Malacoda: Towards High-Level Compilation of Network Security Applications on Reconfigurable Hardware

Sascha Muehlbach
Center for Advanced Security Research Darmstadt (CASED)
Secure Things Group
64293 Darmstadt, Germany
sascha.muehlbach@cased.de

Andreas Koch
Technische Universitaet Darmstadt
Department of Computer Science
Embedded Systems and Applications Group
64289 Darmstadt, Germany
koch@esa.cs.tu-darmstadt.de

ABSTRACT

While the use of reconfigurable computing for tasks such as packet header processing or deep packet-inspection in high-speed networks has been widely studied, efforts to extend the technology to application-level processing have only recently been made. One issue that has prevented wider use of reconfigurable platforms in that context is the unfamiliar programming environment: Such systems commonly require expertise in computer architecture and digital logic design generally foreign to networking experts. To make the technology more accessible to potential users, we present the high-level domain-specific language Malacoda for application-level network processing and an associated compiler that automatically translates Malacoda descriptions into high-performance hardware blocks for insertion into an FPGA-based processing platform. We evaluate our approach on the use-case of a hardware-accelerated secure honeypot-in-a-box, programmed in Malacoda, and implemented on the NetFPGA 10G board. Results from a live-test of the system connected to a 10G Internet uplink complete the evaluation.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Design Aids—*Automatic Synthesis*; C.2.0 [**Computer Communication Networks**]: General—*Security*

General Terms

Design, Security

1. INTRODUCTION

Modern high-speed networks tax the capabilities of conventional software-based solutions to provide the required performance. Reconfigurable technology has long been used for packet-header processing (e.g., switching, routing, firewalls etc.) and has also been successfully applied to Deep Packet-Inspection (DPI), e.g., to accelerate the Snort Network Intrusion Detection System (NIDS) [21] using dedicated hardware [7].

But high performance is not the only advantage of using reconfigurable logic for network processing. Especially in the security domain, the *absence* of a general-purpose software-programmable microprocessor, which could be compromised by an appropriate attack injecting malicious code, can be exploited to security-harden front-line systems. An application that perfectly fits here are honeypot systems. Honeypots emulate vulnerable applications to attract potential attackers and can be used for various purposes: gathering information about new attacks, collecting statistics about propagation of Malware, or even for active network protection (e.g., [16]). Due to their exposed placement, a common risk for software-based honeypots is their possible subversion into attacking other hosts, if the systems are not carefully monitored. This is even more crucial if the honeypot is participating in an active defense scenario.

We exploit both the high processing performance as well as the improved security of dedicated hardware to showcase the implementation of MalCoBox, a hardware honeypot-in-a-box [13]: Following the well-known low-interaction honeypot approach [6, 4], the MalCoBox emulates an attack surface of hundreds of thousands of vulnerable hosts executing applications having security flaws and collects the malicious attack packets, which can then be studied by security researchers to derive anti-virus signatures or other defenses. Since the system processes data at wire speed, it cannot be overwhelmed, e.g., by a distributed denial of service (DDoS) attack. Furthermore, due to the lack of a compromisable processor, the honeypot cannot be abused by attackers for malicious activities.

While a prototype based on our underlying high-speed network processing platform NetStage has already been presented previously [15], its practical use was limited due to the programming requiring experience in hardware design and the associated tool flows, in addition to networking expertise. Even with these skills, the low-level implementation of accelerated protocol handlers in hardware description languages (HDL) such as VHDL or Verilog still takes significant effort. This is doubly detrimental for our honeypot use-case. On one hand, network security researchers will generally lack the hardware design experience. On the other, with the dynamic attack landscape of the Internet, new vulnerability emulations must be created quickly in order to keep up.

ANCS'12, October 29–30, 2012, Austin, Texas, USA.

As a solution, the major new contributions of this work are:

- A specialized high-level language (Malacoda) for concisely describing service emulations and application-level (ISO / OSI Layer 7) vulnerabilities for the honeypot.
- An associated compiler for creating fast hardware units executing on NetStage.
- A fully functional implementation of the described honeypot system on the NetFPGA 10G card, programmed in Malacoda, and stress-tested in a real data center environment.

A brief overview of the NetStage platform is given in Section 3. Section 4 covers the Malacoda language for programming in the honeypot domain, followed by a description of the current prototype compiler in Section 5. We evaluate our approach using it for an implementation of the hardware honeypot-in-a-box on the NetFPGA 10G board and discuss the results of a long-term live evaluation run in Section 6, before we conclude and look forward to further research.

2. RELATED WORK

Making reconfigurable network processing platforms more accessible for non-hardware designers has been the subject of considerable research. This has ranged from focused (but very effective) approaches such as compiling the Snort payload signature ruleset (regular expressions) into hardware accelerators [7] to more general-purpose solutions such as G [3] and Chimpp [22].

G is a general-purpose language, but specialized for packet-header processing. It allows users to flexibly specify the packet format (fields and positions) and conditional rules for modifying these fields depending on packet contents. The programs are then compiled into hardware units to be integrated into a larger system (not described in [3]). While capable of payload processing, G lacks regular expression handling and extended support for protocols above the level of processing individual incoming packets. The focus on header processing in G is also emphasized by the example applications, which deal with switching or MPLS routing [3].

Chimpp is more general framework in that it relies on an XML description for the composition of arbitrary packet-handling hardware blocks. These blocks can be of various granularities (e.g., ARP lookup or simple TTL decrement), but must be implemented manually in a synthesizable HDL. Chimpp only supplies the interfacing / composition capabilities. The authors propose a basic library of modules with focus on routing applications for the NetFPGA platform [11], which they use to build an IP router and NAT gateway as sample application. However, modules for higher-level payload processing are not provided. The packet header-processing roots of Chimpp are apparent when considering that it takes it inspiration from Click [8], a popular software framework for the description of routing operations.

NetThreads [9] uses an alternative approach to improve the programmability of the reconfigurable network processing system: Instead of generating custom logic, NetThreads defines specialized 4-way multi-threaded processors on the NetFPGA platform, which are then software-programmable in languages such as ANSI C. While this offers networking experts a familiar programming environment, for complex tasks the performance of the system does not reach the performance of dedicated hardware accelerators. E.g., for a sample application that does regular expression matching to classify HTTP packets, a performance of roughly 2000 Packets/s is given in [9], which is comparable to a throughput of approx. 16 Mb/s for 1024B packets. Without dedicated hardware accelerators for regular expression or protocol processing, it appears questionable that the approach has performance benefits exceeding those of existing hardwired network processor ASICs [24, 18].

While it would be possible to support a C programming environment for custom-generated reconfigurable network processing units by using one of the commonly available C-to-Gates compilers (e.g., [12, 23]), Brebner [3] shows a productivity gain of more than 6x for using a domain-specific language such as G versus a similar implementation in hardware-synthesizable C (which also has to take the language idiosyncrasies of the specific C-to-Gates compiler into account).

In summary, we are aiming for a system with the flexibility of NetThreads (supporting full protocol interaction processing) and the conciseness of a domain-specific language such as G. To this end, we require not only header, but more advanced payload processing capabilities such as regular expression matching and state tracking. The descriptions should be compiled into multi-threaded hardware units tightly integrated into our high-performance 10G network processing architecture, which provides the underlying general-purpose Internet communication functionality. None of the prior solutions match these requirements.

In terms of the honeypot application, MalCoBox represents the first attempt (to our knowledge) to implement such a system entirely on dedicated hardware. In contrast to the work of Pejovic et al. [20], where memory table-based state machine are interpreted to describe the client-server interaction, we rely on dedicated hardware accelerators for this task. Pejovic et al. implemented a prototype on a Virtex-4 FPGA, but unfortunately did not publish any performance benchmarks. While their table-driven approach is easier to implement in hardware, we expect memory bandwidth to become a bottleneck when network speeds of 10+ Gb/s are considered. Also, they propose the use of a PowerPC CPU to implement parts of the higher-level protocols, which we strictly avoid on our architecture for security reasons.

3. PLATFORM ARCHITECTURE

Figure 1 shows the current architecture of NetStage [15] for the NetFPGA 10G card, which meets the requirements of a basic network server application. Core features include a specialized implementation of the basic Internet protocols (IP, ARP, ICMP, UDP, TCP) as well as facilities for routing packets, scheduling time-based events (e.g., for packet retransmissions), and per-thread state (context) storage. An external management interface allows the monitoring and control of the system independently of the production network.

The network communication core (Fig. 1.a) implements the low- and mid-level Internet protocols. All of the modules have separate 128b wide transmit/receive datapaths, allowing full-duplex operation and reach 20 Gb/s throughput at the nominal 156.25 MHz clock frequency of the network in-

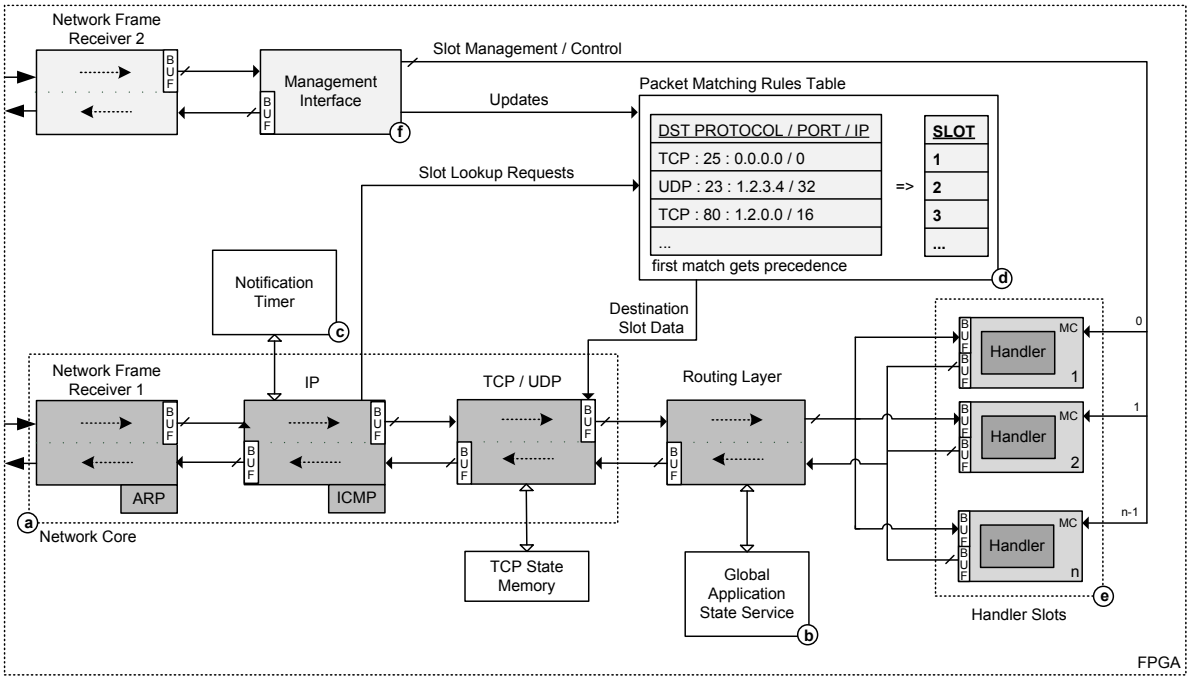

Figure 1: Core architecture of NetStage

terfaces (note that higher speeds of the core are possible, see Section 6). All of the processing stages are decoupled using buffer queues to limit the impact of throughput variations (e.g., during dynamic partial reconfiguration of new Handlers into Handler Slots, Fig 1.e).

While the core provides all communication facilities, it does not deal with the actual application-level protocol processing, which is provided in the form of dedicated hardware *Handlers*. These are attached to the packet routing layer in pre-defined *Slots*, allowing them to be easily replaced (including by partial reconfiguration). The automatic creation of these Handlers as part of the honeypot application forms the main subject of this work. While some of the platform features relevant for that discussion will also be presented here, the platform architecture and capabilities are described in greater detail in [15].

The base NetStage architecture is highly portable: Initially, it has been evaluated on the BEE3 hardware platform [2], fully exploiting the four Virtex 5 devices. For the current research, it was ported to the NetFPGA 10G board [17], trading reconfigurable area for access to fast FPGA-external QDRII SRAMs to provide more context storage and timed events. Additionally, remote management of the complete system is simplified since the NetFPGA 10G card is easily plugged into a standard 2U rackmount server.

3.1 Reconfigurable Protocol Handlers

Protocol Handlers (Fig 1.e) are responsible for the actual application-level processing of network data. In the honeypot scenario, each protocol Handler emulates a certain service and/or one or more application vulnerabilities. Specifically, the Handlers react to incoming packets and generate response packets according to predefined rules that can also track per-session state. However, the handler hardware units themselves do not have a direct access to a long-term memory storing application-level session information for multiple connections. Instead, all context data is stored externally in the Global State Memory (see Section 3.3) and provided to the Handler along with the session packets. This allows multi-threaded processing in each Handler, where packets of different sessions are processed on the same hardware in an interleaved manner.

Figure 2 shows the architecture of a Handler. It consists of the actual protocol state machine, an (optional) regular expression matching engine, and an (optional) set of response packets described as stored templates. These three components need to be customized for each application, which previously required writing RTL HDL, but is now automatically performed using our new compiler (Section 5). A Slot "wrapper" acts as standardized interface, which provides buffering (implemented as ring buffer) of incoming and outgoing messages and simplifies the attachment of Handlers to the core system.

3.2 Message-based Communication

Core stages and Handlers use a message-based communication scheme for data exchange. Generally, messages encapsulate packet payloads by prefixing them with an Internal Control Header (ICH, see Figure 3). Additionally, the system uses ICH-only messages to transport control data independent of network traffic.

The routing of messages between the core and the Handlers is determined by a Packet Matching Rules Table (Fig. 1.d) that selects a target Slot considering the protocol, port, and IP address of an incoming packet. The use of netmask-based prefix matching allows to bind a Handler to an entire subnet of addresses, which is essential for the honeypot to span large address ranges. Slot lookup is efficiently im-

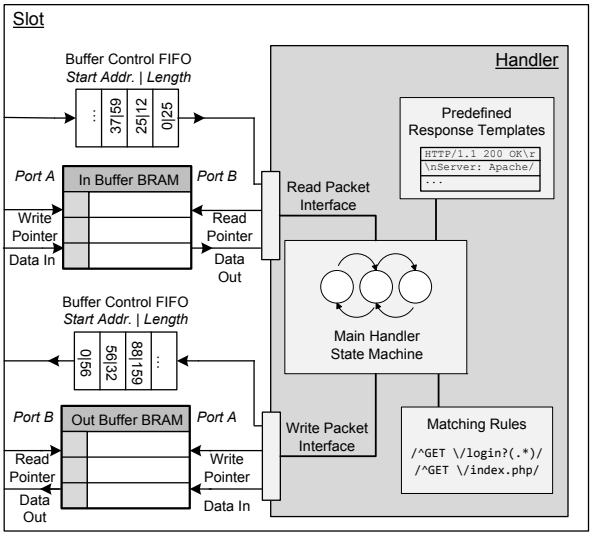

Figure 2: Slot wrapper and Handler architecture

plemented using a content-addressable memory (CAM) and fully pipelined. The rules are configured using the management interface. In contrast to [15], the routing layer has been separated from the core in the NetFPGA implementation to allow for efficient handling of more application state data without burdening the core.

3.3 Global State Memory

Instead of storing per-session state locally in the Handler, this is done centrally in the Global State Memory (Fig. 1.b). This simplifies Handler design as well as swapping in and out Handlers in a dynamic partial reconfiguration (DPR) scenario [15].

The state data is transported as part of the ICH to and from the Handlers. The routing layer manages reading and writing state data from an external QDRII SRAM. Memory addresses are given by the packet source-address / protocol / port combination (hashed together). The SWR control flag in the ICH is used to request that the ICH Application Data Region is written back to the Global State Memory. The size of the state data required for each Handler is defined at design time, stored together with the routing rules, retrieved when performing a Slot lookup, and entered in an ICH field.

Access to the state data itself is pipelined between the core and the routing layer. To further improve platform efficiency, the routing layer contains two queues: one for packets requiring state data, a second one for packets without state data. The latter can then be processed without waiting for state data to become available.

The NetFPGA 10G implementation of the platform supports up to fifteen 128b data words of per-session state (240 bytes), which is generally sufficient for our honeypot use-case to hold passwords, session IDs, or session states.

3.4 Notification Timer

Each Handler can produce response packets at least at line speed. While this is advantageous for high-speed environments, scenarios are conceivable where a client would be overwhelmed with packets at these rates, leading to packet loss or failing communication. Therefore, the platform can

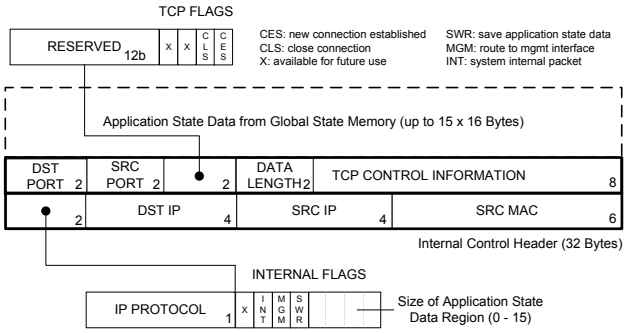

Figure 3: Internal Control Header (ICH) preceding each message

throttle the data transfer by letting the producer thread on the Handler sleep before sending the next packet.

This mode of operation is supported by the Notification Timer (Fig 1.c), which allows a connection (thread) on a Handler to sleep, freeing the Handler for the next connection, and waking up the sleeping thread after the required time (selectable from two globally configured time intervals) has passed. The thread desiring to sleep sends an appropriate control message as internal message (with the ITP flag set) holding both its internal state as well as a selector for the desired timer to the core for requesting a later wake-up. On wake-up, the Notification Timer sends an internal wake-up message to the Handler, restoring the state (context) of the original thread and allowing it to continue execution.

The same functionality is also used for implementing TCP retransmission of unacknowledged packets. Instead of storing the previously crafted response packet each time it is sent out (which would waste external memory), the Handler creates a notification packet. This allows it to rebuild the packet after a certain time period, if the packet has not been acknowledged in the meantime. For the honeypot application, a useful sleep time is $50\mu s$ for throttling and the retransmission timer is set to 200ms.

4. PROGRAMMING IN MALACODA

Many of the solutions to simplify FPGA programming (see Section 2) are achieving good results when focusing on particular problems, e.g., by introducing a Domain Specific Language (DSL). Such a DSL has advantages both for the programmer as well as the compiler. A DSL allows the programmer to describe a specific problem in his domain, while the compiler can generate highly efficient hardware circuits due to its more precise knowledge about language use and the target architecture. Furthermore, by using a DSL, characteristics of the target hardware platform (e.g., multi-threading etc.) can be reflected already in the language specification, and need not be retrofitted as pragmas or library calls.

Together, these aspects make a very strong point for using a DSL to allow network engineers to describe new Handlers in their traditional application domain. An automatic compiler can then generate high-performance hardware blocks matching the execution model of the NetStage architecture (see Section 3.1). This approach has initially been presented at the conceptual level in [14]. The resulting feedback from

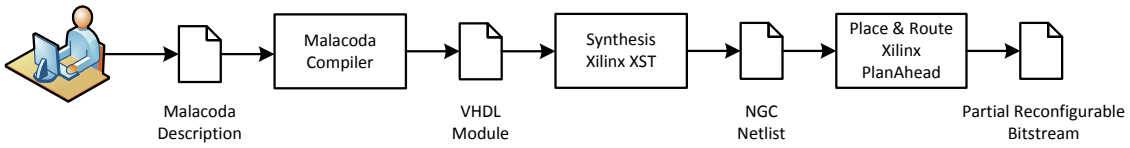

Figure 4: Tool flow for generating hardware from Malacoda programs

domain experts has since led to the creation of the Malacoda DSL and its compiler, both presented here for the first time.

4.1 Background

Service emulations in current software honeypots [6, 4] are generally described in the implementation language of the core honeypot system (commonly, script languages such as Tcl/Tk or Perl), there is no unified way of describing such emulations. Furthermore, such general-purpose programming languages are difficult to efficiently compile into hardware. On the other hand, to lower the barrier of entry for Malacoda, its syntax was inspired by Perl, which should be familiar to most network engineers and security researchers.

In contrast to a general-purpose processor, the complexity and performance of dedicated hardware is highly dependent on the current task to be executed. There are limiting factors, e.g., available FPGA resources, that often require a trade-off between resource use and performance. Based on an analysis of existing honeypot scripts and service emulations, we have designed Malacoda with a balancing of these trade-offs firmly in mind. The analysis has shown the following major operations to be essential for modeling a wide range of service Handlers for the honeypot scenario:

- Describe states and transitions that reflect the communication session.
- Evaluate the incoming request packet and craft a proper response packet by filling-in static template data and inserting parts from the original request packet, based on certain rules.
- Notify an administration station about certain protocol stages of a client conversation.

We will focus both language specification as well as compiler construction on these crucial base functionalities.

4.2 Syntax

Malacoda describes the sequential operations processed in a single NetStage thread. Parallel operations are executed by the automatic multi-threading performed by the NetStage core. Listing 1 shows a sample Malacoda description emulating a Telnet login into a shell.

A Malacoda description starts with a name, followed by an optional section to define state variables. The basic template of a Malacoda description is a protocol dialogue that contains multiple states and (conditional) transitions. A dialogue models the communication session required for emulating a certain service or vulnerability. Each state is identified by an assigned name, with **DEFAULT** indicating the initial state. That state is entered on a newly opened TCP connection or for any arriving UDP packet.

The body of a state description consists of actions (commands and assignments). An assignment to the reserved variable **state** indicates a transition to the indicated state after processing the current packet. State actions include the sending and receiving of packets, while conditional execution is expressed as **if**/**elsif**/**else** constructs. Additionally, the language supports regular expression matching using a subset of the Perl operators.

Listing 1: Sample Malacoda program

```
// Emulate login to a root shell
TELNET {
  // define variables
  dynamic username[14];
  // main fsm
  dialogue {
    // default and initial state for a new
        connection
    DEFAULT:
      addresponse("login:");
      $state = LOGIN;
    // next state
    LOGIN:
      // extract user name
      $username = chomp($INPKG);
      addresponse("password:");
      log("TELNET: Login attempt detected");
    ...
    SHELL:
      // emulate Unix uname command
      if ($INPKG =~ /^uname -a/) {
        // send the system identification
        addresponse("Linux myhost 2.6.35.6 ...");
        addresponse("\n");
        addresponse("[localhost]# ");
      }
    ...
  }
}
```

4.3 Malacoda Commands

Malacoda allows the following commands in state actions currently. *SOURCE* can be either a string (of ASCII characters or byte values, expressed by prefixing two hex digits with \\), or a variable name (reserved or user-defined). Note that a response packet may be incrementally constructed with multiple commands. It will only be sent once all actions have been processed for a state. Furthermore, all commands implicitly operate on the output buffer.

- **addresponse**(*SOURCE*): Append a byte sequence to the response packet buffer.
- **addresponse**(*SOURCE*, *s*, *n*): Copy *n* bytes starting at index *s* from *SOURCE* to the response packet.
- **addresponse**(**file**:*STRING*): Send a given byte sequence defined at compile-time in an external file (useful for larger responses that would make the Malacoda program hard to read if embedded into the Malacoda source).

- **log** (*SOURCE*): Send log packet with the given byte sequence to management interface.
- **if/elsif/else** (*expression*): Conditionally execute commands depending on the value of *expression*.
- **replace**(*s*, *SOURCE*): Replace a single byte or a byte sequence of the response packet with the value given by *SOURCE* starting at index *s*.
- **close**: Send a close connection notification with this response packet to the client (only available for TCP connections).

Beyond the special commands, the Perl command **chomp** is supported to remove any newline character from a byte string.

4.4 Expressions

Malacoda supports arithmetic, regular expression, and comparison operators in expressions. The current version of the language uses unsigned byte sequences as the fundamental data type. The reserved variable **$INPKG** indicates the entire payload of the current input packet.

Sub-ranges of a variable may be selected by the [ ] operator: $\$VARIABLE[n]$ selects an individual byte of a variable, while $\$VARIABLE[a,b]$ selects the given byte sub-sequence of a variable (from index *a* to index *b*, inclusive). Individual bits of a byte (e.g., required to set a flag in a custom application protocol), are accessed with $\$VARIABLE[n][p]$, where p is the index of the bit.

4.5 User-Defined Variables

For advanced emulations, Malacoda allows the explicit storage of per-session state in user-defined variables. These are held in the Global State Memory (see Section 3.3) for the duration of the entire client session. Variables store unstructured byte sequences that are interpreted in context of their current operator. However, they can be declared differently depending on whether they have a variable length (up to a static upper limit ≤ 255) or a fixed length (Listing 2). In the first case, an additional byte of storage is used to track the length of the variable. Longer values will simply be truncated to the maximum variable at the fixed length.

Listing 2: User-defined Variables

```
// variable with dynamic length (in bytes)
dynamic variable1[8];
// variable with fixed length (in bytes)
fixed variable2[4];
```

5. COMPILING MALACODA

The Malacoda compiler has the following design goals:

- Make the MalCoBox hardware honeypot accessible to security and network engineers without hardware design expertise.
- Enable hardware-experienced engineers to quickly generate template code for Handlers that can be later manually optimized for more complex under-the-hood operations.

The resulting compile flow is organized as shown in Figure 4 and produces synthesizable VHDL descriptions. Each

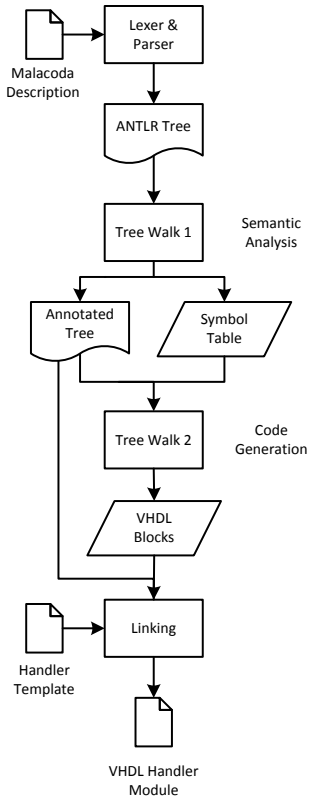

Figure 5: Malacoda compiler organization

VHDL module implements a single Handler and can be compiled into a bitstream using the standard FPGA development tools (e.g., Xilinx ISE [27]). Experienced hardware engineers can easily add custom code to the VHDL handler representation (e.g., for handling special-cases such as handler-local calculations), while non-hardware developers can rely on automatic scripts to generate the FPGA bitstream. Note that our support for dynamic partial reconfiguration on the Xilinx platform requires the use of the PlanAhead [26] floorplanning tool in addition to the usual logic synthesis and FPGA mapping steps.

5.1 Compiler Design

The construction of the Malacoda compiler is shown in Figure 5. Due to the highly specialized nature of Malacoda, much of the complexity of general-purpose high-level language compilers can be avoided. Most of the basic compiler operations are performed using Java code automatically generated by the ANTLR v3 compiler-construction tool [19]. It not only generated the lexer and parser from a formal representation of Malacoda, but also the creation of the Abstract Syntax Tree (AST) used as intermediate representation.

During the semantic analysis pass, the AST is traversed to build a symbol table of states, variables, and regular expressions. Furthermore, the pass discovers the basic blocks and their control predicates, storing this data by annotating the AST. That information is exploited in the code generation pass, which expands a pre-defined Handler template in VHDL by replacing placeholders with the actual signal declarations, output assignments etc. The template already

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

5.2 Handler Execution Model

The execution model of a compiled Handler is split into two phases: reading an input packet and writing an output packet, word by word. At run-time, the generated hardware initially evaluates all conditions in the handler in parallel by reading the entire incoming packet. This is possible, since Handler state is only updated atomically on a state transition. Malacoda assignment semantics are thus similar to the non-blocking assignments in VHDL and Verilog. After reading the entire packet, the condition results are evaluated in program order, selecting a state and predicating the execution of the state actions, which are then executed sequentially in the second phase. To reduce Handler complexity, the compiler does not yet optimize to stop reading packet data if no conditions remain that could potentially match. E.g., if a Handler checks a single condition in the first 128b of the packet, the remaining words of the packet could be skipped (if no other command requires reading them).

However, the current lack of this optimization will not lead to major slowdowns, since the majority of the processing time is often spent constructing output packets by copying the data from the Handler-internal template storage to the output queue in the second phase. In this phase, the generation of the output packet defines the state sequence. Actions are reordered to execute in the order their output occurs in the response packet. This avoids idle cycles by continually streaming data to the packet-under-construction.

If the packet-under-construction has grown to the MTU size, it is transmitted (performing a segmentation-like operation). The building of the response then continues in a new packet. Depending on the user-defined policy, this next packet is either constructed and sent out immediately, or explicitly delayed using an internal timer notification request message. For TCP connections, an internal timer notification request message is always generated to schedule a possible retransmission of the constructed packet after the appropriate time-interval. Finally, if any log packet has been assembled, it is now output to the management interface.

5.3 Regular Expression Matching

The compiler generates a dedicated matching engine for each regular expression in a Malacoda program. Currently, these engines are implemented as simple FSMs with maximally parallel comparators to ensure that a result is available immediately after the last word of a packet has been read. While this approach is feasible for the current prototype which focuses on basic character and string matching (e.g., the compiler currently does not support character classes), it would be worthwhile to integrate more refined matching architectures that are both smaller and support a larger set of regular expression operators (e.g., [5, 25]).

5.4 Packet Construction

Response packets can be generated by copying data from stored templates which are modified on-the-fly using Malacoda commands (e.g., **replace**) at run-time. The compiler can implement these template ROMs (which are also 128b wide) either as LUTs (for small templates) or BRAMs (for larger ones). Depending on the amount of template data,

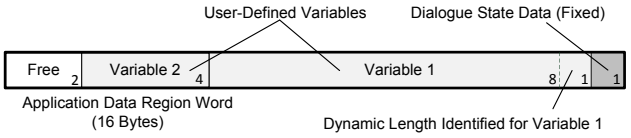

Figure 7: Variable allocation in Global State Memory

the compiler decides which model fits better. Based on the synthesis results (see Section 6.3), the upper limit for a LUT-based implementation has been set to 1 KB of static data.

For some protocols (e.g., DNS), the response packet contains much of the data received in the request packet. The compiler optimizes this special case by detecting if an **addresponse($INPKG)** command (that copies from the input buffer to the output buffer) occurs in the Malacoda program. In that case, dedicated wiring is generated to perform this copying of data in parallel hardware while the packet is read during the condition evaluation phase. Note that this operation is speculative: If a control condition would actually select a different execution path at run-time, the copied packet is removed from the output buffer simply by resetting its write pointer.

Similarly, the **replace** operations in the program are not mapped to byte-wise copy-and-select steps when reading a template. Instead, they too are turned into a dedicated wiring/logic network that modifies all bytes of a 128b template ROM data word in parallel (e.g., by permutation, insertion, deletion, replacement) to achieve a high throughput. To this end, the compiler needs to differentiate between fixed and variable length output packets, shown in Figure 6: If the length of the output packet is fixed (e.g., when always copying a fixed number of bytes from the input packet or by sending only data from a response template), the compiler can create the logic required to route data into the output packet to address fixed write offsets.

This task is more difficult for variable packet sizes (see Fig. 6-b). If data from a template should be appended to a response packet that already contains a variable-length variable, the byte offset of 128b word of template data in the output buffer depends on the number of bytes previously written to the buffer. If this case is detected at compile-time, the compiler generates a wide barrel-shifter that can move the template data to the appropriate offset within the output buffer within a single cycle. Since the barrel-shifter requires many FPGA resources, it is only created if required by the current Malacoda program.

Log packets to the management console are generated in a similar fashion, but they will always be tagged with the IP source and destination address/port information of the original packet, as well as a system-wide 128b time-stamp for better correlation of log messages with traffic dumps.

5.5 Global State Memory Allocation

The compiler also allocates proper space in the Global State Memory (see Section 3.3), both for the reserved internal (e.g., **state**), as well as for the user-defined variables (Figure 7). Dynamic variables occupy their maximum length plus a byte tracking their current length, static variables always have a fixed size. After allocating all variables, the compiler reports the number of state words required for this

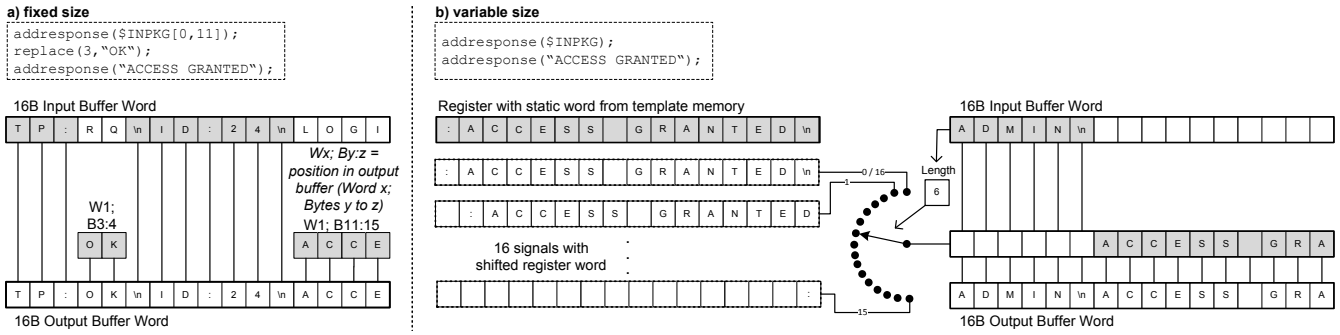

Figure 6: Construction of output packet for fixed and variable-sized response

particular Handler. This information is later required when configuring the routing rules (see Section 3.3).

The state variable is always used automatically if more than the DEFAULT state exists within a dialogue. Symbolic state names are binary-encoded into integers.

6. EXPERIMENTAL RESULTS

6.1 Hardware and Environment

The system has been implemented on the NetFPGA 10G board [17]. It uses two 10G Ethernet interfaces, a Xilinx Virtex-5 TX240 FPGA, and three 72 Mb QDRII SRAMs (holding the Global State Memory, the Notification Timer event queues, and the TCP connection state). A MicroBlaze CPU is present in the system, but only configures the Ethernet SFP+ link parameters, it does not have any contact with network data.

For the honeypot live-test, the NetFPGA card has been placed in a 2U Linux server, equipped with an additional 10G network card that connects to the management interface. For analysis, we implemented a monitoring option for NetStage that copies every packet received from the public interface to the management port. These packets are then captured on the Linux host using the standard tcpdump command. Separation between actual management traffic and this mirroring traffic is achieved by considering the different MAC addresses.

The entire project has been synthesized with Xilinx ISE / EDK 13.3 [27] and mapped with PlanAhead 13.3 [26]. We have configured our platform with six Handler Slots, each with a maximum of 6080 LUTs and 8 BRAMs of FPGA resources. The network core runs at a target frequency of 175 MHz, giving a raw internal throughput of 22.4 Gb/s.

6.2 Synthesis Results

Table 1 shows area results for the portable NetStage core and the board-specific interface and control logic. We then list the FPGA areas required when building the system for dynamic reconfiguration using six Handler Slots, or alternatively, when statically compiling our six sample Handlers (see next Section) into the design.

Note that the platform itself, including the message routing network for six Handlers, requires only 22% of the LUTs in the TX240 FPGA. However, the heavy use of buffers imposes a high demand for BRAMs. Even when reserving space for dynamic reconfiguration in the form of six Slots, the total design requires less than 50% of the TX240 LUTs,

Table 1: Synthesis results for system and components

Module	LUT	FF	BRAM
NetStage core (w/o Handler)	23,278	29,512	156
Infrastructure	9,504	11,029	28
Platform w/o Handlers	**32,782**	**40,541**	**184**
+ 6 Slots	**69,262**	**77,021**	**232**
+ 6 Handlers	**46,372**	**46,066**	**182**

leaving ample space for more or larger Handler Slots, or additional core functionality. The statically configured version does not suffer from Slot-internal fragmentation and is even smaller, but no longer has the self-adaptation capability using DPR.

6.3 Handlers

For our evaluation, we have selected six different Handlers for the emulation of typical network services or actual vulnerabilities:

- Web server: Imitates a webmail service running on a vulnerable web server (identified by a corresponding version header), collecting information about web server attacks.
- Telnet: Emulates a faux system administration CLI accepting any login / password to gather data about what combinations attackers try and commands being executed after login.
- Mail server: Pretends to be an open relay simply accepting every mail (SMTP protocol) to gather information about spam attempts.
- MSSQL Slammer detection: Responds to MSSQL Ping and detects a malicious packet as sent by the Slammer [10] worm.
- SMB login detection: Emulates the first steps of the protocol until client login. Used to gather information about attack attempts on the SMB service.
- DNS server: Emulates a DNS server that resolves a single domain. Used to collect information about DNS attacks.

Table 2: Synthesis results for compiled Handler modules

Handler	Opt.	LUT	FF	BRAM	Max Freq. MHz
SMB	LUT	3,383	1,624	0	185
DNS	LUT	2,864	1,447	0	223
MSSQL	LUT	1,894	1,288	0	212
Telnet	LUT	3,921	1,643	0	175
Mail	LUT	2,460	1,543	0	193
Web	BRAM	2,285	1,355	4	203
Mail	BRAM	2,432	1,584	4	183
Web	LUT	5,796	1,346	0	193

Each Handler has been programmed in Malacoda and compiled into hardware using the developed compiler. Table 2 shows the required resources for each of them. For Mail and Web, we also list alternate results when manually choosing a different LUT / BRAM implementation option for the Template ROMs (see Section 5.4).

Compared to the other Handlers, the MSSQL emulation requires only few LUTs. The size of response templates has a major impact on the Handler size. E.g., the Telnet Handler has 570 B of replies in 18 templates, while the MSSQL Handler has only one, since it just detects an attack and logs its occurrence. Furthermore, the Web server Handler demonstrates that implementing large portions of response templates (in that case, 7 KB) in BRAM instead of LUTs has a significant advantage in terms of resource usage. In terms of code complexity, the Malacoda Handler descriptions have 16 . . . 80 lines of code, while the resulting VHDL modules have 625 . . . 2220.

To evaluate the efficiency of the compiler, we compare an automatically compiled Handler to a manually developed one. We cannot use the previous Handlers originally developed for [15], since the new features introduced with the NetFPGA port cause changes in the Handler-internal structure. Thus, we created a special Malacoda description for the Web (HTTP) Handler of [15] and stripped from the compiled VHDL code the functionality specifically required for the NetFPGA version of NetStage, thus leaving a version comparable to the original one (which targeted the BEE3 platform). This compiled version of the Web Handler is slightly larger (1,570 LUTs, 665 FFs) compared to the original manual implementation (1,026 LUTs and 586 FFs), but both achieve nearly the same performance when creating response packets (the compiled version needs two additional cycles). The overhead in LUTs is due to a more complex regular expression matching implementation and additional logic in the generic implementation of output packet generation. Since the compiler has been optimized for generating high-performance hardware, the increased use of resources is acceptable here, since the Handlers are still relatively small compared to the overall FPGA capacity.

6.3.1 Performance

The latency of an individual Handler consists of a fixed number of clock cycles for administrative functions (processing header data, register notifications), and a variable number of clock cycles depending on the size of the packet for content-related activities (as the compiler generates hardware that processes an entire 16 B input / output word in one clock cycle). Table 3 shows performance data for example packets processed by the Web, Mail and DNS Handler. Note that these latency limits are maintained up to the throughput limit, as the Handlers are implemented using dedicated (non-shared) resources that do not depend on the system load. The maximum throughput can be calculated from the current core target frequency of 175 MHz.

Table 3: Latency for selected operations

Handler	Operation	Latency [Cycles]
Web	120 B Request	$8 + \lceil 120\text{ B}/16\text{ B} \rceil$
	1024 B Response	$11 + \lceil 1024\text{ B}/16\text{ B} \rceil$
Mail	14 B Request	$9 + \lceil 14\text{ B}/16\text{ B} \rceil$
	16 B Response	$8 + \lceil 16\text{ B}/16\text{ B} \rceil$
DNS	33 B Request	$8 + \lceil 33\text{ B}/16\text{ B} \rceil$
	99 B Response	$5 + \lceil 99\text{ B}/16\text{ B} \rceil$

Here, the Web Handler achieves a raw data throughput of 14.5 Gb/s including administrative messages (but excluding packet header processed by the core), which would be sufficient to saturate the 10G link. Administrative messages (e.g., TCP retransmission notification) are transmitted using the same channel as network data, thus reducing the core network throughput for outgoing packets. Handlers which generate a smaller volume of output data are affected more by the administrative overhead: At 175 MHz, the Mail Handler could reply to SMTP HELO messages with an external throughput of only 5 Gb/s (including the 54 B of protocol headers added by the core). This could be improved by an additional pipeline stage inside the Handler (leading to a throughput of 11 Gb/s in the Mail example). Note that the Mail Handler requires an extra cycle of latency for the administrative operation of accessing the global application state memory, but completely avoids TCP throttling notifications, as current response packets always fit in one network packet.

The UDP-based DNS service does not need any notification messages at all. Therefore, the number of fixed clock cycles for response generation is further reduced. In this example, the DNS Handler achieves an external throughput (including the 42 B of protocol headers) of 10.6 Gb/s.

6.4 Live Test

For the live test, the NetFPGA 10G card has been connected to a 10G data center uplink at a major German university, with two dedicated /25 subnets (= 256 IPs) assigned to the honeypot. The public network traffic for the honeypot was dumped for later analysis on the management server. The Handlers were configured to listen on all IP addresses and the test was run for one month. During that time, 1.74 Million connection requests were reaching the honeypot. Table 4 lists the Top-10 services requested, as well as numbers for the remaining services for which the honeypot has active Handlers (active Handlers are shaded gray).

With a connection rate of more than 50%, the Microsoft SMB protocol is leading the list. This is unsurprising, since due to its widespread use and various known vulnerabilities, SMB is a promising target for attackers. In total, our four

Table 4: Number of connections by service

Nr.	# Conn.	Port	Service
1.	977,549	445/TCP	MS-DS (SMB)
2.	167,430	80/TCP	HTTP
3.	82,882	139/TCP	NETBIOS Session
4.	36,167	3389/TCP	MS WBT Server
5.	31,093	1433/TCP	MS SQL Server
6.	30,966	8080/TCP	HTTP Alternate
7.	27,063	22/TCP	SSH
8.	20,118	23/TCP	Telnet
9.	15,618	210/TCP	Z39.50
10.	13,627	25/TCP	SMTP
44.	1838	1434/UDP	MS SQL Monitor
189.	243	53/UDP	DNS

Table 5: Number of monitoring events

Event	# Occurrences
Webserver: GET URL	118,384
MSSQL: Slammer Worm	1,588
SMB: Login Attempt	24,566
Mailserver: Mail Queued	3,778
Telnet: Login	11,438

TCP-based Handlers are among the Top-10, such that the honeypot had a good coverage of network traffic.

In addition to counting the raw connections, we also implemented monitoring points inside the Handlers, using the Malacoda **log** command to log when a certain step has been reached. The occurrence of these events is given in Table 5. While some portion of the connections from Table 4 was coming from simple portscans, many of the clients actually interacted with our honeypot. We discuss these results in the following subsections.

6.4.1 Web Server

Around 70% of the clients were requesting a particular URL. The majority of these were attempts to reach a vulnerable service (e.g., phpMyAdmin). On the other hand, we did not observe clients trying to log into the webmail facade served by the Handler. It appears that automatic attack tools do not try to take advantage of a login form, and that human attackers did not interact with this part of the honeypot.

In addition to the HTTP web service, we also observed many TCP SYN ACK requests (362,381) hitting our system. These packets were not initiated by our honeypot (since we never sent out SYN request), but instead originate from SYN flooding attacks to real web servers by attackers using spoofed IP addresses belonging to our darknet. This is a well known procedure; commercial service providers exist that detect DDoS attacks by monitoring such darknets using distributed sensors [1].

6.4.2 Mail Server

Around 25% of the connecting clients actually tried to send a mail. The contents of these mails appear to be initial identification messages from spam engines, checking whether a server that looks like an open relay actually does deliver the mails. The mails were addressed to cryptic recipient addresses hosted at public webmail services. However, we refrained from actually delivering these messages to avoid impacting the owner institution of our darknet IP range. In a later test, these probe mails could be selectively forwarded to induce the attackers to send real spam to the honeypot, and allow it to collect any attached malware.

6.4.3 DNS Server

Attackers only had limited interest in the DNS server emulation. All requests indicate coming from automatic vulnerability scanners and simply request arbitrary domain names. However, we observed spoofed response packets similar to those hitting the web server. We received 13,804 DNS responses from real DNS servers, despite our system never requesting a lookup.

6.4.4 MSSQL Slammer Detection

Even nine years after the large outbreak and the massive effort to remove the worm from infected systems, we count 20-40 Slammer requests per day trying to infect the honeypot. They originate from other infected systems or automatic scripts all over the world (60% from China, 19% from India and 5% from the United States). This shows the difficulty of eradicating a worm such as Slammer once it has been released on a large scale.

6.4.5 Telnet Shell Emulation

More than 50% of all connecting clients tried to log in. The majority used the username / password combination root / admin or simply root with no password. The commands that were executed after the client has logged in were, e.g., `echo test` or `echo connectioncheck`, most likely coming from automatic scripts looking for open Telnet servers.

6.4.6 SMB Login Detection

This Handler was the most frequently accessed service of our honeypot setup. While the majority of requests were simple port scans, some of the clients were actually following the protocol interaction and tried an anonymous login without password.

We did not perform more detailed analysis, as the SMB protocol with its many variable field lengths and partial encryption is not handled very efficiently by the current Malacoda compiler prototype. These limitations will be addressed in future revision of the system.

6.5 Summary

Summing up, the results from the compiler and the live evaluation clearly show the feasibility of the described architecture. The hardware honeypot can be operated just as any low-interaction software honeypot, collecting data unattended for long time periods, but without the risk of the system becoming compromised. The implementation on the NetFPGA 10G has proven its stability and the Internet protocol implementation of the NetStage communication core demonstrated its ability to establish communication sessions with many different clients on the Internet. Due to the simplified programming interface offered by Malacoda, the system has a high potential in research, education, and production environments. The compiled Handlers have a similar performance to manually optimized ones, but a significantly reduced development effort.

7. CONCLUSION AND FUTURE WORK

With NetStage, we have already demonstrated the high potential of reconfigurable computing beyond the commonly used switching, routing, and deep packet-inspection applications. Using MalCoBox, our honeypot-in-a-box, as a demonstrator, we exploit hardware-accelerated operations not only for higher performance, but also for hardened security.

This work has begun to address a common problem limiting the use of reconfigurable technology for data processing purposes, namely the lack of high-level design tools. We have approached this issue by defining Malacoda, a domain-specific language focused on network processing, specifically for active security applications such as honeypots. It allows networking experts not proficient in hardware design to easily and concisely describe the protocol interactions typical for the emulated network services of a honeypot.

The associated compiler, even though it is just a prototype, has already succeeded in compiling Malacoda programs into high-performance Handlers for execution on the platform, fully exploiting its capabilities for multi-threading, parallel execution, and deep pipelining. The evaluation of the actual implementation on the NetFPGA 10G platform and the long-term live test not only demonstrate the practical feasibility of the developed architecture, but also show directions for future research.

The focus will be on improving the compiler, e.g., by integrating more efficient regular expression matching hardware, better optimization during condition evaluation, and optional optimization for single connection throughput by deeper pipelining. Beyond these issues, support for cryptographic operations, more arithmetic and logic computations, as well as the ability to seamlessly access manually designed IP blocks from Malacoda code, are already planned.

8. ACKNOWLEDGMENTS

This work was support by CASED (www.cased.de) and Xilinx, Inc.

9. REFERENCES

[1] ARBOR Networks. Active Threat Level Analysis System (ATLAS). Available online at: atlas.arbor.net.

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

[3] G. Brebner. Packets everywhere: The great opportunity for field programmable technology. *Proc. Intl. Conf. on Field Programmable Technology*, pages 1–10, 2009.

[4] Dionaea. Available online at: dionaea.carnivore.it.

[5] T. Ganegedara, Y.-H. E. Yang, and V. K. Prasanna. Automation Framework for Large-Scale Regular Expression Matching on FPGA. In *Proc. Intl Conf. Field Programmable Logic and Applications*, pages 50–55, 2010.

[6] Honeyd. Available online at: www.honeyd.org.

[7] T. Katashita, Y. Yamaguchi, A. Maeda, and K. Toda. FPGA-Based Intrusion Detection System for 10 Gigabit Ethernet. *IEICE Trans. Information and Systems*, E90-D:1923–1931, 2007.

[8] E. Kohler, R. Morris, B. Chen, et al. The Click modular router. *ACM Trans. Computer Systems*, 18:263–297, 2000.

[9] M. Labrecque, J. G. Steffan, G. Salmon, et al. NetThreads: Programming NetFPGA with Threaded Software. In *Proc. NetFPGA Developers Workshop*, 2009.

[10] D. Litchfield. Microsoft SQL Server 2000 Unauthenticated System Compromise. Available online at: http://marc.info/?l=bugtraq&m=102760196931518.

[11] J. Lockwood, N. McKeown, G. Watson, et al. NetFPGA - An Open Platform for Gigabit-Rate Network Switching and Routing. In *Proc. Intl. Conf. Microelectronic Systems Education*, pages 160 –161, 2007.

[12] Mentor Graphics. Catapult C. Available online at: www.mentor.com.

[13] S. Mühlbach, M. Brunner, C. Roblee, and A. Koch. MalCoBox: Designing a 10 Gb/s Malware Collection Honeypot using Reconfigurable Technology. In *Proc. 20th Intl. Conf. on Field Programmable Logic and Applications*, pages 592–595, 2010.

[14] S. Mühlbach and A. Koch. A novel network platform for secure and efficient malware collection based on reconfigurable hardware logic. In *Proc. 2011 World Congress on Internet Security*, pages 9–14, 2011.

[15] S. Mühlbach and A. Koch. NetStage/DPR: A Self-adaptable FPGA Platform for Application-Level Network Security. In *Proc. 7th Intl. Symposium on Reconfigurable Computing: Architectures, Tools and Applications*, pages 328–339, 2011.

[16] Mykonos. Mykonos Web Security. Available online at: www.mykonossoftware.com.

[17] NetFPGA 10G. Available online at: www.netfpga.org.

[18] NetLogic Microsystems. NETL7 Layer 7 knowledge-based processor. Available online at: www.netlogicmicro.com.

[19] T. Parr. ANTLR Parser Generator v3, 2008. Available online at: http://www.antlr.org/.

[20] V. Pejovic, I. Kovacevic, S. Bojanic, et al. Migrating a Honeypot to Hardware. In *Proc. Intl. Conf. on Emerging Security Information, Systems, and Technologies*, pages 151–156, 2007.

[21] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proc. 13th USENIX Conf. System Administration*, LISA '99, pages 229–238, 1999.

[22] 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*, pages 36:1–36:10, 2010.

[23] Synopsys. Synphony C. Available online at: synopsys.com.

[24] Tilera. TILE64 processor. Available online at: tilera.com.

[25] H. Wang, S. Pu, et al. A modular NFA architecture for regular expression matching. *Proc. Intl. Symposium on Field Programmable Gate Arrays*, pages 209–218, 2010.

[26] Xilinx, Inc. *UG632: PlanAhead User Guide v. 13.3*, 2011.

[27] Xilinx, Inc. *UG681: ISE Design Suite Software Manuals and Help v. 13.3*, 2011.