E0 80 AF`
		`F0 80 80 AF`

Table 2.1: Representations (hexadecimal) for the control characters “Dot” and “Slash” in UTF-8, in the shortest form and non-shortest forms. These characters can be abused to perform directory traversal attacks.

2.2.3 Change of Size of Requests

Description and examples: Unicode characters typically do not have a fixed length (except UTF-32, all encodings use variable length). Typically in legacy encodings (like ASCII or ISO-encodings), the character size was fixed to 1 byte. In UTF-8, the character width is variable, a single character can be up to 4 bytes long If a developer assumed one byte per characters and only checks input for the number of characters, but not for their size, the buffers he reserved in the memory will probably not be big enough to keep all input data. So, buffer overflow attacks become possible as the attacker just needs to send enough multi-byte characters which will be counted as 1 byte, but expanded to several bytes. The effects of buffer overflow attacks reach from Denial of Service (DoS) (cause a system / application to crash) up to code injection and privilege escalation attacks. A similar attack is possible when single characters are converted to several characters in another character set. As an example, one could imagine a UTF-8 to ASCII converter which automatically converts ß (German Eszett) to *ss*, umlauts to *ae*, *oe* and *ue* or the fi-Ligature to fi.

Possible countermeasures: Programmers must reserve enough space for their string variables[6] when converting Unicode characters to other encodings, resp. stop the conversion when it contains too many characters.

As with the homograph attacks (2.2.1), an IPS could of course block such traffic. But it is also arguable if it makes sense, because the traffic might be legitimate.

Possible attacks and impact on our system: An attacker can send forged requests to our web server which contains multi-byte UTF-8 characters, to provoke a buffer overflow in the internal request handling procedures. He can then try to inject and execute his own executable code. Should he succeed, he will be in control of a process with the full access rights of the www user. This will gain him at least as much power as with the directory traversal attack described in Section 2.2.2.

We rate the likelihood of this attack medium, since provoking a privilege escalation (which implies a high reward for the attacker) using buffer overflows requires a deeper understanding of the web server's inner workings, thus it is not trivial. The attacker's risk that the attack gets noticed because the application crashes is much higher that the probability of a successful privilege escalation attack. The impact of a successful attack is similar, to a successful directory traversal attack: medium-high.

2.3 Choosing an Attack to be Implemented

Table 2.2 summarises the attacks described in the previous sections. The highest risk was assessed to the injection of control characters, in our case notably the directory traversal attack.

Attack (section describing it)	likelihood	impact	risk
Homograph attack (2.2.1)	low	low	low
Injection of control characters (2.2.2)	high	med-high	high
Change of size of requests (2.2.3)	medium	med-high	medium

Table 2.2: Risk assessment of the evaluated attacks on Unicode resp. UTF-8.

As the web server to be protected only needs to understand ASCII, the most simple form of an IPS could simply reject all traffic containing non-ASCII characters. But in these days, where Unicode is omnipresent, this may not be wanted in general.

So we chose to implement a detection of UTF-8 non-shortest form attack.This is a more generic approach, as non-shortest form representations are almost never legitimate traffic.

2.4 UTF-8 Non-Shortest Form Attack

This section explains how the UTF-8 non-shortest form attacks works, based on the UTF-8 encoding convention already explained in section 1.4.2. As one may observe from table 1.1, there is not only one possibility to encode short code points, but up to four. Code points shorter than 16 bits may be represented either in their shortest possible form, or using one of the longer representations possible, by padding the unused x's with 0. Table 2.3 illustrates this with the example of the character U+002E, "full stop".

[6] A Table with maximum expansion factors can be found in [14].

Example: U+002E, "full stop"
Code point number: $46_{decimal}$ $=2\mathrm{E}_{hex}$ $=10\ 1110_{binary}$

char. length	binary representation				hex represent.
1 byte				0010 1110	2E
2 bytes			1100 0000	1010 1110	C0 AE
3 bytes		1110 0000	1000 0000	1010 1110	E0 80 AE
4 bytes	1111 0000	1000 0000	1000 0000	1010 1110	F0 80 80 AE

Table 2.3: Example of a "full stop" character, encoded in the shortest form (first line) and the three non-shortest forms.

Only the shortest form representation is valid UTF-8, but there are implementation of UTF-8 interpreters which also (wrongly) interpret the non-shortest (a.k.a. overlong) form as UTF-8 characters, which potentially opens the door for attacks.

2.5 Attacker model

Any security measure can only protect against certain kinds of attacks and attackers. In order to reasonably define and test the functionality of the IPS, we assume the following attacker model: The attacker is an active, external attacker, who tries to break into the system by exploiting vulnerabilities. He is able to fabricate packets and send them to our system. It is assumed he knows about the webserver's vulnerabilities to injection or directory traversal attacks, using UTF-8 encoded characters.

The IPS shall protect the webserver from these kind of attacks. It does not have to protect against any other kind of attackers or attacks. For instance, the IPS need not be able to detect or protect against man in the middle attacks, denial of service attacks or any kind of cryptographic attacks.

If an attacker manages to overload an IPS, there are two possibilities: The IPS can either switch to a state where it lets all packets pass unchecked, or switch to a state in which it blocks all traffic. Latter protects the system against attacks possible by overloading the IPS but, on the other hand, also makes DoS attacks more easy. We consider a DoS the less harmful impact on our system than a successful attack on the webserver, so we choose this option.

Chapter 3

Design and Implementation

This chapter describes the design and implementation of the EmbedNet IPS FB. We will first list all design goals (Section 3.1) and then explain the actual design and implementation steps of the different IPS components (Section 3.2ff.). Section 3.6explains the detection of the UTF-8 non-shortest form attack and in Section 3.7f. the vulnerable web server and other tools will be explained.

For the sake of linguistic simplicity, we will refer to the "EmbedNet IPS FB" simply as the "IPS".

3.1 Design Goals

3.1.1 Attack to Detect

As chosen in 2.3, the IPS has to detect the 8-Bit UCS Transformation Format (UTF-8) non-shortest form attack. The attack is explained in Section 2.4.

3.1.2 Extendability

In the context of future Internet technologies like LANA/EmbedNet, new threats are likely to appear each day in the wild. So an IPS must be easy to adapt and extend. It shall be simple to add more content analysis blocks to the IPS in order to extend it. Interfaces must be clearly defined and the implemented content analysis shall be easily reusable for new implementations.

3.1.3 Hardware and Software Implementation

The IPS has to be implemented in hardware and in software, with identical behaviour. We expect the IPS to perform faster in hardware than in software. On the other hand, software resources can be reconfigured much faster and with higher flexibility than hardware. By providing both implementations, users may dynamically switch to the hardware version if the IPS is under heavy load. On the other hand, they can use the software version if only a few packets need to be inspected, and let the hardware do other tasks, e.g. de- or encryption.

3.1.4 Single Packet Operation

The IPS operates on single packets. LANA/EmbedNet provides many new possibilities and no standard protocol for connection-oriented communication (like TCP in the current Internet) is defined yet. So it makes no sense yet to try to associate various packets and inspect them together. The only thing which makes sense from today's point of view is to look at individual packets.

Each packet has to be received and checked for attacks. If an attack is found, the entire packet shall be dropped. If no attack is found, the entire packet has to be forwarded. Especially, the IPS shall not alter packets, e.g. by trying to remove or otherwise disable the attack from the packet.

3.1.5 Configurable Header Length

It may be desirable to inspect only the payload of packets. The IPS shall provide the possibility to skip a certain amount of header bytes. This number has to be configurable at run-time. The correct usage of this variable is in the responsibility of the system implementer, especially this length shall not be set to a bigger value than the size of the packet which shall be inspected.

3.1.6 Statistics and Configuration

The number of received, forwarded and dropped packets shall be counted. These counters shall be accessible by the software at run-time.

3.2 Overall System Design

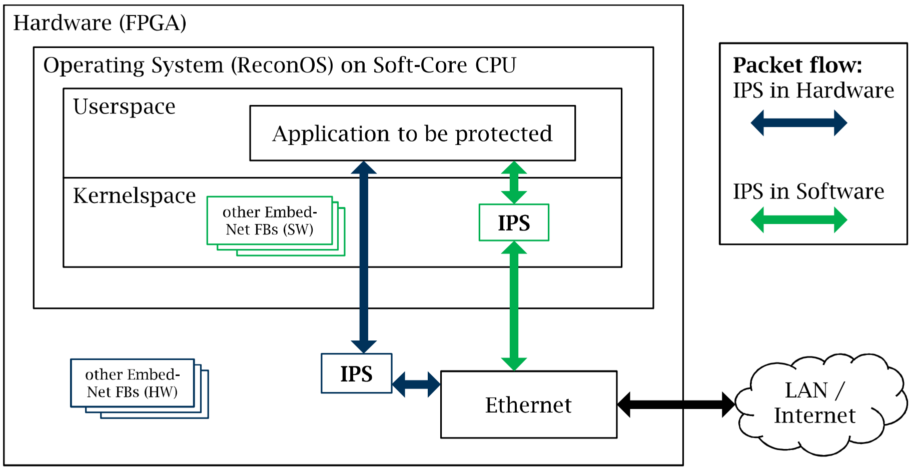


Figure 3.1: Overview of the entire system. The IPS can either be run in hardware (i.e. directly implemented on the FPGA), or in Software (i.e. running on CPU, on the FPGA). The application to be protected always runs in software.

Figure 3.1 shows an overview of the entire system. The IPS is one of several EmbedNet FBs. As explained in the Introduction (1.3.5), different FBs can be used at the same time and they can be dynamically loaded and unloaded according to the current needs of the system.

There are EmbedNet FBs which exist in hardware only (e.g. the Ethernet FB, as it must be connected to the physical Ethernet interface) or in software only, or in both. Programming a EmbedNet FB for hardware and software allows even more flexibility, as hardware-accelerated blocks can be used for load-intensive applications and software blocks for seldomly used applications. When we speak of a "Hardware EmbedNet FB" this means that the FB is implemented directly on the FPGA, using a Hardware Description Language (HDL). On the other hand, if we speak of a “software FB” it means that this FB runs as executable program code within the CPU/RAM of the FPGA. The application to be protected always runs in software.

As already stated in the design goals (goal 3.1.3), the IPS has to be done in hardware and in software. The rest of this chapter is organised in the following way: Section 3.3 explains the design of the hardware block, Section 3.4 the software block and in Section 3.6 the design of the UTF-8 non-shortest form attack detection will be explained.

3.3 Hardware IPS Design and Implementation

In this section, we will explain the hardware design using a top-down approach. The Very High Speed Integrated Circuit Hardware Description Language (VHDL) will be used as HDL to implement the hardware. This language was chosen because the existing interfaces are also written in VHDL.

3.3.1 IPS Hardware Thread

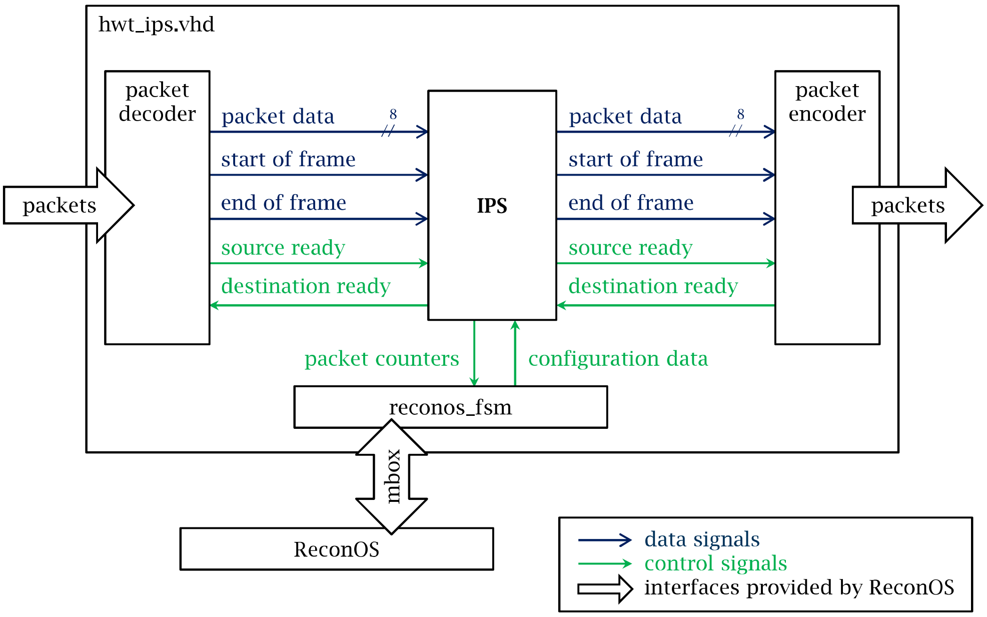


Figure 3.2: hwt_ips.vhd – The ReconOS hardware thread entity provides communication between the software and the hardware. It contains the actual IPS entity.

The IPS hardware thread entity, *hwt_ips.vhd* is the outermost entity of our design. Its layout is shown in Figure 3.2. It provides an abstraction layer to wrap the actual user logic of the IPS as a hardware thread. Communication with the software is handled by the OS synchronisation state machine *reconos_fsm*, which, in turn, makes use of ReconOS message boxes and shared memory interfaces explained in Section 1.3.4. Furthermore, the entity also takes care of receiving and sending packet data from and to other hardware threads via the NoC, using the packet decoder and packet encoder entities explained in Section 1.3.6.

Within the entity, the Xilinx LocalLink Interface [18] is used to transfer packet data. The LocalLink Interface provides point-to-point communication within Xilinx FPGAs, Xilinx Intellectual Property (IP) cores also use this Interface. Besides transferring actual data signals in form of a data bus, this interface also defines flow control signals (*source ready*, *destination ready*) and framing control signals (*start* *of frame*, *end of frame*). In our context, the *start of frame* and *end of frame* signals correspond to the start resp. the end of a network packet. We use a data bus width of 8 Bit, which means that 1 Byte can be transferred per clock cycle. The LocalLink interface provides even more signals, which we will not explain because they are not used in this thesis.

As most functionality in this entity is provided by ReconOS, there are only a few adaptions required within this entity. The obvious change is to include and instantiate one IPS entity and connect its input resp. output to the packet decoder and packet encoder. Furthermore, we need to add and connect 4 additional signals, 3 signals for packet counters and one for the header length (design goals 3.1.5 and 3.1.6). In order to send and receive these values to/from the software, also the *reconos_fsm* state machine needs to be extended.

For better clearness, the *start of frame*, *end of frame* and actual *packet data* signals will be drawn combined, as a single *packets* signal, in the following figures.

3.3.2 IPS Main Entity

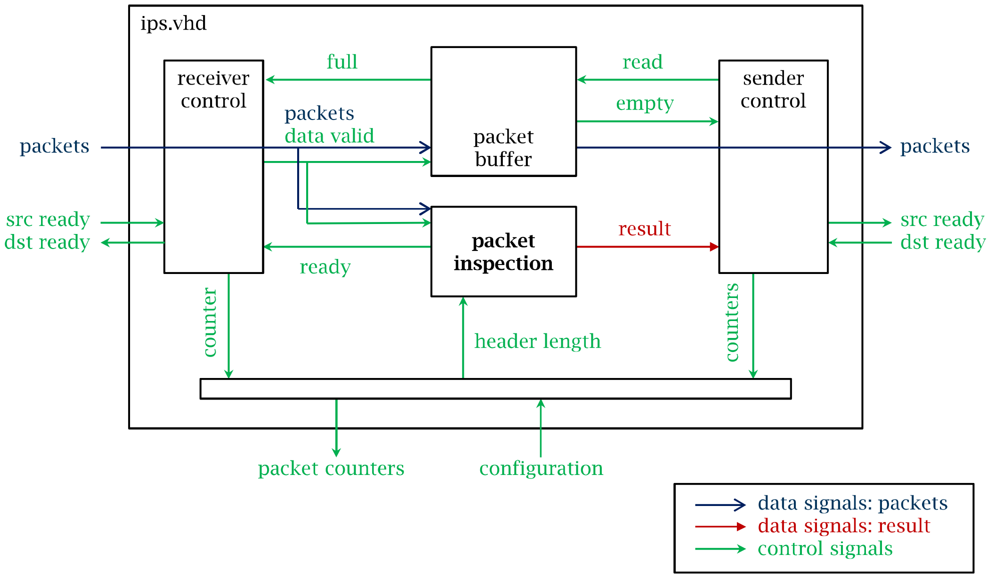


Figure 3.3: ips.vhd – The IPS Main entity

Figure 3.3 shows the main entity of the IPS. It handles the receiving of incoming packets, forwards input data to the packet inspection entity and buffers the packets while they are checked for attacks. When packet inspection is finished, it sends resp. drops the packets, depending of the result of the packet inspection. The following sections explain the mentioned parts.

3.3.3 Receiver Control

As the name says, the receiver control part is responsible to receive packets. When all submodules of the IPS are ready to process packet data, it communicates this to the sender by setting the *source ready* bit to high.

It also provides an IPS-internal *data valid* signal, used by the other submodules. This signal set to high means that the data they receive are currently valid, i.e. all other submodules are ready and the IPS is currently receiving valid data. Furthermore, receiver control counts the packets received by the IPS (design goal 3.1.6).

3.3.4 Packet Buffer

As stated in design goal 3.1.4, the IPS has to be able to drop an entire packet if it contains an attack. Even if the hardware is able to process data at line rate, as they arrive, the IPS must not forward the packet unless it knows it does not contain an attack. The worst case scenario is that the attack is "hidden" in the byte that arrives last at the IPS, so we must reserve enough space to buffer at least one packet with the maximum possible size, Maximum Transmission Unit (MTU). We use the MTU of Ethernet as a reference, which is between 1500 and 1982 bytes [19][1], and chose to implement a 2000 byte deep First In, First Out (FIFO) buffer. The FIFO enables us to process packets exactly in the order they arrived, which is desirable. Packet can be queued (received) and dequeued (either sent or dropped) at the same time (except, of course, when the queue is full resp. empty), so we implicitly apply pipelining here. Figure 3.4 illustrates the timing behaviour of multiple packets processed by the IPS. Changing the size of of this buffer in the future is possible by adapting the value of one single constant in the entire VHDL code.

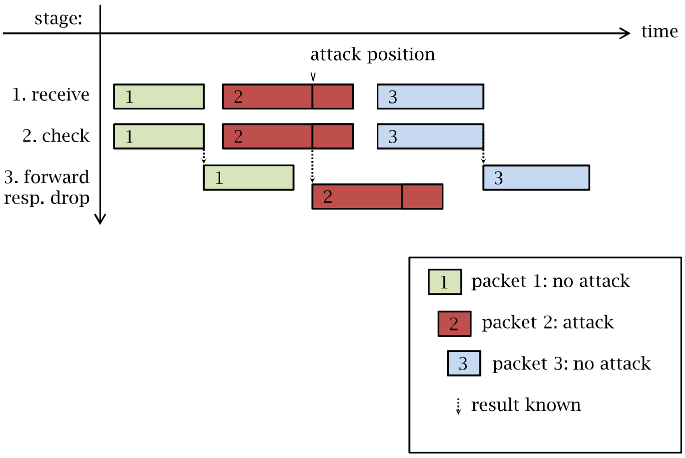


Figure 3.4: Packets can be received, processed and forwarded / dropped in parallel, by using the pipelining technique.

Besides the actual data bytes, also the *start of frame* and *end of frame signals* must be stored within this buffer, in order to clearly identify single packets. We define that all data in the buffer has to be valid packet data. This has the benefit that we don't need to explicitly buffer the *data valid* signal and there is no need to check data validity at the output of the buffer anymore. But it also implies that *receiver control* must only enqueue valid data bytes into the buffer.

Improvements made at ReconOS' FIFO Entity To implement the buffers in our design, the hardware *fifo32.vhd* entity of the ReconOS repository was used, and some adaptions were made: We implemented an overflow/underflow protection (which can be turned on or off using a generic variable). Additionally to the *fill* and *free spaces* signals, two new 1-bit signals *full* and *empty* were introduced which are set to '1' if the queue is full resp. empty. Further, the data width of the queue is no more fixed to 32 Bits, but can be set using a *generic* instruction.

The new file is included in the directory of the hardware IPS[2]. All changes are backwards compatible so it can be imported as the prototype FIFO queue to the ReconOS repository instead of the old version, if this is desired.

[1] Not considering Gigabit Ethernet Jumbo frames.

[2] path_to_cdrom/IPS-HW/hwt_ips_v1_00_a/hdl/vhdl/fifo32.vhd

3.3.5 Packet Inspection Entity

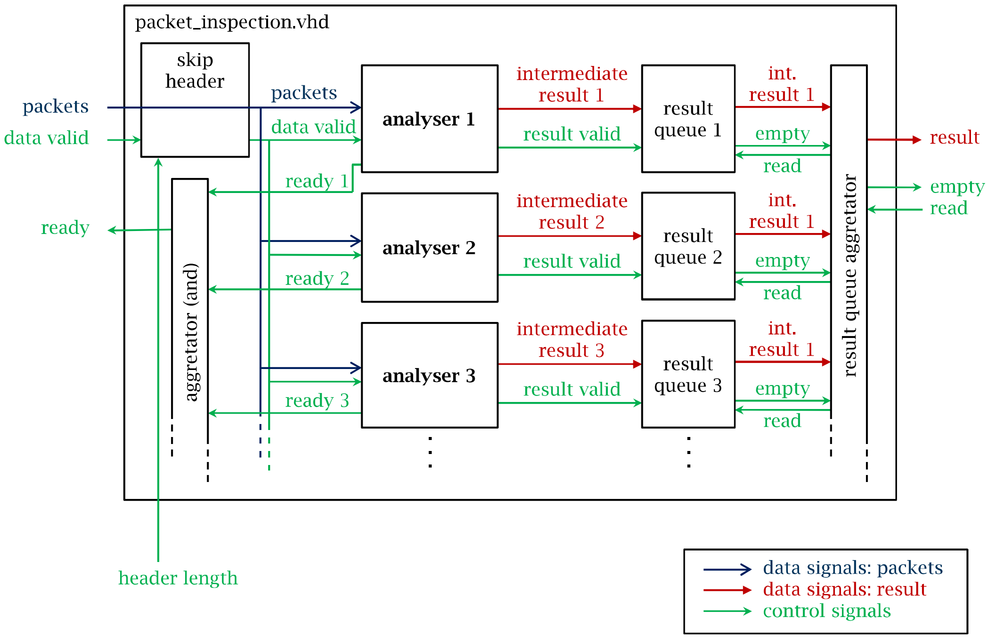


Figure 3.5: packet_inspection.vhd - The packet inspection entity distributes data to the different content analyses and aggretates their intermediate results.

The “heart” of the IPS are the content analysis blocks, which check the payload for actual attacks. This entity contains and manages the different content analyses. One instance of each content analysis is created by this entity, this means each of them gets its own hardware resources assigned at synthesis time, thus they can process the same packet in parallel. The packet inspection entity receives packets as input and broadcasts them to the different content analyses. All content analysis instances receive the packet data as input and return an intermediate result to the packet inspection entity, i.e. if they found “their” attack within the packet or not. Each of these intermediate results need to be queued in a separate queue, first because the content analyses do not necessarily finish all at the same time and secondly because we need to wait until the next higher level entity (i.e. *sender control*) is ready to process the result. From the outside, the output interface looks like one FIFO queue of results, so as soon as all intermediate results are present, they are aggregated and the next upper entity can dequeue the result from this virtual queue.

The packet inspection entity expects exactly one intermediate result for each packet. This means that the content analysis blocks must ensure that they reply each result during exactly one clock cycle, not more.

Note that the content analysers do not output any packet data, but only results. The IPS only needs to forward or drop packets (design goal 3.1.4), so no output except a result is needed from them.

Furthermore, this entity skips a user-defineable amount of (header-)bytes at the beginning of each packet (design goal 3.1.5). It manages this by overriding the internal *data valid* signal to be *low* on all bytes which don’t have to be checked.

3.3.6 Sender Control

Apart from the actual content analysis, this process is the most essential part of the IPS entity. The *sender control* process controls the data flow of the entire IPS. It retrieves the result from the *packet inspection* entity and sends or drops packets, according to the result. Furthermore, it counts the number of dropped resp. forwarded packets. The output of the IPS is connected to the packet encoder using the Xilinx LocalLink Interface, as explained in Section 3.3.1.

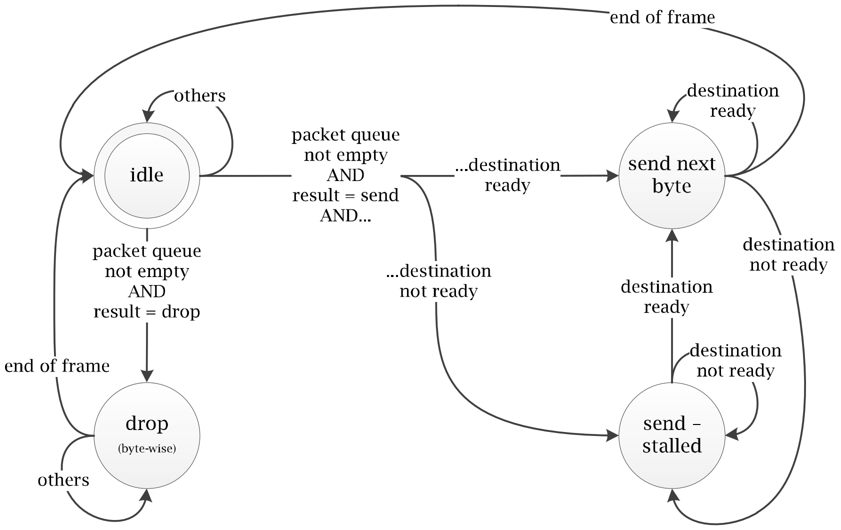


Figure 3.6: The *sender control* state machine controls the data flow of the IPS entity: It retreives the result from the *packet inspection* entity and sends or drops packets, according to the result.

Figure 3.6 shows the state diagram of the *sender control* Finite State Machine (FSM). Its initial state is the the *idle* state. In this state, the packet queue or the result queue are empty, so the sender currently cannot do anything and needs to wait for a new packet to arrive. When a packet arrives, the packet queue is being filled, but the result queue is still empty, because the result of the content analyses is not yet available. As soon as both queues contain data, *sender control* can start its work: Depending on the result, it must either forward (i.e. *send*) or *drop* the packet.

In the *drop* state, the current packet is known to contain an attack, hence it has to be dropped. “To drop” in this context means that *sender control* has to flush the current packet from the packet queue, s.t. the next packet in the queue can be accessed, but without actually sending the current packet to the output. It does this by dequeuing single bytes from the packet queue until the *end of frame* signal. The way to achieve this is by connecting the data bus directly to the output, and just by setting the *source ready* signal to low, *sender control* instructs the receiving entity that the data currently arriving is invalid and has to be ignored.

When the current packet is known not to contain an attack, it can be forwarded (sent). Sending is a bit more complex than dropping, because it may happen that the receiver is not ready while the IPS would like to send data. This is why there are two states for sending, *send_nextbyte* and *send_stalled.* As the names suggest, in the *send_nextbyte* the next byte is read from the packet queue and forwarded to the receiver, as opposed to the *send_stalled* state, where no byte must be read from the queue, since the receiver is not ready to process it.

State machine implementation The Sender Control FSM is a Mealy state machine. It is implemented using two concurrent processes, a synchronous, memorising (i.e. only executed on the rising clock edge) process and an asynchronous, memoryless (i.e. purely combinatorial) process. In the synchronous process, only the states and packet counters are updated. All other signals are updated in the combinatorial process.

3.3.7 Interface to the Content Analysis Blocks

In the hardware, each content analysis block is implemented as a separate VHDL entity. The hardware interface for the content analysis blocks is defined as follows:

```
entity ca_prototype is
  port (
    rst              : in  std_logic;
    clk              : in  std_logic;
    rx_sof           : in  std_logic;
    rx_eof           : in  std_logic;
    rx_data          : in  std_logic_vector(7 downto 0);
    rx_data_valid    : in  std_logic;
    rx_ca_ready      : out std_logic;
    tx_result        : out std_logic;
    tx_result_valid  : out std_logic
  );
end ca_prototype;
```

Listing 3.1: Interface to the hardware content analysis blocks.

All `rx_` signals belong to the LocalLink interface, explained in Section 3.3.1. These are the input of the content analysis block, used to send the packet from the to the content analysis block. `rx_data_valid` and `rx_ca_ready` are the flow control signals *source ready* and *destination ready*, whereas `rx_sof` and `rx_eof` are the framing control signals *start of frame* and *end of frame*. Figure 3.7 illustrates how a packet sent via the LocalLink interface looks like.

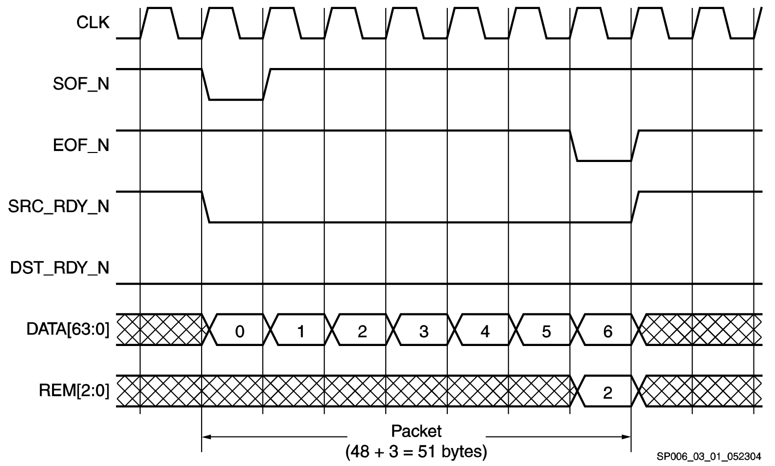


Figure 3.7: Example of a packet sent via the LocalLink interface. The REM signal is not used by our system. (Figure taken from [18, p. 31])

The `tx_` signals are the output of the content analyser, i.e. the result it has found. They are defined as

```
constant GOOD_FORWARD : std_logic := '1';
constant EVIL_DROP    : std_logic := '0';
```

Listing 3.2: Definition of constants for “good” and “evil” in hardware.

and only considered valid when the `tx_result_valid` bit is also set to ’1’. In the implementation one must consider that packet inspection entity expects exactly one intermediate result for each packet: The content analysis block must ensure it replies each result during exactly one clock cycle.

CA, in this context, stands for content analyser, RX and TX for receiver and sender.

The signals `rst` and `clk` are the reset and clock signal. The IPS is a synchronous design, i.e. there is one clock signal for the entire entity. The same applies for the reset signal: The `rst` signal is only set when the entire IPS is reset, not between single packets.

Content analysers shall not rely on a specific minimum length of packets (e.g. 64 bytes, as known from the Ethernet specification [19]), because the skipped header will not be “visible” for them. A minimum length of 1 byte per packet shall be assumed. Further, one needs to consider that the *start of frame* signal is not “visible” either when a header is skipped.

With this explanation of the content analysis blocks’ interface, we conclude the hardware IPS. The following section will explain the design and implementation phase for the software IPS.

3.4 Software Design

The software version of the IPS is based on a Linux kernel networking module, for the LANA networking stack. Many parts of the the framework were given and simplified software design, as compared to the hardware design. Sending and receiving packets is done by the kernel, we can concentrate on the actual attack detection.

3.4.1 Constructor Function

When the kernel loads a module (e.g. because the user instructed it to do so using the *insmod* command [20]), it calls the function *init_fb_ips_module*. After loading, the module can be instantiated, using the *fbctl* tool. On instantiation, the constructor function *fb_ips_ctor* is called. We don’t need any specific code to be run at loading time, only at instantiation, we need to reserve space for the counters and the header length variable. All other data or variables are only needed on a per packet basis, hence they don’t need to be reserved at this time. To simplify debugging, some feedback output may be provided to the user when the module was successfully loaded and instantiated.

3.4.2 IPS Main Function

The *netrx* function is called by the kernel each time a packet is received. This is where the IPS’ work starts. Besides other information, the kernel provides the module the starting pointer to the memory area where the current packet is stored, and its size. All packet handling can then be performed by this function.

Processing packets basically requires the following sequential steps:

1. Increses received packets counter each time the function is called.

2. Determine which part to check for attacks (skip header, design goal 3.1.5).
3. Inspect the packet for attacks.
4. Instruct kernel either to drop or to forward the packet. Increase resp. counter.

As opposed to the hardware version where all content analyses can be run in parallel, the software may only run one packet analysis at once[3]. It makes thus sense to first run the analyses which do not require much computation time and skip the remaining analyses, should an attack be found. As only one packet is processed at once, there is no need to queue the results.

After the result is known, the main function either instructs the kernel to drop the packet or returns a success value, which instructs it to forward the packet.

3.4.3 Configure Header and get Statistics

In order to configure the IPS and get information from it, we use the *procfs* interface. This interface creates a node in the file system which, when read by the user, prints a content specified in the kernel module and, when written to it, stores the written content inside the kernel module's memory. We use the *procfs* interface to display packet counters (*fb_ips_proc_show* function) or to receive a new value for the header length (*fb_ips_proc_write* function).

3.4.4 Interface to the Content Analysis Blocks

In the software, each content analysis is implemented as a C function, usually stored in a separate file.

The called function will receive two arguments:

```
int ca_prototype( unsigned char * buffer,
                  unsigned int    packet_length);
```

Listing 3.3: Interface to the software content analysis blocks.

The pointer `buffer` points directly to the start of the payload, stripping the header has already been done by the calling function. The number of bytes which have to be checked are stored in the integer `packet_length`. As with the pointer, also this is the length without the header.

The content analysis returns a value of type *int*, which is defined as 1 when the packet is "good" and 0 for "evil":

```
const int GOOD_FORWARD  = 1;
const int EVIL_DROP     = 0;
```

Listing 3.4: Definition of constants for "good" and "evil" in software.

When implementing new content analyses, do not rely on a specific minimum length of packets (e.g. 64 byte), because the skipped header will not be "visible" for the content analysis block. Assume a minimum length of 1 byte per packet.

3.5 Software Measurements

To perform the measurements of the software, some code has to be added which stores the current time stamps on different execution stages of the IPS and computes the elapsed time. In order to minimise the impact of this code on the overall performance, all variables are instantiated before the actual computation phase and

[3] The system we use runs on a single core processor

all output is made after the computation phase. The output is made s.t. the result data can be imported and processed as a Comma-Separated Values (CSV) file. This was done by printing all results which belong together, on one line, separated by semicolons[4]. To perform the measurement, the output of the serial console must be recorded (e.g. using minicom's log feature) and then evaluated.

In order to differentiate between the lines containing the CSV data and any other output which may be present on the serial console's recording (from the setup phase, other measurement output, etc.), we added keyword `[fb_ips_meas]` to each line containing measurement data, so they can be filtered for this keyword.

Further, there shall not be any unneccessary commands during the measurement phase. Especially there shall be no `printf` commands, since they need a lot of time (data must be sent over a slow serial terminal) and would distort the measurement.

This concludes the explanations of the software IPS. The following section will explain how to detect the UTF-8 non-shortest form attack.

3.6 UTF-8 Non-Shortest Form Detection

In this section, we explain the detection of the UTF-8 non-shortest form attack. Table 3.1 shows how UTF-8 characters are represented as binary data. For a general explanation of how UTF-8 works, please refer to Section 1.4.2 of this document. As explained there, UTF-8 stores characters using 1, 2, 3 or 4 bytes. Part a) of the table shows these different possibilities: Line 1 shows a 1-byte character, line 2 a 2-byte character, etc. The bytes which are available for encoding bits of the code-point number are marked with an x in the table and the number of possible bits is marked in the rightmost column.

a)

1 byte	0xxx xxxx	7 bits
2 bytes	110x xxxx 10xx xxxx 7th bit	11 bits
3 bytes	1110 xxxx 10xx xxxx 10xx xxxx 11th bit	16 bits
4 bytes	1111 0xxx 10xx xxxx 10xx xxxx 10xx xxxx 16th bit	21 bits

b)

1 byte	0xxx xxxx	7 bits
2 bytes	1100 000x 10xx xxxx 7th bit	11 bits
3 bytes	1110 0000 100x xxxx 10xx xxxx 11th bit	16 bits
4 bytes	1111 0000 1000 xxxx 10xx xxxx 10xx xxxx 16th bit	21 bits

Table 3.1: Binary representation of Unicode characters in UTF-8. Underlined in table b) are the bits which are 0 when a non-shortest form attack is present.

[4] Although the format is called *Comma*-separated values, we use semicolons, since the comma is too ambigous and could be mistaken as a decimal point or 1000er separator.

An overlong representation of a character is present, if a character which is represented using n_1 bytes could also have been represented using $n_2 < n_1$ bytes. For instance, a code point ≤ 127 only needs 7 bits, i.e. it is possible to represent it using a 1-byte character. If the same character is represented using 2 bytes, only the last 7 x are filled with the code point number and all the remaining x (on the left) are set to 0, as underlined in line 2 of table b). The same line of thought was applied to code points which require 11 and 16 bits, and can thus be represented using 2 or 3 bytes, as marked in line 3 and 4 of table b).

One may observe that all relevant information, if a UTF-8 non-shortest form attack is present or not, can be found within the first two bytes of the character, no matter what the values of the other x are. The corresponding algorithm is shown in Listing 3.5. Note the first statement in the switch structure: Ignoring all latter bytes of a multibyte character is possible, since the first byte always must have been checked. Of course, other kind of attacks can be hidden in such a packet, but no non-shortest form attack.

To implement this in hardware, the pseudocode can be translated into the state machine shown in Figure 3.8. Note that every time the algorithm needs to check the following byte, an additional state is needed in the FSM since the data arrive bytewise in the entity. When the end of the packet is reached (indicated by the *end of frame* bit), the FSM returns the result: If it is still in in the *good* state, no attack was found. In the *evil* state, an attack was found and in the two remaining states, marked in yellow, the FSM cannot certainly decide wether there is an attack or not, because there is a byte of information missing in order to decide.

The UTF-8 non-shortest form detection FSM is a Mealy state machine and is implemented the same way as explained for the sender control FSM, in Section 3.3.6.

```
for each byte in the packet, switch:


  The byte is "10xx xxxx":
    Byte is the latter byte of a multy-byte character.
    Can be ignored, the first bytes have already been checked.


  The byte is "0xxx xxxx":
    Byte contains a 7-bit ASCII character, is ok.
    If this was the last byte of the packet:
      Classify packet as good!


  The byte is "110x xxxx":
    Byte is the first byte of a 2-byte character.
    if byte is "1100 000x":
      This is a 7-bit character represented with 2 bytes ...
            instead of 1.
      Classify packet as evil!
      Quit for-loop, there is no need to check more bytes.

    else:
     A regular 2-byte character, is ok.
      If this was the last byte of the packet:
    Classify packet as good!

```

```

The byte is "1110 xxxx":
  Byte is the first byte of a 3-byte character.
  This character can be up to 12 bits long.
  Need to check the following byte.
  If this was the last byte of the packet:
    No classification possible!

  else if the following byte is "100x xxxx":
    This is a 11 bit character represented with 3 bytes ...
          instead of 2
    Classify packet as evil!
    Quit for-loop, there is no need to check more bytes.

  else:
    This is a regular 3-byte character
    If this was the last byte of the packet:
      Classify packet as good!


The byte is "1110 xxxx"
  Byte is the first byte of a 4-byte character.
  This character can be up to 18 bits long.
  Need to check the following byte.
  If this was the last byte of the packet:
    No classification possible!

  else if the following byte is "1100 xxxx"
    This is a 16 bit character represented with 4 bytes ...
          instead of 3
    Classify packet as evil!
    Quit for-loop, there is no need to check more bytes.

  else:
    This is a regular 4-byte character
    If this was the last byte of the packet:
      Classify packet as good!
```

Listing 3.5: Pseudocode of the UTF-8 non-shortest form detection.

3.7 Vulnerable Application: Web Server

As introduced in Chapter 2.1, a simple web server was implemented. To protect against directory traversal attacks, this web server performs input validation in a naïve way, by rejecting all requests which contain the `../` string. An attacker may pass this naïve check by sending `../` using non-shortest forms (this is where the IPS will jump in).

The success of the attack is based on the fact that the victim software interprets the UTF-8 non-shortest form, i.e. internally converts it to the shortest form or to a legacy character set like ASCII, after the input validation has been done. For instance, we could imagine our web server which only performs the naïve check described above, and an implementation of the file system driver which internally converts non-shortest forms to shortest forms.

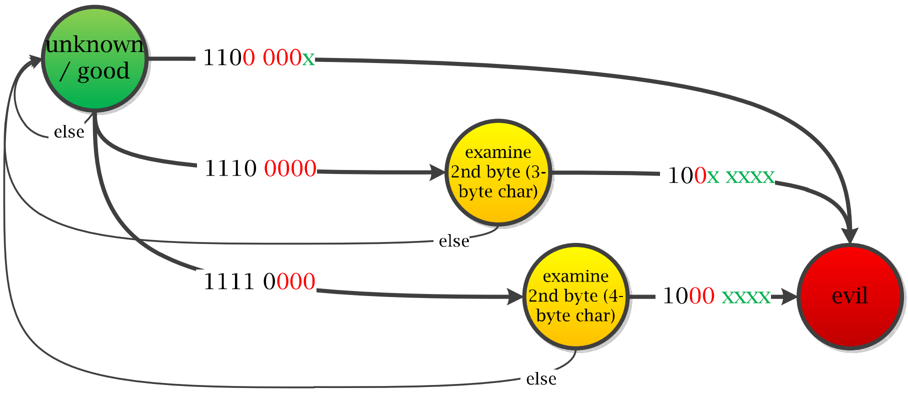


Figure 3.8: Simplified version of the UTF-8 non-shortest form detection FSM: When the *end of frame* is reached, the FSM returns the result, according to the current state: No attack (green), attack found (red), no certain decision possible (yellow).

As we were not able to find common software which has this vulnerability (luckily), we implemented the naïve check and a UTF-8 to ASCII conversion in the web server by ourselves. The web server deliberately performs the two tasks in the wrong order to simulate the behaviour of a vulnerable software, i.e. it does

```
utf8−to−ascii(input−validation(input));
```

instead of

```
input−validation(utf8−to−ascii(input));
```

We are perfectly aware of the fact that writing software which is deliberately vulnerable is not a good thing per se, but as the primary goal of the web server is to provide a demonstrator how the attack works and that the IPS works, we considered it passable in this particular case.

3.8 Tools

In the course of this thesis, we wrote several scripts. A scapy-based script which sends different kind of packets for measurement can be found in at the resp. measurement in the *measurements* folder on the CD-ROM.

Besides this, we wrote two scripts which can simplify the life of a developer. Some commands which have to be added to the *.bashrc* file, in order to run the Xilinx tools (as described in the Getting Started Guide in Appendix A), seem to confuse certain KDE programs. Thus, we wrote a small *fpga-tools* script which creates a temporary *.bashrc* file and launches a new bash shell with this temporary *.bashrc* file. The lines to add to your *.bashrc* can be configured in a separate configfile. Should you experience any problems with programs after editing your *.bashrc* file, this script is a valuable alternative.

The second script we'd like to mention is a script which generates a Bitfile for the current state of your VHDL design. As hardware synthesis is a quite time-consuming process, you may run the Bitfile generation in background and continue working on your original files in the meantime. Multiple instances of the script can run at the same time (provided your workstation has enough RAM), e.g. if you't like to try the same hardware, but with some different variables or parameters. The script, named *synthetisise_current_state*, copies the current contents of your ReconOS Git repository to a temporary folder and then starts the synthesis. The finished bitfile will be copied back to a folder of your choice. Further, you may enter a comment for each run s.t. you will be able to find the correct bitfile, even if you start multiple instances of the script.

Chapter 4

Validation and Evaluation

This chapter describes the validation and evaluation of the IPS. Section 4.1 describes the validation, i.e. how the correct functionality of the IPS was verified. Section 4.2 describes the evaluation and measurement methodology and 4.3 presents the results. An interpretation will be provided in Section 5.1 of the following chapter.

All measurement scripts and data can be found on the attached CD-ROM in the folder *measurements*.

4.1 Validation

To verify the correct functionality of the IPS, known test packets were sent to the IPS. As the IPS shall not alter the content, the (known) input packet must arrive at the output either unchanged or not at all. Should an altered packet arrive at the output, then there is certainly an error. We first developed and tested the IPS without any content analyses present, by manually generating the resp. good and evil signals. The outcome was compared by manually checking the output packet and forwarded/dropped packet counters.

4.1.1 Test setup

The hardware was developed with the help of iSim[1], a VHDL simulator software by Xilinx provided within the ISE design Suite. Most of the validation could be done within this simulator.

As soon as the IPS behaved correctly in the simulation, the correct functionality in the FPGA was verified: The IPS was synthesised and configured s.t. it forwards all packets to the h2s interface (explained in Section 1.3.3) provided by ReconOS. Packets were generated by a Linux Workstation, sent via Ethernet, to the IPS and then to the s2h. The software part of the s2h was configured s.t. it outputs all packet content on the console.

The software IPS was validated in a very similar way: After compiling and loading the software, packets were sent from the Linux Workstation, via Ethernet, to the IPS and then received by an application which prints them on the console.

So both, the software version as well as the hardware version, can then be checked for correctness by inspecting the console output of ReconOS, a technique usually used in software development only.

[1] `http://www.xilinx.com/tools/isim.htm`

4.1.2 Validation of the Attack Detection

To verify the correct functionality of attack detection, we used the methods and test setups introduced above and sent packets containing malicious characters. The most obvious solution would be to send all possible UTF-8 characters, including non-shortest forms, and check the result. As it is not feasible to check all possible attack using visual inspection though, we decided to send a character of each possible bit length. A condensed list of characters used for verification is showed in Table 4.1, the lines marked with *nsf* contain non-shortest forms (the complete lists can be found in appendix B.1). Further, by configuring the value for the header length and altering the position of the attack within the packet, we verified the correct functionality of the “skip header” feature.

4.1.3 Validation Result

Hardware and software work as specified, except for a bug in the hardware which causes the first byte to arrive twice. In the given time frame, we were neither able to reproduce this bug in the simulator, nor to find out what caused it, or to find a workaround which fixes it. See Section 5.3 for details.

4.2 Evaluation Methodology

4.2.1 Test setups

Hardware The hardware is discussed theoretically, with the help of the previously introduced iSim simulator. To do this, a testbench was created which can send packets to the IPS' input as well as receive its output. The testbench is at the same hierarchy level as the hardware thread (cf. 3.3.1) in the final design.

Software - best case For the software, we provide a best-case estimation by measuring the processing time for the content analysis only. From this, we deduce the maximum possible throughput.

As we only measure the actual processing time, it does not matter if the packets come from the software within the FPGA or are sent from the Linux Workstation to the FPGA. For sake of flexibility we decided thus to use the Linux Workstation to send the packets.

Software - real world scenario By performing the same measurement not only for the content analysis, but for the processing time of several packets, we further estimate the throughput for a real-world scenario. For this measurement, we measured the time needed to process 1000 packets and deduced the throughput from the measured values.

The packets sent to the IPS in this scenario are created by a user space program running on the FPGA.

4.2.2 Data measured

For all presented test setups, we measure resp. simulate the processing time and then deduce the throughput from these values.

4.2.3 Performed tests and expected results

This section explains which kind of packets are sent to the IPS and what the expected results are.

binary representation	hex	nsf
0000 0000	0x00	no
0000 0001	0x01	no
0000 0010	0x02	no
(3 more characters) ⋮	⋮	no
0010 0000	0x20	no
0100 0000	0x40	no
1100 0000 1000 0000	0xc080	yes
1100 0000 1000 0001	0xc081	yes
1100 0000 1000 0010	0xc082	yes
(3 more characters) ⋮	⋮	yes
1100 0000 1010 0000	0xc0a0	yes
1100 0001 1000 0000	0xc180	yes
1100 0010 1000 0000	0xc280	no
1100 0100 1000 0000	0xc480	no
1100 1000 1000 0000	0xc880	no
1101 0000 1000 0000	0xd080	no
1110 0000 1000 0000 1000 0000	0xe08080	yes
1110 0000 1000 0000 1000 0001	0xe08081	yes
1110 0000 1000 0000 1000 0010	0xe08082	yes
(8 more characters) ⋮	⋮	yes
1110 0000 1001 0000 1000 0000	0xe09080	yes
1110 0000 1010 0000 1000 0000	0xe0a080	no
1110 0001 1000 0000 1000 0000	0xe18080	no
1110 0010 1000 0000 1000 0000	0xe28080	no
1110 0100 1000 0000 1000 0000	0xe48080	no
1110 1000 1000 0000 1000 0000	0xe88080	no
1111 0000 1000 0000 1000 0000 1000 0000	0xf0808080	yes
1111 0000 1000 0000 1000 0000 1000 0001	0xf0808081	yes
1111 0000 1000 0000 1000 0000 1000 0010	0xf0808082	yes
(13 more characters) ⋮	⋮	yes
1111 0000 1000 1000 1000 0000 1000 0000	0xf0888080	yes
1111 0000 1001 0000 1000 0000 1000 0000	0xf0908080	no
1111 0000 1010 0000 1000 0000 1000 0000	0xf0a08080	no
1111 0001 1000 0000 1000 0000 1000 0000	0xf1808080	no
1111 0010 1000 0000 1000 0000 1000 0000	0xf2808080	no
1111 0100 1000 0000 1000 0000 1000 0000	0xf4808080	no

Table 4.1: Characters used for testing the UTF-8 non-shortest form detection. These characters contain all possible bit lengths of UTF-8 code points.

Packet size The IPS will be tested using different packet sizes. We chose to test it using the packet sizes recommended in RFC 2544 [21]. The expected result in the software is that the computation time increases linearly to the packet size, plus a constant time needed for packet handling, calling routines, etc. The hardware, on the other hand, was designed to be able to process packets in the rate they arrive, so we expect the same computation time, no matter what the packet size is.

Location of attack within the Packet The test packets which contain attacks schall contain them at different locations in the packet. We expect that the software will need less CPU time (and thus, perform better) when the attack is on a early position in the packet, since it can skip checking the remainder of the packet for attacks. The hardware, on the other hand, already checks packets in real time, so no better throughput is possible. The only thing we can do is lowering the latency as packets can be dropped earlier.

4.3 Results

4.3.1 Hardware

Based on the knowledge from the design and implementation phase, we can theoretically deduce the performance of the IPS.

For the following computations, we define (loosely based on [22]):

- Momentary throughput θ_m: Maximum data rate at which packets can be processed. $\theta_m = \frac{1}{T}$ with $T =$ the minimum number of clock cycles between two subsequent data items. As we like to know the result in (G)Bit per second, we define this value as the number of bits processed $[\theta_m] =$ Bits per clock cycle.
- Average throughput θ: Data rate when the IPS is in a steady state, i.e. averaged over a long time period. $[\theta] =$ Bits per clock cycle.
- Latency L: Number of clock cycles a data item needs from entering the IPS until it leaves the IPS. L is not defined for packets being dropped as they never leave the IPS at all. $[L] =$ clock cycles.
- Packet Size s_p: Size of a data packet. $[s_p] =$ bits.
- s_b: Size (depth) of the packet buffer. $[s_p] =$ bits.
- f: Clock Frequency at which the FPGA operates. $[f] =$ Hz.
- L_{SI} and θ_{SI}: L resp. θ using SI units (seconds) as a time basis instead of clock cycles.

From the design and implementation phase we know that the IPS consists of memorising[2] and of purely combinatorial parts. The longest path in the combinatorial parts restricts the maximum clock frequency f_{max} at which the FPGA can operate. The synthesis tools compute this value for us. The combinatorial parts, on the other hand, define the L and θ as follows:

Our circuitry operates byte-wise, i.e. the momentary throughput while processing data is:

$$\theta_m = 8\frac{bits}{clockcycle} = 1\frac{byte}{clockcycle}$$

[2] Parts of the VHDL code where data is stored in in registers

The possible packet sizes are (cf. 3.3.2):

$$s_p \in [64, 1500]bytes = [512, 12000]bits$$

The number of clock cycles needed to process one packet consist of a part which linearly increases with increasing packet length plus a constant overhead of 1 clock cycle per packet. This clock cycle occurs because the *sender control* FSM (cf. Section 3.3.6) always transitions to the idle state after sending/dropping is finished, even if there is already a new packet waiting in the queue.

$$C_p = \frac{s_p}{\theta_m} + 1 = [65, 1501]clockcycles$$

So the average throughput is:

$$\theta = \frac{s_p}{C_p} = [0.985, 0.999]\frac{byte}{clockcycle} \approx 1\frac{byte}{clockcycle}$$

We can observer that small packets perform worse because the 1 additional clock cycle accounts for more, referred to the size of a packet.

The latency is a variable depending on the current packet's size and buffer utilisation. The best case is that the IPS is idle, i.e. the packet buffer is empty. Then it equals to the current packet's size:

$$L = \frac{s_p}{\theta_b} \in [64, 1500]clockcycles$$

The worst case is that at arrival time of a packet, the packet buffer is completely filled. Then, the IPS first needs to send/drop the packets in the buffer. Note that it does not matter how long the current packet is, as it will take as least as long to flush the buffer than to process the packet ($s_b \geq s_{p,max}$).

$$L = \frac{s_b}{\theta_{worst}} = 2031clockcycles$$

Combining the last two results, we can give upper and lower bound for the Latency of 1 packet:

$$L \in [64, 2031]clockcycles$$

Synthesis report During synthesis, the Xilinx tools create several report files. By examining the report file for the *hwt_ips* pcore, we can obtain the utilisation and timing reports for our design. On our FPGA (cf. 1.5), our design uses the following portion of the device:

```
Slice Logic Utilization:
  Number of Slice Registers:    491  out of  301440    0%
  Number of Slice LUTs:        1240  out of  150720    0%
    Number used as Logic:       786  out of  150720    0%
    Number used as Memory:      454  out of  58400     0%
      Number used as RAM:       454
```

Listing 4.1: Device utilisation report for our design on the Xilinx Virtex-6 FPGA.

and the following timing report can be obtained:

```
Minimum period: 6.237ns (Maximum Frequency: 160.325MHz)
Minimum input arrival time before clock: 3.840ns
Maximum output required time after clock: 5.518ns
Maximum combinational path delay: 2.335ns
```

Listing 4.2: Timing report for our design on the Xilinx Virtex-6 FPGA.

From the first line we can obtain the maximum clock frequency at which the hardware IPS can operate: $f_{clk,max} = 160.325MHz$.

computation of maximal throughput and latency Using this value for the maximal clock frequency, we can compute the values for Latency and Throughput using SI units as a time basis:

$$L_{SI,min} = \frac{L}{f_{clk,max}} \in [0.3992, 9.356]\mu s$$

$$\theta_{SI,max} = \theta \cdot f_{clk,max} \in [157.9, 160.4]MBits/s$$

The minimum clock frequency needed to achieve Ethernet line rate of 1 GBit/s (10^9 Bits/s) is:

$$f_{clk,min} = 126953125Hz \approx 127MHz$$

4.3.2 Software - best case

For the software, we measure the processing time for a single content analysis, using the “Jiffies” time reference. A Jiffie is the time between two timer interrupts, they are a recommended way to measure time in kernel modules [20, chapter 7]. The formula to convert from Jiffies to Seconds is:

$$t_{seconds} = \frac{t_{jiffies}}{HZ}$$

where HZ is a platform-dependent constant. For our platform it is defined as $HZ = 100$ which means that 1 Jiffie equals to 10 milliseconds.

Using this method, we measured the processing time for one content analysis, using different packet sizes. It became apparent quite soon that a single content analysis is too short to be measured accurately, as the time elapsed was always zero jiffies, independently of the size of the used packet. So we decided to execute the content analysis several times using a for-loop. This for-loop executes the content analysis n_{test} times. n_{test} can be configured in the .c file, we configured it to be $n_{test} = 1000$ for our measurements.

The measurement was performed by sending packets of packet lengths from the mentioned RFC: (64, 128, 256, 512, 1024, 1280, 1500) bytes. Further we sent packets which contain the attack at different positions inside the packet: at beginning of the payload, after (32, 64, 128, 256, 512, 1024, 1280) bytes and at the end of the payload. Each packet was sent 100 times and mean value

$$\bar{x} = \left(\frac{1}{n} \cdot \sum_{i=1}^{n} x_i\right)$$

and standard deviation

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

were computed. The results are shown in Figure 4.1, the tables with the measurement results can be found in appendix B. Note that the time values in the graph are Jiffies for 1000 runs. In order to know the computation time in SI units, for 1 run, we need to apply the formula

$$t_{seconds} = \frac{t_{jiffies,measured}}{HZ \cdot 1000runs}$$

e.g. for packet size = 1500 bytes and no attack, the content analysis needs 113.97 Jiffies = 1.1397 ms on average. This would correspond to a throughput of:

$$\theta_{SI} = \frac{1500bytes}{0.0011397s} = 10554089\frac{bits}{sec} \approx 10.6\frac{Mbits}{sec}$$

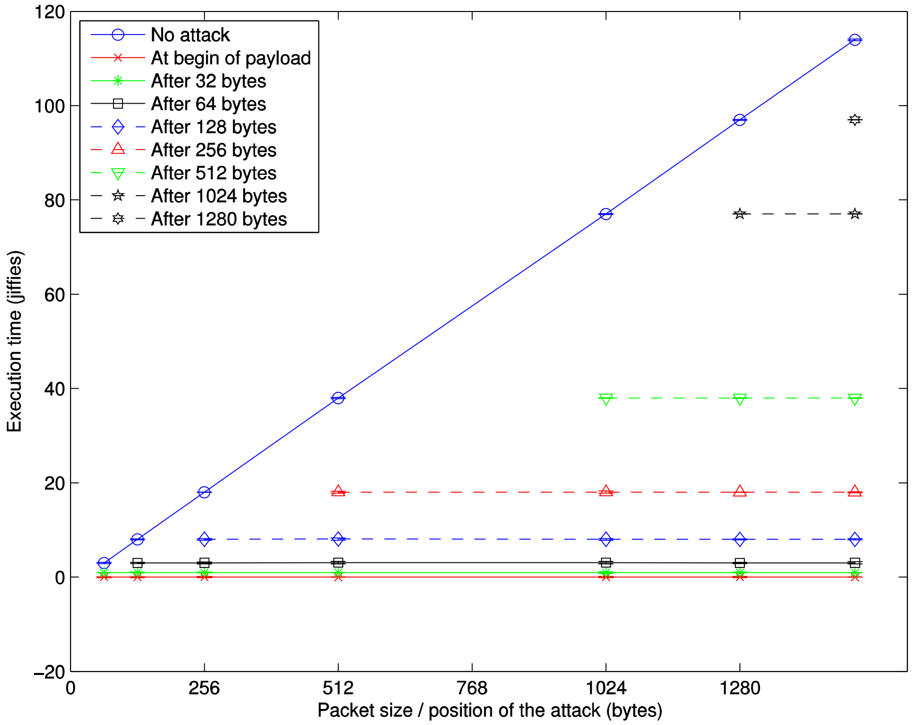


Figure 4.1: Execution time of the UTF-8 nonshortest form detection for different packet sizes. Each line corresponds to a positions of the attack within the packet.

Result The graph in Figure 4.1 clearly shows that the execution time is linear to the packet size and decreases when the attack is on an early position in the packet. This is what we expected.

4.3.3 Software - Real World Scenario

Due to different reasons, it was not possible to automate these measurement, so each of them must be carried out manually. Thus, it is not feasible to measure all possible packet sizes and positions of attack in the packet.

We performed the following measurement: Measure the time needed to process 1000 packets, without attack, and with size = 64; 1500 bytes. As the packets without attack take the most time to be analysed, this measurement can be considered as a worst case approximation. Each measurement was repeated several times and no statistic spikes could be observed. Table 4.2 summarises the results. One can observe that the software IPS is about 5 times faster for the biggest packet size, than for the smallest packet size.

packets sent:	measured duration [min, max]	throughput [max, min]
1000 packets, 1500 bytes each	[164, 165] jiffies = [1.64, 1.65] s	[7.317, 7.237] MBit/s
1000 packets, 64 bytes each	[34, 35] jiffies = [0.34, 0.35] s	[1.51, 1.46] MBit/s

Table 4.2: Measurement results for the real world scenario.

Chapter 5

Conclusion and future work

5.1 Interpretation

Measurements have shown that the hardware supports at least 0.7 GBit/s at 100 MHz clock (so most probably the full 1 GBit/s line rate if clocked at 125 MHz), whereas the best case throughput for our software IPS is about 10.6 MBit/s. In latter best case, clearly the CPU is the limiting factor, as we only measured the time for the content analysis when calculating this best-case throughput and not the time needed for packet handling and similar overhead. Extrapolated to a IPS, which does check for several hundred attacks, not just one kind of attack, we can assume that the computation will need about 100-1000 times more time. Thus, a best case throughput of 10-100 kBit/s and a latency of several 10 to 100 milliseconds must be expected, which is unacceptably slow.

We assume that this extremely low throughput is caused by the MicroBlaze CPU. This CPU is a soft-core CPU, optimised for high configurability and flexible implementation in FPGAs [23], but the computing power is not comparable to modern (ASIC) CPU cores. To achieve higher throughput, another CPU must be used.

On the other hand, the hardware utilisation for the IPS pcore currently is below 1 %, most of which are occupied by the packet buffer which is only needed once, no matter how many content analysers there are. Extrapolating the hardware usage for a IPS which detects 100-1000 attack could in fact fill a considerable amount of the FPGA's slices, but (provided the content analysis blocks are well implemented) would still provide the full 1 GBit/s throughput.

5.2 Conclusion

In this thesis we showed that an IPS for future Internet architecture, using hardware and software implementations, is feasible. The hardware version reaches line rate (1GBit/s) throughput when clocked at 125 MHz, and could be clocked to even higher frequencies, up to 160 MHz. So there is still some spare, the performance is unlikely to degrade even if more content analysis blocks are added.

5.3 Known Bugs

5.3.1 Double Byte Bug

On forwarding packets, the hardware sends the first byte twice. We could namely isolate the problem to be related to the sender, but we didn't manage to reproduce this bug in the simulator, nor to solve it in the hardware, in the given time. The possible workarounds we already tried are marked as comments in the source code of the *ips.vhd* file, see espectially the *sendercontrol_memless* process.

Since this clearly is an error and not intended behaviour, we consider the urgency to fix this bug as high.

5.3.2 Hardware “gives away” one Clock Cycle per Packet

As already stated in the evaluation section (4.3.1), there is a constant overhead of 1 clock cycle per packet, because of a state transition in the *sender control* state machine, which is theoretically not needed. The *sender control* FSM could be further optimised s.t. it does not perform this transition, which would result in the ideal overhead of 0, which is a theoretic performance gain of up to 1.6 %, but at the cost of higher complexity and higher logic cost.

We assume that if this flaw really caused a problem in practise, it would cause a $\leq$ 1.6 % packet loss rate due to the packet buffer being full. Since we could not observe this, we conclude that the effects are not noticeable, and the urgency to fix this inelegance is very low.

5.4 Future work

5.4.1 Implement more Attacks

Obviously, an IPS which only detects one attack is not feature complete. New content analysis blocks must be written and added, in order to create an IPS suited for deployment in the wild. As LANA / EmbedNet currently are in the phase of a research project and not intended for deployment in the wild, we consider the urgency for this implementation as low.

Further, a way should be found to automate the inclusion of new attacks in the IPS. Currently, each new content analysis needs to be manually included in the IPS. This is quite cumbersome, especially in the hardware version, because each new content analysis implies including and instantiating new entities, adding new signals, and adapting logic functions. We propose a program or script which automatically generates the necessary code inside the *m_fb_ips.c* and *content_analysers.vhd* files (on compile- resp. synthesis-time), based on the files currently present in the folders intended for the content analysers. For a few (say < 10) content analyser blocks, it is well possible to add them by hand. Neverthless, this measure would highly facilitate the future development of content analyser blocks, so we consider the urgency to implement this as medium.

5.4.2 Support for Data Streams

As defined in design goal 3.1.4, the IPS currently only supports single packets. This approach is limited in several aspects, for instance our current attack detection would trigger false positives on certain multi-byte characters, when the first byte is in one packet and the latter bytes in a successor packet.

We could imagine a network-based IPS which operates on data streams. The concepts of the packet buffer and providing exactly one result per packet would have to be revised, though. Instead of dropping entire packets when an attack is detected, a certain amount of the data stream would have to be dropped.

5.4.3 Run-time Reconfiguration of Hardware FBs

Latest FPGAs provide the possibility to reconfigure a part of the hardware at run-time. Current research in EmbedNet aims towards using this feature to load and unload hardware FBs at run-time. In the course of this development, also the IPS hardware FB could be loaded and onloaded at run-time. This provides the advantage to be able to use the hardware version if many data are to be checked, and the software version if only a few data must be analysed.

Appendix A

How to get started (manual)

A.1 Prerequisites

A.1.1 Linux Workstation and other Hardware Requirements

Besides from the actual FPGA development board (see Section A.1.4), you will need some kind of workstation computer :-). We recommend a machine which is equipped with at least 4 GB of RAM, better is 6 GB or more since the synthesis of hardware designs for FPGAs is very RAM-intensive. Any recent x86-compatible CPU should suffice. To connect the FPGA board, you need 2 free USB ports (or a hub).

In order to perform tests with the non-TCP/IP networking stack of EmbedNet, we strongly recommend not to use the same Network Interface Card (NIC) which you use for your regular Internet traffic, but to use a separate Gigabit NIC instead, which will be connected exclusively to the FPGA board. So, install an additional NIC if needed. We will refer to this network as the LANA network resp. LANA NIC from now on.

Further, you will need a CF card (2 GB will suffice) and a matching card reader. As the files of this thesis and the Xilinx tools come on CD-ROM, you will also need an optical drive.

Our test machine is equipped with a Intel DP35DP Mainboard[1], Intel Core2Quad Q6600 CPU (@2.40GHz) and 8 GB of RAM, the LANA NIC is a Intel 82541PI Gigabit Ethernet Controller[2].

A.1.2 Install and configure Linux on your Workstation

Basically, any version of Linux compatible to the Xilinx Tools (see below) should work. However, this manual was tested on a Debian GNU/Linux[3] version 6.0 “Squeeze” (amd64), the latest version of Debian at the time this thesis started. All the latest updates and patches of this version were installed.

After installing Linux with your favourite (graphical) user interface, install the following development and networking tools. The following command should install the packages on Debian-based systems, but need to be adapted an other Linux distributions.

```
aptitude install build-essential git tcpreplay
```

Further packages you might need are:

[1] http://www.intel.com/p/de_DE/support/highlights/dsktpboards/dp35dp

[2] http://www.intel.com/design/network/products/lan/controllers/82541pi.htm

[3] http://www.debian.org/

```
aptitude install wireshark scapy mz minicom
```

To make sure your brand new LANA network stays free of any (potentially perturbing) non-LANA traffic, disable (or uninstall) all avahi/ZeroConf related applications. We further recommend to disable (or uninstall) all IPv6 related traffic and to assign a fixed IPv4 address to your second NIC. Doing latter will prevent your workstation from sending any DHCP requests on your LANA NIC. If you decided to use Debian, the Debian Wiki[4] is a good place to start your search on how to achieve the mentioned network configurations.

Verify that your LANA network is free of any unwanted traffic by connecting and powering on the FPGA board (see Section A.1.4) and inspecting the network traffic using a network analyser, e.g. Wireshark.

A.1.3 ReconOS Git repository and Get Started

Clone the ReconOS Git repository from GitHub to a directory on your workstation. We will refer to that directory by `path_to_reconos` from now on.

```
git clone git://github.com/EPiCS/reconos.git
```

We will refer to the "ReconOS Get Started" several times in this document. This file is named `GETSTARTED` and can be found in the `path_to_reconos` directory. Alternatively, you may also use this HTTP link[5].

This manual was written, based on Git commit 78ed2dc... You can jump to the state of this git commit using the command:

```
git checkout 78ed2dc12401e584b11e38518cb1765081096ab3
```

If you just wish to view the state of certain files in this commit, we recommend looking them up on GitHub[6].

A.1.4 Setup FPGA Board and Xilinx Tools

Unbox your Xilinx ML605 evaluation board[7] and attach the included power supply. Connect the Ethernet interface to your LANA NIC and the USB JTAG and USB UART to your computer, using the included cables.

Install the Xilinx Tools, MicroBlaze GNU tools and Xilinx USB driver, as explained in the ReconOS Get Started. You may Skip the part which explains how to install ModelSim, we don't need it. As an alternative to adding all the commands to your .bashrc file, which are mentioned in the ReconOS Get Started, we wrote script which can do this automatically. This script is described in Section 3.8.

A.1.5 Setup ReconOS

Setup ReconOS as described in the "Setup ReconOS V3" section of the ReconOS Get Started. Please do not run the sort demo or any of the NFS server related steps, we don't need all this and will now deviate a little bit from the ReconOS Get Started.

Download and extract the Linux Kernel. We will refer to the Linux directory as `path_to_linux`.

```
wget -qO- http://pc-techinf-25.cs.upb.de/ml605-linux/linux
    -2.6-xlnx.tar.bz2 | tar xjv
```

[4] `https://wiki.debian.org/`

[5] `https://github.com/EPiCS/reconos/blob/master/GETSTARTED`

[6] `https://github.com/EPiCS/reconos/tree/78ed2dc12401e584b11e38518cb1765081096ab3`

[7] `http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm`

Copy the device tree file from the ReconOS protocol_graph design into the Linux directory:

```
cp path_to_reconos/demos/protocol_graph_b/hw/device_tree/noc
    .dts path_to_linux/arch/microblaze/boot/dts/
```

Compile the Linux Kernel:

```
cd path_to_linux;
make clean;
make CROSS_COMPILE=microblaze-unknown-linux-gnu- ARCH=
    microblaze -j8 simpleImage.noc
```

You might have received a CF card with the ReconOS root filesystem from your advisor, work colleague, or other person who introduced you to ReconOS and EmbedNet. If not, make your own one. Create an empty ext2 file system on the CF card and mount it on our workstation. Download the contents of the root filesystem and store them to your CF card (these commands might need root privileges):

```
wget -qO- http://pc-techinf-25.cs.upb.de/ml605-linux/
    rootfs_mb.tar | tar -xv;
cd rootfs_mb;
mv * /cf_card_mount_point/
```

A.1.6 Setup LANA

To setup LANA, run the steps described in the “Building Linux and LANA” section of the ReconOS Get Started. Make sure you execute the steps which are marked for ML605 resp. cross-compilation, not the host-only deployment. Especially don’t forget to adapt all mentioned Makefiles.

A.1.7 Well done ...

... you just finished the preparative work and are now ready to include the hardware IPS, software IPS, or both, to your EmbedNet environment.

A.2 Include hardware IPS into EmbedNet

1. Install the software tools and clone the ReconOS Git repository, as described in the previous section.

2. Copy the folder *hwt_ips_v1_00_a* from the CD-ROM to the *pcores* Folder inside *protocol_graph_b*:

```
cp -r path_to_cdrom/IPS-HW/hwt_ips_v1_00_a
    path_to_reconos/demos/protocol_graph_b/hw/edk/pcores
    /
```

3. Adapt the *system.mhs* file in the *edk* folder, s.t. the IPS pcore is included during hardware synthesis. If you are using the Git commit mentioned in A.1.3, you may apply the patch file included with the hardware files:

```
patch path_to_reconos/demos/protocol_graph_b/hw/edk/
    system.mhs < path_to_cdrom/IPS-HW/system_mhs.patch
```

This patch removes the AES FB [12] from the design and includes the IPS FB instead. (As there are no more free hardware slots in the current design, it is currently not possible to operate the AES FB and IPS FB at the same time.)

4. Generate the new Bitstream using the Xilinx tools:

```
cd path_to_reconos/demos/protocol_graph_b/hw/edk/;
make clean; make
```

Generating the bitstream will take a while[8]. During this time, you might want to have a look at the next section, describing how to configure the hardware IPS.

A.3 Hardware IPS Usage

The hardware IPS can be configured, and statistics can be retreived, from within the OS. ReconOS provides means to communicate with such hardware threads, like message boxes or shared memory, as explained in Section 1.3.4. In the software, the communication is handled within a kernel module. An example of this kernel module *hw_sw_interface.c* can be found on the CD-ROM, this section explains how to use this kernel module.

1. Copy the example *hw_sw_interface.c* from the CD-ROM:

```
cp -r path_to_cdrom/IPS-HW/hw_test_ips path_to_reconos/
    demos/protocol_graph_b/linux/
```

2. Open the newly copied *hw_sw_interface.c* using your favourite text editor. The relevant code is located below the keyword *config_and_statistics* (just search for this keyword).

3. Different instructions can be sent to the IPS. They are sent using a 32-bit message box (in the IPS, we are only using message boxes, no shared memory). The messages are sent and received to hardware slot E (*&e_mb_put*) as we configured the IPS to be in hardware slot E in the MHS file.

   - The first message which is sent to the IPS instructs it where to forward its output packet data (using a hardware-internal address). In our example, we instruct it to forward the packets to the hardware-to-software interface (h2s) *addr_h2s*. Note that in this example, the Data is just forwarded to another hardware FB, not to the kernel or any user space program.

     ```
     mbox_put(&e_mb_put, addr_h2s);
     ```

     For the beginning, you probably don't want to change anything on this command.

   - The second message instructs the IPS how long the header of our packets are. Bytes in the header are not checked for attacks, this is useful if you only want to check the payload of packets for attacks. The message consists of a ASCII 'h' (for "header") in bits 31 downto 24, followed by the unsigned integer header length on the remaining bits 23 downto 0. For instance, the following commands set the header length to be 16 bytes:

[8] About 40 minutes on our test system.

```
tmp = ('h' << 24) | 16;
mbox_put(&e_mb_put, tmp);
```

Adapt the header length to your needs. **Beware** not to set the header bigger than your minimum packet size - 1. The IPS will fail to identify attacks correctly, or even crash, because the content analysis blocks assume a minimum length of 1 byte per packet which they should check. For the beginning, let's check the entire packet for attacks, so set the header length to 0:

```
tmp = 'h' << 24;
mbox_put(&e_mb_put, tmp);
```

- The last message instructs the IPS to return status and statistical values. It consists of a ASCII 's' (which stands for "statistics" or "status") on bytes 31 downto 24:

```
tmp = ('s' << 24);
mbox_put(&e_mb_put, tmp);
```

When it receives this message, the IPS will answer using 5 message boxes, in the following order: Number of **received**, **forwarded** (sent) and **dropped** packets, currently configured **header length** and currently configured **address** where to forward the packet data. In the example, each of the message boxes is read and the value is printed on the console:

```
ret = mbox_get(&e_mb_get);
printk(KERN_INFO "Received Packets: %u\n", ret);
ret = mbox_get(&e_mb_get);
printk(KERN_INFO "Forwarded Packets: %u\n", ret);
ret = mbox_get(&e_mb_get);
printk(KERN_INFO "Dropped Packets: %u\n", ret);
ret = mbox_get(&e_mb_get);
printk(KERN_INFO "Header length: %u\n", ret);
ret = mbox_get(&e_mb_get);
printk(KERN_INFO "Global/local_addr: %u\n", ret);
```

If you don't need all the output, feel free to comment the resp. *printk* commands. But beware not to comment any of the 5 *mbox_get* commands: The IPS always returns exactly 5 values, which all have to be fetched, otherwise it will stall and not accept any further messages through *mbox_put* anymore. So, for the beginning, you probably don't want to change anything else here, either.

4. What else does this example code do?

- Each packet that arrives in the h2s FB will be printed on the console by the software. The software requests the data from the h2s and prints them on the console, using the following commands:

```
ret = mbox_get(&b_mb_get);
for (i = 0; i < 100; i+=4)
     printk(KERN_INFO "\%x \%x \%x \%x\n",
         shared_mem_h2s[i], shared_mem_h2s[i+1],
         shared_mem_h2s[i+2],  shared_mem_h2s[i+3]);
```

Note that this message box adressed to the h2s (*&**b**_mb_get*), not to the IPS (*&**e**_mb_get*) hardware FB.

- The Ethernet FB forwards the first packet that arrives to the h2s (not the IPS!), and it will thus always be printed on the console (even if it contains an attack). Then, the Ethernet FB is reconfigured s.t. it forwards all data to the IPS:

  ```
  u32 config_eth_address = addr_ips;
  mbox_put(&a_mb_put, config_eth_address);
  ```

  All following packets arriving in the Ethernet FB will be forwarded to the IPS, which in turn has already been configured to forward its packets to the h2s. This means that all packets not containing attacks will still arrive at the h2s (thus printed on the console). All packets containing an attack will be dropped by the IPS and never arrive at h2s (thus, not printed on the console).
- All the commands are inside an infinite loop, to keep the example code as simple as possible. In a more real-world scenario, of course, one could start the kernel module in background und send/receive data from it, e.g. using procfs.

5. Compile the *hw_sw_interface.c* and copy the resulting *.ko* file to the CF card:

```
cd path_to_reconos/demos/protocol_graph_b/linux/;
make clean; make;
mkdir path_to_CF_card/hw_test_ips/;
cp hw_sw_interface.ko path_to_CF_card/hw_test_ips/;
```

6. Insert the CF card in the FPGA board. Upload your Bitfile (which, hopefully, has been generated in the meantime) and Kernel to the FPGA:

```
dow path_to_reconos/demos/protocol_graph_b/hw/edk/
    implementation/system.bit;
dow path_to_linux/arch/microblaze/boot/simpleImage.noc;
```

7. Connect to the serial console of the FPGA board, e.g. using Minicom:

```
minicom -o -b 9600 -D /dev/ttyUSB0
```

8. Enter the following commands in Minicom:

```
mount -n -o remount,rw /
ifconfig lo up
./load_fsl.sh
./load_getpgd.sh
insmod libreconos.ko
insmod hw_test_ips/hw_sw_interface.ko
```

The last command starts the example you just compiled and copied to the CF card.

9. Send some packets from the workstation to the FPGA board. You may take the example packets provided on the CD-ROM and send them, e.g. using tcpreplay (replace ethX by network interface connected to the FPGA board):

```
tcpreplay -i ethX path_to_cdrom/sample-packets/
    good_packet.pcap;
tcpreplay -i ethX path_to_cdrom/sample-packets/
    evil_packet.pcap;
```

Each time you send a "good" packet (i.e. without an attack), it will arrive at the software and its content will be printed on the console, together with the IPS' statistics. Each "evil" packets will be dropped by the IPS, thus nothing will happen on the console. You may, however, observe that the counter of dropped packets has increased, next time you send a "good" packet. Note that in the example packets provided, the attack is on the 7th byte of the packet. If you have set your header length to be bigger than 6 in step 3, the attack is in the header and all packets will be classified as "good" packets.

A.4 Include software IPS into ReconOS

1. Install the software tools and clone the ReconOS Git repository, as described in Section A.1.3.

2. Copy all files from the folder *kernel-module* to the *net* folder inside *linux*:

```
cp path_to_cdrom/IPS-SW/kernel-module/* path_to_reconos/linux/net/
```

3. Adapt the *Makefile* in the *net* folder, s.t. the IPS kernel module is made too.

```
<favourite_editor> path_to_reconos/linux/net/Makefile
```

   Add the following two lines below the existing *obj-m* lines of the Makefile:

```
obj-m         += fb_ips.o
fb_ips-objs   := m_fb_ips.o ca_utf8_nonshortest_form.o
```

4. In *xt_nocx.c*, comment (or delete) the code which configures the AES FB [12] (this might not be needed anymore in future versions of the ReconOS Git):

```
<favourite_editor> path_to_reconos/linux/net/xt_nocx.c
```

   Comment the env. 30 lines between the line

```
//#ifdef AES_CONFIGURED
```

   and the line

```
//#endif
```

5. As a shortcut for steps 3 and 4, you may just apply the following two patches:

```
patch path_to_reconos/linux/net/xt_nocx.c < path_to_cdrom/IPS-SW/xt_nocx_c.patch;
patch path_to_reconos/linux/net/Makefile < path_to_cdrom/IPS-SW/makefile.patch;
```

A.5 Software IPS Usage

1. Compile the Linux networking modules, including the IPS module, and copy them to the CF card:

```
cd path_to_reconos/linux/net;
make clean; make;
mkdir path_to_CF_card/sw_ips_module/;
cp *.ko path_to_CF_card/sw_ips_module/;
```

2. Insert the CF card in the FPGA board. Upload your Bitfile and Kernel to the FPGA:

```
dow path_to_reconos/demos/protocol_graph_b/hw/edk/
    implementation/system.bit;
dow path_to_linux/arch/microblaze/boot/simpleImage.noc;
```

3. Connect to the serial console of the FPGA board, e.g. using Minicom:

```
minicom -o -b 9600 -D /dev/ttyUSB0
```

4. Enter the following commands in Minicom:

```
mount -n -o remount,rw /
ifconfig lo up
./load_fsl.sh
./load_getpgd.sh
insmod libreconos.ko
insmod sw_ips_module/lana.ko
```

These commands load the ReconOS base system and the LANA core module. Now, load the FBs by entering:

```
insmod sw_ips_module/fb_dummy.ko
insmod sw_ips_module/fb_ips.ko
insmod sw_ips_module/fb_pflana.ko
```

These commands load the FBs into the Kernelspace, but they are not yet added to the protocol stacks. Build the protocol stack by entering the following *fbctl* commands:

```
cd /
./fbctl add eth1 ch.ethz.csg.dummy
./fbctl add fb1 ch.ethz.csg.ips
./fbctl flag eth1 hw
./fbctl unflag fb1 hw
./fbctl bind fb1 eth1
```

The first two commands add the FBs to the protocol stack. The *flag* and *unflag hw* commands instruct EmbedNet to use the hardware `eth1` FB, which is the Ethernet FB, and the software `fb1` block, which is the IPS. The last command connects a communication channel between the Ethernet and IPS FBs.

5. The IPS FB is now ready. To change the configuration how long the header of our packets are, write a new integer value in the according file in *procfs*. **Beware** not to set the header bigger than your minimum packet size - 1. The IPS will fail to identify attacks correctly, or even crash, because the content analysis functions assume a minimum length of 1 byte per packet which they should check.

To begin, we will set the header length to 0, i.e. the entire packet will be checked. Enter the following in Minicom:

```
echo 0 > /proc/net/lana/fb1
```

To print the statistics of received, forwarded and dropped packets, read the contents of the file in the *procfs*. Enter the following in Minicom:

```
cat /proc/net/lana/fblock/fb1
```

6. Send some packets from the workstation to the FPGA board. You may take the example packets provided on the CD-ROM and send them, e.g. using tcpreplay (replace ethX by network interface connected to the FPGA board):

   ```
   tcpreplay -i ethX path_to_cdrom/sample-packets/
       good_packet.pcap;
   tcpreplay -i ethX path_to_cdrom/sample-packets/
       evil_packet.pcap;
   ```

   For each packet, the IPS will print a debug message on the Console, indicating if it classified the packet as a "good" packet (i.e. without an attack), or as an "evil" packet. Observe the console output as you send different packets. You may also check the statistics or play around with the header length, as explained in step 5.

   Note that in the example packets provided, the attack is on the 7th byte of the packet. If you have set your header length to be bigger than 6, the attack is in the header and all packets will be classified as "good" packets.

A.6 How to add new Content Analysis Blocks

To add more content analysis blocks to the IPS, write a new VHDL entity for the hardware and a corresponding C function for the software version. The interfaces used for content analysis blocks are described in Section 3.3.7 for the hardware and 3.4.4 for the software. It is advised to create one *.vhd* file for each hardware content analyser and a *.c* and *.h* file for each software content analyser. **Please** always write a hardware and corresponding software content analyser, not just one of the two, as the two IPS implementations should always be equivalent s.t. they can be dynamically exchanged.

naming convention Think of a meaningful name for the new content analysis block. Meaningful names for content analysers contain at least the name of the attack they can detect. Example for non meaningful names are the name of the author, or names like "test", "attack detection" and suchlike.

Copy new *.vhd* files to the folder

and name them `ca_<name>.vhd`.

Copy new *.c* and *.h* files to the folder

```
path_to_reconos/linux/net/
```

and name them `ca_<name>.c` resp. `ca_<name>.h`.

Also name the functions resp. entities according to the name of the block.

Files and folders As long as new content analysis blocks are not automatically added to the system (cf. Future Work), this needs to be done manually. Search for the keyword "TODO automatise" in the source code of `packet_inspection.vhd` resp. `m_fb_ips.c` and you will find all instances where you need to add your content analysis block, with an example how to add it.

Appendix B

Tables

B.1 Verification of UTF-8 non-shortest form detection.

Tables B.1 through B.4 show all characters used to verify the correct functionality of the UTF-8 non-shortest form attack detection.

binary representation	hex	nsf
0000 0000	0x00	no
0000 0001	0x01	no
0000 0010	0x02	no
0000 0100	0x04	no
0000 1000	0x08	no
0001 0000	0x10	no
0010 0000	0x20	no
0100 0000	0x40	no

Table B.1: All 1-byte characters used for testing.

binary representation	hex	nsf
1100 0000 1000 0000	0xc080	yes
1100 0000 1000 0001	0xc081	yes
1100 0000 1000 0010	0xc082	yes
1100 0000 1000 0100	0xc084	yes
1100 0000 1000 1000	0xc088	yes
1100 0000 1001 0000	0xc090	yes
1100 0000 1010 0000	0xc0a0	yes
1100 0001 1000 0000	0xc180	yes
1100 0010 1000 0000	0xc280	no
1100 0100 1000 0000	0xc480	no
1100 1000 1000 0000	0xc880	no
1101 0000 1000 0000	0xd080	no

Table B.2: All 2-byte characters used for testing.

binary representation	hex	nsf
1110 0000 1000 0000 1000 0000	0xe08080	yes
1110 0000 1000 0000 1000 0001	0xe08081	yes
1110 0000 1000 0000 1000 0010	0xe08082	yes
1110 0000 1000 0000 1000 0100	0xe08084	yes
1110 0000 1000 0000 1000 1000	0xe08088	yes
1110 0000 1000 0000 1001 0000	0xe08090	yes
1110 0000 1000 0000 1010 0000	0xe080a0	yes
1110 0000 1000 0001 1000 0000	0xe08180	yes
1110 0000 1000 0010 1000 0000	0xe08280	yes
1110 0000 1000 0100 1000 0000	0xe08480	yes
1110 0000 1000 1000 1000 0000	0xe08880	yes
1110 0000 1001 0000 1000 0000	0xe09080	yes
1110 0000 1010 0000 1000 0000	0xe0a080	no
1110 0001 1000 0000 1000 0000	0xe18080	no
1110 0010 1000 0000 1000 0000	0xe28080	no
1110 0100 1000 0000 1000 0000	0xe48080	no
1110 1000 1000 0000 1000 0000	0xe88080	no

Table B.3: All 3-byte characters used for testing.

B.2 Measurements: Software, best case

The tables show the measured times for the software ideal case, as described in Section 4.3.2.

Table B.5 shows the mean value

$$\bar{x} = \left(\frac{1}{n} \cdot \sum_{i=1}^{n} x_i \right)$$

and Table B.6 the standard deviation

$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2}$$

of processing times, in Jiffies, for $n_{test} = 1000$. From each type of packet, 100 packets were sent and evaluated. Each column corresponds to a packet size, while each line corresponds to a position of the attack within the packet. The mean values divided by 100 correspond to the execution time in milliseconds for $n_{test} = 1$. e.g. for packet size 1500 bytes and no attack, the content analysis needs 1.1397 ms.

Table B.7 shows the same values for 1000 packets without an attack. This scenario corresponds to the first line of Table B.5 and B.6.

binary representation	hex	nsf
1111 0000 1000 0000 1000 0000 1000 0000	0xf0808080	yes
1111 0000 1000 0000 1000 0000 1000 0001	0xf0808081	yes
1111 0000 1000 0000 1000 0000 1000 0010	0xf0808082	yes
1111 0000 1000 0000 1000 0000 1000 0100	0xf0808084	yes
1111 0000 1000 0000 1000 0000 1000 1000	0xf0808088	yes
1111 0000 1000 0000 1000 0000 1001 0000	0xf0808090	yes
1111 0000 1000 0000 1000 0000 1010 0000	0xf08080a0	yes
1111 0000 1000 0000 1000 0001 1000 0000	0xf0808180	yes
1111 0000 1000 0000 1000 0010 1000 0000	0xf0808280	yes
1111 0000 1000 0000 1000 0100 1000 0000	0xf0808480	yes
1111 0000 1000 0000 1000 1000 1000 0000	0xf0808880	yes
1111 0000 1000 0000 1001 0000 1000 0000	0xf0809080	yes
1111 0000 1000 0000 1010 0000 1000 0000	0xf080a080	yes
1111 0000 1000 0001 1000 0000 1000 0000	0xf0818080	yes
1111 0000 1000 0010 1000 0000 1000 0000	0xf0828080	yes
1111 0000 1000 0100 1000 0000 1000 0000	0xf0848080	yes
1111 0000 1000 1000 1000 0000 1000 0000	0xf0888080	yes
1111 0000 1001 0000 1000 0000 1000 0000	0xf0908080	no
1111 0000 1010 0000 1000 0000 1000 0000	0xf0a08080	no
1111 0001 1000 0000 1000 0000 1000 0000	0xf1808080	no
1111 0010 1000 0000 1000 0000 1000 0000	0xf2808080	no
1111 0100 1000 0000 1000 0000 1000 0000	0xf4808080	no

Table B.4: All 4-byte characters used for testing.

size:	64	128	256	512	1024	1280	1500
none	3.02	8.01	18.00	37.98	77.01	96.99	113.97
begin	0.01	0.01	0.03	<0.01	0.02	0.02	<0.01
32	1.00	0.99	0.99	1.00	0.96	0.99	1.00
64	.	3.02	3.04	3.09	3.07	3.02	3.05
128	.	.	8.03	8.08	8.03	8.01	8.02
256	.	.	.	18.04	18.05	18.00	18.01
512	.	.	.	.	37.99	37.99	37.99
1024	.	.	.	.	.	77.00	77.00
1280	.	.	.	.	.	.	97.00
end	3.00	8.02	17.99	37.98	77.00	96.95	113.98

Table B.5: Mean of processing times in Jiffies. These values divided by 100 correspond to the execution time in milliseconds for $n_{test} = 1$.

size:	64	128	256	512	1024	1280	1500
none	0.1407	0.1000	0.0	0.1407	0.1000	0.1000	0.1714
begin	0.1000	0.1000	0.1714	0.0	0.1407	0.1407	0.0
32	0.0	0.1000	0.1000	0.0	0.1969	0.1000	0.0
64	.	0.1407	0.1969	0.2876	0.2564	0.1407	0.2190
128	.	.	0.1714	0.2727	0.1714	0.1000	0.1407
256	.	.	.	0.1969	0.2190	0.0	0.1000
512	.	.	.	.	0.1000	0.1000	0.1000
1024	.	.	.	.	.	0.0	0.0
1280	.	.	.	.	.	.	0.0
end	0.0	0.1407	0.1000	0.1407	0.0	0.2190	0.1414

Table B.6: Standard deviation of processing times (in Jiffies).

size:	64	128	256	512	1024	1280	1500
mean:	3.0190	8.0080	17.9960	37.9590	77.0020	96.9810	113.9580
std-dev:	0.1365	0.0891	0.0631	0.1983	0.0447	0.1365	0.2007
median:	3	8	18	38	77	97	114

Table B.7: Mean, median and standard deviation of processing times (in Jiffies), for 1000 packets of each size, without an attack.

Appendix C

Task Description

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Master Thesis

Intrusion Prevention for Flexible Protocol Stacks

Stefan Kronig

Advisor: Ariane Keller, ariane.keller@tik.ee.ethz.ch
Co-Advisor: Markus Happe, markus.happe@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

28 March 2013 - 28 September 2013

1 Introduction

This master thesis is in the context of the EPiCS project. The goal of the EPiCS project is to lay the foundation for engineering the novel class of proprioceptive computing systems. Proprioceptive computing systems collect and maintain information about their state and progress, which enables self-awareness by reasoning about their behaviour, and self- expression by effectively and autonomously adapting their behaviour to changing conditions.

In this thesis we focus on the networking aspect of EPiCS, which we call *EmbedNet*. EPiCS uses the network architecture developed in the ANA project as a basis. The ANA network architecture is a novel architecture that enables flexible, dynamic, and fully autonomous formation of network nodes. It splits network functionality into individual building blocks that can be combined to tailor made protocol stacks. Additionally, in EmbedNet the mapping of the building blocks to either hardware or software can be done at runtime. The objective of this Masters Thesis is to develop and evaluate an Intrusion Prevention block that can be used in hardware as well as in software.

2 Assignment

This assignment aims to outline the work to be conducted during this thesis. The assignment may need to be adapted over the course of the project.

2.1 Objectives

The goal of this Master thesis is to develop an Intrusion Prevention block that prohibits attacks against UTF-8 encoding. First, an overview of attacks against UTF-8 should be performed which includes an assessment of the effort require to defend against the attack. Second, based on this overview, an attacker model should be defined, and a hardware as well as a software block should be developed that defend the system from the attacker. The hardware and software blocks should offer identical functionality, so that the mapping can be done at run time. The scope of the Intrusion Prevention block will be increased in several iterations. Third, a test system should be developed which demonstrates that the defence against the attacks works.

2.2 Tasks

This section gives a brief overview of the tasks the student is expected to perform towards achieving the objective outlined above. The binding project plan will be derived over the course of the first three weeks depending on the knowledge and skills the student brings into the project.

2.2.1 Familiarization

- Study the available literature on ANA, EmbedNet and Reconos [1, 2, 3, 4].
- Setup a FPGA development environment with the Xilinx tools and cross compilation tools.
- Familiarize yourself with git/github and clone the ReconOS source code repository [5].
- Verify your tool chain by running the protocol_graph demo.
- In collaboration with the advisor, derive a project plan for your master thesis. Allow time for the design, implementation, evaluation, and documentation of your software.

2.2.2 Attacker Model

- Familiarisation with attacks against UTF-8 encoding.
- Classification of the attacks and assessment of the attacks with regards to impact of a successful attack and effort to defend against the attack.
- Development of an attacker model and selection of the attacks to be defended.

2.2.3 Architecture and hardware design

- Develop an architecture with which you can show that your defend mechanisms work.
- Develop the hardware architecture for your Intrusion Prevention block. If possible, the hardware block should be able to process packets at line rate (1GB/s).
- Develop the software architecture for your Intrusion Prevention block. The hardware and software block should offer exactly the same functionality.
- Optional: Change your attacker model to a more sophisticated attacker and develop the architectures for the hardware and software block defending against the new attacker.
- Optional: Look at other attacks (non UTF-8) and develop hardware and software blocks that defend against these attacks.

2.2.4 Implementation

- Determine an appropriate version control system and set it up for further use. You might consider using git and branch the official reconOS git repository into your git repository.
- Implement the testbed in which you will show that your defend mechanisms work.
- Implement the hardware block.
- Implement the software block.
- Optional: Implement hardware and software blocks that defend against more sophisticated attacks.

2.2.5 Validation

- Validate the correct operation of your implementation after each implementation step. Use for your evaluation different packet sizes (short, long, even or odd number of bytes, etc.).
- Quantify the maximum troughputs of the hardware block and software block for selected packet sizes.
- Check the resilience of the implementation, including its configuration interface, to uneducated users.
- Optional: Build a demo system with a graphical user interface that shows which attacks were successful and which were blocked.

2.2.6 Evaluation

- Do a performance evaluation of your implementation. This includes a stress test, in order to verify that your blocks do not introduce any instabilities into the overall system.

2.2.7 Documentation

- Provide appropriate source code documentation.
- Write a step-by-step how to that describes the compilation of your code, the loading of the code into the hardware and the execution of your code.
- Write a documentation about the design, implementation, validation and evaluation of your work.

3 Milestones

- Provide a project plan, which identifies the milestones.
- Two intermediate presentations: Give a presentation of ten minutes to the professor and the advisors. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.
- Final presentation of 15 minutes in the CSG group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.
- Any software that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.
- Final report: The final report must contain a summary, the assignment, the time schedule and the Declaration of Originality. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.

4 Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps. The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.
- The student is encouraged to write all reports in English; German is accepted as well.
- The core source code will be published under the GNU general public license.

5 References

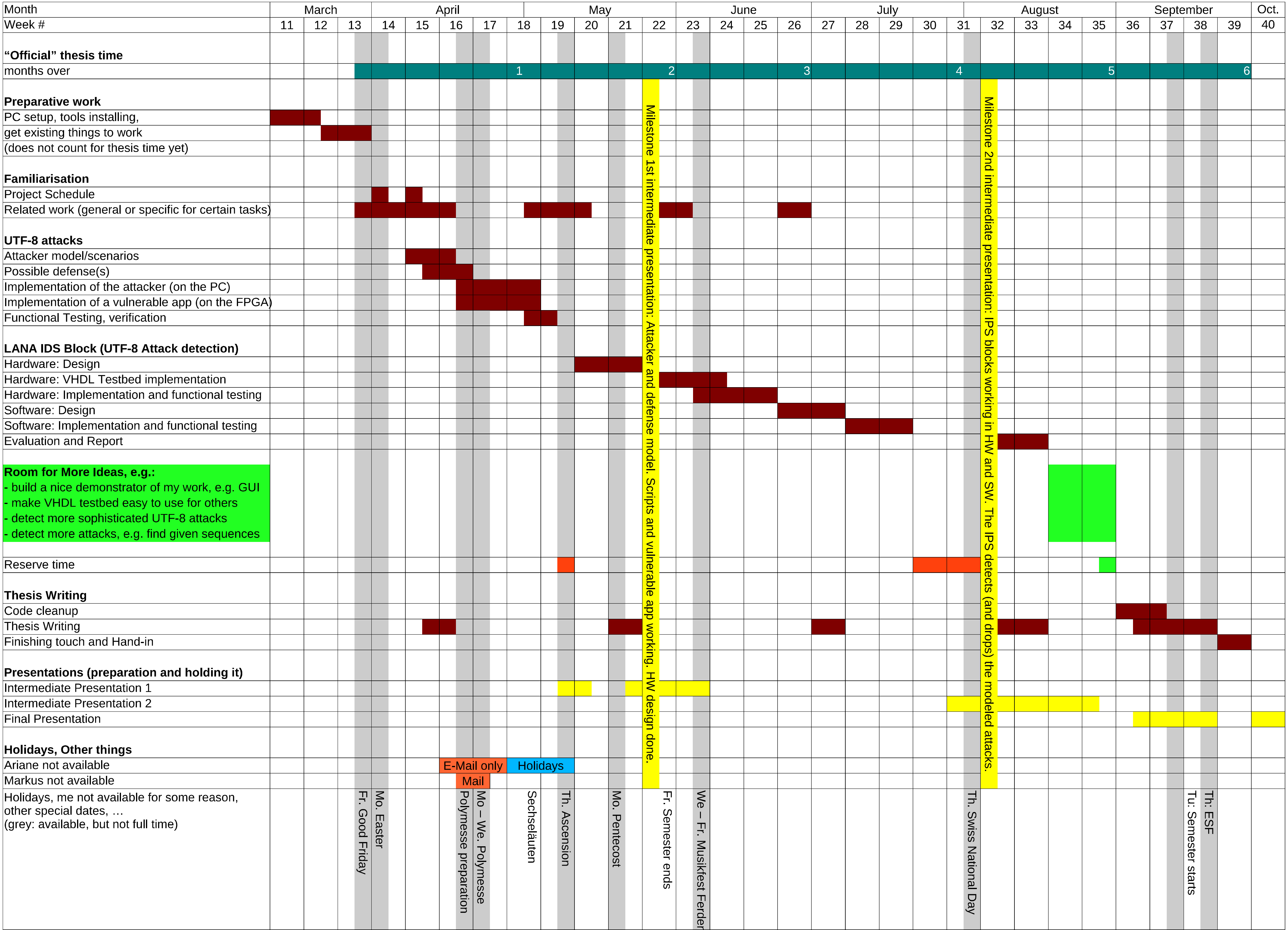
[1] ReconOS: Multithreaded Programming for Reconfigurable Computers: Description of the hardware/software architecture
[2] Reconifgurable Nodes for Future Networks: Description of how we would like to use the HW/SW architecture to build reconfigurable networks
[3] The Autonomic Network Architecture (ANA): Description of the ideas and SW prototype for configurable networks
[4] https://github.com/EPiCS/epics-org/blob/master/deliverables/D3-1_Architecture_And_Tool_Flow/D3-1_Architecture_And_Tool_Flow.pdf
[5] https://github.com/EPiCS/reconos/
Webpages:
http://www.ana-project.org
http://www.epics-project.eu
http://www.reconos.de

Appendix D

Project Schedule

Project Schedule

Master's Thesis, Stefan Kronig, kronigs@ee.ethz.ch



Acronyms

ANA Autonomic Network Architecture.

ASCII American Standard Code for Information Interchange.

CF Compact Flash.

CSV Comma-Separated Values.

DoS Denial of Service.

EPiCS Engineering Proprioception in Computing Systems.

FB Functional Block.

FIFO First In, First Out.

FPGA Field Programmable Gate Array.

FSM Finite State Machine.

GPGPU General Purpose computing on Graphics Processing Units.

h2s Hardware to Software FB.

HDL Hardware Description Language.

IDN Internationalised Domain Names.

IDPS Intrusion Detection and Prevention System.

IP Intellectual Property.

IP Internet Protocol.

IPS Intrusion Prevention System.

LANA Lightweight ANA.

MTU Maximum Transmission Unit.

NIC Network Interface Card.

NoC Network on Chip.

OS Operating System.

s2h Software to Hardware FB.

SoC System on Chip.

TCP Transmission Control Protocol.

UCS Universal Character Set.

UI User Interface.

UTF-8 8-Bit UCS Transformation Format.

VHDL Very High Speed Integrated Circuit Hardware Description Language.

XSS Cross site scripting.

References

[1] Julie D. Allen [et al.], *The Unicode Standard, Version 6.2.0.* The Unicode Consortium, 2012. Also available online at `http://www.unicode.org/versions/Unicode6.2.0` (last accessed in September 2013).

[2] various / MITRE, “CVE-2009-1535.” `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1535` (last accessed in September 2013), 2009.

[3] various / MITRE, “CVE-2010-3870.” `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3870` (last accessed in September 2013), 2010.

[4] D. Borkmann, “Lightweight autonomic network architecture,” Master’s thesis, ETH Zurich, jan 2012. Available online at `ftp://ftp.tik.ee.ethz.ch/pub/students/2011-FS/MA-2011-01.pdf` (last accessed in September 2013).

[5] G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May, “The autonomic network architecture (ana),” *Selected Areas in Communications, IEEE Journal on*, vol. 28, no. 1, pp. 4–14, 2010.

[6] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *ACM Trans. Comput. Syst.*, vol. 18, pp. 263–297, Aug. 2000.

[7] T. Becker, A. Agne, P. R. Lewis, R. Bahsoon, F. Faniyi, L. Esterle, A. Keller, A. Chandra, A. R. Jensenius, and S. C. Stilkerich, “Epics: Engineering proprioception in computing systems,” in *Proc. Int. Conf. on Computational Science and Engineering (CSE)*, p. 353–360, IEEE Computer Society, dec 2012.

[8] E. Lübbers and M. Platzner, “Reconos: Multithreaded programming for reconfigurable computers,” *ACM Trans. Embed. Comput. Syst.*, vol. 9, pp. 8:1–8:33, Oct. 2009.

[9] A. Agne, “Reconos tutorial - presentation.” Can be found in the ReconOS Git repository. Direct Link to Github: `https://github.com/EPiCS/reconos/blob/master/docs/tutorial/tutorial.ppt` (last accessed in September 2013), 2012.

[10] A. Keller, D. Borkmann, and S. Neuhaus, “Hardware support for dynamic protocol stacks,” in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ANCS ’12, (New York, NY, USA), pp. 75–76, ACM, 2012.

[11] R. Huber, “A dynamic hardware architecture for future networks,” Master’s thesis, ETH Zurich, jun 2012. Available online at `ftp://ftp.tik.ee.ethz.ch/pub/students/2011-HS/MA-2011-27.pdf` (last accessed in September 2013).

[12] Y. Yang, "Hardware encryption for embedded systems," semester thesis, ETH Zurich, feb 2013. Available online at `ftp://ftp.tik.ee.ethz.ch/pub/students/2012-HS/SA-2012-16.pdf`, last accessed in September, 2013.

[13] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol – HTTP/1.0." RFC 1945, May 1996. online, `http://www.ietf.org/rfc/rfc1945.txt` (last accessed in April 2013).

[14] M. Davis and M. Suignard, "Unicode Security Considerations," Unicode Technical Report #36, The Unicode Consortium, July 2012. Available online at `http://www.unicode.org/reports/tr36/` (last accessed in April 2013).

[15] G. Fumera, I. Pillai, and F. Roli, "Spam filtering based on the analysis of text information embedded into images," *Journal of Machine Learning Research*, vol. 7, pp. 2699–2720, Dec. 2006. Available online at `http://jmlr.csail.mit.edu/papers/v7/fumera06a.html` (last accessed in April 2013).

[16] E. Gabrilovich and A. Gontmakher, "The homograph attack," *Commun. ACM*, vol. 45, pp. 128–, Feb. 2002.

[17] M. Davis and K. Whistler, "Unicode Normalization Forms," Unicode Technical Report #15, The Unicode Consortium, Aug. 2012. Available online at `http://www.unicode.org/reports/tr15/` (last accessed in September 2013).

[18] Xilinx Inc., *LocalLink Interface Specification*, sp006 (v2.0) ed., july 2005. Available online at `http://www.xilinx.com/aurora/aurora_member/sp006.pdf`, last accessed in September, 2013).

[19] "IEEE Standard for Ethernet, section 1," *IEEE Std. 802.3-2012*, p. 56, 2012. Available online at `http://standards.ieee.org/about/get/802/802.3.html`, last accessed in September, 2013.

[20] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, Third Edition.* O'Reilly Media, Inc, 3 ed., 2005. Also available online at `http://lwn.net/Kernel/LDD3/` (last accessed in September 2013).

[21] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices." RFC 2544, Mar. 1999. online, `http://www.ietf.org/rfc/rfc2544.txt` (last accessed in September 2013).

[22] H. Kaeslin, *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication.* Cambridge University Press, 2008.

[23] Xilinx Inc., *MicroBlaze Processor Reference Guide*, ug081 (v9.0) ed., 2008. Available online at `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/mb_ref_guide.pdf`, last accessed in September, 2013).