A Reconfigurable Hardware Platform for Secure and Efficient Malware Collection in Next-Generation High-Speed Networks

Sascha Mühlbach
Secure Things Group
Center for Advanced Security
Research Darmstadt (CASED)
Email: sascha.muehlbach@cased.de

Andreas Koch
Embedded Systems and Applications Group
Dept. of Computer Science
Technische Universität Darmstadt
Email: 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 can be done, e.g., by using honeypots. As an alternative to software-based honeypots, we propose a single-chip honeypot appliance that is entirely hardware-based and thus significantly more resilient against compromising attacks. Additionally, it can easily keep up with network speeds of 10+ Gb/s and emulate thousands of vulnerable hosts. As base technology, we employ reconfigurable hardware devices whose functionality is not fixed by the manufacturing process. Furthermore, a special technology available in modern reconfigurable devices allows to alter part of the functionality even during operation and will be used to create a hardware virtualization layer. Beyond the technical aspects, we present improvements to the platform, aiming to simplify management and updates. To this end, we introduce the domain-specific language VEDL, which can be used to describe the honeypot behavior in a high-level manner by security experts not proficient in hardware design.

1. Introduction

Malicious software, short "malware", is doubtlessly one of the main threats to computer users on the Internet today. It is spread by security flaws in regular software applications, which can, e.g., be exploited via network communications through the Internet. To defend against malware attacks, security researchers continuously collect as many malware samples as possible for analysis. Timely malware capture and analysis has become essential to establish adequate defenses, e.g., creating new signature files for anti-virus programs.

The setting of honeypots, which emulate vulnerable applications, is one method of gathering large amounts of attack code automatically. To present a large attack surface, honeypots are connected directly to the Internet and often respond to entire ranges of IP addresses.

Software packages such as Nepenthes [1] or Honeyd [2] exist for setting-up such honeypot systems. However, software running on general-purpose processors always runs the risk of being compromised beyond the purpose of the honeypot and used, instead, as a launch-pad for further attacks against the Internet or the researcher's own internal network. 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.

2. Research Rationale

Against this background, we introduced the idea of a malware collection honeypot, composed entirely of dedicated hardware blocks instead of software components [3]. As dedicated hardware is built to perform a "specific" task, it cannot be abused by an attacker to perform arbitrary operations. Thus, the entire system is significantly more resistant to compromising attacks than a system based on general purpose CPUs that can run any code. Additionally, with the high level of parallel processing achievable in hardware devices, the system can handle frequent connections to many endpoints, making it ideal for use in 10+ Gb/s networking environments.

As honeypot systems need to be continuously updated with upcoming exploits, the hardware blocks have to be changed on a regular basis. We achieve this by using Field-Programmable Gate Arrays (FPGAs), which offer a functionality similar to hard-wired chips, but can be altered after fabrication. These updates can be even performed during operation without any system outage, thus offering the possibility to build a hardware virtualization layer on top of the

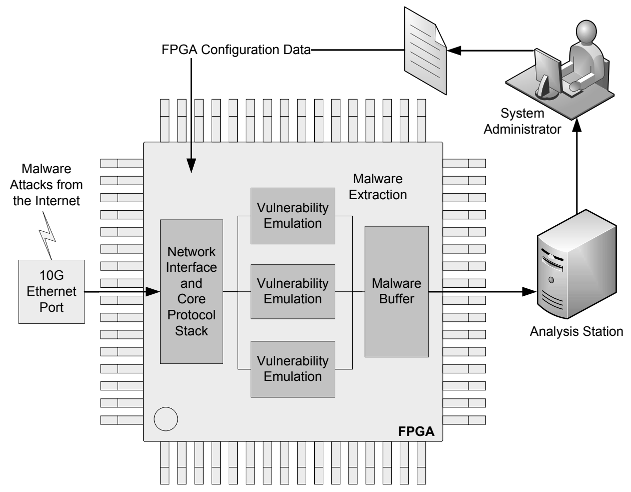


Figure 1. Hardware-based malware collection scenario

underlying physical chip.

However, actually implementing functionality on FPGAs is more complex than performing software updates. It requires expertise in digital logic design, computer architecture, and using languages (e.g., VHDL, Verilog) and tool flows unfamiliar to most software developers. As an alternative, this work introduces the new domain-specific Vulnerability Emulation Description Language (VEDL), which allows security experts to concisely describe vulnerabilities, but is suitable for automatic compilation to hardware emulation modules to be used in the honeypot system.

The paper is organized as follows: Section 3 gives an overview of the core platform architecture and its major characteristics, while Section 4 concentrates on the networking aspects. Section 5 explains the concept of the so-called vulnerability emulation handlers (VEH), which actually realize the honeypot behavior, followed by details of the new high-level description language. Section 6 covers the architectural details of the hardware virtualization layer. In Section 7, we will briefly present results of our prototype system, implemented on an actual FPGA networking platform, before closing with a conclusion and an outlook towards further research in Section 8.

3. Platform Architecture

Figure 1 shows the application scenario. The hardware-based honeypot will act just as any software honeypot would do. The FPGA contains all functionality for the honeypot and the network connectivity on a single chip. The core of the system is a high-speed hardware implementation of the basic Internet communication protocols and supporting services. Currently, the core contains functions to handle the IP, UDP and TCP protocol as well as ARP and ICMP messages. While the first is essential to act as an autonomous networked system, the latter is used especially in this application scenario to provide ICMP PING responses to attackers probing for IP addresses.

Attached to the core are several independent Vulnerability Emulation Handlers (the VEHs, see Section 5), each dedicated to emulate a specific security flaw of an application. If a request exploiting any of these flaws is detected by the emulation engines, the pertinent data received is stored and forwarded to a management station, where the potential malware can be extracted and analyzed further using special tools (e.g., CWSandbox [4]). While the core provides base functionality, the vulnerability emulations contain the actual honeypot functionality and need to be updated when new vulnerabilities are discovered. For seamless operation, we have improved our initial platform to support run-time updates of these VEH hardware blocks without stopping the system by using Partial Reconfiguration (see Section 6), a technique that alters the functionality of only part of the FPGA while the rest continues to operate.

3.1. Hardware Implementation

Figure 2 shows the block diagram of the major hardware components. The structure follows our flexible NetStage Architecture [3], which is a hierarchical design of dedicated processing elements (called "handlers"), that perform their specific tasks in-line with the data flow to support high transmission rates. These handlers employ many acceleration technologies available on dedicated hardware, such as pipelining and parallel processing (e.g., for fast regular expression matching). Buffers assure proper inter-stage decoupling and limit the impact of data rate variations.

NetStage can be divided into three major parts: the network core (operating at 20 Gb/s), the management section (e.g., for system updates) and the vulnerability emulation section (operating at least at 10 Gb/s). The latter contains slots that can be filled (programmed) with specific VEH hardware engines. The communication between the core and the VEHs is based on messages. These messages contain the network packet data and a custom internal header that contains control data. Packets can be routed through the architecture based on certain criteria (port, IP address etc.). With the performance of the dedicated hardware, very flexible routing rules are supported. The NetStage-internal routing is also updated during the reconfiguration process, e.g., when new VEHs should be inserted in the data flow.

The core also contains a global application memory supporting stateful operation for all VEHs that need it. As the VEHs can be reprogrammed in-system, they should not have internal state (e.g., session information). Thus, VEH

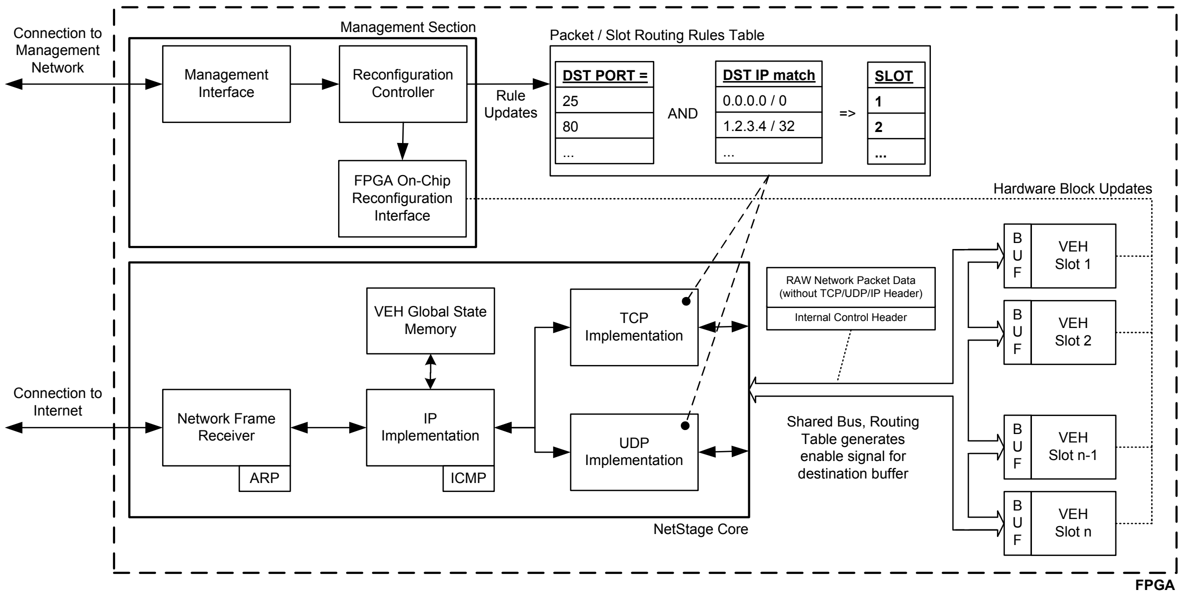


Figure 2. Core architecture of the malware collection network platform

state is stored centrally in the global memory and attached to every message that passes the core (state thus accompanies packets on its flow through Net Stage). In that fashion, VEHs can be updated without any impact on the connections currently active.

4. Networking Features

The system contains all required protocols to support autonomous Internet communication. As we want to present the illusion of large networks of vulnerable hosts (possibly tens of thousands), we need to actively deal with incoming ARP requests and send replies for all the IP addresses managed by the honeypot. To ease the configuration, we implemented an ARP responder that replies to every request for an IP address by denoting the network interface of the honeypot as responsible device (simply swapping source and destination addresses). To prevent the honeypot from taking over IP addresses already used in the network, the administrator can specify certain IP addresses or ranges to which the honeypot will not respond (e.g., one of the routers). Using this approach, the hardware honeypot appliance can be simply put into a network behind a router and will respond to any IP addresses that are forwarded by the router to that network.

As an example, Figure 3 shows the hardware implementation diagram of the ARP response generation process according to the simple swap technique described above. Packet data is arriving continuously with a word size of 8 bytes per clock cycle. With pipelining and parallel pattern matching, we can generate the ARP response in the same number of clock cycles as the request is received. This zero cycle overhead allows the module to run at full line speeds, while the small pipeline latency of only one clock cycle keeps the buffer requirements low. This is a good example of the advantages we can enjoy when building a networking system on dedicated hardware. Other hardware blocks of our system all follow a similar approach.

For IP and UDP packets, address and port information accompanies the packets through NetStage in the internal control header. Thus, reply packets can be sent without referring to extra state storage.

TCP requires more complex handling. A straightforward approach would require too much state storage for the number of connections we want to handle (hundreds of thousands). While it is possible to integrate full TCP functionality into NetStage (such as [5]) we decided to follow a different approach. As an alternative, we created a lightweight hardware-based TCP stack [6] highly optimized for our application scenario, which has significantly lower resource demands than the general solution.

Our approach relies on a stateless server approach: Instead of storing session information, the sequence (SEQ) and acknowledgment (ACK) numbers contained within the TCP header of the client's request packet are used to reconstruct the current state, allowing appropriate response packets to be sent back. Specifically, the ACK and SEQ numbers are stored within the internal control header together together with further TCP data (e.g., window size). The TCP handler can now perform header processing simi-

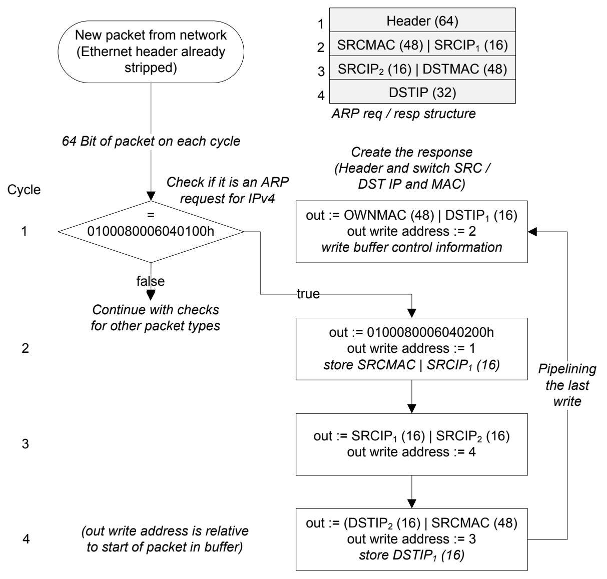

New packet from network (Ethernet header already stripped)
64 Bit of packet on each cycle
Cycle
Check if it is an ARP request for IPv4
= 0100080006040100h
false
Continue with checks for other packet types
true
Header (64)
SRCMAC (48) | SRCIP1 (16)
SRCIP2 (16) | DSTMAC (48)
DSTIP (32)
ARP req / resp structure
Create the response (Header and switch SRC / DST IP and MAC)
out := OWNMAC (48) | DSTIP1 (16)
out write address := 2
write buffer control information
out := 0100080006040200h
out write address := 1
store SRCMAC | SRCIP1 (16)
out := SRCIP1 (16) | SRCIP2 (16)
out write address := 4
out := (DSTIP2 (16) | SRCMAC (48)
out write address := 3
store DSTIP1 (16)
Pipelining the last write
(out write address is relative to start of packet in buffer)

Figure 3. Zero-cycle overhead ARP response generation

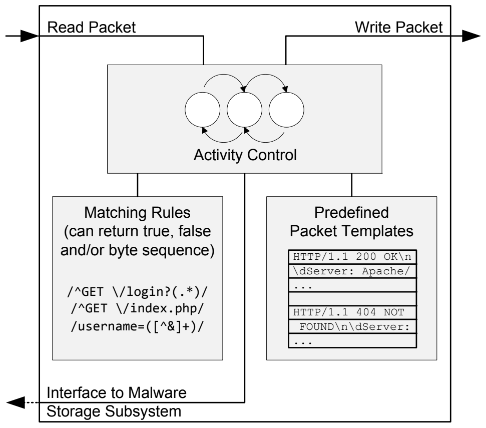

Read Packet
Write Packet
Activity Control
Matching Rules (can return true, false and/or byte sequence)
/^GET \/login?(.*)/
/^GET \/index.php/
/username=([^&]+)/
Predefined Packet Templates
HTTP/1.1 200 OK\n\dServer: Apache/
...
HTTP/1.1 404 NOT FOUND\n\dServer:
...
Interface to Malware Storage Subsystem

Figure 4. Schematic overview of the VEH components

lar to that of the UDP and IP handlers, only for TCP control messages is a dedicated handler required. As we do not store the sequence numbers, we use a technique similar to SYN cookies [7] to detect the third packet of the three-way-handshake. If a new connection is established, a dummy message is created and routed to the responsible VEH. This is required for application emulations where the client waits for a greeting of the server directly after connection establishment.

This alternative implementation is able to handle hundreds of thousands of concurrent connections at line speeds of 10 Gb/s. However, while it is *compatible* with regular TCP clients, it has some limitations. E.g., packets arriving out-of-order are not detected. We try to anticipate this by setting the maximum sequence size equal to the window size. Thus, for consecutive transmission of segments, only a single packet should be on the line at once. While this does reduce the performance achievable for a *single* connection, it does not affect our main use-case, as it is very unlikely that we will receive requests from a *single* attacker at the line speed of 10 Gb/s. Finally, there will be a small number of clients that will fail to establish a connection given our protocol design choices. Since, at worst, we will not receive malware from them, this is also an acceptable trade-off.

5. Vulnerability Emulation Handler (VEH)

As described above, the actual honeypot functionality is implemented in independent dedicated hardware modules: the VEHs. To assure security and performance, the VEH functions are hard-wired and only changed by reconfiguration. Each VEH handles a single vulnerability (or a class of related ones) and implements only the minimum functionality required to trigger an exploit. Our VEHs react only to incoming packets and never initiate outgoing connections on their own.

VEHs share the common structure shown in Figure 4. The external interface consists of connections to the input and output buffers holding the packets for this VEH. A central state machine manages reading and writing from/to these buffers and implements the behavior to emulate the vulnerable application. A set of parallel string matching units supports the state machine by extracting information from incoming packets to initiate state transitions, and to detect malicious requests. Outgoing packets are composed by filling-in the appropriate fields in stored packet templates.

Figure 5 shows the implementation of a simple web server emulation. It consists of a ROM holding the predefined HTML pages to be served as well as the HTTP header information that will be added to each response (e.g., the server version). The VEH is not implementing the full HTTP protocol, instead it composes response messages according to predefined rules out of the fragments stored in the ROM. As part of the main state machine, a string matching section determines the correct response to incoming requests. This VEH could, e.g., be used to emulate a web mail service and look for brute-force login attacks, or requests that try to exploit a vulnerability of either the web server software or the web application.

Instead of sending predefined HTTP response headers, the hardware is also capable of generating replies dynamically on the fly. However, such dynamic assignments require more hardware resources than static assignments and

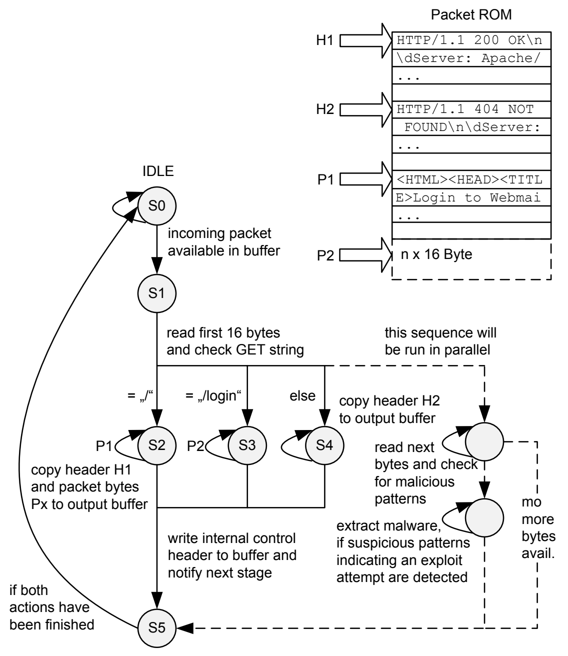

Packet ROM
H1
HTTP/1.1 200 OK\n
\dServer: Apache/
...
H2
HTTP/1.1 404 NOT
FOUND\n\dServer:
...
P1
<HTML><HEAD><TITL
E>Login to Webmai
...
P2
n x 16 Byte
IDLE
S0
incoming packet available in buffer
S1
read first 16 bytes and check GET string
this sequence will be run in parallel
= „/“
= „/login“
else
copy header H2 to output buffer
P1
S2
P2
S3
S4
read next bytes and check for malicious patterns
copy header H1 and packet bytes Px to output buffer
extract malware, if suspicious patterns indicating an exploit attempt are detected
mo more bytes avail.
write internal control header to buffer and notify next stage
if both actions have been finished
S5

Figure 5. Implementation of a web server VEH

should only be used in moderation.

5.1. VEH Description Language (VEDL)

Despite the performance of VEHs realized in dedicated hardware, the large effort to implement new VEHs is not suitable for the quickly changing security landscape. With VEDL, we propose a domain-specific language both suited for concise vulnerability description by non-hardware experts, as well as automatic compilation to NetStage VEH blocks. In contrast to compiling general purpose languages such as C to hardware, the VEDL-to-VEH translation process is much easier. This is both due to limiting the input language to a specific problem domain, as well as the internal structure of VEHs, which follows a similar pattern across all VEHs (FSM, pattern matchers, actions).

For our initial VEDL release, we chose the Mealy finite state machine as the underlying compute paradigm. While more complex schemes are of course possible, none of the VEHs we have examined so far would have required them. The outputs of the FSM correspond to actions and state transitions. Both can depend on conditions (e.g., pattern matches) and are initiated only as reactions to incoming packets. The following actions are currently supported:

- **send:** Send a predefined response packet. Optionally, dynamic custom data extracted from the *current* request (e.g., User Name or Session ID) or calculated by the system (e.g., Date), could be added.
- **close/reset:** Send a TCP close or reset notification.
- **log:** Copy the current request to the malware memory, in which data can be accumulated.
- **notify:** Notify management station that a potentially malicious request has been detected and can be retrieved from malware memory.
- **state:** Set state for the next state transition.

Actions can be arbitrarily combined (e.g., also occur more than once). Conditions can check both packet contents (deep packet inspection) as well as explicitly stored state:

- Matching a byte pattern (can also be an ASCII string) within the data of the request.
- Matching a field of the internal control header (which also contains per-connection state from the global state memory).

Regular expression matching is based on a subset of the Perl regular expression syntax. The special variable “$_” contains the data of the current request. The data of the control header is accessible through special keywords for each field. Individual conditions can be composed using logical operators to form more complex expressions. If no conditional matches, the FSM will remain in its current state (but it can generate different outputs).

Listing 1 shows an excerpt of a simple SMTP mail receiver in VEDL that copies the mail data to the malware memory. Note that the description starts with the information to configure the VEH routing table.

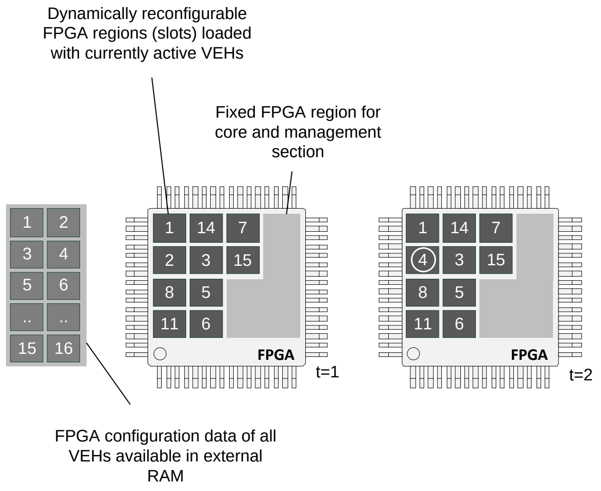
6. Hardware Virtualization

Based on the experimental results evaluating the MalCoBox prototype (see Section 7), we estimate that we can fit around 40 VEHs on a current medium-sized FPGA chip. While this should already be sufficient for a broad range of application scenarios, there will be some use-cases demanding a larger number of available VEHs, especially when the system should run unattended for a long period of time. This can be achieved by exploiting the capability of modern FPGAs to partially reconfigure the device with new functionality while the rest of the chip keeps performing the originally configured operations [8]. This mechanism can not only be used to easily update the system functionality, but also to realize a fully autonomous self-adaptable system that can alter parts of its functionality during operation to better handle changing conditions.

We will employ this technology to extend our base platform with a hardware virtualization layer that can be used

```
SMTP_VEH {
  # configuration section for the routing table
  config {
    protocol TCP;
    port 25;
    ip any;
  }
  # activity definition of the state machine that
       represents the vulnerability emulation
  fsm {
    # this state is always evaluated
    *:      if (newConnection) {
              send("220: bee3 SMTP");
              state = WELCOME;
            }
    ...
    DATA:   if ($_ =~ /^DATA/) {
              send("354 GO ON");
              state = MAIL;
            }
    MAIL:   log;
            if ($_ =~ /\r\n.\r\n$/) {
              notify;
              send("250 OK queued");
              state = NEXT;
            }
    ...
```

Listing 1. Sample VEDL description of a mail server emulation



Dynamically reconfigurable FPGA regions (slots) loaded with currently active VEHs
Fixed FPGA region for core and management section
1
2
3
4
5
6
..
..
15
16
1
14
7
2
3
15
8
5
11
6
FPGA
t=1
1
14
7
4
3
15
8
5
11
6
FPGA
t=2
FPGA configuration data of all VEHs available in external RAM

Figure 6. Partitioning the FPGA into dynamically reconfigurable regions for the VEHs

to swap VEHs in and out based on the current demand [9]. With this approach, it is possible to effectively expand the capacity of the underlying physical chip by time-multiplexing some of the actual hardware ressources.

On the technical level, we divide the dynamically reconfigurable part of the FPGA into a number of rectangular regions that contain a well-defined mix of processing elements (logic and memory blocks). These regions are reserved for holding VEHs, with each region being empty or holding a VEH. A region thus corresponds to a handler slot in Figure 2. The remaining FPGA area remains static during normal operation and holds the core and management logic (see Figure 6 for an example with 10 slots). The configuration data for the dynamically reconfigurable regions is generated by extending the standard Xilinx FPGA implementation tool flow with special development tools such as PlanAhead [10]. For each region, partial configuration data is generated for all VEHs that could be placed there.

The partial hardware configuration data files are stored outside of the FPGA (e.g., in an external memory) and are loaded into the device by using the on-chip reconfiguration interface (ICAP). To be able to reconfigure the VEH modules during runtime, we need to ensure proper isolation between the VEH modules and the static core system to avoid accidental data transmission or packet loss (as the reconfiguration process is not an atomic operation on the physical level). This is accomplished by the use of buffers and hardware switches, which temporarily "break" the connections on the chip between the core and the VEH regions while a dynamic partial reconfiguration takes place.

On the functional level, the ability to dynamically replace VEH modules is supported by the following design principles of VEHs:

1. A VEH itself is both session- and state-less, and only active while it is processing a packet. It can thus be re-used to handle different connections (e.g., clients), instead of being assigned exclusively for a specific connection for its entire duration.

2. Session and state information for a connection (e.g., client transactions) is stored *outside* the VEHs in a global application state memory provided by the static core architecture. The actual session state data is piggy-backed on to every incoming packet when it is forwarded to the appropriate VEH.

3. Execution of a VEH is only started by incoming packets.

Property 1 is essential to allow frequent swapping: A slot is active only for a short amount of time (a packet is processed in a couple of microseconds), and no state has to be preserved when swapping VEHs. Property 2 ensures that the session state data is not affected by the swapping of VEHs. Finally, Property 3 allows the use of a VEH deployment strategy based on the characteristics of the currently observed data traffic (see next Section).

Table 1. Routing Rules Matching table with socket listen information

Rule ID	Protocol	Port	IP	Netmask	Rule Group ID
1	0x06	80	1.2.3.0	255.255.255.0	1
2	0x11	23	4.5.6.7	255.255.255.255	2
3	0x06	25	1.5.6.0	255.255.255.0	3
4	0x06	25	1.6.6.0	255.255.255.0	3
5	0x06	22	2.2.0.0	255.255.0.0	4
...	...	...	...	...	...

Table 2. Group Counter Table with packet counter and target slot routing information

Rule Group ID	VEH ID	Counter	Slot
1	3	241	1
2	4	12	0
3	14	178	2
4	12	24	0
...	...	...	...

6.1. Dynamic VEH Replacement Strategy

To support autonomous swapping of VEHs during the operation of the malware collection honeypot, we employ a self-adaptation algorithm to activate the VEHs that are required by the currently incoming data traffic. To this end, the packet forwarding rules table (see Figure 2) will now contain destination socket information for all VEHs known to the system (both the on-chip ones as well as the ones available in the external configuration store). Each entry of the routing rules matching table (see Table 1) is associated with a packet counter (see Table 2), which counts the number of packets matching the corresponding rule in a given time interval. The counters are reset on a regular basis (currently every 10s). In this manner, we measure the demand of current network traffic for all handlers (again, both on-chip and off-chip VEHs). The packet counters are held separately from the rules table to support the grouping of rules with a single counter (e.g., to have a single VEH listen to multiple IP addresses or ports). Counters and rules are associated by having identical values for their Rule Group ID (which corresponds to the row in the Group Counter Table).

As the deployment is based on the counter values, the group entry does also contain the VEH ID responsible for all the rules belonging to that group as well as the current target slot information if the VEH is currently active (swapped-in on-chip). Otherwise, the slot column in the group counter table contains zero. This information is used by the core to route the packet to the destination slot (for an on-chip VEH), or to make adaptation decisions (possibly swapping in a new VEH).

For our protoype, we use an adaptation algorithm that implements a least recently used strategy (see Listing 2). The group counter table is scanned periodically (currently every 1ms, thus allowing up to 1000 swap transactions per second) for the off-chip handler with the highest packet count, and the on-chip handler with the lowest packet. In a second step, it is checked if the high-count off-chip handler has a higher packet count than the low-count on-chip one. If yes, the high-count off-chip handler replaces the low-count on-chip handler (see Figure 6 at time steps t=1 and t=2). Using this algorithm, we always have the Handlers on-chip that have the highest activity index.

A drawback of this algorithm is, that attacks which send only very infrequent requests (so-called *low-and-slow* attacks) are not caught using our initial adaptation scheme, as the required handler will be swapped out before it has completely seen the attack, or never even swapped in. The first weakness could be alleviated somewhat by simply giving each handler a time-to-live (TTL) value that ensures it is present a minimum interval of time (to actually catch more of the low-and-slow attack). The second weakness could be addressed by having a higher level of decision-making based on longer-term statistics: It could ensure that handlers for rarely occuring traffic do get a bump up in priority once in a while to make sure that every kind of traffic is at least occasionally observed. These issues will be addressed in future work.

7. Experimental Results

Our prototype implements the NetStage Architecture on the BEE3 [11] reconfigurable computing platform, currently on a mid-size FPGA chip. Table 3 provides a brief overview of the technical characteristics. A 10 Gb/s point-to-point CX4 Ethernet link connects the BEE3 to a dedicated eight-core Xeon Linux server for traffic generation.

We have developed a number of VEHs for evaluating the system: The first one emulated a vulnerability [12] present

```
deployed_id , undeployed_max , undeployed_id , i = 0;
deployed_min = MAX;

/* search for replacement candidates */
while (i < total_counter_rows) {
  if (counter_value(i) < deployed_min) && (slot(i) >
      0) {
    deployed_min = counter_value(i);
    deployed_id = i;
  } elsif (counter_value(i) > undeployed_max) && (slot
      (i) = 0) {
    undeployed_max = counter_value(i);
    undeployed_id = i;
  }
  i++;
}

/* trigger partial reconfiguration */
if (undeployed_max > 0) then
  if (empty slots available) {
    deploy off-chip Handler(undeployed_id) to free
        slot;
  } elsif (undeployed_max > deployed_min) {
    swap on-chip Handler(deployed_id) with off-chip
        Handler(undeployed_id);
  }
}
```

Listing 2. Self-reconfiguration algorithm

in a number of SIP applications. Other VEHs emulate vulnerable MS SQL servers [13] and simple mail and web servers. Our initial concerns that the dedicated hardware approach would require too much FPGA areas were unfounded. As an example, the MSSQL VEH requires less than 1% of the FPGA area. Since the other VEHs do have similar sizes, even our current single chip solution can handle ca. 40 VEHs running in parallel (assuming a chip utilization of 80% and a average VEH size of 1.5%), as the NetStage core and the management interface require just 20% of the entire FPGA. Given that the release of much larger FPGA devices (e.g., Xilinx Virtex-7 series) is imminent, we are convinced that our approach will scale even to significantly more complex VEHs. In case that the space of a single FPGA device is not sufficient, the virtualization layer allows to extend this number to hundreds of available VEHs by the use of an external SDRAM as configuration store. In that case, the additional logic to support the adaptation process requires around 4% of the FPGA.

The latency of a packet processed by the system depends on the occupation of the data paths. In the best case scenario (all paths have empty buffers), the latency for an ICMP PING packet is around 0.8 μs. For a web server request this optimal latency is 2.4 μs. Even when the system is fully loaded (buffers filled to 90%), latency reaches just 17 μs.

To give an example of the performance of the system, we used the ApacheBench 2 testing tool to submit a million requests to the web server VEH. The hardware honeypot consistently replied in 22 μs (measured on the Linux server) with no packet loss, while a software Apache running on the Linux server required 100 μs.

Table 3. Hardware honeypot specifications

FPGA Type	Virtex 5 LX155T
Connectivity	10G CX4 Ethernet
Core Frequency	156.25 MHz
Data Path Width	128 Bit
Core Processing Speed	20 Gbit/s
Max. simultaneous VEHs	ca. 40
Max. swap operations / s	ca. 1000
Sample VEH Sizes	SIP VEH: 1.15%
(in % of Chip)	MSSQL VEH: 0.81%
	WEB VEH: 0.95%
	MAIL VEH: 0.78%

8. Conclusion and Future Work

Our proposed platform demonstrates the potential of hardware-accelerated networking operations, even for such complex scenarios as honeypot operation. It combines both raw performance (internally 20 Gb/s) as well as increased security: Since the system does not have a general-purpose processor executing software, it cannot be subverted in this fashion.

With our current work on VEDL, one of the major hurdles to actually employ the hardware honeypot in a production environment for capturing malware samples is significantly lowered. By using the partial reconfiguration capability of today's FPGAs, individual VEHs can be swapped in and out of the system without having to take it offline.

Our next research steps will improve the current state of the prototype and perform more complete evaluations. For the latter, we will attach the hardware honeypot to a true uplink for stress testing. Other work will deal with advancing the state of VEDL and the VEH compiler.

9. References

[1] "Nepenthes," Website, available online at nepenthes.carnivore.it (Access date: 10 Sept., 2010).

[2] "Honeyd," Website, available online at www.honeyd.org (Access date: 10 Sept., 2010).

[3] 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.

[4] "Cwsandbox," Website, available online at www.sunbeltsoftware.com (Access date: 10 Sept., 2010).

[5] 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.

[6] S. Mühlbach and A. Koch, “An FPGA-based Scalable Platform for High-Speed Malware Collection in Large IP Networks,” in *FPT ’10: Proceedings of the 2010 International Conference on Field Programmable Technology*. IEEE Computer Society, 2010.

[7] W. Eddy, “TCP SYN Flooding Attacks and Common Mitigations,” RFC 4987 (Informational), Internet Engineering Task Force, Aug. 2007.

[8] D. Dye, *WP374: Partial Reconfiguration of Virtex FPGAs in ISE 12*, Xilinx, Inc., 2010.

[9] S. Mühlbach and A. Koch, “NetStage/DPR: A Self-adaptable FPGA Platform for Application-Level Network Security,” in *ARC ’11: Proceedings of the 7th International Symposium on Applied Reconfigurable Computing*. Springer, LNCS, 2011.

[10] *UG632: Xilinx PlanAhead User Guide v. 12.3*, Xilinx, Inc., 2010.

[11] *BEE3 Hardware User Manual*, BEEcube Inc., 2008.

[12] M. Thumann, “Buffer Overflow in SIP Foundry’s SipXtapi,” Website, available online at http://www.securityfocus.com/archive/1/439617 (Access date: 15 Sept., 2010).

[13] 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.