A Reconfigurable Platform and Programming Tools for High-Level Network Applications demonstrated as a Hardware Honeypot

Sascha Mühlbach, Andreas Koch

***Abstract*—The security of computer systems and networks is severely threatened today by the combination of novel attack patterns and high traffic volumes. Together, this often exceeds the capabilities of purely software-based network security systems. As an alternative, hardware acceleration has been employed, e.g., for performing deep-packet inspection and pattern matching as well as general packet-header processing. While such implementations, capable of handling lower protocol layers, have been extensively studied in research and industry, their extension to higher communication layers has only rarely been addressed. Such capabilities, including the application level (OSI Layer 7), are the focus of this work. We present the NetStage platform, employing reconfigurable computing for high-throughput low-latency network processing, as well as associated development tools that allow networking domain experts to easily customize the system. As a use-case, we consider the realization of high-performance attack-resilient honeypots based on NetStage. To this end, we introduce the Malacoda language, its programming tools, and the generated target microarchitecture. We then evaluate the performance of Malacoda-generated vulnerability emulation handlers running on the NetStage platform.**

***Index Terms*—Network Security, 10G, FPGA, Network Stack, Deep Packet Inspection**

I. Introduction

THE relevance of the Internet for both business and private use has grown dramatically in the past decades. Social networks, video calls, instant messaging, and many more services have fundamentally changed the way how people communicate. Online banking, online shopping and eGovernment solutions simplify many day-to-day tasks. However, on the flip side of these advances, the large volume of accompanying financial activities attracts many types of criminals. Due to the global distribution of the Internet, it also presents a global attack surface on its services.

Attackers exploit weaknesses (bugs, design errors, etc.) of computer systems to break into remote systems and leverage this control to steal sensitive data such as passwords or credit card information. As shown by numerous studies [1], [2], the risk is omnipresent. A common tool for starting attacks is so-called "Malware", programs that automatically exploit bugs of computer systems for injecting and executing malicious code provided by the attacker. Pandalabs estimates that in 2012 32% of the worldwide PCs are infected with some malware [2].

Setting up proper security mechanisms has therefore never been more important than today. Firewalls and Intrusion Detection Systems (IDS) are two technologies that are employed on the network layer to secure the communication infrastructure from illegal access to systems and applications. However, the data volume transferred on today's high-speed networks (10 to 40 Gbit/s already in common use at the datacenter/carrier levels [3], with 100 to 400 Gbit/s on the upswing) presents a significant challenge to current security measures. Conventional software-programmable processors are not able to keep up with these speeds. An evaluation [4] of the popular network intrusion detection system Snort [5] showed that such a software system cannot handle high-speed traffic without noticeable packet loss on a regular server system.

To support such network security mechanisms on the infrastructure level, dedicated hardware accelerators have been proposed to offload compute-intensive tasks from the processor. As a key technology, Field Programmable Gate Arrays (FPGAs) are of particular interest to this end. FPGAs are integrated circuits whose functionality is not fixed during the manufacturing process, but instead can be flexibly reconfigured for specific applications afterwards.

Hardware accelerators based on FPGAs have often been employed for computation [6] or lower-layer acceleration tasks on the packet level [7], [8]. However, advances in chip design and fabrication technology have led to very powerful reconfigurable devices that have become capable of performing more complex operations. The use of FPGAs to accelerate such higher-level applications will be discussed in this work. Specifically, we employ the term *high-level* to refer to any system that *actively* takes part in a communication session as an endpoint, instead of simply monitoring traffic flowing by (as most current IDSs do).

The NetStage platform presented here allows the rapid deployment of hardware-accelerated attack-resilient interactive communication applications. The specific attack-resilience we aim for is against malware injected into the host running the networking applications. This resilience is achieved by the complete omission of all software-programmable processors from the data and control planes.

S. Mühlbach is with the Center for Advanced Security Research Darmstadt (CASED), Darmstadt, Germany.
E-mail: sascha.muehlbach@cased.de

A. Koch is with Technische Universität Darmstadt, Embedded Systems and Applications Group, Darmstadt, Germany. E-Mail: andreas.koch@esa.informatik.tu-darmstadt.de

As NetStage does not contain any such processor (even for operating system-functions), it cannot be subverted by an attacker into executing malicious code. The FPGA configuration itself remains secure, as the actual configuration port (e.g., ICAP on Xilinx devices) is completely isolated from network traffic. To the best of the author's knowledge, no possibility of remote attack exists for altering the FPGA configuration without access to the port.

Note, however, that NetStage itself does not prevent weaknesses in the system (e.g., in protocol handler state machines) that could lead to it being exploited for attacks against other hosts (e.g., by packet multiplication). Evaluating such attack scenarios could be an interesting topic for future research but lies out of the scope of this work.

As a proof-of-concept for a high-throughput but security-critical application, the hardware honeypot MalCoBox has been developed for the NetStage platform. A honeypot is a network security device that emulates thousands of vulnerable servers and is placed at an exposed position in the network in order to attract attackers. One goal of such a honeypot is to gain knowledge about attack patterns (e.g., collect the malware injected through the emulated vulnerabilities). The high volume of requests that could reach such a honeypot when connected to a high-throughput link, together with the increased risk of such an exposed system to become compromised itself, turns a honeypot into a promising candidate for a hardware implementation.

Despite the advantages of hardware-accelerated solutions, the high programming complexity of dedicated hardware systems is an ongoing issue. This is especially true for systems such as the MalCoBox, where the programmers will be domain experts in network security, but not proficient in the digital logic design and computer architecture fields commonly required to program FPGA-based computers. On the other hand, the dynamic threat landscape on the Internet requires frequent vulnerability emulation updates to keep up with attackers.

To resolve this quandary, this work discusses the domain-specific Malacoda language for abstractly formulating honeypot behavior. Malacoda is translated by its associated compiler from a high-level description of vulnerability emulations into high-performance hardware handlers executing on the NetStage platform. Together, NetStage and Malacoda address some of the productivity deficiencies often remarked to be major hindrances for the more widespread use of reconfigurable computing in communications applications [9].

We present our findings as follows: Section II reviews related work in the areas of network processing using FPGAs, hardware compilers, and honeypot systems. Section III gives an overview of the general architecture of the NetStage platform, the actual communication core, as well as various supporting services. For the conciseness of this text, many details are discussed only in more specialized prior publications, which will be referred to frequently. Section IV covers the new language Malacoda and the corresponding compiler architecture for NetStage. Section V gives synthesis and performance characteristics of the hardware implementation. Section VI finally concludes with a summary and a perspective towards future work.

II. Related Work

This section discusses prior and related work in the three topic areas of this article: Hardware support for network security, high-level hardware compilation, and network honeypots.

A. Hardware Support for Network Security

The use of FPGAs to accelerate network security applications has been a popular field of study for many years now. A recent survey by Chen et. al. [10] provides a good overview of the employment of FPGAs for packet classification, pattern matching, TCP stream processing, and Internet worm and DDoS attack detection and containment (as one example for anomaly detection). However, these works cover packet operations mainly on a lower network operation level[1], relying just on traffic monitoring without interaction.

1) Network Communication Support: A key requirement for the implementation of higher-level network communications on FPGAs is an efficient implementation of the basic Internet protocol stack [11], especially the transmission control protocol (TCP) [12]. Unfortunately, this has only rarely been addressed in the research community.

Schuehler and Lockwood [13], [14] began focusing on FPGA-based TCP processing for gigabit line rates in 2002. But their intention was to monitor TCP streams in switches and routers, instead of enabling endpoint communication. The same holds for later proposals [15], [16]. Dollas et. al. [17] were among the first researchers aiming at providing an entire communication stack, but their solution (achieving around 350 Mbit/s) does not reach the performance of high-speed networks and the project was discontinued. As an intermediate solution, TCP offload engines have been proposed that support CPUs for compute-intensive protocol processing (e.g., checksum calculation) for high-speed operation, but leave complex parts of the protocol (e.g., flow control) up to the CPU [18].

However, in the past two years, industrial development activity has ramped up. With the release of more powerful FPGAs and corresponding hardware platforms, and supposedly driven by the exploding interest in FPGA-based high-frequency trading (HFT) applications [19], a number of companies presented or announced [20], [21], [22] complete TCP/IP stacks for the use in FPGAs at data rates of 10 Gbit/s or more. Most of these cores support configuration parameters (e.g., number of supported connections) to allow the designer to adapt the core to the particular needs [22]. However, a deeper study in research continues to be hindered by these proprietary cores only being available under commercial licenses, with strict non-disclosure rules.

[1]For purposes of this discussion, we consider passive monitoring to be lower level, and active communications higher level operations.

2) FPGA-based Development Platforms: In addition to core components such as pattern matching and communication support, hardware applications require further infrastructure beyond the actual FPGA chip(s) to actually realize entire systems. In the area of FPGA-based networking, the most popular research platform is NetFPGA [23]. The NetFPGA base board consists of an FPGA connected to multiple network interfaces and external memory, with a 10G-capable version becoming available in 2011. An attractive aspect of NetFPGA is that, in addition to stable and affordable hardware, a rich set of open source software tools is available.

A commercial platform with a focus similar to NetFPGA is NetCope [24] from INVEA-TECH. NetCope also consists of an FPGA-based IP core and supporting software components. However, NetCope primarily targets commercial research and production use, e.g., for telecommunication providers, and was initially implemented only on the INVEA-TECH Combo Board hardware platform [25]. With the release of the NetFPGA-10G card, INVEA-TECH presented a port of NetCope to the NetFPGA-10G, available under special academic licensing [26]. The combination of the industrial-strength NetCope core with the open NetFPGA-10G research platform could also be suitable for further academic research.

3) High-Level Hardware-Based Applications: The implementation of complete communication applications entirely using reconfigurable logic has only rarely been addressed (in comparison to, e.g., regular expression matching). After initial proposals, e.g., by Fallside [27], only limited research effort has been directed at the topic. Most attempts perform only compute-intensive tasks on the FPGA and employ embedded general-purpose processors (GPP) for complex protocol processing (e.g., [28], [29]. While this is a practical approach, it does not reach the performance and security levels achievable using completely specialized hardware architectures.

A recent work from Lockwood et al. [30] considers the area of high-frequency trading applications. They describe a platform allowing financial experts to build hardware-based automatic trading applications. For that use-case, the main benefit of the hardware lies in the low latency that could be achieved by implementing the network communication stack and the application logic directly on the FPGA. For the application layer, Lockwood et al. follow an approach similar to the one we have used for Malacoda. They also advocate the use of a domain-specific language, both for being more accessible to application experts, as well as for better quality-of-results compared to compiling a general-purpose language.

B. High-Level Hardware Compilation

Two major approaches for high-level hardware compilation have been followed: Compiling from an established high-level general-purpose programming language (e.g., C), or accepting special domain-specific languages (e.g., [31]) as source. While general programming languages have the advantage that they can express a wide range of algorithms, domain specific languages score with their focus on concisely describing solutions in a particular problem area, where they achieve a higher programming and compilation efficiency.

1) General Purpose Languages: In the embedded systems domain, the C language and its derivatives (e.g., SystemC) are still dominant. The compilation of these languages to hardware is the subject of intense research in projects such as C-to-Verilog [32], Comrade [33], or Nymble [34]. In recent years, the technology has matured sufficiently to be usable in commercial systems. Vivado HLS [35] by Xilinx is a generic C to HDL (VHDL, Verilog) compiler employing many of the techniques first proposed in academic research. CatapultC [36] and Synphony C [37] are comparable, but according to an evaluation by Meeus [38], more advanced knowledge in digital system design is required to achieve a similar quality of results.

Coming from the hardware design perspective, but with a significantly raised level of abstraction (e.g., a powerful type system, the abstract description of parallelism and concurrency, etc.), the Haskell-based BlueSpec System Verilog (BSV) [39] is another language aiming to improve the productivity of hardware design. However, since BSV originated in digital logic design, it is easier for the BSV compiler to generate circuits that are competitive with manually optimized implementations.

2) Domain-Specific Languages: The language G [40] has been designed specifically for packet *header* processing on FPGAs. It supports a flexible specification of the packet format (fields and positions) and of conditional rules for modifying these fields depending on packet contents. G has some limited capability for *payload* processing, but lacks advanced facilities such as direct regular expression (RE) handling.

PacketC [41] is a language developed by Cloudshield [42] to be used on their heterogeneous multiprocessor machines (containing FPGAs, processors, and TCAMs) for high-speed network packet processing. Their programming model uses coarse-grain, SPMD (single program, multiple data) parallelism. On the hardware side, FPGAs (e.g., for ingress filtering and header evaluation) and special microcode control the packet pipeline. A single instance of a packet program is then executed on multiple network processing cores. PacketC is based on C for the basic syntax (operators, conditional statements etc.), but omits operations requiring complex hardware implementations (e.g., pointers, address operators, and dynamic memory allocation). For better packet processing support, domain-specific data types and operators have been added.

In contrast to hardware compilers generating complete architectures from scratch, Chimpp [43] follows a more general approach. It relies on an XML description for the composition of arbitrary pre-implemented packet-handling hardware blocks. These blocks can be of varying granularity (e.g., ARP lookup or simple TTL decrement), but they must be manually provided as synthesizable HDL descriptions. Chimpp itself provides only the interfacing and composition

capabilities.

Gorilla [44] is another C-style language for describing data parallel applications (such as network processors) for compilation to an FPGA on the functional level. Programmers can use domain-specific functions, which are mapped to a library of dedicated hardware accelerators. These are provided to the tool as parameterized templates that have been written and optimized by hardware experts for high performance operation. In this manner, Gorilla follows a similar concept as Chimpp, but offers more fine-grained control of the application (due to the more flexible parametrization). The Gorilla compiler accepts the application description together with the template library and generates synthesizable Verilog code.

C. Honeypots

While a wide spectrum of honeypot approaches exists, the focus of this work are so-called *low-interaction* honeypots [45]. These emulate selected parts of an operating system (e.g., the network stack) and / or selected parts of a vulnerable application. These honeypots are well suited for unattended operation and assisting a network intrusion detection system (NIDS) by providing information about current worm activity, spam attempts, or collected malware (using malware collection honeypots). Popular open source honeypot systems are HoneyD [46], Nepenthes [47] and Dioneae [48]. HoneyD offers a generic framework for the development of honeypot scripts, while Nepenthes and Dioneae primarily focus on collecting malware using emulated application vulnerabilities.

To the best of the authors' knowledge, there exists only one prior published work exploring the concept of executing honeypot functionality on bare hardware. The work of Pejovic et al. [49] describes a hardware honeypot implementation based on interpreting state machine transition/output tables stored in memory. These FSMs represent conditions and actions orchestrating the client-server interaction [50]. The research group implemented a hardware prototype using a Virtex-4 FPGA, but unfortunately did not publish any performance benchmarks. While the table-driven FSM approach is easier to program than the fully specialized hardware architecture we propose, it could become bottlenecked by limited memory bandwidth once network speeds exceeding 10 Gbit/s are considered.

III. NetStage Architecture

The realization of active high-level network security applications entirely on dedicated hardware requires a base architecture capable of autonomous Internet communication without CPU intervention. A major component of such a platform is a complete, yet efficient implementation of the basic Internet protocol stack [11]. Since at the beginning of this work, no suitable solution was available off-the-shelf (see also Section II-A1), we developed our own implementation, called NetStage (described in greater detail in [51], [52], [53]).

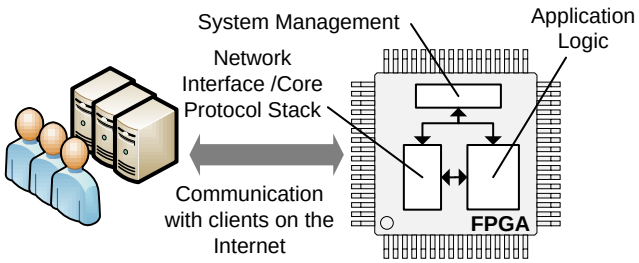


Fig. 1. NetStage operating scenario

NetStage is a generic high-speed network processing platform implementing the endpoint-oriented communications scheme shown in Figure 1. It supports UDP and TCP processing at 10 Gbit/s line rates on FPGAs using a protocol implementation customized for hardware mapping [51]. The NetStage Core, which provides the actual TCP/IP stack in hardware, can be used to support arbitrary network operations on the FPGA. In addition to the low-level network interface and protocol processing, NetStage also offers utility facilities (such as integrated per-connection state storage) as well as extensive simulation capabilities for the development and debugging of new hardware-accelerated network services.

For FPGA-acceleration, the application logic is integrated into NetStage as so-called Handlers. These act as network endpoints / servers and are mapped to independent hardware blocks. The blocks support the transparently virtualized execution of the actual per-connection processing, using time-multiplexing for different connections to the same application. Handlers communicate with the NetStage Core via a unified message-based interface that is easily extended with new capabilities.

This section gives an overview of the hardware architecture, Section IV will describe some of the programming tools for the platform. More details on the architecture and a number of actual hardware prototypes have been published in [52], [53], [54], with the hardware-optimized lightweight TCP/IP implementation covered in greater detail in [51].

A. Packet Processing Architecture

Figure 2 sketches the design of the NetStage platform architecture. Two dedicated network interfaces (Fig. 2-a) support the physical separation of public Internet and internal management traffic.

Public network packets that arrive at the system are processed by the Core 2-b) and forwarded to the corresponding application Handler 2-e). Routing the packets to the right application is the task of the Routing Layer 2-d), based on an internal Routing Table 2-f) that holds the destination information for each application. When NetStage is operated as network communication endpoint, the routing rules correspond to network sockets identified by the combination of destination IP address, protocol, and port. The Routing Table can be dynamically controlled via the management interface.

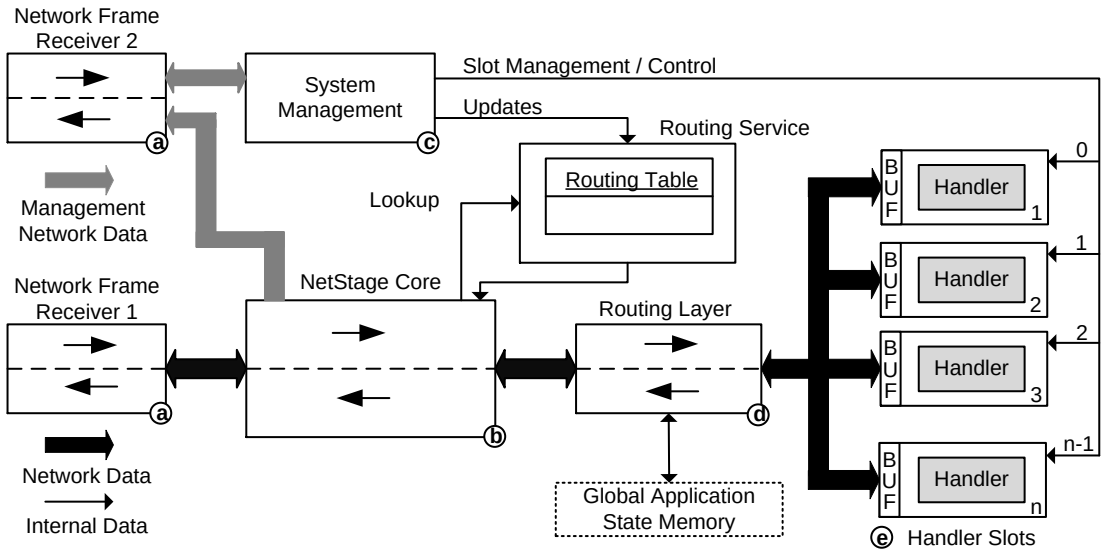


Fig. 2. NetStage platform architecture overview

For increased flexibility, the application Handlers are not attached directly to the Core, but are instead placed into so-called Slots. These provide buffered connectivity (to avoid stalls in the main datapath) between the routing layer and the Handlers. The buffers are implemented as special ring buffers [52], in contrast to the simple FIFOs often found in flow-based networking platforms (e.g., [55]). Slots and the routing layer are connected by a shared bus for direct communication between the Core and any Handler. The interconnect is optimized for operating NetStage as communications endpoint, allowing a simplified optimization of the routing layer. Full duplex communication at native interface speeds is supported by separate data paths for the receive and transmit directions.

The internal data bus has a width of 128 bits. Typically, the Core is running at the same clock frequency of the network interface, which is 156.25 MHz for 10G operation. Using a single clock avoids clock domain crossings and simplifies FPGA placement and routing. The combination of bus width and clock rate leads to an internal data path throughput of 20 Gbit/s for each direction (incoming and outgoing), which is sufficient to satisfy a line rate of 10 Gbit/s while providing some headroom to recover for throughput variations (stalls) in the Core or Handlers.

As a configurable option, the NetStage Core can be provided with a unidirectional connection to the management interface. This would allow the sharing of the Core between Handlers and system management components for sending IP-based control messages over the private management interface (shaded light grey in Figure 2).

B. Message-based Internal Communication

For future scalability, NetStage already employs a message-based scheme over the bus-based internal interconnect. This will allow an easy transition to a network-on-chip in the future, but none of our current applications have required this extra flexibility yet. NetStage messages encapsulate packet payloads by prefixing them with an Internal Control Header (ICH, see Figure 3), similar to the approach proposed in [23]. The wrapped packets bundle NetStage-internal control data with the payload for efficient processing.

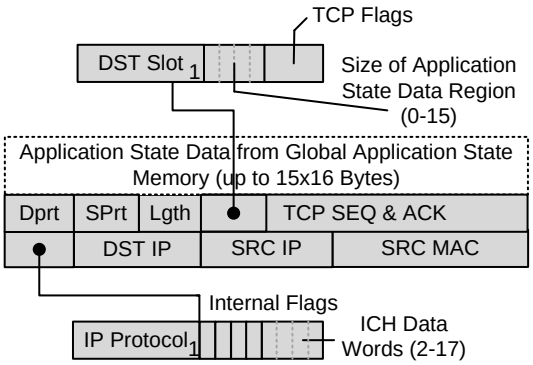


Fig. 3. NetStage Internal Control Header (ICH)

However, the NetStage ICH goes beyond the traditional models of routing and control information. In addition, it supports Layer 7 operations by also containing the *application-specific* state data. This allows to keep both the Core as well as many of the Handlers completely stateless, relying entirely on the ICH for quick reply packet generation. The ICH encapsulates both incoming and outgoing traffic.

C. NetStage Communication Core

The NetStage Core implements ARP, IP, ICMP, UDP, and TCP processing to perform autonomous Internet communication. In contrast to a software implementation (where optional functionality can easily be provided using libraries), an all-hardware architecture requires FPGA device

resources even for seldom-used functionality. This requires a careful trade-off between functionality and available hardware area, often leading to the omission of rarely-used functions from the hardware architecture. NetStage has been subjected to such a trade-off, enabling it to support autonomous low-latency high-bandwidth operations for a multitude of parallel connections (100,000+), while fitting onto off-the-shelf reconfigurable computing boards. Specifically, the following requirements have been imposed on the base architecture:

- The main data path should be able to handle network traffic at a line rate of 10+ Gbits/s.
- Overall multi-connection performance is more important than individual per-connection performance.
- Parallel support for multiple applications and network endpoints (sockets).
- Basic operation should possible even with limited external resources (e.g., using only on-chip memory)
- To maintain security, only dedicated hardware modules will come in contact with network traffic.
- All communication is initiated by the external clients, NetStage only acts as a responder.

The Core has a modular architecture, reflecting the structure of the protocol stack, which allows easy extension on all protocol layers. The hardware implementations of the different layers, called Stages, are interconnected via on-chip memory buffers and operate in a streaming fashion, similar to techniques described in prior work such as [23].

Handling many parallel TCP connections normally requires high memory bandwidth on the server for keeping track of the per-connection state. NetStage implements a stateless TCP variant in hardware [51], avoiding the need to maintain local counters on the server to track the connection state. The approach exploits, that TCP transmits the current acknowledgment number not only with acknowledgment packets, but also with each data packet. Assuming that there are no outstanding packets that have already been sent by the server, the server-side SEQ number can thus be reconstructed entirely from the incoming data without the need for local storage in memory. In normal operation, both for connection-establishment (using hardware-based SYN cookies), as well as for steady-state request-response traffic, NetStage adheres to the TCP specifications and has proven to be inter-operable with a wide spectrum of clients (see Section V).

The absence of state keeps NetStage from detecting and reordering packets arriving out-of-order. This is worked-around by the Core always offering a window size equal to the maximum segment size on the link, allowing at most one packet to be in transit for larger transfers. We accept the corresponding reduction in per-connection throughput to achieve higher overall throughput for a large number of parallel connections, the scenario NetStage is optimized for.

NetStage could be extended to full TCP functionality, but this would require the presence of sufficient amounts of low-latency off-chip memory (e.g., RLDRAM, or even better QDRII+ SSRAM) on the target hardware platform. But since not all of our supported target platforms have such off-chip memory (e.g., the Beecube BEE3 has only DDR2 SDRAM), it is just optional for our current implementation (allowing better performance if fast external memory is present, though).

D. Application-Specific Handlers

The application-specific service Handlers are responsible for the actual Layer 7 processing of network data. Figure 4 shows the architecture of an example Handler in the MalCoBox honeypot application (also called Vulnerabilty Emulation Handler, VEH). Such Handlers react to incoming packets and generate response packets according to predefined rules.

They comprise the actual protocol state machine, an (optional) RE matching engine, and an (optional) set of response packets, described as stored templates. These three components are customized for each application. Logically, it appears that each connection is processed by its own Handler. Physically, multiple connections are time-multiplexed onto the same Handler at line-rate, transparently virtualizing the underlying FPGA hardware and giving the appearance of multi-threaded execution.

For maximum threading throughput, Handlers should never block. However, if necessary, variable-latency handlers are supported, but will lead to packets queuing up in the inter-stage buffers until a non-blocked Handler for the required protocol becomes available. To ameliorate this situation, our current NetStage prototypes have an internal throughput of double the external line rate, allowing Handlers to catch-up with backlogged packets.

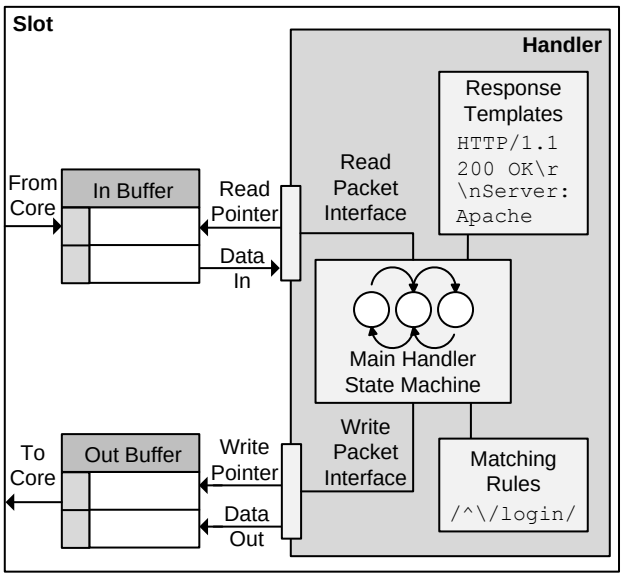


Fig. 4. Example application handler structure

E. Supporting Services

In addition to the packet pipeline in the NetStage Core, the system also provides a number of support services.

1) Global Application State Memory: FPGA-based custom computer architectures allow the matching of memory systems to the needs of the application (e.g., [56]). The NetStage memory system is highly specialized for the storage of per-connection state for a multitude of parallel connections. The Global Applications State Memory (GASM), introduced in [51], stores this application state in a central location and unburdens individual Handlers from having to implement their own memory interfaces. Additionally, by allowing the Handler internals to be stateless, the overhead of saving/restoring Handler state when exploiting dynamic partial reconfiguration of Handlers [53] is completely avoided.

Initially, the GASM was realized in on-chip BRAM blocks [51] and limited to storing just 16 bytes of data per connection. Current versions are also able to use low-latency off-chip memory (such as that available on the NetFPGA 10G card) to quickly access up to 15 words of 16 bytes each per connection.

The per-connection data is retrieved from the GASM when a packet arrives, and accompanies it through the receive pipeline, bundled into the ICH attached to the packet. State changes performed in the Handler are committed to the GASM when a reply (or an explicit empty state change packet) leaves the Handler by way of the transmit pipeline. When the ICH-wrapped packet passes the GASM, the altered state contained in the ICH is written to memory.

2) Mirror Port Functionality: For debugging and monitoring, NetStage can use one of its 10G interfaces as a mirroring port for the public network interface, independently of the mirror capabilities of an uplink switch. The mirroring (if enabled in the current NetStage configuration) is performed using dedicated hardware and does not affect system throughput or latency. The mirroring port is limited to an aggregate throughput of 10 Gbit/s, even though the NetStage Core supports 10 Gbit/s full-duplex traffic.

3) Low-Level Simulation Environment: When developing new Handlers at the register-transfer level using hardware-description languages such as Verilog or VHDL, the capability to perform system-level debugging becomes crucial for developer productivity.

To this end, a HDL simulator [57] was extended with an operating system-level virtual network interface using the Linux TAP mechanism. This allows the simulated HDL models of NetStage and the Handler under development to participate in actual network communication, receiving and sending real traffic. Despite the limited throughput and high latency, this capability has proven invaluable for the development of the NetStage Core as well as application-specific Handlers.

IV. High-Level Programming of NetStage

Even with the extended simulation/debugging support, developing Handlers at the RTL level is an activity generally unfamiliar to most networking experts. It is much preferable to allow the domain experts to express their application at a familiar abstraction level, leading to programming in so-called domain-specific languages (DSL).

As a first step in this direction, we developed Malacoda [58][54], a DSL that allows network security engineers to easily describe vulnerability emulations, which are then automatically compiled to Handlers for integration into the NetStage packet pipeline. Other DSLs could be formulated for other application domains, e.g., complex event processing, deep-packet inspection, or content-based routing.

In practical use, Malacoda has also proven useful to assist experienced hardware designers by quickly generating Handler skeletons in HDL, which could then be quickly extended manually with special-purpose features.

A. The Malacoda Language

When designing Malacoda, we surveyed how network security experts commonly customize honeypots. As an example, the Nepenthes honeypot [47] is programmed in scripts that follow a dialogue-based approach, where an incoming packet is inspected for patterns, which trigger the sending of appropriate response packets, mimicking the behavior of a vulnerable network application. The scripts are commonly structured as follows:

- Describe a sequence of steps (states) reflecting the progress of a communication session.
- Evaluate the incoming request packet.
- Craft a response packet from static template data, filling-in placeholders with dynamically computed data (e.g., excerpts from the request packet).
- Notify an administration station if specific states have been reached during a communication session.

As Perl [59] is a popular programming language in the system administration and networking domain, Malacoda provides a Perl subset suitable for easily expressing honeypot behaviors, and amenable for compilation into high-performance NetStage Handlers. Special commands, summarized in Table I, provide frequently used actions.

TABLE I
Malacoda Commands

addresponse(*SOURCE*)	Append a given byte sequence to the response packet buffer.
addresponse(*SOURCE*, s, n)	Copy n bytes starting at index s from *SOURCE* to the response packet.
log (*SOURCE*):	Create a log packet on the administrative interface from the given byte sequence.
replace(s, *SOURCE*)	Replace a single byte or a byte sequence in a response packet under construction with the given value starting at index s.
close:	Send a close-connection notification (only available for TCP).

Listing 1 shows an excerpt from a Malacoda description emulating a simple Telnet login into a shell which accepts any user / password combination. The keyword **dialogue** begins the activity description, which consists of a sequence of interactions with the client.

Listing 1. Sample Malacoda Telnet emulation

```
// Emulate login to a root shell
TELNET_VEH {
  // stateful dialogue description
  dialogue {
  //initial state
  DEFAULT:
    addresponse("Connected to localhost.
        localdomain\nlogin:");
    $STATE = LOGINWAIT;
    log("TELNET: Connection");
    ...
  SHELL:
    if ($INPKG  =~ /^ls/) {
      addresponse("web-password.txt\n");
      addresponse("[localhost]#");
    }
    elsif($INPKG  =~ /^whoami/) {
  ...
```

Fundamentally, the Malacoda dialogue is expressed as a finite-state machine (FSM), with NetStage tracking the current state per connection in the GASM (see Section III-E1). In Malacoda, these states are identified by symbolic names and set using the reserved variable **$STATE**. A new connection always starts in the initial state identified by the name **DEFAULT**. The initial state is also entered for connectionless traffic (e.g., arriving UDP packets). The response packet may be constructed incrementally using multiple **addresponse** calls, it will be sent automatically once all actions in a state have completed. The incoming request packet can be accessed using the reserved variable **$INPKG**, and used, e.g., in RE matching. More complex response packet templates can be read from external files, preserving the clear structure of the Malacoda source code.

In addition to the system-maintained reserved variables, Malacoda allows the definition of user-defined variables. They can be declared having flexible (a maximum length needs to be set) as well as fixed lengths, as shown in the following example:

```
dynamic variable1[8]; //flexible length
fixed variable2[4]; //fixed length
```

In Malacoda, all variables have global scope within a script and exist during the entire lifetime of the connection, possibly spanning multiple packets. The declarations only determine the size of the storage space reserved in the GASM, as Malacoda is typeless and treats all variable values as raw (hardware-level) byte sequences. They can, e.g., be used for arithmetic as well as string-operations, with the context of each use determining the interpretation of the byte sequences (see the next section for details).

Listing 2 shows an excerpt of a DNS server, demonstrating the use of **$INPKG** both for RE matching, as well as to pre-populate the contents of the response packet (which is selectively overwritten afterward by the **replace** command).

Listing 2. DNS emulation in Malacoda

```
// Emulate DNS response
DNS_VEH {
  dialogue {
  // A stateless Handler only has the default
      state
  DEFAULT:
    if ($INPKG[31] = "0") {
      if ($INPKG =~ /^.{13}www\\08malcobox\\02de
          /) {
        addresponse($INPKG);
        replace(3,"
            \\81\\80\\00\\01\\00\\01\\00\\02
            \\00\\02");
        addresponse("\\c0\\0c
            \\00\\01\\00\\01\\00\\00\\1c
            \\20\\00\\04\\51\\a9\\91");
        ...
        log("DNS: Request for www.malcobox.de");
        ...
```

B. Compiler Design

The Malacoda compiler translates the source code to hardware-synthesizable VHDL, which can then be submitted to standard FPGA vendor tools for mapping to the actual FPGA device(s). Due to the highly specialized nature of Malacoda, much of the complexity of conventional high-level language compilers [60] can be avoided. ANTLR v3 [61] was used in the development of the compiler to generate not only the lexer and parser, but also the complete Abstract Syntax Tree (AST) representation.

The AST is traversed to perform semantic analysis. In addition to building a symbol table of FSM states and variables, the pass collects information about a number of language constructs central tables. These include all conditions in the program (including regular expressions), all output packets (response and log), as well as all variable assignments. This is a departure from traditional compiler organization as, due to the specialized execution semantics of Malacoda, most of these constructs will be executing in parallel later. Additionally, the total number of bytes used for response packets is tracked for later optimization of the packet construction logic. The AST is decorated (shown in Figure 5) with basic block boundaries and the control predicates of each basic block.

The top VEH node has the individual state nodes as children. Each state node corresponds to a state block of the VEH dialog description. The state nodes can contain either **if/elsif/else** constructs or *grouped statements* nodes as children. **If/elsif** nodes contain a reference to their control condition and may have nested grouped statement nodes or another **if/elsif/else** block as children. A *grouped statement* collects all single occurrences of a particular statement within a basic block. E.g., it will be often the case that there are multiple **addresponse** statements in a basic block (to increase the readability of the VEH description). As only a single response packet will be created per execution cycle, all of these separate **addresponse** statements for constructing that packet will be aggregated into a single grouped statement that performs the construction task of the formerly separate statements.

The decorations in the AST are evaluated during code generation. VHDL code is generated by expanding VHDL code templates (Figure 6), replacing placeholders with actual signal declarations and output assignments. The static

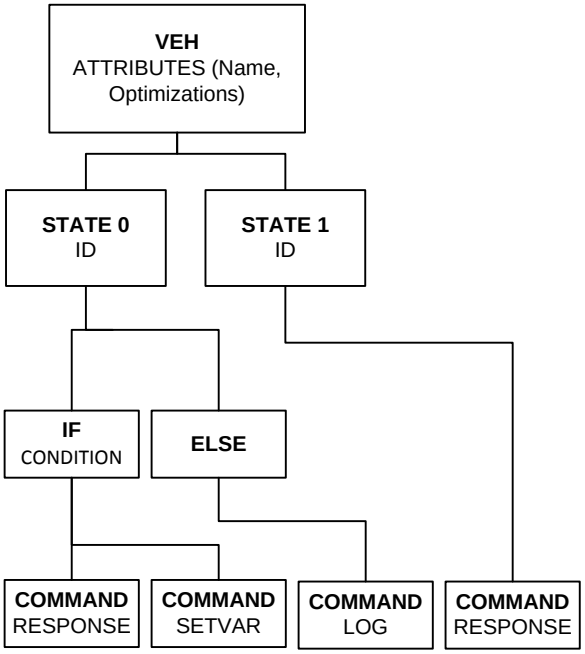


Fig. 5. Decorated abstract syntax tree for Malacoda descriptions

part of the template describes the buffered interface to the NetStage core and a skeleton FSM for receiving and sending messages, which is then extended with Handler-specific processing.

Code generation proceeds as follows: First, the static contents of response and log packets are inserted into the template. Then, all individual conditions are translated into their underlying hardware (e.g., regular expression matchers, simple comparators), computing boolean signals. A condition evaluation block combines these predicates into the more complex expressions that actually control the execution of the basic blocks within a state (also considering intra-state control flow, if any). The contents of the basic blocks are assembled from code block templates for the different possible statements. These include packet generation (response and log) and variable assignments. Note that a Malacoda program will at least contain the **DEFAULT** state, but may include an arbitrary number (within device limits) of additional states.

C. Target Microarchitecture

In order to reach the required throughput, the processing engines created by the compiler have an internal data width of 16 Bytes and aim to produce/consume one of these words per clock cycle. To this end, intra-word parallelism is exploited whenever possible. For example, RE matching tries to perform parallel matching of search strings (e.g., by generating dedicated comparators for each offset of a literal search string in the target string). The composition of an output packet from individual data sources is also performed in parallel. In the DNS server example in Listing 2, the sequence **addresponse/replace/addresponse** is compiled into a wide combinational circuit creating the entire

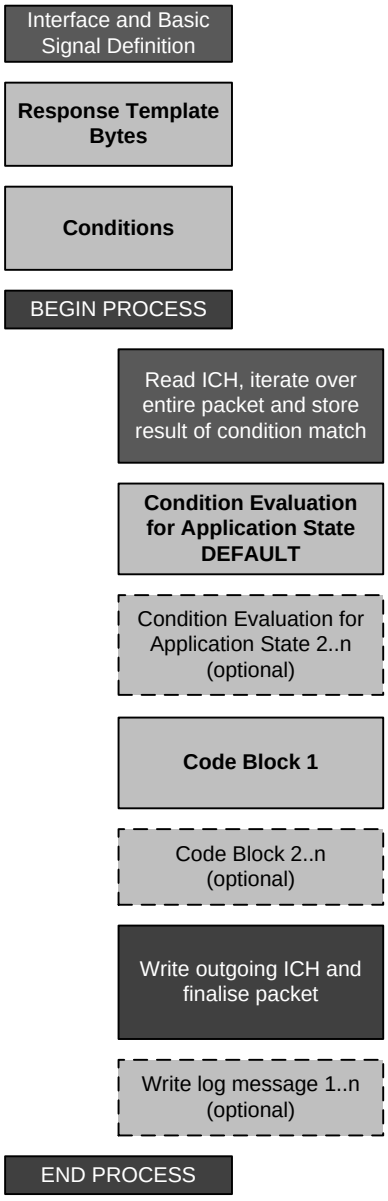


Fig. 6. Handler template

output packet in a single clock cycle (see Section IV-C4 for more details).

1) Handler Execution Cycle: The VHDL code that is generated by the Malacoda compiler reflects the structure of the basic Handler module described in Section III-D. Packets are read from an input buffer, processed by the corresponding hardware block, and any responses generated are finally written back to an output buffer. A central state machine controls the entire operation.

Figure 7 shows a sample execution cycle of a compiled Handler. Similar to execution models of VLIW processors or the Verilog non-blocking assignment, writes to Malacoda variables only take effect after the end of an execution cycle, all in parallel. Thus, the processing engine compiled for a Handler can begin a new execution cycle by evaluating all current conditions in the Handler in parallel. The Malacoda statements associated with each true condition

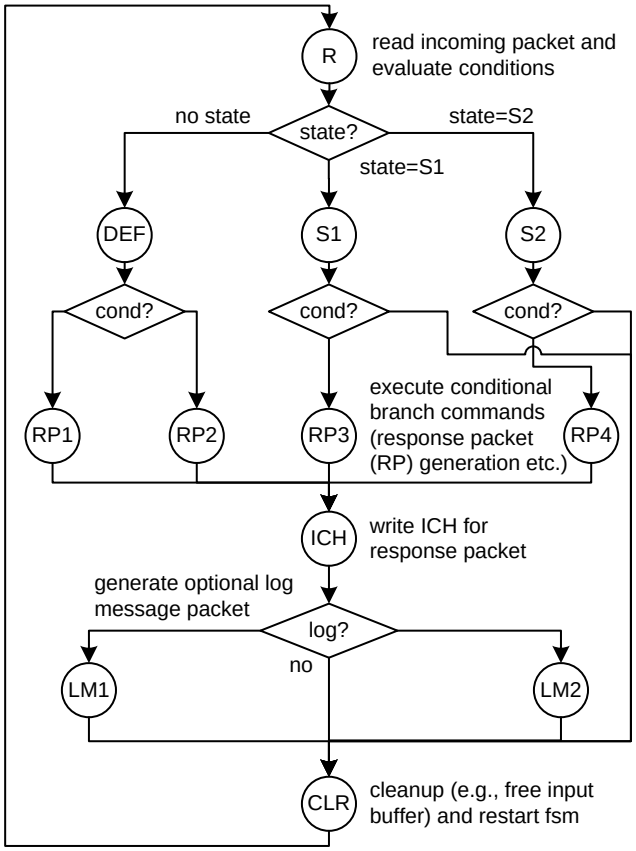


Fig. 7. Sample execution cycle of a compiled Handler

are then executed in program order, sequenced as separate states of the control FSM. Multiple statements performing writes to different parts (bytes) of a wide variable (or reply packet) are executed in parallel, as described in the prior section.

2) Conditions and Regular Expressions: The compilation of Malacoda conditions yields different hardware for basic and complex conditions. Basic conditions are those requiring matching a constant byte sequence at a constant offset within a variable or packet, including across 16B word boundaries for wider variables (or packets). These conditions, collected from the entire Malacoda source code of the Handler, are directly turned into wide comparators, all evaluated in parallel.

All other conditions are considered complex, and are implemented using RE matchers. The efficient hardware realization of RE matching has been the subject of intense study, e.g., in [62], [63]. However, these solutions are not ideal for typical Malacoda applications. First, they are generally optimized to support a large number of search strings (e.g., the entire set of Snort [5] patterns). Malacoda handlers, on the other hand, usually require just 5 to 10 regular expressions each. Second, the larger base area of the more powerful matching engines is not amortized over so few search strings.

In contrast, the RE matchers compiled from Malacoda rely on simple FSMs and thus require only limited hardware area for administrative purposes. The actual matching, however, is done using custom-generated networks of maximally parallel comparators for each RE, allowing single-cycle matching of all search hits within a single 16B input word. While delivering the required throughput of at least 10 Gbit/s, such an approach is indeed only practical for the limited number of search strings contained in a typical Malacoda Handler.

3) Variable Access and Allocation: Each Malacoda variable is implemented in two registers. One holds the current value as retrieved from GASM and bundled with the ICH-wrapped packet, the second one the new value to write to GASM at the end of the current execution cycle. At the start of each execution cycle, both registers will be initialized to the GASM-retrieved value. Separate registers are used to avoid inadvertent overwrites in Handlers with variable latency-execution cycles and reused (time-multiplexed) hardware. Thus, the semantics of variable writes taking place only after the end of the cycle can always be guaranteed.

The compiler allocates per-connection variables in the GASM. For simplicity, dynamic variables are allocated with their maximum sizes and contain a byte tracking their length, allowing up to 255 Bytes per variable. Static variables always have a fixed size and avoid the length byte. Thus, a simple contiguous addressing scheme can be used in the compiler.

4) Response Packet Generation: To achieve high throughput and aiming for non-blocking Handler execution, all operations affecting a single 16B data word execute in the same cycle. This is illustrated in the example shown in Figure 8 and will now be discussed in greater detail.

Response packets are generated by copying static data from stored templates (see Figure 8), which can then be further modified using Malacoda commands (e.g., using **replace**) at run-time. The Malacoda compiler implements the storage of the 16B templates either as LUTs (for small templates), or as BRAMs (for larger ones).

The actual generation of response packets not only has to deal with the retrieval of data from a template, but also its run-time modification. While the latter is quite easy when just overwriting data at fixed offsets in the template data (shown for **replace** in the figure), the system also has to deal with variable-length parts of the packet and dynamic variables. Figure 8 shows this as appending a template to a variable-length input package (retrieved from **$INPKG**)

In that case, the current offset into the 16B word needs to be tracked during packet construction and the dynamic parts have to be placed into the output packet at dynamically calculated offsets (see Figure 8). To achieve this within one clock cycle, a barrel shifter is used to make the template data available shifted to any of the 16 positions in the 16B output word. Appending it to the packet-under-construction then just consists of selecting the correctly (for the current offset) shifted version of the template and selecting these bytes as contents of the output word.

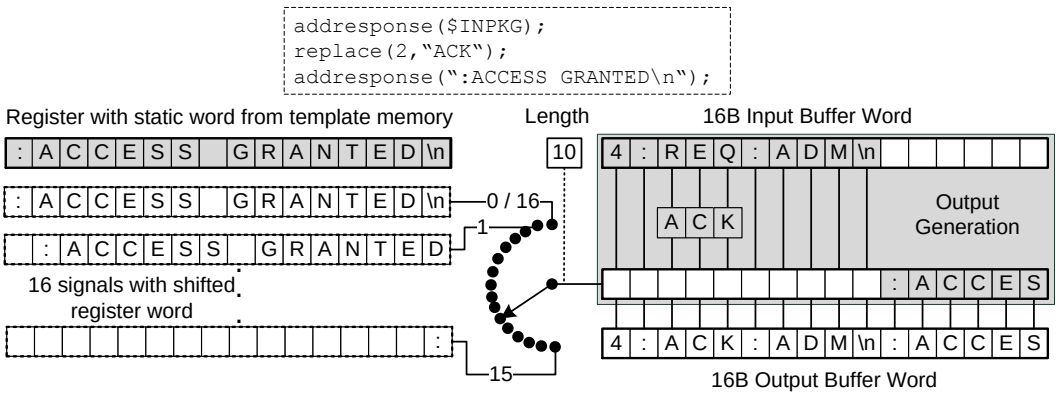


Fig. 8. Single-cycle output word generation

D. Compiler Optimization

Beyond the straighforward compilation flow sketched in Section IV-B, Malacoda performs a number of highly domain-specific optimizations.

For some protocols (e.g., DNS), the response packet contains much of the data received in the original request packet. The compiler detects this by looking for an **addresponse($INPKG)** command (which copies the input buffer to the output buffer) in the Malacoda program. If such a construct is detected, dedicated wiring is generated to perform this zero-overhead forwarding in hardware as soon as an input packet arrives. This avoids having to read the entire packet again when building the output packet. Additional static or dynamic data can be appended at the end of the copied block. Note that this operation is speculative: If a control condition would actually select a different execution path, the prematurely copied packet is instantly discarded from the output buffer simply by resetting the buffer write pointer. But no execution time is wasted copying the packet data. Note that if an **addresponse($INPKG)** construct is not present in the Malacoda program, the costly forwarding wiring is not created.

The compiler also selects appropriate storage for the packet templates: LUT-based storage is fast (allows 0-cycle combinational access), but eventually becomes inefficient for larger templates. On-chip BRAMs can easily hold these, but add an additional cycle to state execution for data access. The Malacoda compiler switches between both storage methods depending on the template size: Experiments have shown that templates smaller than 1024 bytes are best stored in LUTs, all larger templates will be maintained in BRAM.

The barrel shifter-approach (shown in Figure 8) allows fast dynamic editing of variable contents. However, it requires significant chip area. The compiler thus specifically checks whether such editing is actually required in the current Malacoda program. In the absence of such commands, the shifter will be replaced by appropriate static wiring instead.

Hardware support for persistent state storage using the GASM is also created on demand: If the Malacoda program contains only the **DEFAULT** state and does not declare any custom variables, the logic for GASM access (read and store to ICH) is omitted to conserve hardware resources.

V. Experimental Evaluation

This section discusses a system-level evaluation of NetStage, the Malacoda compiler, and the Malcobox honeypot application. It is structured into two subsections presenting the characteristics of the hardware implementation and the packet-processing performance. Experiences gathered during a one-month live test of the Malacoda-compiled Malcobox honeypot connected to a direct 10G Internet uplink are given in [54].

A. Reconfigurable Target Platform

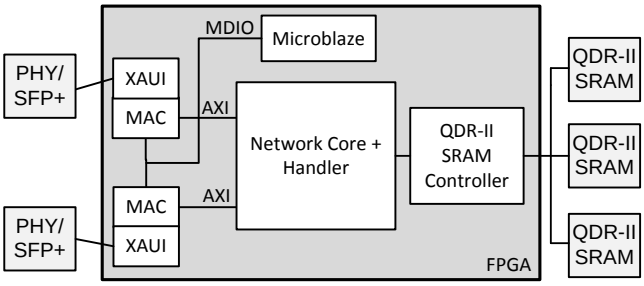


Fig. 9. NetStage on a NetFPGA 10G platform

For these experiments, we have used the NetFPGA 10G card as reconfigurable processing platform. Similar to the NetFPGA Loopback example design [64], we employ a MicroBlaze soft-core processor embedded in our design for quick and easy configuration of the SFP+ transceiver module parameters. Note that the MicroBlaze does not touch any packet data at all, thus maintaining our security requirement of using only hardware resilient against malware injection attacks for the actual packet processing. The system uses two 10G ports, one for the Internet uplink, the second one for the administrative interface. To gauge the impact of having available low-latency off-chip memory, we can also selectively enable the use of the QDRII SSRAM memory present on the card.

The original NetFPGA 10G example design used different clocks for the network interface and the core packet processing logic. While this allows lower-speed core logic to be integrated into the system, it leads to deteriorated place-and-route results. Since the NetStage core was carefully optimized to also reach the 156.25 MHz of the network interface, we have been able to eliminate the separate core clock (and the associated clock domain crossing logic) in NetStage, and achieved significantly improved mapping results as a consequence.

B. Hardware Synthesis Results

Logic synthesis and FPGA mapping has been performed using the ISE, PlanAhead, and EDK tools from the Xilinx software suite, using version 13.3 [65]. Two different configurations have been synthesized: a) contains only the basic components, while b) is a full featured design that also uses the external QDRII SSRAMs on the NetFPGA 10G card for holding the GASM and has a statistics module enabled. Both configurations contain six Handler Slots. Table II lists the corresponding results.

TABLE II
LOGIC SYNTHESIS RESULTS FOR NETSTAGE COMPONENTS

Module	LUT	Reg. Bits	BRAM	Clock [MHz]
(a) w/o external SSRAM and w/o statistics				
NetStage Core	16,441	19,363	107	164
Empty Slot	535	684	4	286
Core/Slot Routing	2,962	2,955	25	175
NetStage Total	**19,402**	**22,935**	**135**	**164**
(b) external QDRII SSRAM and statistics module enabled				
NetStage Core	29,903	29,696	85	164
Empty Slot	937	976	4	286
Core/Slot Routing	3,953	3,306	25	190
NetStage Total	**35,945**	**36,188**	**113**	**164**

Without external state data, the NetStage core requires 10% of the LUT resources and 33% of the BRAMs. The high number of BRAMs reflects the many internal buffers linking the different modules in the core. In relation to the FPGA size, the total size of the infrastructure including core, management, slots, and routing, is relatively small (just 12% of LUTs), which leaves sufficient area for the Handlers. Compared to earlier NetStage versions [66], [51], the increased number of register bits here is due to additional pipeline registers, inserted to achieve more reliable timing closure on the different NetStage platforms.

When adding the statistics option in configuration b), the logic/register resources increase by more than 50% compared to configuration a). This is due to the many performance counters which are then inserted into the system. Additional area is required for making the counters readable over the administrative interface. For both configurations, the critical path is identical. It passes through the highly parallelized implementation of the IP-layer checksum.

Comparing these synthesis results to related work is impeded both by the different feature sets of other approaches, as well as the scarcity of published results in general. Dini Group, Inc., [21] reports a resource usage of 3,889 FFs and 6,885 LUTs for their single connection core, with software-support for connection establishment and ARP / ICMP. Intilop [20] gives only a number of 'less than 30,000' slices for their full-featured hardware TCP core. NetStage in configuration a), which appears to be the most similar one, requires 23,859 slices.

TABLE III
SYNTHESIS RESULTS FOR COMPILED HANDLER MODULES

Handler	LUT	FF	BRAM	Max. Clock [MHz]
SMB	2,371	1,497	4	202
DNS	2,821	1,444	0	204
MSSQL (Slammer)	1,841	1,289	0	225
Telnet	3,910	1,642	0	176
Mail	2,464	1,541	0	204
Web	2,394	1,357	4	214

To collect application-level results, six different Handlers, emulating typical network services or actual vulnerabilities, have been developed and compiled using Malacoda [54]. They include a Web server, a Telnet CLI, a Mail server, a DNS server, an SMB login monitor and a Slammer worm [67] detector. Note that these Handlers are not toy examples, they are full-capability implementations that have been used in a real data center environment to capture attack attempts.

Table III shows the synthesis results for these Handlers. Since they all follow the same microarchitecture template (see Section IV-B), they are close in size and require just 2 to 4 percent of the device resources each. Comparing the quality-of-results of the Malacoda-compiled handlers with the original manual implementations [66] is difficult due to the alterations (described in detail in [54]) that had to be made to the Handler microarchitecture. The modifications consist mainly of more pipeline registers, but also of a better modularization of functionality in the Malacoda-generated microarchitecture, as compared to the carefully hand-optimized designs of our earlier work.

In practice, the TX240T device on the NetFPGA 10G card supports around eight Malacoda-compiled Handlers, before place-and-route will run into difficulties. The final static[2] bitstream of the Malacoda-compiled design, including the basic infrastructure from configuration b) and the six Handlers, requires a total of 46,582 LUTs, 46,019 FFs and 184 BRAMs.

C. Packet Processing Performance

While the performance of the NetStage Core (latency and throughput) is mostly independent of the application, the performance of each single Handler does depend on

[2]The use of dynamic partial reconfiguration in NetStage is described in [53]

its individual complexity. Table IV shows the application-level throughput at the Handler in Gb/s, running at the current system target frequency of 156.25 MHz, and using the message sizes listed in brackets. Factors affecting performance are a fixed number of clock cycles overhead per packet for administrative functions (e.g., processing the internal control header data, preparing notifications), and a variable number of clock cycles dependent on the payload size of the packet for content-related activities. Note that these performance numbers are guaranteed, as all processing blocks are implemented using dedicated (non-shared) resources that do not suffer from increased system load.

TABLE IV
PERFORMANCE FOR EXAMPLE HANDLER OPERATIONS

Handler Operation	**Gb/s**
Web GET (78 Byte IN, 405 Byte OUT)	15.6
Send Mail (800 Byte IN, 14 Byte OUT)	16.9
Telnet uname (6 Byte IN, 30 Byte OUT)	7.0

As Table IV shows, the constant number of overhead cycles becomes especially costly for very small payloads. However, only that specific Handler is slowed down, the NetStage core continues to process connections (including the TCP/IP stack) to other Handlers at the full speed of up to 20 Gb/s. Should tiny-payload performance be critical for certain applications, the limitation could be worked around by running multiple independent instances of the Handler, and performing load-balancing between them for higher aggregate throughput.

Next to throughput, latency is another key characteristic of a network processor architecture. As before, the lack of published third-party results hampers an objective comparison. Only Dini Group, Inc., reports information on the latency of its IP block between the arrival of first byte at the core input and the availability of the first byte on the application side (time to first byte, TTFB) for a 100 Byte packet payload, giving a latency of 120ns [68]. NetStage, which is optimized for a multitude of parallel connections as opposed to Dini's single-connection architecture, still manages to achieve a TTFB of 270ns for the same 100 Byte TCP payload size. Again, the added latency is due to the store-and-forward design of NetStage.

To put this into perspective, note that on a regular Linux server system, even with a highly optimized network card and software stack such as Myricom DBL (Datagram Bypass Layer), the user-level read latency is about 1.5μs [69]. This demonstrates the significant performance advantage of hardware-level network processing over purely software-based solutions.

VI. SUMMARY AND OUTLOOK

The security of computer systems and networks is one of the key issues for the future growth of the Internet. But novel attack patterns and huge traffic rates easily overload purely software-based network security solutions.

The approach proposed here not only offers performance efficiency exceeding that of many software solutions, but also a resiliency against malware injection attacks (e.g., due to buffer overflows), as no software-programmable processors exists in the system that could be compromised.

Our reconfigurable NetStage architecture allows rapid prototyping of FPGA-based applications supporting autonomous Internet communication without any CPU. NetStage encompasses a core implementation of the fundamental Internet communication protocols, a set of supporting services, a flexible interface for embedding application-layer protocols directly into the processing pipeline, and utilities to support development for the platform. The entire system has been evaluated as base for Malcobox, a completely hardware implemented honeypot, under real-world conditions [54].

A commonly voiced complaint against solutions relying on reconfigurable computing is the lack of high-level programming tools, requiring domain experts to also be experienced hardware designers. We have addressed this for the honeypot domain by defining the Malacoda language and implementing the associated compiler, allowing domain experts to program the system in a familiar notation, yet exploiting the performance and attack resilience of the NetStage base.

Further research is promising both on the hardware architecture as well as compiler sides. For the former, adding IPv6 support as well as transparent decryption/encryption are of particular interest for many applications. For the latter, the current proof-of-concept nature of the Malacoda compiler offers much space for improvement, e.g., with regard to more general variable accesses or better support for logical and arithmetic functions.

ACKNOWLEDGMENT

This work has been supported by the Hessian Ministry for Science and the Arts under the LOEWE program and Xilinx, Inc.

REFERENCES

[1] Symantec, "Internet threat report 2011," 04 2012.
[2] Pandalabs, "Pandalabs security report," 2012.
[3] G. Chanda, "The market need for 40 gigabit ethernet (white paper)," Cisco Systems, 2012. [Online]. Available: http://www.cisco.com
[4] F. Alserhani, M. Akhlaq, I. U. Awan, J. Mellor, A. J. Cullen, and P. Mirchandani, "Evaluating intrusion detection systems in high speed networks," in *Proceedings of the 5th. International Conference on Information Assurance and Security - Vol.02*, 2009, pp. 454–459.
[5] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX conference on System administration*, 1999, pp. 229–238.
[6] H. Lange and A. Koch, "An execution model for hardware/software compilation and its system-level realization," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*. IEEE, 2007, pp. 285–292.
[7] T. Katashita, Y. Yamaguchi, A. Maeda, and K. Toda, "Fpga-based intrusion detection system for 10 gigabit ethernet," *IEICE Trans. Information and Systems*, vol. E90-D, pp. 1923–1931, 2007.
[8] G. S. Jedhe, "A scalable high throughput firewall in fpga," in *16th International Symposium on Field-Programmable Custom Computing Machines*, 2008.

[9] M. Blott, "Fpgas head for the cloud," *Xcell journal*, vol. 80, pp. 20–23, 2012.

[10] H. Chen, Y. Chen, and D. Summerville, "A survey on the application of fpgas for network infrastructure security," *Communications Surveys Tutorials, IEEE*, vol. 13, no. 4, pp. 541–561, Fourth 2011.

[11] R. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122 (Standard), Internet Engineering Task Force, Oct. 1989, updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633. [Online]. Available: http://www.ietf.org/rfc/rfc1122.txt

[12] J. Postel, "Transmission Control Protocol," RFC 793 (Standard), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168, 6093, 6528. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[13] D. Schuehler and J. Lockwood, "Tcp-splitter: A tcp/ip flow monitor in reconfigurable hardware," in *High Performance Interconnects, 2002. Proceedings. 10th Symposium on*, 2002, pp. 127 – 131.

[14] D. V. Schuehler, "Techniques for processing tcp/ip flow content in network switches at gigabit line rates," Ph.D. dissertation, Sever Institute of Washington University, 2004.

[15] T. H. Vu, N. Q. Tuan, T. N. Thinh, and N. T. H. Nguyen, "Memory-efficient tcp reassembly using fpga," in *Proceedings of the Second Symposium on Information and Communication Technology*, ser. SoICT '11. New York, NY, USA: ACM, 2011, pp. 238–243. [Online]. Available: http://doi.acm.org/10.1145/2069216.2069261

[16] R. Yuan, Y. Weibing, C. Mingyu, Z. Xiaofang, and F. Jianping, "Robust tcp reassembly with a hardware-based solution for backbone traffic," in *Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on*, july 2010, pp. 439 –447.

[17] 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.

[18] H. Jang, S.-H. Chung, and D.-H. Yoo, "Design and implementation of a protocol offload engine for tcp/ip and remote direct memory access based on hardware/software coprocessing," *Microprocess. Microsyst.*, vol. 33, no. 5-6, pp. 333–342, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2009.03.001

[19] C. Leber, B. Geib, and H. Litz, "High frequency trading acceleration using fpgas," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, sept. 2011, pp. 317 –322.

[20] Intilop, "10 g bit tcp offload engine (toe) - hardware ip core - top level product specifications." [Online]. Available: http://www.intilop.com/resources/product_briefs/10G.pdf

[21] D. Group, "Tcp offload engine ip (toe) product overview," 2012. [Online]. Available: http://www.dinigroup.com/new/TOE.php

[22] PLDA, *QuickTCP Core - Reference Manual*, PLDA, 2012.

[23] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "Netfpga–an open platform for gigabit-rate network switching and routing," in *Proc. of the 2007 IEEE International Conference on Microelectronic Systems Education*, ser. MSE '07. IEEE Computer Society, 2007, pp. 160–161.

[24] INVEA-TECH a.s., "Netcope product brief," [accessed 15 Jul 2011]. [Online]. Available: http://www.invea-tech.com

[25] INVEATECH, "Combo product brief." [Online]. Available: http://www.invea-tech.com/products-and-services/combo-fpga-boards

[26] P. Korcek, V. Kosar, M. Zadnik, K. Koranda, and P. Kastovsky, "Hacking netcope to run on netfpga-10g," in *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 217–218. [Online]. Available: http://dx.doi.org/10.1109/ANCS.2011.40

[27] H. Fallside, "Internet connected fpl," in *FPL 2000*, 2000.

[28] I. Gonzalez, F. Gomez-Arribas, and S. Lopez-Buedo, "Hardware-accelerated ssh on self-reconfigurable systems," in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, dec. 2005, pp. 289 – 290.

[29] R. Sadoun, "An fpga based soft multiprocessor for dns/dnssec authoritative server," *Microprocessors and Microsystems*, vol. 35, 2011.

[30] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. A. Vissers, "A low-latency library in fpga hardware for high-frequency trading (hft)," in *IEEE 20th Annual Symposium on High-Performance Interconnects (HOTI 2012)*, 2012, pp. 9–16.

[31] D. Hildenbrand, J. Pitt, and A. Koch, "Gaalop-high performance parallel computing based on conformal geometric algebra," *Geometric Algebra Computing, ISBN 978-1-84996-107-3. Springer-Verlag London Limited, 2010, p. 477*, vol. 1, p. 477, 2010.

[32] N. Rotem, "C to verilog." [Online]. Available: http://www.c-to-verilog.com/

[33] H. Gadke and A. Koch, "Comrade - a compiler for adaptive computing systems using a novel fast speculation technique," in *Proceedings of the International Conference on Field Programmable Logic and Applications, 2007. FPL 2007*, 2007.

[34] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch, "Hardware/-software co-compilation with the nymble system," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*. IEEE, 2013, pp. 1–8.

[35] *UG902: Vivado High Level Synthesis User Guide*, Xilinx.

[36] *Catapult Product Family Datasheet*, Calypto Design Systems, Inc, Available online at: calypto.com. [Online]. Available: calypto.com

[37] *Synphony C Compiler Datasheet*, Synopsys, Inc, Available online at: www.synopsys.com. [Online]. Available: www.synopsys.com

[38] W. Meeus, K. V. Beeck, T. Goedem, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *International Journal of Design Automation for Embedded Systems (DAES)*, 2012.

[39] Bluespec, "Bluespec vs. c/c++/systemc modeling - white paper." [Online]. Available: http://www.bluespec.com

[40] G. Brebner, "Packets everywhere: The great opportunity for field programmable technology," *Proc. Intl. Conference on Field Programmable Technology*, pp. 1–10, 2009.

[41] P. Jungck, R. Duncan, and D. Mulcahy, *packetC Programming*, 2012.

[42] R. Duncan and P. Jungck, "packetc language for high performance packet processing," in *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, june 2009, pp. 450 –457.

[43] E. Rubow, R. McGeer, J. Mogul, and A. Vahdat, "Chimpp: a click-based programming and simulation environment for reconfigurable networking hardware," in *Proc. 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2010, pp. 36:1–36:10.

[44] M. Lavasani, L. Dennison, and D. Chiou, "Compiling high throughput network processors," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145709

[45] N. Provos and T. Holz, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley Professional, 2007.

[46] N. Provos, "A virtual honeypot framework," honeyD.

[47] P. Baecher, M. Koetter, M. Dornseif, and F. Freiling, "The nepenthes platform: An efficient approach to collect malware," in *In Proceedings of the 9 th International Symposium on Recent Advances in Intrusion Detection (RAID*. Springer, 2006, pp. 165–184.

[48] Dionaea, "Dionaea documentation." [Online]. Available: http://dionaea.carnivore.it

[49] V. Pejovic, I. Kovacevic, S. Bojanic, C. Leita, J. Popovic, and O. Nieto-Taladriz, "Migrating a honeypot to hardware," in *SECURE-WARE '07: Proc. Intl. Conf. on Emerging Security Information, Systems, and Technologies*. IEEE Computer Society, 2007, pp. 151–156.

[50] C. Leita, K. Mermoud, and M. Dacier, "Scriptgen: an automated script generation tool for honeyd," in *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005, pp. 203–214.

[51] S. Mühlbach and A. Koch, "An fpga-based scalable platform for high-speed malware collection in large ip networks," in *Field-Programmable Technology (FPT), 2010 International Conference on*, dec. 2010, pp. 474 –478.

[52] S. Mühlbach, M. Brunner, C. Roblee, and A. Koch, "Malcobox: Designing a 10 gb/s malware collection honeypot using reconfigurable technology," *International Conference on Field Programmable Logic and Applications*, vol. 0, pp. 592–595, 2010.

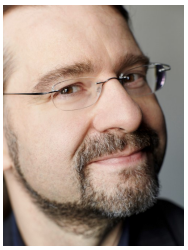
[53] S. Mühlbach and A. Koch, "A dynamically reconfigured network platform for high-speed malware collection," in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, dec. 2010, pp. 79 –84.

[54] ——, "Malacoda: towards high-level compilation of network security applications on reconfigurable hardware," in *Proceedings of the Symposium on Architecture for Networking and Communications Systems, ANCS '12*, 2012, pp. 247–258.

[55] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "Netfpga: reusable router architecture for experimental research," in *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, ser. PRESTO '08. New York, NY, USA: ACM, 2008,

pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/1397718.1397720
[56] H. Gadke-Lutjens, B. Thielmann, and A. Koch, "A flexible compute and memory infrastructure for high-level language to hardware compilation," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 475–482.
[57] M. Graphics, *Modelsim SE Documentation*.
[58] S. Mühlbach and A. Koch, "A novel network platform for secure and efficient malware collection based on reconfigurable hardware logic," in *Internet Security (WorldCIS), 2011 World Congress on*, feb. 2011, pp. 9 –14.
[59] Perldoc, "Perl language reference." [Online]. Available: http://perldoc.perl.org/perlsyn.html
[60] A. Koch, "Adaptive computing systems and their design tools," in *Dynamically Reconfigurable Systems*, M. Platzner, J. Teich, and N. Wehn, Eds. Springer Netherlands, 2010, pp. 117–138. [Online]. Available: http://dx.doi.org/10.1007/978-90-481-3485-4_6
[61] T. Parr, *The Definitive Antlr Reference: Building Domain-Specific Languages*, 2007. [Online]. Available: http://www.antlr.org/
[62] Y.-H. Yang and V. Prasanna, "High-performance and compact architecture for regular expression matching on fpga," *Computers, IEEE Transactions on*, vol. 61, no. 7, pp. 1013 –1025, july 2012.
[63] K. Wang, Y. Qi, Y. Xue, and J. Li, "Reorganized and compact dfa for efficient regular expression matching," in *Communications (ICC), 2011 IEEE International Conference on*, june 2011, pp. 1 –5.
[64] NetFPGA, "Netfpga 10g loopbacktest example design." [Online]. Available: https://github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-10G-10G-Ethernet-Interface-Loopback-Test
[65] *Xilinx ISE 13.3 - Synthesis and Simulation Design Guide*, Xilinx.
[66] S. Muehlbach and A. Koch, "Netstage/dpr: A self-adaptable fpga platform for application-level network security," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, A. Koch, R. Krishnamurthy, J. McAllister, R. Woods, and T. El-Ghazawi, Eds. Springer Berlin / Heidelberg, 2011, vol. 6578, pp. 328–339. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19475-7_35
[67] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *Security Privacy, IEEE*, vol. 1, no. 4, pp. 33 – 39, july-aug. 2003.
[68] D. Group, "Toe latencies product brief," 2012. [Online]. Available: http://www.dinigroup.com/product/data/DNTOE/files/TOE_latencies_v103.pdf
[69] Myricom, "Myricom dbl downloads." [Online]. Available: https://www.myricom.com/dbl.html

Sascha Mühlbach Sascha Mühlbach received his diploma in computer science and engineering in 2006 from the Technical University Hamburg-Harburg (Germany). He has been working for two major internet and telecommunication providers as system engineer and technical product manager, before he joined the Center for Advanced Security Research Darmstadt (Germany) in 2009 as a doctoral researcher in the field of hardware-based network security for high-speed environments. His current research interests include reconfigurable hardware architectures for network security systems and hardware-efficient network intrusion detection algorithms.



Andreas Koch Andreas Koch received his diploma in informatics and his doctorate in engineering in 1992 and 1997, respectively, both from the Technical University Braunschweig (Germany). He then joined the University of California at Berkeley as a Post-Doctoral Scholar and returned to Braunschweig in 1999, continuing his research on hardware architectures and design tools. After his habilitation in 2005, Andreas Koch joined the Technische Universität Darmstadt (Germany) as Computer Science faculty, heading the newly founded Embedded Systems and Applications Group. His current research interests include hardware/software co-compilers, computer architecture and compute-intense embedded systems. He is a member of ACM, GI, IEEE and a Principal Investigator at the Center for Advanced Security Research (CASED) in Darmstadt.