An FPGA-based Scalable Platform for High-Speed Malware Collection in Large IP Networks

Sascha Mühlbach [#1], Andreas Koch [*2]

[#] *Secure Things Group*
Center for Advanced Security Research Darmstadt (CASED)
[1] sascha.muehlbach@cased.de

[*] *Embedded Systems and Applications Group*
Dept. of Computer Science, Technische Universität Darmstadt
[2] koch@esa.cs.tu-darmstadt.de

***Abstract*—With the growing diversity of malware, researchers must be able to quickly collect many representative samples for study. This is commonly achieved by harvesting the malware from honeypots: Insecure systems presenting a wide attack surface to the public Internet, aiming to attract attackers. However, software-based honeypots have both performance issues in light of 10+ Gb/s networks, as well as difficulties in preventing the compromise of the honeypot system itself. We present an architecture for a honeypot using dedicated hardware instead of a general-purpose processor. Our system is fast enough to keep up with high-speed networks and more resilient against subversion attempts than existing software solutions. It consists of a high-speed implementation of the Internet protocol stack attached to hardware-based emulations of vulnerable applications. A specialized implementation of the TCP protocol, capable of managing hundreds of thousands of simultaneous connections, allows the system to span large honeynets. The practical feasibility of the approach has been demonstrated on a real FPGA platform connected to a 10 Gb/s network interface.**

I. Introduction

To defend against Malicious software, short 'malware", anti-virus programs scan PCs continuously for patterns or anomalous behavior indicating a possible malware infection. But malware is evolving very quickly. Research indicates that the number of different malware variants detected per year has risen significantly from around 100k to 3M between 2005 and 2009 [1]. Timely malware capture and analysis has become essential to establish adequate defenses. One possibility is to use dedicated computer systems, that emulate vulnerabilities of applications to attract attackers.

Software packages [2], [3] exist for setting-up honeypot systems. However, software running on general-purpose processors always runs the risk of being compromised beyond the purpose of the honeypot and using it, instead, as a launch-pad for further attacks against other hosts on the Internet. Often, careful manual monitoring is required to detect and shut down a rogue honeypot. Furthermore, software-based solutions are severely taxed by current networking speeds of 10+ Gb/s.

As a solution, we proposed the idea of a malware collection honeypot realized entirely on dedicated hardware in [4]. The core of the system is a high-speed implementation of the basic Internet communication protocols. Attached to this core are several independent emulation engines (called vulnerability emulation handler, VEH), each dedicated to emulate a specific security flaw of an application.

However, the initial proof-of-concept implementation was covering the UDP protocol only and omitted the TCP protocol due to its complexity. To close this gap, we will now present a special stateless TCP hardware implementation, which is highly tuned to the application domain and able to handle hundreds of thousands of concurrent connections at line speeds of 10 Gb/s.

This paper is organized as follows: Section II briefly describes the core architecture and its major characteristics. Section III covers the details of our stateless TCP implementation, showing capabilities and limitations. Section IV discusses implementation specifics of the system on a real FPGA platform and is followed by a discussion of experimental results in Section V. We close with a conclusion and an outlook towards further research in the last Section.

II. System Architecture Overview

The TCP implementation will be integrated into our Net-Stage Architecture [4], which is a hierarchical design reflecting the levels of the Internet protocol stack. Within each stage, dedicated processing elements (called "handlers") perform their specific tasks in-line with the data flow. The handlers of adjacent stages are loosely coupled using buffers. In this fashion, new handlers can easily be "plugged-in" at the appropriate stage.

A. Core System

The core system consists of all handlers responsible for the Internet protocol stack. This includes the IP, UDP, and TCP protocol handlers as well as the ARP and ICMP implementation needed for the autonomous operation as a networked system.

Within each stage, packets are classified according to their protocol information, processed (e.g., checksummed) and then forwarded to the next handler by assigning the output data to the corresponding buffer. As a hard constraint, all handlers must provide a processing speed of at least the data rate of the

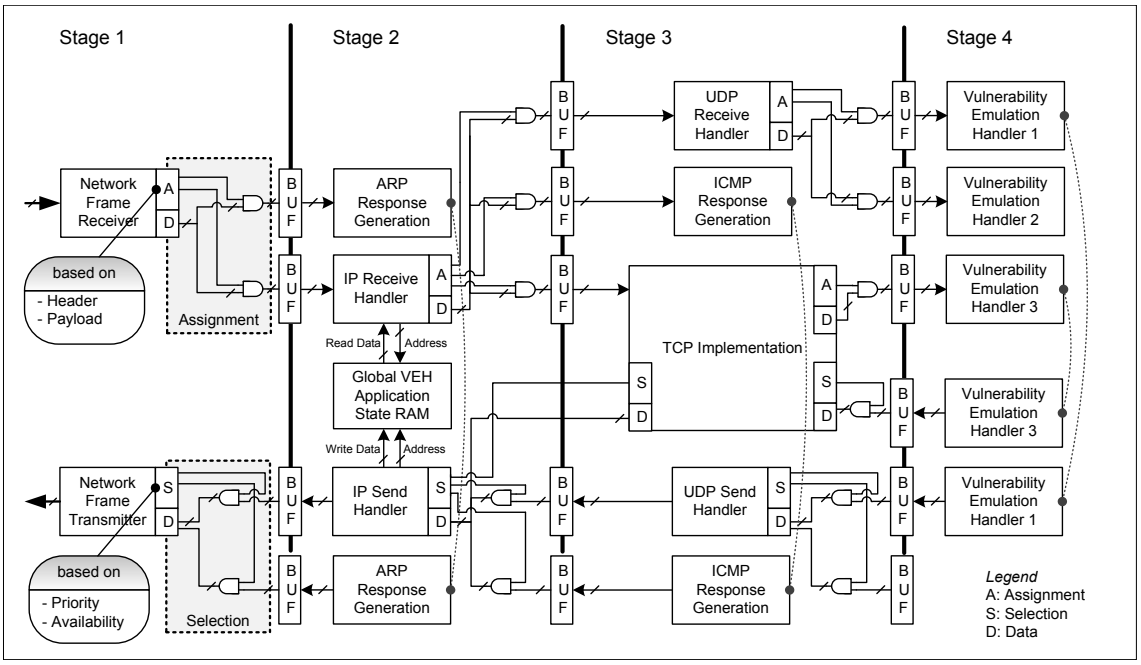

Fig. 1. NetStage Architecture of the malware collection network platform including the TCP implementation

network interface (10 Gb/s, in our case) to avoid congestion in the data path. As packets proceed upwards, lower-level protocol headers will be stripped away and replaced by an internal control header, which accompanies the packet throughout the various stages, carrying architecture-specific data.

B. Vulnerability Emulation Handlers

Packets passing the lower core stages will finally be forwarded to a responsible VEH which performs the actual malware detection and extraction. To this end, Stage 3 handlers have mapping tables that use matching rules to assign packets to specific Stage 4 VEHs. Due to the finely grained parallel processing possible in dedicated hardware, these rules can be very flexible and exploit both header and payload matching. E.g., HTTP requests could be forwarded to different VEHs based on the URL, emulating web servers running on Windows or Linux.

C. UDP and TCP Protocol Handling

Our architecture can easily handle UDP data streams (which are stateless), but TCP requires significantly more efforts. We went to considerable length to maintain our high-performance stateless architecture, while remaining compatible with TCP-speaking partners (despite violating some protocol specifications, which would be very expensive and/or slow to perform, see Section III for details).

D. VEH Global Application State Management

To keep our architecture stateless as much as possible, we only evaluate state information at the VEH level and only if this is absolutely required by the emulated application. To support this, we place a global state memory on-chip in the second stage along the IP handler (see Figure 1). When an incoming IP packet passes the IP receive handler, the current state information for this connection is retrieved from the state memory and attached to the packet in a custom field of the internal control header.

III. TCP Implementation

Originally, TCP is a stateful protocol which establishes a reliable communication channel between two communication partners on top of the (itself unreliable) IP protocol. Fundamentally, the reliability is achieved by two counters on each side, one for the cumulative number of bytes received and one for the cumulative number of bytes transmitted since establishment of connection [5]. The TCP stack divides the data to be sent into TCP "segments". Whenever a TCP segment is transmitted, it carries the current byte offset (called sequence number, short: SEQ) of the data within this segment relative to the beginning of the connection as part of the TCP header. If all expected bytes were received, the receiver notifies the sender of the successful in-order reception of bytes by sending its current number of cumulative received bytes including the actual ones as acknowledgment (ACK) back to the sender. By tracking these numbers, both sides can provide an ordered data stream and determine if there a segments outstanding or lost.

A. Stateless Design

By its design, TCP normally requires to maintain state information (e.g., the counter values) for each open connection. However, for applications such as our malware collection in large IP networks, potentially hundreds of thousands of open connections need to be managed at 10+ Gb/s. This would be

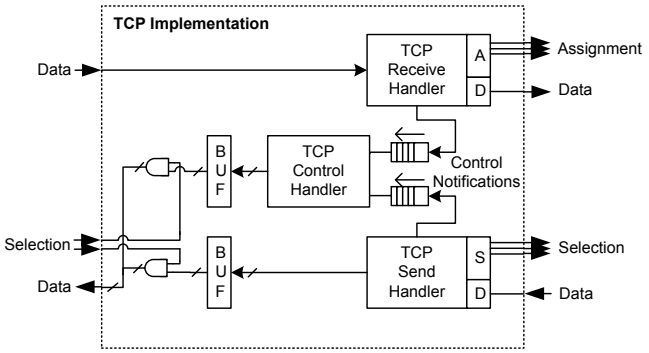

Fig. 2. Components of the TCP implementation

very difficult to handle with conventional TCP implementations (e.g., [6]). Instead, we will use a custom implementation highly tuned for our application domain, which emphasizes multi-connection performance over single-connection throughput. The latter is irrelevant to our domain, as it would involve the highly unlikely use-case of a *single* attacker interacting with us at full line-speed.

To this end, we will avoid storing local state and instead re-use TCP header information to reconstruct the session state. Concepts of stateless TCP implementations have been proposed in the past, e.g., as part of a TCP port scanner [7]. Another approach moves the state-tracking functionality to the client [8]. But since that requires modified clients, this technique is inapplicable to our application.

For our solution to the stateless TCP problem, we exploit the nature of interactions with the honeypot: We will only react to incoming request packets and never self-initiate traffic. This means that we can always rely on the ACK and SEQ header data of the incoming packets (which is retained in the internal control header) to reconstruct the connection state. As sketched in Figure 2, the TCP implementation consists of receive, send, and control handlers, easily integrated into our layered system architecture (Fig. 1). While the send and receive handlers manage checksum calculation and header inspection, the control handler is responsible for the connection establishment process and handling of other control messages.

B. TCP Connection Establishment

TCP connections are established using the well-known "three-way handshake". Client and server exchange random initial sequence numbers (identified by a set SYN flag in the TCP header), and mutually acknowledge them. Random numbers are used to defend against connection hijacking attacks, where the attacker predicts the next sequence number and uses it to inject a packet of his own into the connection. Instead of storing the sequence numbers (which would again require memory accesses), we use a simplified mechanism (shown in Figure 3) similar to SYN cookies [9].

After the connection is established, the corresponding VEH is notified by generating an internal dummy packet indicating a new connection by a special flag which is set within the

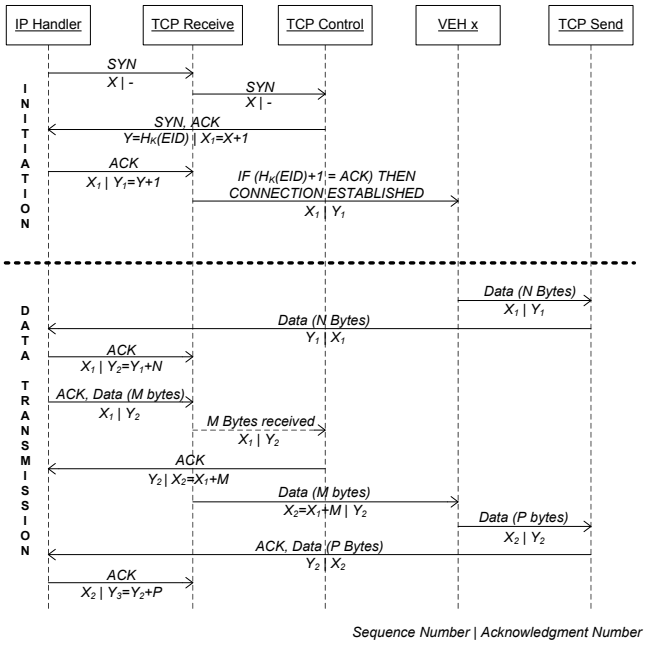

Fig. 3. TCP sequence of the stateless implementation

control header. This dummy packet is required, e.g., to induce the VEH to send out an initial welcome message over the new connection.

C. TCP Data Transmission

Data can now be exchanged over the newly established connection (Fig. 3). To remain stateless, we compute outgoing SEQ and ACK numbers from the incoming packet headers that accompany the packet in the internal control header. While compatible with the TCP protocol, this approach does have limitations:

- *Lost incoming single packets cannot be detected.* Here, we rely on the sender to just retransmit the packet after we have not acknowledged it within the time-out period.
- *Packets arriving out-of-order or packets lost from a packet group cannot be detected.* We avoid this situation (for consecutive transmission of full segments) by offering a constant receive window size equal to the MSS. Thus, at most one packet may be unacknowledged at a time and the sender must wait for our acknowledgment (offering a new window size) before it can send the next packet. This is one of the cases where we have accepted deterioration of per-connection throughput to raise the number of manageable connections.
- *Unnecessarily retransmitted packets are not detected.* This can occur, e.g., if our initial acknowledgment got lost. We will then just acknowledge the packet again, satisfying the sender. However, since we cannot detect the superfluous packet, it will be passed upward in our architecture for processing, possibly triggering actions twice. VEHs must be aware of this possibility. In practice, this is easily achievable since VEHs are often stateless

already, and a duplicate response packet will be silently discarded at the remote site (due to SEQ and ACK numbers being equal to an already received packet).

Independent of our sequence number scheme, we currently aim to save hardware resources by not retransmitting packets. We justify this severe measure with the observation that only about 10% of the TCP connections transferring less than 50 kB of data on the Internet suffer from packet loss at all [10]. Typical vulnerability emulations will exchange a much smaller amount of data within a single connection, either because they are very simple (see Section IV), or because they create a new connection for every request (such as a simple web server). This limitation is not a fundamental characteristic of our architecture: If future live evaluations in a production environment demonstrate the practical need for retransmission, appropriate logic could be added to the handlers without invalidating the general approach.

D. Congestion Control

We handle congestion (respecting the available space in the client receive buffer) in two ways. The simpler one relies on passing the client window size together with the incoming packet in the internal control header, and the corresponding VEH only sending data up to this limit. However, this is not appropriate if the VEH needs to send more data than the client currently has space available for. In this case, we cannot avoid tracking state. However, by moving the state-keeping to the application-level VEH (instead of having it in the TCP layer), we can maintain state selectively only for the few VEHs that actually require it, instead of all (possibly hundreds of thousands) of connections. To notify such a connection-tracking VEH that it can send more data, we do forward to it the dedicated acknowledgment packets (which are discarded for stateless VEHs). When a VEH receives such a notification, new transmit data is quickly generated by the VEH (usually without going to slow external memory) until the window is filled again. This process repeats until all bytes have been sent.

IV. Implementation

The overall system has been implemented on an FPGA-based BEEcube BEE3 reconfigurable computing platform, which is equipped with eight 10 Gb/s network interfaces. Our BEE3 has four Xilinx Virtex 5 FPGAs (2x LX155T, 2x LX95T), of which we are currently only using one.

A. Core Architecture

Physical network connectivity is provided by the Xilinx XAUI and 10G MAC IP cores. To achieve a raw throughput of 10 Gb/s on the 64b data words output by the 10G MAC, the entire system operates on a clock speed of 156.25 MHz. Within the core architecture (IP, UDP, TCP), the processing speed is doubled to 20 Gb/s by widening the data path to 128b. This was done to cope with brief variations in intra-system throughput (e.g., a data-dependent latency in an active VEH) by giving the system the ability to "catch-up" with normal 10 Gb/s traffic by burst-processing the data accumulated in the inter-stage buffers during the short stall.

B. Vulnerability Emulation Handlers

To test the functionality of the system, we have created a number of VEHs emulating a broad spectrum of vulnerabilities and applications. As UDP-based VEHs, we took the SIP VEH from [4] and created an additional VEH, emulating a MSSQL server vulnerability which was the target of the Slammer worm [11]. To test the TCP implementation, we built a web and mail server emulation (see below). The heart of a VEH generally consists of a finite state machine (FSM), which processes incoming packets, creates a response, and extracts the malware (if detected). Extracted malware is currently send to a management station PC via an UDP packet with a hardwired (set during compile time) IP and MAC address. Beyond the FSM, many VEHs also contain fast parallel pattern matchers in dedicated hardware units.

1) Simple Web Server (TCP): With the large number of attacks targeting vulnerable web applications [12], we have implemented a VEH emulating a simple web server. It consists of an on-chip ROM holding predefined HTML pages that the web server should serve, and a string matching section that determines the correct response to incoming requests. The implementation responds to requests for the root URL "/" with a simple login form (size: 1kB) pretending to be a webmail service. The HTTP headers indicating the server version (which should be a vulnerable one) are also stored in the ROM and are sent back together with the HTML page.

Instead of sending predefined HTTP response headers, they could also be generated on the fly. Examples include the current value for the DATE field, or the real Content-Length if some parts of the response packet are also created dynamically (e.g., login information previously submitted by the client).

Even this simple emulation suffices for many use-cases. It can search for extremely long URLs containing special characters (typical of a web server exploit attempt), or pretend to be a real webmail service and logging what type of user / password combinations an attacker is typing in while trying to get access. In many cases, the web server VEH can execute stateless and just rely on the requested URL to determine its next action. This is even true for a login process, which will just redirect the attacker to the next URL.

2) SMTP Mail Server (TCP): The SMTP mail service is also a commonly attacked target. Thus, we implemented a mail server VEH that accepts incoming mails and pretends to be an open relay server. The implementation of the VEH consists of a pattern matching unit that recognizes to SMTP commands and an FSM that emulates a SMTP dialogue. In contrast to the web server VEH, the response packets of the SMTP VEH are small and consist only of a status code and a short string. As spam mails often contain malware or links to malware, the mails received by this emulation can be forwarded to the management station, where they can be further inspected. A future refinement of this VEH could also recognize identical messages in the VEH using structures such as Bloom filters to avoid overwhelming the management station.

TABLE I
SYNTHESIS RESULTS FOR COMPONENTS AND COMPLETE SYSTEM

Stages	Handler	LUT	Reg. Bits	BRAM
Stages 1,2,3	Core System w/o TCP	4,976	4,208	58
	TCP Implementation	3,162	2,553	15
Stage 4	SIP VEH	1,119	364	5
	MSSQL VEH	788	638	5
	Web Server VEH	924	576	9
	Mail Server VEH	755	563	5
Core System incl. VEH		**11,724**	**8,902**	**97**
mapped incl. MAC + XAUI (% of XC5VLX155T)		**14,441** (14%)	**12,323** (12%)	**110** (51%)

V. EXPERIMENTAL RESULTS

The design was synthesized using Synplify Premier 9.6.2 and mapped with Xilinx ISE 12.1, targeting a Virtex 5 LX155T FPGA and aiming for a clock speed of 156.25 MHz. Functionality and performance tests were performed both by simulation as well as on an actual BEE3 machine. The BEE3 was connected using a single 10 Gb/s interface to a dedicated quad-Xeon Linux server, which generated the test traffic load.

A. Synthesis Results

Table I gives a summary for the core, the various VEHs, and the complete system. VEHs sizes vary only a little. Among the four VEHs, the SIP VEH requires the most LUTs, as it contains the most complex pattern matching algorithms. The other VEHs perform a larger number of simple operations, but require more data storage (resulting in a higher number of register bits). BlockRAMs are only needed for the inter-stage input and output buffers, except for the web server VEH, where they hold the website response packets as ROMs.

When considering the core system, around 40% of the LUTs are taken up by the TCP implementation. But due to our stateless design, the size is still small relative to the entire device area. Together with the remaining core handlers (ARP, ICMP, IP, UDP), in total only around 8,000 LUTs are occupied.

B. Stability and Performance

To verify the stability and performance of the system under real conditions, we performed multiple tests on the BEE3 in a live environment. The SIP and MSSQL VEHs were fed with corresponding UDP packets generated synthetically. The correct response(s) of the handlers were verified by monitoring the network traffic.

The Mail Server VEH compatibility was tested by sending mails through Thunderbird and via Postfix. To accept a single mail using a SMTP dialog, the VEH required 83ms, while the Postfix software SMTP service on the Linux server required 153ms.

The Web Server VEH was tested using Firefox and with Apache Bench 2. For the 1 million of pipelined requests of the latter, the VEH replied in 22us (mean), while the software Apache required 100us (mean) to serve a page (fully loading all eight hyperthreaded cores of the Linux server).

VI. CONCLUSION AND FUTURE WORK

We have presented a flexible system architecture to realize a high-speed hardware-accelerated malware collection solution. With its stateless TCP implementation, it is capable of handling large numbers of simultaneous connections. The vulnerability emulation handlers actually performing the malware extraction rely on reconfigurable hardware to efficiently implement complex regular expression matchers or fast FSMs. Thus, the system can keep up with 10 Gb/s network traffic spanning wide address ranges.

Due to its hierarchical and modular architecture, it is easily extensible, but resilient to attackers aiming to compromise the honeypot itself. Results gathered from a real FPGA implementation demonstrate the feasibility of our approach.

In the future, we will develop more vulnerability emulation handlers as well as a complete and autonomous malware analysis flow, linking the hardware-accelerated honeypot with software-based malware analysis tools. We also plan to test our solution in a production environment connected to the Internet (e.g., at a university or large ISP connection).

Mid-term research will also consider dynamic reconfiguration to swap vulnerability handlers at run-time, as well as specialized tools to ease the development of handlers. Additionally, allowing the system to span FPGA boundaries will also be the subject of further investigation.

ACKNOWLEDGMENT

This work was supported by CASED (www.cased.de).

REFERENCES

[1] "Internet security threat report, volume xv," Symantec, 2010. [Online]. Available: http://www.symantec.com
[2] "Nepenthes." [Online]. Available: http://nepenthes.carnivore.it
[3] "Honeyd." [Online]. Available: http://www.honeyd.org
[4] S. Mühlbach, M. Brunner, C. Roblee, and A. Koch, "Malcobox: Designing a 10 gb/s malware collection honeypot using reconfigurable technology," in *FPL '10: Proceedings of the 20th International Conference on Field Programmable Logic and Applications*. IEEE Computer Society, 2010, pp. 592–595.
[5] J. Postel, "Transmission Control Protocol," RFC 793 (Standard), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt
[6] A. Dollas, I. Ermis, I. Koidis, I. Zisis, and C. Kachris, "An open tcp/ip core for reconfigurable logic," in *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 2005, pp. 297–298.
[7] D. Kaminsky, "Scanrand." [Online]. Available: http://www.secureworks.com/research/articles/scanrand
[8] A. Shieh, A. C. Myers, and E. G. Sirer, "Trickles: a stateless network stack for improved scalability, resilience, and flexibility," in *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. USENIX Association, 2005, pp. 175–188.
[9] W. Eddy, "TCP SYN Flooding Attacks and Common Mitigations," RFC 4987 (Informational), Internet Engineering Task Force, Aug. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4987.txt
[10] G. Maier, A. Feldmann, V. Paxson, and M. Allman, "On dominant characteristics of residential broadband internet traffic," in *IMC '09: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009, pp. 90–102.
[11] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Security and Privacy*, vol. 1, no. 4, pp. 33–39, 2003.
[12] S. Gordeychik, "Web application security statistics," 2008. [Online]. Available: http://projects.webappsec.org