Accelerating FPGA Assurance Analysis via Channel Slicing

Stephen Baka, John Hallman
Secure Computing & Communications Division
MacAulay-Brown, Inc., an Alion company
Roanoke, VA, USA
fpga@macb.com

***Abstract*—FPGAs excel at parallelization and are widely used in applications that process high volumes of streamed data. Such designs duplicate their processing functions across multiple channels. A newly developed algorithm slices FPGA firmware along channel boundaries and maps the components between them. This technology simplifies trust and assurance analysis by enabling formal equivalence checking between channels.**

Keywords—FPGA; channel; vetting; trust; assurance; DARPA; formal equivalence checking; firmware; mapping; slicing; NetFPGA; OpenFlow; router; switch

I. Introduction

Field programmable gate array (FPGA) devices are valued for their ability to process large volumes of data with a high degree of parallelism. Modern FPGAs [1] [2] include dozens of high-speed serial transceivers, each capable of multiple gigabits-per-second (Gbps) speeds, enabling device-level processing speeds in the terabit-per-second (Tbps) range. Many communication designs, such as routers and switches, demand packet-processing functions duplicated across several channels where each channel services network traffic on a given Ethernet port operating at speeds up to 100 Gbps.

This paper presents an automated method (*channelize*) for identifying the boundaries of such channels in an FPGA netlist and slicing (i.e., extracting their sub-circuits) them from the firmware for analysis. Analysts provide the method with start points for each channel in the design. The tool slices channels in the user-specified direction (default is both) and maps components between them. The channel mappings can then guide a formal equivalence checker (FEC) to mathematically prove or disprove the logical equivalence of the channels.

II. Background

The DARPA Vetting Commodity IT Software and Firmware (VET) program [3] developed automated methods for vetting commercial off-the-shelf (COTS) commodity information technology (IT) devices for vulnerabilities and determining their risk of use in DoD deployment. As part of this program, MacAulay-Brown (MacB) developed new techniques [4] [5] [6] for vetting FPGA firmware found in commodity IT devices for susceptibility to malicious incursion. The NetFPGA 10G board [7] - a sandbox network card with a Xilinx Virtex-5 TX240T FPGA, four 10GigE SFP+ ports, and a Gen2 x8 PCIe host connection - configured as a generic router (Fig. 1) served as our exemplar device for the program.

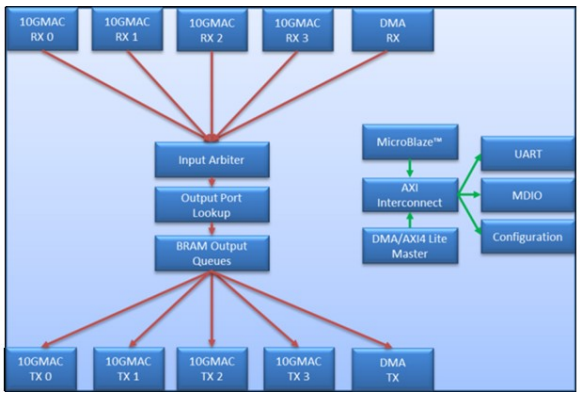

Fig. 1. NetFPGA10G Generic Router Design. Drawing from [8]

Given the IT device selected, as well as the popularity of parallel processing architectures in FPGA designs, it became clear that we needed an automated method for slicing large parallel processing regions from a segment of FPGA firmware. During the VET program, we created the *channelize* tool to address this goal. We hypothesized that if we could slice channels in a parallel system with sufficient fidelity, then we could simplify the vetting process in two significant ways:

Benefit 1 - By comparing channels expected to be equivalent in a design, we quickly identify potential exposures to malice through the differences in the circuit function. We assume that any channel significantly different from the others, may indicate such an exposure.

Benefit 2 - Similarly, if the compared channels were shown to be identical, then the analyst would need to vet just one of the channels to eliminate or prove exposure to malice across all channels, thus shortening the time required to vet this logic.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and SPAWAR Systems Center Pacific (SSC Pacific) under Contract No. N66001-13-C-4045. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

III. METHOD

In this section, we describe 1) the original (greedy) approach used by our channelization method, 2) the impact of mapping effort on its slicing efficacy, and 3) the final approach used for channelization.

A. Original Channelization Approach

Initially, *channelize* used a greedy tracing algorithm (Fig. 2) to slice channels. That is, given two or more user-identified channel starting points, *channelize* would trace forward and backward from each start point, "coloring" each channel until it intersected one of the others. This tended to produce large channel slices since it ignored the underlying logic equations and structure of the channels. Schematic or code comparisons of these larger sliced channels were often more difficult for human analysts to vet. In addition, the lack of mapped I/O or states made FEC untenable for comparing all but the smallest channel slices.

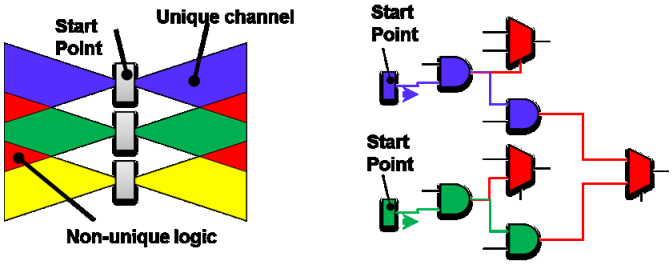

Fig. 2. Greedy channelization (mapping strictness = 0) slices circuits until they intersect (red regions); non-red regions represent unique channels

In order to address these issues, one potential solution adds a mechanism to the algorithm for reporting static characteristics across each channel. Several characteristics were investigated for this purpose, including:

a) Number of inputs
b) Number of outputs
c) Number of sequential components
d) Number of combinational components
e) Number of macro components (e.g., BRAMs)
f) Total number of components
g) Number of individual nets (net bits)
h) Number of multi-bit nets

Given that each channel could synthesize differently, even using identical source code, it was not enough to compare these characteristics directly. Over the course of several experiments, using identical processing engines instantiated across multiple channels, we observed that any one of these characteristics could vary by small amounts (up to 1%).

In the end, we selected five characteristics for comparing unmapped channel slices. If any one of these varied by more than ±5% from the average across four or more channels, our analysts would focus additional effort on identifying and understanding differences in the outlying channel:

a) Number of channel inputs
b) Number of channel outputs
c) Total number of components per channel
d) Total number of nets per channel
e) Number of decoders detected [6] *in each channel*

Note that a goal of DARPA VET was to rapidly demonstrate or rule out a device's exposure to malice. Given this constraint, we chose to de-emphasize channel level analysis when these characteristics fell within expected tolerances. Although this methodology proved useful in addressing vetting speed, it had the potential to miss subtle exposures to malice.

B. Mapping Effort Levels

To increase our confidence in sliced channel comparisons, we needed a means for mapping and comparing channels. We experimented with several channel-mapping techniques implemented as strictness levels (Table 1), that would force *channelize* to color similar components between channels as it traced out from the start points. For each mapping technique, a component was only included in a given channel if it:

a) was not in another channel's fan cone *AND*
b) mapped to a component in at least one other channel

When a single channel diverged from the others, its slicing stopped. When multiple channels diverged from the others, but remained mapped to one another, the channels were split into sets and each set continued slicing and mapping separately.

Table 1. Mapping techniques planned for channelize method.

Strictness	Technique	Metric
0	greedy	***Ignore mapping***; trace until channels intersect
1	similar	***Circuit regions*** are heuristically similar
2	equivalent	***Circuit regions*** must be logically equivalent
3	one-to-one	***Components*** must be logically equivalent
4	exact	***Components*** must be exactly the same

The type of mapping used had a dramatic effect on the depth to which channels were sliced. For example, if the algorithm was told to exactly map components (strictness = 4) across the channels, the cuts often end up too shallow to realize either of the two aforementioned benefits (§II. Background). This is not surprising given the number of optimizations that FPGA synthesis tools perform in order to meet a system's targeted timing, area, and power goals. Thus, even if all channels in a multi-channel system are logically equivalent, they can differ in structure significantly.

To address the shallow-slicing issue, a less rigid mapping method (e.g., one-to-one, strictness = 3) was added to the algorithm, so that some of this channel implementation variation might be overcome at the component level. For example, a LUT with the equation `A & B & C` (i.e., a `LUT3` component with `INIT` value of `8'h80`) is equivalent to 3-input AND gate driven by A, B, and C. Similarly, two flop types, one with a reset port and one without, could be equivalent, assuming the former has its reset port tied to a benign value and both flops share the same power-up value. This one-to-one mapping technique enabled *channelize* to slice more deeply into each channel, thus providing more points of controllability and observability.

C. Final Channelization Approach

By the end of the DARPA VET program we were utilizing a combination of *channelize* mapping methods (§III.B) and static channel characteristics (§III.A) to identify exposures to malice. Once the starting points (often FPGA pins) for each channel in a design were identified, *channelize* would use its greedy algorithm to bound the state space for each channel and produce a set of static characteristics for comparison. These channels were cut once for both packet processing directions (i.e., transmit and receive).

This first channelization step not only provided analysts with guidance concerning the viability of using channel comparisons to vet for malice, but also a means for identifying other major functions in the network device. For example, if each channel ended in a BRAM component, we could hypothesize that these were the Ethernet MAC's Rx FIFOs and that the logic immediately following it was likely the device's Rx Arbiter. Similarly, in the transmit direction the inputs to each sliced channel likely delineated their boundary with the Output Port Lookup function, which was responsible for IP-level packet processing.

With the greedy channels cut, we would then rerun *channelize* in one-to-one mapping mode to establish a useful set of mapped inputs, outputs, and states. These mappings provided known points of controllability and observability to FEC tools when comparing channels. This pass of the tool also established the "regions of uncertainty" for each channel. In other words, the set of components belonging to each greedy channel slice less their one-to-one mapped component sets, where other VET-developed methods [3] were applied for identifying exposures to malice.

IV. Evaluation and Results

Our *channelize* method was tested over the course of six "engagements" on the DARPA VET program that were designed to evaluate our vetting approaches. An independent adversary team (§Acknowledgements) created up to three FPGA designs per evaluation period, targeting the Xilinx Virtex-5 TX240T, to test our ability to both detect Trojans and avoid false alarms. All evaluations included a "deployment scenario" that defined how the commodity IT device was to be used in the field (e.g., "port 0 is connected to the Internet").

In all scenarios, the commodity IT device (i.e., NetFPGA-10G card) was assumed to be plugged into the PCIe slot of a trusted host computer running in a physically secure facility. In other words, an adversary could only access the installed cards remotely. It was assumed that the same adversary had means to apply their own bitstreams to the commodity IT device prior to its installation, such as through an imagined product patch.

In this paper, we will discuss the contributions of our *channelize* method to four of our six evaluation periods (i.e., "engagements") and not our programmatic results as a whole.

A. First Evaluation

In our first real use of the *channelize* algorithm, we were tasked with evaluating three finite impulse response (FIR) filter designs that included roughly three-, eight-, and ninety-four-thousand circuit components, respectively. Design 1 was a bank of 16 filters with 64 coefficient taps and 18-bit precision and shared input and output interfaces. Design 2, a somewhat larger bank of 38 filters with 59 taps and 22-bit precision, included input/output FIFOs for high-speed data streaming. Finally, Design 3 was significantly different from the other two designs in that it was a fully parallel, 6-channel, 22-tap, direct transpose implementation, with a password authentication mechanism.

This early version of *channelize* only included the greedy (i.e., non-mapped) mode of operation. During the evaluation, only Design 1 was successfully sliced and produced fourteen channels of 86 components and two channels of 137 (Channels 11 and 13). Although none of the channel slices resulted in the discovery of exposure to malice, we did unknowingly uncover some intended yet non-functional malice (*Malice 1*) on channels 11 and 13 where the Red Team mistakenly lost a portion of their modifications during synthesis. Channelization did not catch a second piece of malice (*Malice 2*) since the modification had no effect on the number of channel components, inputs, outputs, or nets. Design 2 channel slicing failed due to a bug in the code. For Design 3 we did not run *channelize* and instead chose to focus our vetting efforts on the shared authentication circuitry.

Malice 1: *Swap adjacent FIR outputs.* This malice intended to swap two multiplier output bits from FIR11 with the two from FIR13. The Red Team mistakenly lost the malice during synthesis, resulting in benign circuits that differed in structure from the other fourteen channels.

Malice 2: *Feedback from output MUX to FIR input.* This malice summed the output from the shared filter bank MUX back into to the FIR6 input. This results in incorrect output and a longer response path whenever FIR6 was selected, and required twice as much time to completely flush from FIR6.

B. Second Evaluation

Our next evaluation consisted of a single NetFPGA 10G generic IP router [8] design containing over 140,000 circuit components. This design presented a challenge for our channelization algorithm in the form of several large multi-gigabit transceiver components. These extremely large components include hundreds of I/O ports and are often interconnected to form wider data paths. In this case the design included four 2-bit transceivers per channel for each direction (transmit and receive). Structurally, this meant that all channels intersected almost immediately and no useful channels were sliced.

Table 2. Second evaluation's comparison of *channelize* results highlights a possible exposure to malice in Channel 0's receiver.

Channel	Components	Inputs	Outputs	Nets
Rx<3:1>	3729	212	210	4656
Rx<0>	*3692*	***195(-6%)***	***154 (-21%)***	*4568*
Tx<3:0>	3275	197	16	3641

After the evaluation we developed an automated means of subdividing the transceiver components into separate transmit, receive, configuration, test, and clocking blocks. This allowed channelize to produce sizable channels (Table 2). Note that each Ethernet channel was independently sliced in both the transmit and receive directions due to their intersection in the network layer logic (Fig. 1). This post-mortem highlighted a clear structural difference between the Channel 0 GigE receiver

and those for Channels 1:3. Red team post-evaluation reports indicate that of the nine instances of malice placed in the router, just one (*Malice 3*) targeted the Channel 0 receiver and none directly targeted the transmitters.

Malice 3: *Time Bomb Resets Input Queue.* This malice fires at regular intervals (roughly every 7 trillion clock cycles, or every 21 hours at 160MHz) and resets the receive queue associated with port 0. It was implemented with a residue number system (RNS) ring counter utilizing just 12 LUTs configured as shift registers and the circuit's only input was a clock.

C. Third Evaluation

Our next evaluation period included three FPGA designs that implemented a network switch (Fig. 3) [9] [10] [11] based on the OpenFlow Switch Specification [12]. We received a trusted variant of the design (Table 3), to work out any performance issues with our VET-developed tools, followed by three untrusted variants of the switch.

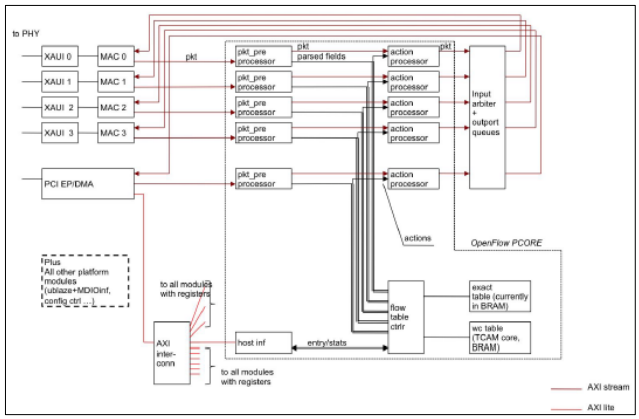

Fig. 3. NetFPGA-10G OpenFlow Switch was used as an exemplar COTS device for the third and fourth evaluations. Drawing from [11]

Per the "engagement" rules of the program, we could not use the trusted bitstream as a reference for comparison nor were we allowed to compare the three untrusted designs for differences. Our "deployment scenario" stated that the switches would be configured with Port 3 connected to the Internet and Ports 2:0 only ever connected to trusted networks.

Table 3. Channelization results for a trusted OpenFlow Switch.

Channel	Components	Inputs	Outputs	Nets
Rx<3:0>	4808	246	466	6128-6148
Tx<3:0>	3294	197	16	3660-3663

Of the three untrusted OpenFlow Switches, the *channelize* method only pointed to possible malice in Design 2 (Table 4). This design included two malicious circuits (*Malice 4* and *Malice 5*) targeting the Channel 3 GigE receiver logic. We discovered both of these circuits through multiple techniques, including *channelize*. In this case, the static channel data, and stated deployment scenario, focused our vetting efforts on the Channel 3 receiver and accelerated the time of discovery. Design 2 contained three additional malicious circuits that were not discoverable through channelization techniques.

Table 4. Third evaluation's comparison of *channelize* results highlights a possible exposure to malice on Channel 3 of OpenFlow Switch Design 2.

Channel	Components	Inputs	Outputs	Nets
Rx<3>	***5422 (+9%)***	*246*	***1020 (+69%)***	***6742 (+7%)***
Rx<2:0>	4801	246	465	6122-6126
Tx<3:0>	3275	197	16	3641

Malice 4: *Network-based OpenFlow rule updates.* This malice monitored packets received on port 3 for hexadecimal value 0x4d414c4943452036 aligned on a 64-bit boundary. This triggered a modification of the OpenFlow routing tables using the next 640-bits of packet data, bypassing the trusted means of updating routing tables via host.

Malice 5: *Forward dropped packets from port 3 to port 0.* After approximately 21 hours, all received packets received and dropped on port 3 were forwarded to port 0 instead.

The other two designs had fairly balanced channel slices (Table 5) in both directions and did not suggest any exposure to malice in their GigE packet processing logic. Design 1 contained four items of malice. Design 3 contained one malicious modification and one chaff circuit meant to distract our circuit analysis. Of these, only one piece of malice (*Malice 6*) could have been discovered via channelization and wasn't. The reason for this is not entirely clear, but may have been related shared logic interconnecting the four GigE receivers and the DMA engine. This would result in an intersection of channels and prevented the algorithm from further slicing prior to the malice.

Table 5. Third evaluation's comparison of *channelize* results suggests an absence of malice in GigE regions of OpenFlow Switch Designs 1 & 3.

Design	Channel	Components	Inputs	Outputs	Nets
D1	Rx<3:0>	4808	246	466	6128-6152
D1	Tx<3:0>	3294	197	16	3660-3670
D3	Rx<3:0>	4801	246	465	6121
D3	Tx<3:0>	3275	197	16	3641-3643

Malice 6: *Special packet clears output FIFO.* This malice monitored packets received on port 3 for a sequence of thirty-one consecutive 77-byte packets. Once triggered, any pending host-bound packets were flushed from the DMA (PCIe) queues.

D. Fourth Evaluation

Our next evaluation period again tasked us with evaluating three OpenFlow Switch designs, but each included a simple payload encryption function for protecting trusted traffic forwarded to untrusted networks. A set of host-accessible configuration registers determined which Ethernet ports were connected to trusted networks. The deployment scenario stated that software running on the trusted host computer would configure ports 0 and 1 as trusted and ports 2 and 3 to the (untrusted) Internet.

Channelization of Design 1 indicated a structural difference on Port 3 in both directions (Table 6); specifically the number of inputs to the channel slices. The design contained two items of malice, the first of which (*Malice 7*) only applied to the IP portion of the packet. Because the IP traffic is handled by a routing table shared across the channels, however, channelization stopped short of slicing this malice. The second

malice (*Malice 8*) was successfully sliced via channelization. The trigger for this malice resided in Channel 3's receiver, while its effects applied the transmitters for Channels 2 and 3.

Table 6. Fourth evaluation's comparison of *channelize* results highlights a possible exposure to malice on Channel 3 of OpenFlow Switch Design 1.

Channel	Components	Inputs	Outputs	Nets	Decodes
Rx<3>	*5063*	***211 (+6%)***	*400*	*6376*	*547*
Rx<2:0>	5063-5064	195	400	6352-6353	547
Tx<3>	*3553*	***142 (+6%)***	*11*	*3947*	*611*
Tx<2:0>	3542	131	11	3907-3908	611

Malice 7: *Apply OpenFlow rules that rely on source port ID to the wrong packets*. This malice triggered on a special sequence of IP packet headers received on any port. Once triggered it caused traffic received on port 2 or 3 (Internet) to be treated as if they were received on port 0 or 1 (trusted).

Malice 8: *Disable encryption of data exiting ports 2 and 3*. This malice triggered when 101 packets received on port 3 that were forwarded to the host (PCIe) for processing. It then disabled payload encryption in transmitters for ports 2 and 3.

Design 2's channel slices were all nearly identical (Table 7) and suggested no malice existed in the device's GigE logic. We learned after the completion of the evaluation that Design 2 was completely free of malice, included extra benign logic, and was engineered to test our false-positive reporting rate.

Table 7. Fourth evaluation's comparison of channelize results suggests an absence of malice in GigE regions OpenFlow Switch Design 2.

Channel	Components	Inputs	Outputs	Nets	Decodes
Rx<3:0>	4859-4863	193	297-309	6487-6544	442-443
Tx<3:0>	3542	131	11	3907-3925	611

Finally, Design 3 channelization analysis (Table 8) pointed at large structural differences in port 2's GigE receiver logic. Port 3's receiver was also slightly larger than average, but not enough to meet our threshold (5% delta in one or more channel characteristics) for increasing its analysis priority. Early analysis of port 2's receiver logic uncovered malice that allowed an adversary to write arbitrary rules directly to the OpenFlow tables (*Malice 9*).

Additional analysis uncovered a second item of malice in the port 2 receiver that allowed an adversary to update the list of ports a given packet could be forwarded to (*Malice 10*). A deeper investigation indicated that this second Trojan was also triggerable via traffic received on port 3. This result is interesting because it explains the port 3 receiver's slightly larger size (as compared to ports 0 and 1).

Table 8. Fourth evaluation's comparison of *channelize* results highlights a possible exposure to malice on Channel 2 of OpenFlow Switch Design 3.

Channel	Components	Inputs	Outputs	Nets	Decodes
Rx<3>	5151	196	402	6548	566
Rx<2>	***6059 (+14%)***	*199*	***447 (+8%)***	***7462 (+12%)***	*568*
Rx<1:0>	5064	195	402	6352	547
Tx<3:0>	3542	131	11	3907	611

Malice 9: *Create an arbitrary rule from packet data*. If a packet coming from port 2 contains the hexadecimal string 0x4953494841564f43 aligned on a 64-bit boundary, write subsequent packet data to the flow table using the existing pipeline for updates from the trusted host.

Malice 10: *Data in packet specifies destination port of the following packets*. When eight 1149-byte packets are received on ports 2 or 3, use the last byte of the 8th packet to specify additional output ports.

V. Conclusions and Future Work

The process of vetting today's complex electronic devices needs automated solutions to keep up with critical deployment schedules. The *channelize* tool described in this paper demonstrates how to simplify the analysis and raise assurance levels in a very efficient manner. Through several experiments the results of *channelize* engender optimism as a means for slicing a design into manageable pieces and locating potential malice. The results have provided success, and have shown opportunities for additional improvement. Future work will focus on algorithmic capability and scalability improvements.

With enhancements, *channelize* will eventually include a capability that enables it to map equivalent regions of components (strictness = 2), where each region may contain a different quantity and structure of components. Although a formal equivalence mapping mode for channelize was not pursued on the VET program, it may be possible, with API access to a commercial grade FEC tool, to add this capability.

Another possible mapping technique we may investigate is "fuzzy" mapping (strictness = 1), where *channelize* would use some heuristics to map "similar" regions of logic. The goal of this type of mapping would be to sacrifice some of the speed associated with "greedy" (strictness = 0) channelization in favor of a best-effort mapping attempt for each component or region of components. This mapping technique would use both greedy and one-to-one (strictness = 3) mapping as its basis. The former would define the boundaries of the mapping space, while the latter would provide the algorithm both with a starting point for heuristic mapping and contribute to the scores used to compare component regions.

The *channelize* capability, along with other MacB-developed technologies from DARPA VET, has been successfully transitioned to several US Government labs, where we hope to improve its usability and value on real-world FPGA firmware trust and assurance efforts.

Acknowledgements

We thank DARPA I2O for their support in performing the research presented in this paper. We also thank the University of Southern California's (USC) Information Sciences Institute (ISI) for providing the designs used to evaluate our methods. Finally, we would like to thank Timothy Dunham and Dr. Scott Harper for their technical contributions to MacB's work on the DARPA VET program.

References

[1] Xilinx, Inc., "UltraScale+ FPGAs Product Tables & Product Selection Guide," 2018. [Online]. Available: https://www.xilinx.com [Accessed Sept. 2018].

[2] Intel Corporation, "Intel Stratix 10 Tx Product Table," [Online]. Available: https://www.intel.com. [Accessed Sept. 2018].

[3] Defense Advanced Research Projects Agency, "Vetting Commodity IT Software and Firmware (VET)," [Online]. Available: https://www.darpa.mil/program/vetting-commodity-it-software-and-firmware. [Accessed Sept. 2018].

[4] S. Harper, J. Hallman and S. Baka, "Vetting FPGA firmware for commodity devices," in Proceedings of the Government Microcircuit Applications & Critical Technology (GOMACTech) Conference, Orlando, FL, 2016.

[5] W. Batchelor, F. Tuttle, S. Baka and S. Harper, "Inexact subgraph matching for digital circuit analysis," in Proceedings of the Government Microcircuit Applications & Critical Technology (GOMACTech) Conference, Reno, NV, 2017.

[6] T. Dunham, S. Baka, J. Hallman and S. Harper, "Malicious trigger discovery in field programmable gate array (FPGA) firmware," in Proceedings of the Government Microcircuit Applications & Critical Technology (GOMACTech) Conference, Miami, FL, 2018.

[7] NetFPGA.org, "NetFPGA Systems - NetFPGA 10G Details," [Online]. Available: https://netfpga.org/site/#/systems/3netfpga-10g/details/. [Accessed Sept. 2018].

[8] NetFPGA.org, "NetFPGA 10G Reference Router," 2013-2017. [Online]. Available: https://github.com/NetFPGA/NetFPGA-public/wiki/Home_NetFPGA-10G-Reference-Router. [Accessed 2018].

[9] Stanford University, "NetFPGA 10G OpenFlow Switch," 2013-2017. [Online]. Available: https://github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-10G-OpenFlow-Switch.

[10] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller and N. McKeown, "Implementing an OpenFlow Switch on the NetFPGA platform," in Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'08), San Jose, CA, 2008.

[11] T. Yabe, "OpenFlow Implementation on NetFPGA-10G - Design Document," [Online]. Available: https://github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-10G-OpenFlow-Switch.

[12] Open Networking Foundation (ONF), "OpenFlow Switch Specification," 26 March 2015. [Online]. Available: https://www.opennetworking.org.