NetStage/DPR: A Self-adaptable FPGA Platform for Application-Level Network Security

Sascha Mühlbach[1] and Andreas Koch[2]

[1] Secure Things Group, Center for Advanced Security Research Darmstadt (CASED), Germany
[2] Embedded Systems and Applications Group, Dept. of Computer Science, Technische Universität Darmstadt, Germany

Abstract. Increasing transmission speeds in high-performance networks pose significant challenges to protecting the systems and networking infrastructure. Reconfigurable devices have already been used with great success to implement lower-levels of appropriate security measures (e.g., deep-packet inspection). We present a reconfigurable processing architecture capable of handling even application-level tasks, and also able to autonomously adapt itself to varying traffic patterns using dynamic partial reconfiguration. As a first use-case, we examine the collection of Malware by emulating an entire honeynet of potentially hundreds of thousands of hosts using a single-chip implementation of the architecture.

1 Introduction

With the growing reliance of business, government, as well as private users on the Internet, the demand for high-speed data transfer has ballooned. On a technical level, this has been achieved by improved transmission technologies: 10 Gb/s Ethernet is already in widespread practical use at the ISP and data-center levels, standards for 40 Gb/s and 100 Gb/s speeds have already been formulated.

The data volume transferred at these speeds presents a significant challenge to current security measures, especially when going beyond simple firewalls and also considering payload inspection, or even application-level protocols. Conventional software-programmable processors are sorely pressed to keep up with these speeds. A recent evaluation of the popular network intrusion detection system Snort showed that such a software system can scan only up to 200 Mb/s without noticeable packet loss on a standard CPU [1].

As an alternative, both software-programmed dedicated network processing units (NPUs), as well as hardware accelerators for these operations have been proposed. The use of reconfigurable logic for the latter allows greater flexibility than hardwiring the functionality, while still allowing full-speed operation [4,11]. Our own research has demonstrated that modern FPGAs can also go beyond these traditional packet-processing tasks by also handling application-level protocols in hardware. As a first use-case of our architecture, we implemented a low-interaction honeypot, which emulates entire networks of hosts with vulnerable applications, entirely in hardware (the “MalCoBox”) [8].

In contrast to related work such as [10], which mainly interpreted RAM-based state transition tables at run-time, our approach fully exploits hardware features such as parallel FSMs and pattern matchers to actually keep-up with data rates of 10 Gb/s and beyond. We thus achieve not only high-performance, but also harden the honeypot itself against hijacking: No general-purpose processor exists that could be subverted to actually execute attack code.

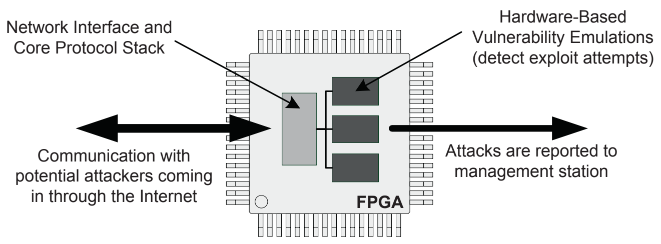

Network Interface and Core Protocol Stack
Hardware-Based Vulnerability Emulations (detect exploit attempts)
Communication with potential attackers coming in through the Internet
Attacks are reported to management station
FPGA

Fig. 1. Honeypot as use-case for FPGA-based network processing

We have since improved our initial architecture by adding networking capabilities such as a domain-specific highly optimized TCP/IP processing in hardware, restructured into a very modular architecture that allows the addition of new application-level services in hardware analogously to setting up software servers, and have added partial dynamic reconfiguration to hot-swap service modules (called *Handlers*) while keeping the rest of the system running [9]. In the honeypot scenario (see Figure 1) these Handlers are named Vulnerability Emulation Handlers (VEHs) to reflect their intended purpose.

Here, we present a refined architecture not only capable of dynamic partial reconfiguration (DPR), but able to autonomously adapt itself to react to varying traffic characteristics: The base architecture is described in Section 2, followed by details on the self-adaptation strategy in Section 3. The honey-pot application is a good use-case for our approach, since a large library of VEHs will not completely fit on the FPGA, and it should operate independently without user intervention. Implementation details and results for the honeypot are presented in Sections 4 and 5. Finally, we close with a conclusion and an outlook towards further research in the last Section.

2 Architecture

Figure 2 shows the architecture of our dynamically partially reconfigurable high-level network platform, called NetStage/DPR. The application-independent core (Fig. 2-a) can process IP, UDP and TCP protocols as well as ARP and ICMP messages. It has an easily extensible hierarchical design [8] that allows the quick addition of new protocols in dedicated modules at all layers of the networking model.

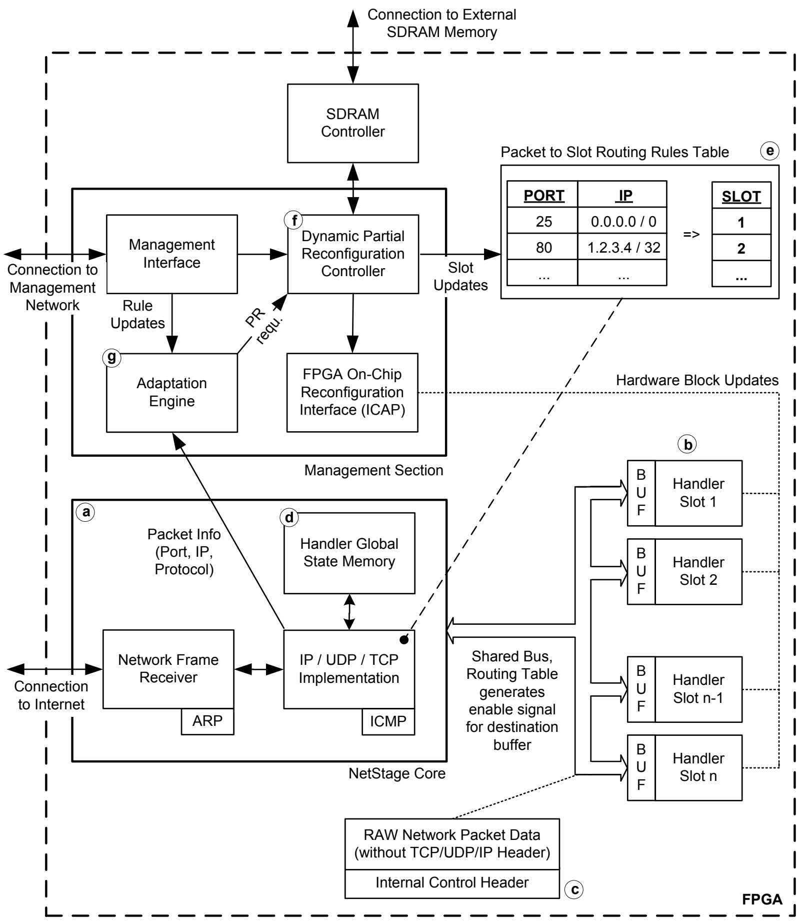

Connection to External SDRAM Memory
SDRAM Controller
Packet to Slot Routing Rules Table
e
PORT
IP
SLOT
25
0.0.0.0 / 0
1
80
1.2.3.4 / 32
2
=>
...
Connection to Management Network
Management Interface
f
Dynamic Partial Reconfiguration Controller
Slot Updates
Rule Updates
PR requ.
g
Adaptation Engine
FPGA On-Chip Reconfiguration Interface (ICAP)
Hardware Block Updates
Management Section
b
B U F
Handler Slot 1
Handler Slot 2
Handler Slot n-1
Handler Slot n
a
d
Packet Info (Port, IP, Protocol)
Handler Global State Memory
Network Frame Receiver
ARP
IP / UDP / TCP Implementation
ICMP
Connection to Internet
Shared Bus, Routing Table generates enable signal for destination buffer
NetStage Core
RAW Network Packet Data (without TCP/UDP/IP Header)
Internal Control Header
c
FPGA

Fig. 2. NetStage/DPR self-adaptable network platform

The Handlers (Fig. 2-b) are connected to the core by using two separate shared buses with a throughput of 20 Gb/s each, one for the transmit and one for the receive side. Buffers decouple the different processing stages and limit the impact of Handler-local stalls in the processing flow. The interface between the buffers and the actual handlers forms a natural boundary for using dynamic partial reconfiguration to swap the handlers in and out as required.

2.1 Handler Interface

All handlers share the same logical and physical interfaces to the core system. The physical interface consists of the connection to the ingress and egress buffer and logistical signals such as clock and reset. However, the handlers actually communicate with the rest of the system purely by sending and receiving messages (not necessarily corresponding to actual network packets). These messages

(Fig. 2-c) consist of an internal control header (containing, e.g., commands or state data) and (optionally) the payload of a network packet. In this fashion, the physical interface can remain identical across all handlers, which considerably simplifies DPR. For the same reason, handlers should also be stateless and use the Global State Memory service (Fig. 2-d), provided by the NetStage/DPR core instead (state data will then just become part of the messages). This approach avoids the need to explicitly save/restore state when handlers are reconfigured.

2.2 Packet Forwarding

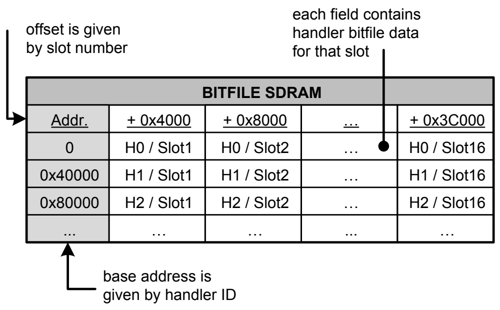
Incoming packets must be routed to the appropriate Handler. However, using DPR, the Handler may actually be configured onto different parts of the FPGA. Thus, we need a dynamic routing table (Fig. 2-e) that directs the message-encapsulated payloads to the appropriate service module. Our routing table has the usual structure of matching protocol, socket, and address/netmask data of an incoming packet to find the associated Handler. Note that a single Handler can thus receive data for an entire subnet. On the sending side, handlers deposit outgoing messages into their egress buffers, where they will be picked up by the core for forwarding (and possible transmission to the network). This is currently done using a simple round-robin approach, but more complex schemes (e.g., QoS-based) could, of course, be added as needed.

If packets are destined for a Handler with a full ingress buffer, they will be discarded. However, since all of our current handlers can operate at least at the line rate, this should not occur during regular operation. Packets for which a Handler is available off-line (not yet configured onto the device) will be counted before being discarded, eventually resulting in configuring the Handler onto the FPGA (bringing it on-line) if enough demand has been detected (see Section 3). Note that this strategy does not guarantee the reception of *all* packets (which is acceptable for the honeypot), but represents a good compromise between speed and complexity. If no appropriate Handler exists (either off-line or on-line on the device), packets will be discarded right away.

2.3 Handler Reconfiguration

Our system can perform the actual DPR operation autonomously of a host PC. A dedicated hardware unit (Fig. 2-f) is used as DPR Controller (DPRC) instead of an embedded soft-core processor, since the latter would not be able to achieve the high reconfiguration speeds we aim for (see Section 5.2 and [7]). Because of the storage requirements the Handler bitstreams are stored in an external SDRAM memory, and fed into the on-chip configuration access port (ICAP) by the DPRC using fast DMA transfers.

The DPRC is also responsible for selecting the specific bitstream to load: For simplicity, our initial implementation requires separate bitstreams for each Handler, corresponding to the physical location of the partially reconfigurable areas (which we call a *Slot*). To this end, the SDRAM is organized in clusters, which hold multiple versions of each Handler, addressed by the Handler ID and the target Slot number (see Figure 3). Again, for simplicity, we set the cluster size to the average size of each Handler's bitstream. In a more refined implementation, we could use a single bitstream for each Handler, which would then be relocated to the target Slot at run-time [2], and bitstream compression techniques [5] to further reduce its size.


offset is given by slot number
each field contains handler bitfile data for that slot
BITFILE SDRAM
base address is given by handler ID

Addr.	+ 0x4000	+ 0x8000	...	+ 0x3C000
0	H0 / Slot1	H0 / Slot2	...	H0 / Slot16
0x40000	H1 / Slot1	H1 / Slot2	...	H1 / Slot16
0x80000	H2 / Slot1	H2 / Slot2	...	H2 / Slot16
...	...	...	...	...

Fig. 3. Addressing of Handler bitstreams in SDRAM

Note that the actual decision when and which slot to reconfigure with which Handler is not made by the DPRC, but by another subsystem (see next Section).

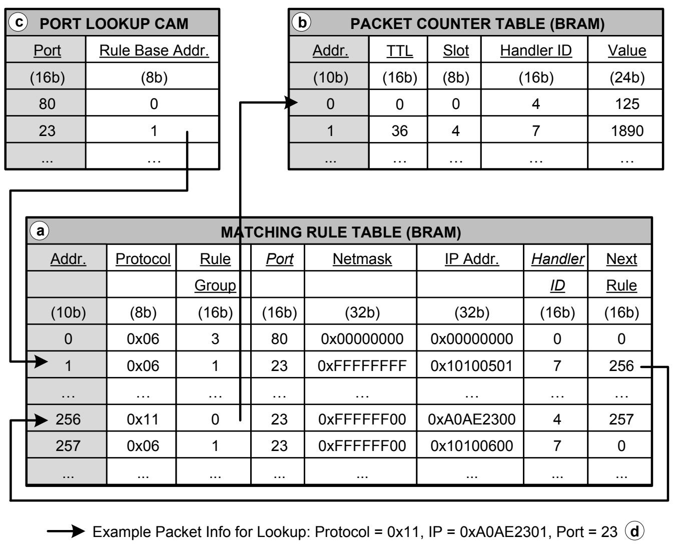
3 Self-adaptation Strategy

We use a rule-based adaptation strategy, implemented in the Adaptation Engine (Fig. 2-g), that interprets packet statistics. Specifically, we consider packets at the socket level (destination protocol, port, and IP address), received in a time interval (currently set to 1s). These statistics are only kept for packets for which a Handler is actually available (either on-line or off-line).

3.1 Rule Representation

Our architecture (see Figure 4) aims for fast rule lookups and statistics updates (few cycles) even for high packet rates (10 Gb/s, packet size < 100 B). It is characterized by the following underlying Assumptions: A) We will have 100...200 handlers, with 5...10 packet matching rules each, and B) the most distinctive criterion of a rule is the port number.

Since the Packet Matching Rules for the counter updates themselves are very similar in nature to the Packet Forwarding Rules (see Section 2.2), we combine both representations into a single Rule Table (Fig. 4-a) that has the Forwarding Rules (to on-line handlers) as a subset of the Matching Rules (encompassing both on-line and off-line handlers). In addition to the socket-level specifications, the Rule Table also holds the ID of the Handler responsible for the rule, and a field for linking rules that should be matched in order. Furthermore, multiple rules can also be aggregated into a Rule Group, which structures the packet counters in the Counter Table (Fig. 4-b): We keep packet counts per Rule Group (instead



c PORT LOOKUP CAM

Port	Rule Base Addr.
(16b)	(8b)
80	0
23	1
...	...

b PACKET COUNTER TABLE (BRAM)

Addr.	TTL	Slot	Handler ID	Value
(10b)	(16b)	(8b)	(16b)	(24b)
0	0	0	4	125
1	36	4	7	1890
...	...	...	...	...

a MATCHING RULE TABLE (BRAM)

Addr.	Protocol	Rule	*Port*	Netmask	IP Addr.	*Handler*	Next
		Group				*ID*	Rule
(10b)	(8b)	(16b)	(16b)	(32b)	(32b)	(16b)	(16b)
0	0x06	3	80	0x00000000	0x00000000	0	0
1	0x06	1	23	0xFFFFFFFF	0x10100501	7	256
...	...	...	...	...	...	...	...
256	0x11	0	23	0xFFFFFF00	0xA0AE2300	4	257
257	0x06	1	23	0xFFFFFF00	0x10100600	7	0
...	...	...	...	...	...	...	...

Example Packet Info for Lookup: Protocol = 0x11, IP = 0xA0AE2301, Port = 23 d

Fig. 4. Rule and Counter Tables, First-level CAM

of per individual rule), thus reducing storage needs for complex Rule Tables. Off-line, but available handlers are indicated by a Target Slot value of zero. For efficiency, we keep a copy of the Handler ID in both Rule and Counter Tables (see Section 3.4). To keep the architecture flexible, we set the size for the fields Rule Group and Next Rule to 16b, even as the current implementation uses a lower number of rules and counter entries.

3.2 Fast Hierarchical Rule Matching

To speed-up look-ups, we exploit our Assumption B to realize a hierarchical matching beginning with the destination port of a packet (see Figure 5): Using a CAM (Fig. 4-c), we can quickly determine the start of the rule chain for this port, which is then processed in order. The linked list allows us to match rules

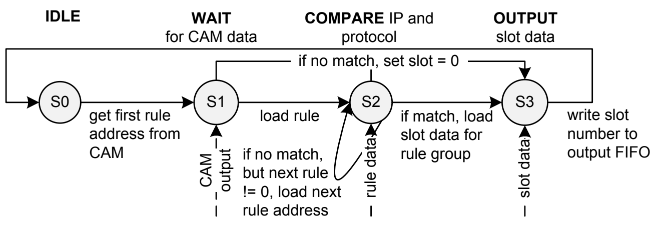

IDLE
WAIT for CAM data
COMPARE IP and protocol
OUTPUT slot data
if no match, set slot = 0
S0
S1
S2
S3
get first rule address from CAM
load rule
if match, load slot data for rule group
write slot number to output FIFO
if no match, but next rule != 0, load next rule address
CAM_output
rule data
slot data

Fig. 5. Look-up of destination slot for a packet

in an explicit priority ordering, normally considering more specific rules (having a longer netmask) before more general ones. Rules will be inserted into the table at the correct position depending on the length of their netmask. For efficiency, the chain starts are all located in the lower 256 entries of the Rule Table (to be directly addressable by an 8b CAM entry), succeeding elements of chains begin at address 256 of the Rule Table. As only the port numbers of existing rules are stored in the CAM, we assume that 256 entries suffice for the current implementation. The arcs in Figure 5 show the lookup path for the given example input tuple (Fig. 4-d).

3.3 Packet Counter

The Counter Table is indexed by the Rule Group that matched an incoming packet to increment its corresponding counter value. The counters are reset after a parametrized time interval (currently: 1 s) to begin the next sampling window.

3.4 Reconfiguration Candidate Selection

The Counter Table (Fig. 4-b) is scanned periodically (every 1 ms in the current implementation, this interval may not be shorter than the Handler reconfiguration time, see Section 5.2) to make adaptation decisions (configuring a new Handler, possibly replacing an existing Handler). To this end, we select as candidate for the *new* Handler the offline one with the highest packet count, as candidate to be *replaced* the online one with the lowest packet count. If the new candidate has a higher count than the replacement candidate, the replacement will occur. If the system still has free Slots available, the new candidate will just be configured in without displacing an existing Handler. This approach can miss some attack patterns (e.g., if the network data contains many *different high-volume* attacks), but is a very resource efficient algorithm, with acceptable trade-offs for the honeypot scenario (which needs to collect each malware just once).

The decision reached will be used to update the Counter Table (which also keeps track of the Handler-Slot associations) and to inform the DPRC (Section 2.3) to perform the actual reconfiguration operation. Having all information for the adaptation decision available within the Counter Table simplifies the process implementation and justifies the redundant storage of values in both Tables (Fig. 4-a) and (Fig. 4-b).

To avoid reconfiguration thrashing (immediately replacing handlers just brought on-line), an optional time-to-live (TTL) value can be associated with a Counter Table entry. For this number of rule evaluation scans, that Handler will be exempt from replacement. The TTL value will be decremented during each scan, once it reaches zero, that Handler can become a replacement candidate again.

4 Implementation Details

We have implemented a NetStage/DPR prototype on a Xilinx Virtex 5 FPGA on the BeeCube BEE3 platform. Xilinx XAUI and 10G MAC IP cores provide the network connectivity at 10 Gb/s line speed, while the NetStage core datapath can handle 20 Gb/s by having a 128b processing width. This allows the core to catch-up with momentary stalls by burst-processing buffered data at the higher rate. These parts of the architecture run at a clock frequency of 156.25 MHz.

The ICAP is fed at its specified limit of 400 MB/s (32b@100 MHz), with the configuration data read from BEE3's DDR2-SDRAM using a dedicated Xilinx MIG block able to supply at least the required 400 MB/s. The core and ICAP clock domains are decoupled using FIFOs.

For fast lookups, the Global State Memory (Section 2.1) is implemented as on-chip BlockRAM. For the honeypot use-case, only very few handlers will actually need to keep state. Even then, data will also need to be retained for only a few seconds. Thus, 64k x 32b = 256 kB of Global State Memory suffices for this application.

Management functions (e.g., updating the Matching Rule Table, loading the bitstream memory, retrieving malware) are performed through a separate network interface. For simplicity, we trust the management network and do not perform security checks on the incoming bitstreams. However, such functionality could be easily added [3].

4.1 Packet Forwarder and Adaptation Engine

Since they rely on the same data structures, the Packet Forwarder and the Adaptation Engine are realized in a common hardware module (Figure 6). It contains the logic for tracking statistics, interpreting rules, and managing Handler-Slot assignments. Dual-port BlockRAMs are used to realize the 1024-entry Rule and 512-entry Counter Tables. Thus, lookups to determine the Slot of the destination Handler for an incoming packet can be performed in parallel to the rule management and counter processes. For area efficiency, the CAM (see Section 3) is shared between the functions. But since the throughput of the system is directly affected by the Packet Forwarding performance, the corresponding slot-routing lookups will always have priority when accessing the CAM. Since the CAM is used only briefly for each process, it will not become a bottleneck.

The Packet Forwarder logic (Fig. 6-a) puts the destination Handler slot for an incoming packet in the output queue. The forwarding look-up is pipelined: by starting the process as soon as protocol, IP address and port number have been received, the looked-up destination slot will generally be available when it is actually required (once the packet has passed through the complete core protocol processing). Since packets will be neither reordered nor dropped before the Handler stage, simple queues suffice for buffering look-up results here.

Since not all incoming packets should be counted (e.g., TCP ACKs should be ignored), the Adaptation Engine uses a separate port (Fig. 6-c) to update the Counter Table only for specific packets.

The Rule Management subsystem (Fig. 6-b) accepts commands from the management network interface through a separate FIFO, and has an internal FIFO that keeps track of available row addresses in the Rule Table. In a similar fashion, a separate FIFO also keeps track of available Slots, so new handlers are preferentially configured into empty slots before replacing existing handlers.

Reconfiguration requests are passed to the DPRC using the DPR Request FIFO as Handler ID and target Slot pairs.

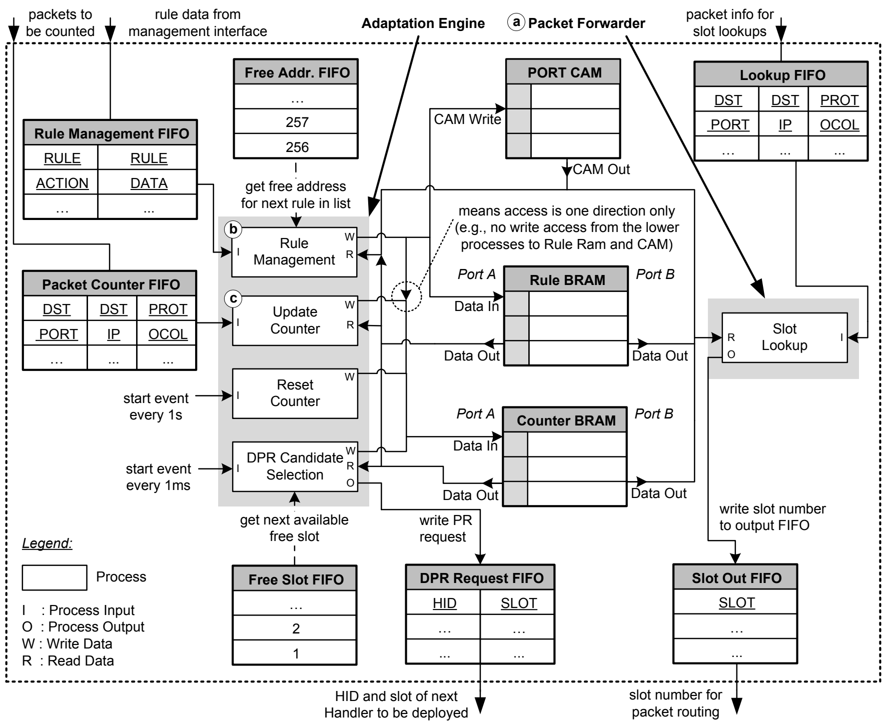


Fig. 6. Packet Forwarder and Adaptation Engine

5 Experimental Evaluation

For testing, we have connected the BEE3 with the MalCoBox application running on the NetStage/DPR architecture by a 10 Gb/s Ethernet point-to-point CX4 link to a dedicated eight-core Linux server for traffic generation.

5.1 Synthesis Results

Table 1 shows the results when implementing the design with Xilinx ISE 12.3. The complete system with management logic for 16 Handler Slots requires roughly one third of the FPGA capacity, the rest is available for the Slots themselves. The critical resource are the BlockRAMs, of which five are required per Slot for ingress and egress buffers. The Packet Forwarding and Adaptation Engine (PF&AE) block is relatively small compared to the overall design.

Table 1. Synthesis results for components and emulation modules

Module	**LUT**	**Reg. Bits**	**BRAM**
NetStage Core System incl. TCP / UDP	9,064	5,804	68
Interfaces for 16 Slots w/o Slot Contents	8,656	3,968	80
Management Section w/o PF&AE	890	1,271	9
Packet Forwarder & Adaptation Engine	3,343	1,647	14
NetStage/DPR Total	**21,953**	**12,690**	**171**
Mapped incl. MIG and MAC/XAUI	**33,208**	**18,788**	**192**
in % of LX155T resources	34	19	90
SIP Emulation Handler [9]	1,082	358	0
MSSQL Emulation Handler [9]	875	562	0
Web Server Emulation Handler [9]	1,026	586	0
Mail Server Emulation Handler [9]	741	362	0

For our tests, we chose four Handlers, each emphasizing a different implementation aspect: vulnerability emulations of a SIP application (focus on pattern matching) [12] and the MSSQL server (focus on byte matching and response generation) [6], as well as emulations of a simple Web Server (focus on large response packets) and Mail Server (focus on multiple protocol steps). The Handler implementations have been taken from [9]. Their characteristics are presented in the second part of the table.

5.2 Dynamic Partial Reconfiguration Results

The 16 Handler Slots have been distributed across the FPGA using PlanAhead 12.3 (see Figure 7). The FPGA regions for each Slot have been sized to 1920 LUTs (just twice as the average module size). All slots have equal area, as the results show that module sizes are relatively close. This simplifies the adaptation process, since otherwise we would need to perform multiple scans when selecting on-line/off-line candidates (one for each different Slot size class).

Table 2 gives the dynamic partial reconfiguration times and the resulting number of possible reconfigurations per second for the ICAP frequency of 100 MHz we use. We show the times not only for the 1920 LUT Slots we have used for the MalCoBox (indicated by *), but also for both smaller and larger choices (the best size is application-dependent). In general, LUTs are not scarce when realizing larger Slots, but the limited number of available BlockRAMs can constrain a design to less than 16 Slots if a Slot requires dedicated BlockRAMs.

Considering the complete adaptation operation, the time required is dominated by the actual reconfiguration time, as ICAP throughput is the limiting factor. All other processes are significantly faster. For example, the process to scan over all 512 Counter Table entries to find the next candidates (see Section 3.4) requires only about 3μs at 156.25 MHz clock speed, a negligible time relative to the reconfiguration time.

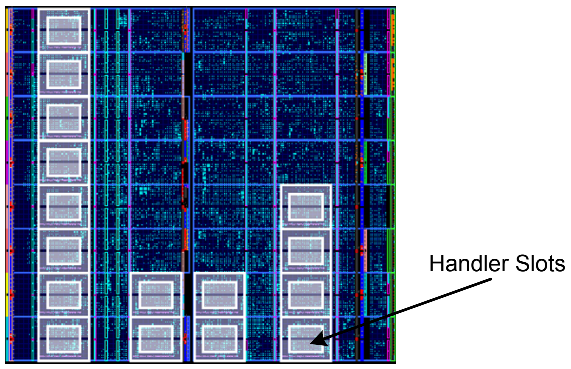


Fig. 7. FPGA Layout and Distribution of Handler Slots

Table 2. Reconfiguration time for different slot sizes

	LUT/ BRAM	**Bitfile Size**	**% of LUTs LX155T**	**Reconfig. Time**	**# Reconfigs/ second**
	1064/0	41 KB	1.09%	106μs	9438
(*)	1920/0	70 KB	1.97%	180μs	5563
	1920/4	96 KB	1.97%	245μs	4081
	4144/0	162 KB	4.26%	416μs	2401

However, as the remaining slots can continue to process packets while one slot is being reconfigured, the DPR performance is more than sufficient for the honeypot scenario. Here, we assume a traffic distribution with a high volume of similar requests trying to exploit well known security flaws (initiated by worms, etc.). Only a small number of packets distributed over time will be attacking (possibly older) vulnerabilities that are not that common, and that will actually need to be reconfigured.

6 Conclusion and Future Work

NetStage/DPR has demonstrated that it is not only feasible to perform application-level (instead of just the packet-level) network processing in FPGAs, but also actively exploit the dynamic partial reconfiguration capabilities of modern devices to build self-adapting architectures.

The base architecture of separate processing stages that allows the flexible integration of new functionality based on a hardware-based network protocol stack is suited to a wide number of domains. Our use-case of malware collection using the MalCoBox application is just one example scenario.

In the next research steps, we will improve the current state of the NetStage/DPR platform by partitioning the system to support multiple FPGAs to further increase the number of Handler slots, and by experiments with dynamic bitstream relocation. Furthermore, the MalCoBox application will be evaluated in a production environment using true real-time Internet traffic to refine the NetStage/DPR adaptation strategy.

Acknowledgments. This work was supported by CASED and Xilinx, Inc.

References

1. Alserhani, F., Akhlaq, M., Awan, I.U., Mellor, J., Cullen, A.J., Mirchandani, P.: Evaluating Intrusion Detection Systems in High Speed Networks. In: Proc. of the 5th. Intl. Conf. on Information Assurance and Security, vol. 02, pp. 454–459 (2009)
2. Flynn, A., Gordon-Ross, A., George, A.D.: Bitstream relocation with local clock domains for partially reconfigurable FPGAs. In: Proc. of the Conference on Design, Automation and Test in Europe, pp. 300–303 (2009)
3. Hori, Y., Satoh, A., Sakane, H., Toda, K.: Bitstream Encryption and Authentication Using AES-GCM in Dynamically Reconfigurable Systems. In: Proc. of the 3rd Intl. Workshop on Security, pp. 261–278 (2008)
4. Katashita, T., Yamaguchi, Y., Maeda, A., Toda, K.: FPGA-Based Intrusion Detection System for 10 Gigabit Ethernet. IEICE - Trans. Inf. Syst. E90-D, 1923–1931 (2007)
5. Koch, D., Beckhoff, C., Teich, J.: Bitstream Decompression for High Speed FPGA Configuration from Slow Memories. In: Proc. of the Intl. Conference on Field-Programmable Technology (2007)
6. Litchfield, D.: Microsoft SQL Server 2000 Unauthenticated System Compromise (2000), `http://marc.info/?l=bugtraq&m=102760196931518&w=2`
7. Liu, M., Kuehn, W., Lu, Z., Jantsch, A.: Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration. In: Proc. of the Intl. Conference on Field Programmable Logic and Applications (2009)
8. Mühlbach, S., Brunner, M., Roblee, C., Koch, A.: Malcobox: Designing a 10 gb/s malware collection honeypot using reconfigurable technology. In: Proc. of the 20th Intl. Conf. on Field Programmable Logic and Applications, pp. 592–595 (2010)
9. Mühlbach, S., Koch, A.: A dynamically reconfigured network platform for high-speed malware collection. In: Proc. of the Intl. Conf. on ReConFigurable Computing and FPGAs (2010)
10. Pejovic, V., Kovacevic, I., Bojanic, S., Leita, C., Popovic, J., Nieto-Taladriz, O.: Migrating a Honeypot to Hardware. In: Proc. of the Intl. Conf. on Emerging Security Information, Systems, and Technologies, pp. 151–156 (2007)
11. Singaraju, J., Chandy, J.A.: FPGA based string matching for network processing applications. Microprocessors and Microsystems 32(4), 210–222 (2008)
12. Thumann, M.: Buffer Overflow in SIP Foundry's SipXtapi (2006), `http://www.securityfocus.com/archive/1/439617`