2021 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM
VEA / CYBER TECHNICAL SESSION
AUGUST 10-12, 2021 - NOVI, MICHIGAN

Real-time Analysis of Vehicle Patches and Binaries

James Brock[1], Jason Dahlstrom[1], Stephen Wille Padnos[1], and Stephen Taylor[2]

[1]Web Sensing LLC, Hanover, NH
[2]Thayer School of Engineering at Dartmouth, Hanover, NH

ABSTRACT
Modern data loggers of industrial bus networks provide a useful tool to record the bus traffic associated critical vehicle systems, but provide little insight into the impact of maintenance patches on the associated system binary codes and system behaviors. This paper describes an emerging DARPA technology, the Tactical Smart Network Interface Card (TSNIC), that provides a secure base from which to deploy, monitor, and interact with patched binaries. Our TSNIC appliance can take either a passive or active presence on the vehicle bus, obviating the need for a vulnerable JTAG interface, and processes diagnostic messages arriving from the patched binary. These messages can provide a wide range of insights into the behavior of the system. The Tactical Smart NIC represents the next-generation of secure and reliable patching technology for military and heavy industrial systems. It provides a unique way for developers, maintainers, and field engineers to gain a new appreciation for the operational impact of their patches. DARPA is actively seeking operational partners for deployment and technology evolution. This paper outlines the existing and planned capabilities of the platform.

Citation: J. Brock, J. Dahlstrom, S. Wille Padnos, S. Taylor, “Real-time Analysis of Vehicle Patches and Binaries”, In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 10-12, 2021.

1. BACKGROUND
The design, validation, operation, and maintenance of military and industrial heavy vehicle systems fundamentally relies on highly connected networks of engine control units (ECUs) and other industrial controllers communicating over industry standard busses (i.e. CAN, J1939, 1553)[1-3]. During the development cycle, these controllers will have an interface, such as JTAG, available for deploying and debugging binaries. However, once installed into a fielded system this interface is usually disabled because of the security risks posed with leaving a direct, open interface to the controller[4]. Unfortunately, the controllers are often extremely difficult to physically access for maintenance and updating. Additionally, the associated bus protocols have known vulnerabilities that malicious actors can exploit[5]. In some cases, a separate auxiliary bus is available for traffic related to diagnostics and maintenance, in addition to the primary control bus. Therefore, the most effective means of communicating with deployed control systems on heavy vehicle

This research is supported under DARPA’s AMP program under contract N66001-20-C-4022.

platforms, and observing the interactions between them, is via these industry standard busses.

Maintenance and update of control system binaries are critical to improving performance, adding capability, and patching vulnerabilities. These necessitate a system, resident on the control bus, capable of deploying binary patches and monitoring the resultant system behavior with precision and accuracy. Current entry-level COTS bus monitors can provide a moderate level of flexibility via custom software, but rely on software timers within the operating system that have poor fidelity for both timestamping input and replaying output[6]. Mid-range bus adapters address the timer issues, but do not provide custom replay of bus activity with accurate timing[7]. The current state-of-the-art capabilities provide limited scripting capabilities for replaying using relatively obtuse lambda-style function definitions[8]. Consequently, there exists a need for programmable appliances, allowing arbitrarily complex functions, leveraging standard, well-established libraries and programming practices.

In response to this need, Web Sensing has developed a new Tactical Smart NIC (TSNIC) product, leveraging state-of-the-art Field Programmable Gate Array (FPGA) technology, under the DARPA SBIR program. This platform, shown in Figure 1, sits on a system bus interacting with a vehicle system, monitoring and validating mission traffic. These capabilities are incorporated into Hidden Hardware Monitors (HHM's)[9,10] embedded within the onboard FPGA.

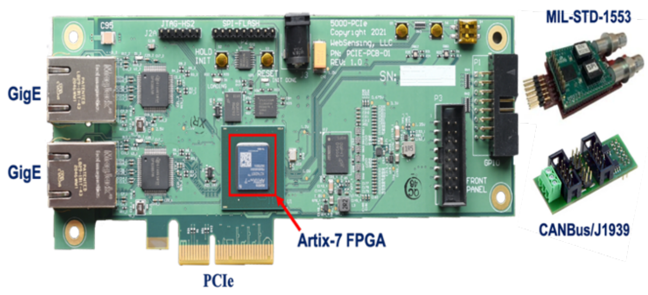


Figure 1: Tactical Smart NIC

The monitors are specified in a normal systems programming language – C/C++/System-C -- and can compute any arbitrary function. After software test and validation, they are *automatically* translated into hardware circuit blocks through a process termed High-Level Synthesis (HLS). The hardware blocks are then dynamically installed within in the FPGA through a process called Partial Reconfiguration[11]. This allows the TSNIC to dynamically adjust the granularity and fidelity of monitoring and control, adaptively react to threat level, and incrementally improve resilience.

This *all-hardware* security approach offers three key security properties: No software is present, thereby mitigating malicious implants and zero-day attacks; all functionality is strictly contained within the security perimeter provided by the chip-boundary, thereby mitigating reverse engineering in the event that a TSNIC is captured in the field; extensive anti-tamper and refresh techniques have already been developed to enhance resilience.

Under the DARPA AMP program we are developing tools, techniques, and procedures (TTP's) for developing, testing, and deploying heavy vehicle HHM's. These monitors are already capable of interacting with ECUs and control systems via industry-standard bus architectures. The TTP's allow any developer to rapidly specify, develop, validate, deploy, monitor, and interact with any vehicle control system.

2. TSNIC ARCHITECTURE

Unlike software diagnostic systems – that allow persistent malicious implants to penetrate into a vehicle system -- the TSNIC forms a *hidden security barrier,* implemented wholly in hardware, thereby eliminating the opportunity for software exploits to tamper with a vehicle or access and deposit implants within vehicle subsystems. The TSNIC is analogous to a logic analyzer for CAN, J1939, and MIL-STD-1553 busses, by virtue of its ability to accurately & precisely timestamp and validate bus traffic, in real time. In addition, it possesses advanced features enabled by its onboard DDR RAM, Gigabit ethernet ports, PCIe interface, and GPIO pins. Collectively, these features enable application-specific analytics engines to be dynamically inserted within the FPGA, through partial reconfiguration, allowing the TSNIC to meet any emerging analytic need or evolving threat. Custom analytics engines are able to utilize the memory and interfaces through common software APIs, such as SocketCAN[12]. An important security feature of the FPGA design is the ability to encrypt any data that may be transmitted or stored outside the chip/security boundary provided by the FPGA. As a result, all of the connections to interfaces and backend storage, through either GigE, PCIe, DDR, or GPIO interfaces can optionally be encapsulated using an IPSec Encapsulating Security Payload (ESP) and encrypted using AES-256.

Figure 2 depicts the overall design methodology. High-Level Synthesis (HLS) allows the TSNIC to be adapted to alternate vehicle platforms and incorporate alternate analytics: HHM's specified in C, C++, or System-C are directly synthesized into hardware blocks. The TSNIC reserves slots within its FPGA for multiple blocks, each of which have access to all of the available resources and interfaces. The interfaces are all presented to developers through intuitive APIs. The TSNIC is able to connect to a broad set of busses by isolating the physical interface from the HHM's through a standard AXI stream interface that is encapsulated in our SocketCAN API. Thus, any physical interface can be replaced without affecting the rest of the hardware design significantly.

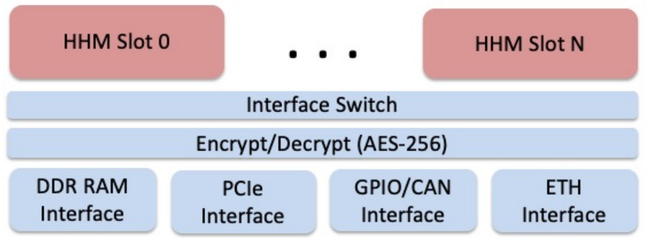


Figure 2: The System Architecture

Since the TSNIC's analytics engines are software-defined but deployed dynamically as hardware, an extremely broad array of possible operations can be performed on the observed bus traffic. The present capabilities provide precise timestamping of bus packets, variable and stack monitoring with basic statistical analysis of the observed values (min, max, avg, range, etc.), as well as bus-trace capture and differencing. Collectively, these features enable precise, detailed observation and analysis of vehicle system binaries operating on the bus in real-time and allow the impact of patching to be assessed. We are also developing more complex analytics engines that take actions on the bus to avoid or mitigate observed erroneous, unwanted, or unsafe vehicle system behavior.

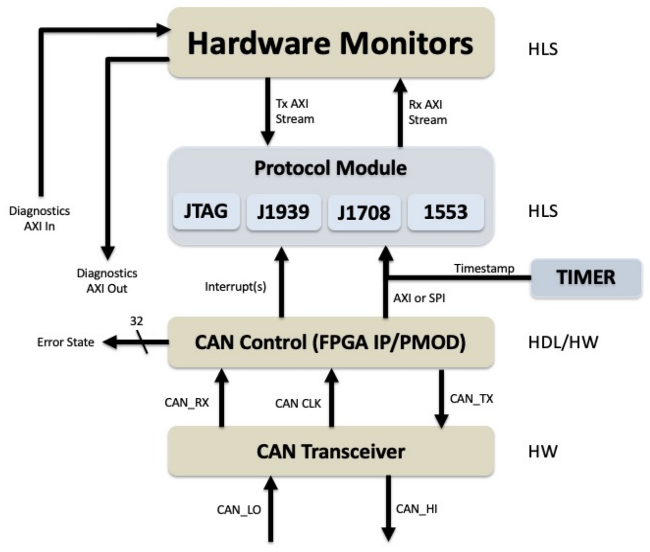


Figure 3: Hardware monitor interfaces

3. HIDDEN HARDWARE MONITORS

Recall that the TSNIC hardware is a platform onto which developers can deploy custom analytics engines called *hidden hardware monitors* (HHM's). These monitors, because they are specified in a standard systems programming language, all utilize a common software API to interface to any connected bus. This API is based on the Berkeley Sockets API[13], widely used and familiar to developers. The full hardware monitor pipeline is shown in Figure 3.

All hardware monitors consume streams (Tx and Rx) through the sockets API; generic memory mapped pointers can be used for arbitrary data I/O to all other hardware resources. This gives hardware monitors a common software framework in which to operate and makes them compatible with any slot in the FPGA. A protocol module sits between the hardware monitor and any given bus interface. This hardware block contains a pre-defined hardware bus module that will translate bus traffic to/from the generic Rx/Tx stream and take necessary actions to adhere to the selected protocol. For example, the J1939 protocol module will maintain a list of claimed addresses and generate appropriate response address claim packets on the bus. This module is also responsible for applying accurate timestamps to incoming traffic as soon as it is received. FIFO buffering of both the received and transmitted messages can be configured to meet a given application's needs. The protocol module is responsible for appropriate error handling in the chosen protocol, however, there are additional hardware mechanisms in the FPGA design to monitor and track system errors, such as dropped packets or error reporting from the HHM itself. Since hardware interfaces are unlikely to change often, the protocol module is statically preconfigured in the FPGA design at build time.

The central problem of concern in the DARPA AMP program is the validation of micropatches in vehicle control system binaries. Elements of the program involve both static analysis through de-compilation/recompilation of binaries, as well as dynamic analysis through real-time monitoring. The TSNIC is the platform to effect dynamic analysis; it sits directly on J1939 CAN bus observing both normal traffic and diagnostic traffic emanating from instrumented binaries and patches.

Early collaboration with other performers on the program has yielded a list of core capabilities that the TSNIC needs to support, so as to integrate into performer workflows and address the needs of the program. These capabilities have all been implemented and are available; they allow:

- Monitoring the state of a variable at run-time
- Support for arbitrarily large diagnostic payloads
- Transmission of stimulus onto the bus with precisely specified timing
- Full bus-trace recording with precise time stamping
- Correlation of a binary's Program Counter (PC) with another variable's state
- Correlation of binary's PC with bus transactions
- Correlation of call stack addresses and data with bus transactions
- Correlate binary clock values with bus transaction timestamp
- First order statistics (min, max, avg) for all of the above observations
- Comparison of data point ordering for all of the above observations
- Comparison of data point timing for all of the above observations
- Differencing between multiple bus-traces
- Remote access to a TSNIC platform attached to challenge problem hardware

On top of these core capabilities, we have developed a cohesive, automated workflow for developing, synthesizing and deploying hardware monitors to the TSNIC. This workflow alleviates developer effort and is shown in Figure 4. It begins on the right of Figure 4. A user, such as Galois, Inc or the University of Michigan on the AMP program, deploys a patched binary to the vehicle system. The binary is instrumented to send whatever diagnostic payload over the bus they desire. To specify a monitor, the user provides a simple C-structure to represent the payload data and a simple JSON file that specifies the features to operate on the data. These definitions are fed into the automated workflow, shown in red, on the bottom half of the image. This flow first takes the users specifications, generates C/C++ source code

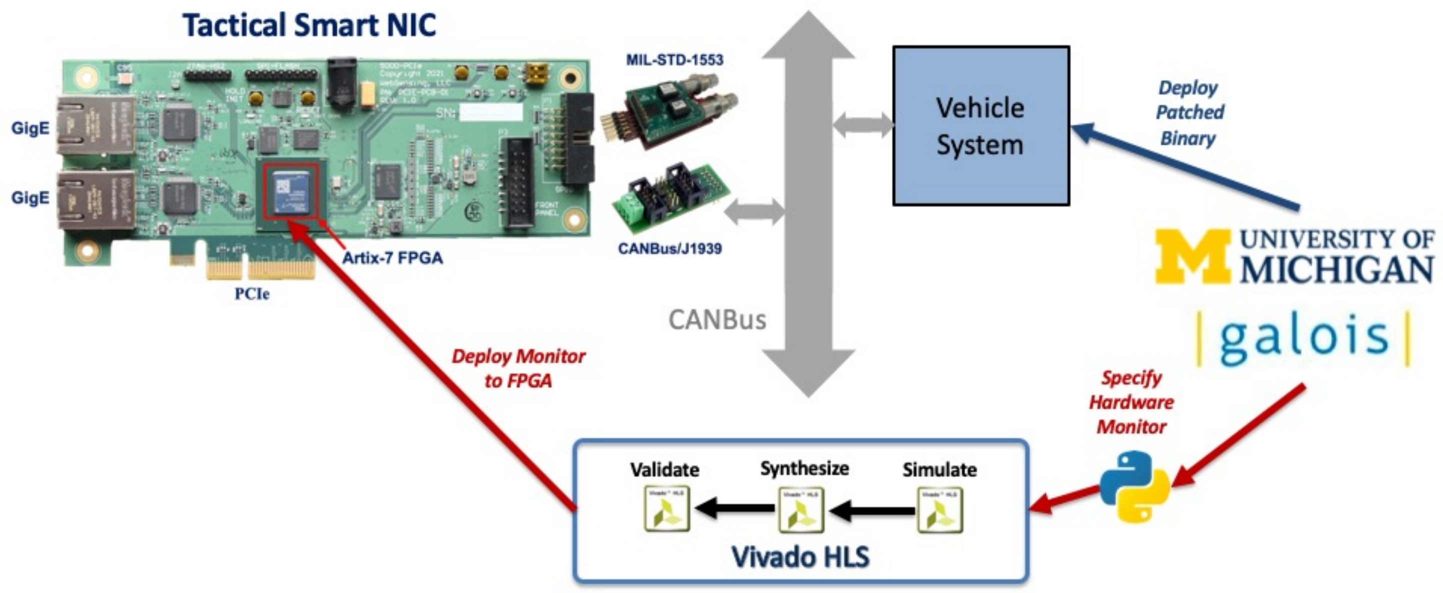
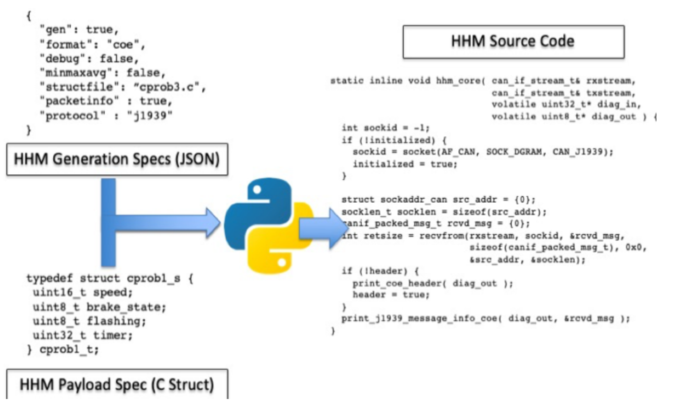


Figure 4: Hidden hardware monitor workflow

for the hardware monitor as well as a basic testbench to validate it. At this point, the developer can choose to add additional functionality and tests to the HHM source. Once the source is complete, our scripts run a software simulation of the HHM with the testbench, synthesizes it into a hardware module, and validates the synthesized hardware using the same testbench to ensure it has retained all expected behavior. The last step is to automatically deploy the HHM into one of the available slots in the TSNIC FPGA design. At this point the TSNIC becomes the custom monitor. It listens to the bus for the messages from the binary, performs the requested analytics, saves the required results, and may optionally interact with the running binary to refine diagnostics or reduce the impact of diagnostic messages.

To automate HHM code generation, the basic functionality of a hardware monitor is supplied though some standard boilerplate code. We have written a series of scripts that generate some basic hardware monitors for passively reading packets off the bus. Options passed to these scripts are provided in a JSON configuration file. The options designate the generation of code implementing the capabilities we currently support, customized to the data the instrumented binary is transmitting for any given use case. An example of one of these configuration files is shown in Figure 5 (top left). The C structure defining the payload being transmitted by the binary is shown in the bottom left, and a snippet of the resultant source code is on the right side.



```
{
  "gen": true,
  "format": "coe",
  "debug": false,
  "minmaxavg": false,
  "structfile": "cprob3.c",
  "packetinfo" : true,
  "protocol" : "j1939"
}
```

```
typedef struct cprobl_s {
 uint16_t speed;
 uint8_t brake_state;
 uint8_t flashing;
 uint32_t timer;
} cprobl_t;
```

```
static inline void hhm_core( can_if_stream_t& rxstream,
                             can_if_stream_t& txstream,
                             volatile uint32_t* diag_in,
                             volatile uint8_t* diag_out ) {
  int sockid = -1;
  if (!initialized) {
    sockid = socket(AF_CAN, SOCK_DGRAM, CAN_J1939);
    initialized = true;
  }

  struct sockaddr_can src_addr = {0};
  socklen_t socklen = sizeof(src_addr);
  canif_packed_msg_t rcvd_msg = {0};
  int retsize = recvfrom(rxstream, sockid, &rcvd_msg,
                         sizeof(canif_packed_msg_t), 0x0,
                         &src_addr, &socklen);
  if (!header) {
    print_coe_header( diag_out );
    header = true;
  }
  print_j1939_message_info_coe( diag_out, &rcvd_msg );
}
```

Figure 5: Hidden hardware monitor code generation

The primary features for which a hardware monitor can be automatically generated are listed below:

- Generate basic HHM code to read bus data

- Interpret bus data as CAN or J1939 formatted packets
- Interpret bus data payloads as a specified payload (variable monitoring)
- Maintain min/max/avg values of all variables over time
- Print packet header and flag information in addition to payload information
- Print information in specified format (pretty print, CSV, COE)

A full description of the JSON code generation options and their effects can be found in our published documentation. For features that print output information, this is achieved via a simple UART interface.

The automated workflows and features, as well as the adherence to industry standard APIs, make the HHM easy to create, modify, validate, and deploy to the TSNIC. Several exemplars are provided with the TSNIC to ease transition and use of the device.

4. PERFORMANCE STUDY

Clearly, the user must ensure that diagnostic traffic does not impact system behavior. Our approach allows the level of traffic to be controlled or completely eliminated reverting to only passive bus tracing. This offers a broad practical trade-off space between full observation and zero impact.

A primary concern has been to characterize the performance of the TSNIC and its ability to transmit, capture, and process J1939 bus traffic in real-time. To study this question, we connected two J1939 interfaces to the board's GPIO ports and placed a BeagleBone Black based Heavy Truck Cape [14] on the bus to have meaningful interactions with, as shown in Figure 6. The TSNIC was then populated with a HHM that would transmit J1939 packets with semi-random payloads out on one interface, and a HHM that would receive, confirm the J1939 header, and then drop the packet connected to the other interface. Since the internal 125MHz clock of the TSNIC is much faster than the bus architecture targeted, this one experiment demonstrates the ability of the TSNIC to transmit at maximum bus load, receive at maximum bus load, and properly buffer packets without generating any internal errors.

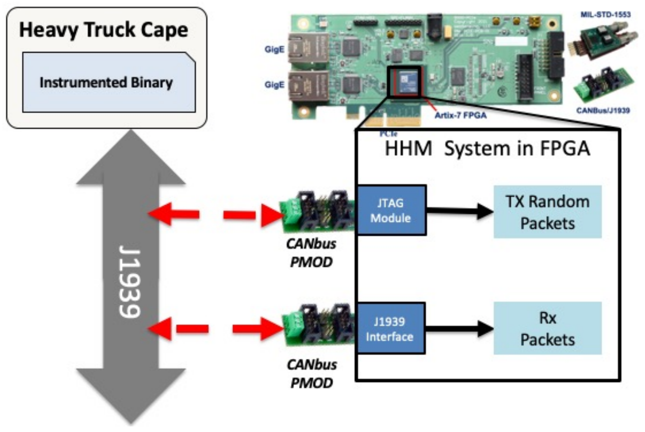
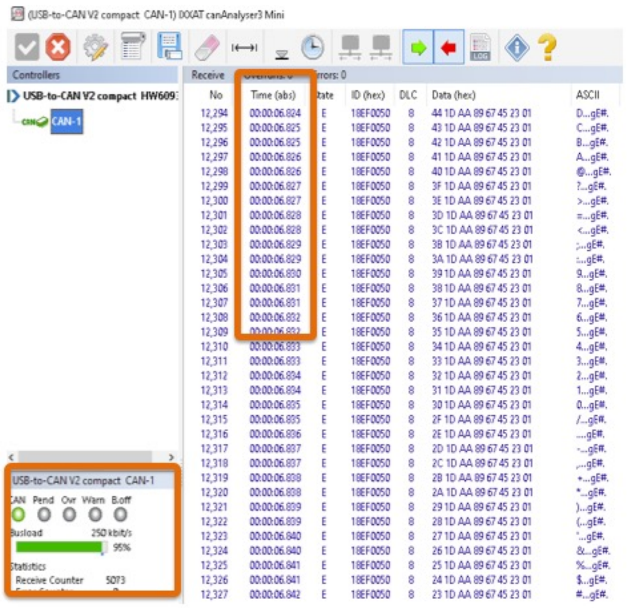


Figure 6: Performance Characterization

We gathered data from two sources during runs of this experiment. First, from a COTS USB-to-CAN bus analyzer, and second from an Integrated Logic Analyzer (ILA) module inserted in the TSNIC FPGA design. The ILA is capable of capturing the FPGA's internal signals in real-time and sending them back to a display. Multiple runs of the experiment confirmed that the output from our two sources was consistent. Sample output is shown in Figures 7 and 8. Figure 7 shows a screenshot of the COTS bus analyzer's output, and demonstrates that the TSNIC transmits traffic on the bus at the maximum feasible capacity: ~95% bus load. The lower orange box shows graphically the current bus load, while the upper orange box highlights the timestamps of successive packets coming across the bus at the expected rate. Figure 8 depicts some of the output of the FPGA's ILA. The upper maroon box shows a packet being sent across successive busses in the HHM pipeline through the protocol module to the HHM. The lower maroon box highlights an error count of 0, indicating that after many packets have already been received, the system has yet to detect any dropped packets or other internal errors.



(USB-to-CAN V2 compact CAN-1) IXXAT canAnalyser3 Mini

Controllers

USB-to-CAN V2 compact HW609

CAN-1

Receive Overruns: 0 Errors: 0

```
No      Time (abs)    State  ID (hex)  DLC  Data (hex)                ASCII
12,294  00:00:06.824  E      18EF0050  8    44 1D AA 89 67 45 23 01   D...gE#.
12,295  00:00:06.825  E      18EF0050  8    43 1D AA 89 67 45 23 01   C...gE#.
12,296  00:00:06.825  E      18EF0050  8    42 1D AA 89 67 45 23 01   B...gE#.
12,297  00:00:06.826  E      18EF0050  8    41 1D AA 89 67 45 23 01   A...gE#.
12,298  00:00:06.826  E      18EF0050  8    40 1D AA 89 67 45 23 01   @...gE#.
12,299  00:00:06.827  E      18EF0050  8    3F 1D AA 89 67 45 23 01   ?...gE#.
12,300  00:00:06.827  E      18EF0050  8    3E 1D AA 89 67 45 23 01   >...gE#.
12,301  00:00:06.828  E      18EF0050  8    3D 1D AA 89 67 45 23 01   =...gE#.
12,302  00:00:06.828  E      18EF0050  8    3C 1D AA 89 67 45 23 01   <...gE#.
12,303  00:00:06.829  E      18EF0050  8    3B 1D AA 89 67 45 23 01   ;...gE#.
12,304  00:00:06.829  E      18EF0050  8    3A 1D AA 89 67 45 23 01   :...gE#.
12,305  00:00:06.830  E      18EF0050  8    39 1D AA 89 67 45 23 01   9...gE#.
12,306  00:00:06.831  E      18EF0050  8    38 1D AA 89 67 45 23 01   8...gE#.
12,307  00:00:06.831  E      18EF0050  8    37 1D AA 89 67 45 23 01   7...gE#.
12,308  00:00:06.832  E      18EF0050  8    36 1D AA 89 67 45 23 01   6...gE#.
12,309  00:00:06.832  E      18EF0050  8    35 1D AA 89 67 45 23 01   5...gE#.
12,310  00:00:06.833  E      18EF0050  8    34 1D AA 89 67 45 23 01   4...gE#.
12,311  00:00:06.833  E      18EF0050  8    33 1D AA 89 67 45 23 01   3...gE#.
12,312  00:00:06.834  E      18EF0050  8    32 1D AA 89 67 45 23 01   2...gE#.
12,313  00:00:06.834  E      18EF0050  8    31 1D AA 89 67 45 23 01   1...gE#.
12,314  00:00:06.835  E      18EF0050  8    30 1D AA 89 67 45 23 01   0...gE#.
12,315  00:00:06.835  E      18EF0050  8    2F 1D AA 89 67 45 23 01   /...gE#.
12,316  00:00:06.836  E      18EF0050  8    2E 1D AA 89 67 45 23 01   ....gE#.
12,317  00:00:06.837  E      18EF0050  8    2D 1D AA 89 67 45 23 01   -...gE#.
12,318  00:00:06.837  E      18EF0050  8    2C 1D AA 89 67 45 23 01   ,...gE#.
12,319  00:00:06.838  E      18EF0050  8    2B 1D AA 89 67 45 23 01   +...gE#.
12,320  00:00:06.838  E      18EF0050  8    2A 1D AA 89 67 45 23 01   *...gE#.
12,321  00:00:06.839  E      18EF0050  8    29 1D AA 89 67 45 23 01   )...gE#.
12,322  00:00:06.839  E      18EF0050  8    28 1D AA 89 67 45 23 01   (...gE#.
12,323  00:00:06.840  E      18EF0050  8    27 1D AA 89 67 45 23 01   '...gE#.
12,324  00:00:06.840  E      18EF0050  8    26 1D AA 89 67 45 23 01   &...gE#.
12,325  00:00:06.841  E      18EF0050  8    25 1D AA 89 67 45 23 01   %...gE#.
12,326  00:00:06.841  E      18EF0050  8    24 1D AA 89 67 45 23 01   $...gE#.
12,327  00:00:06.842  E      18EF0050  8    23 1D AA 89 67 45 23 01   #...gE#.
```

USB-to-CAN V2 compact CAN-1

CAN Pend Ovr Warn Boff

Busload 250 kbit/s 95%

Statistics

Receive Counter 5073

Figure 7: COTS CAN analyzer output of TSNIC

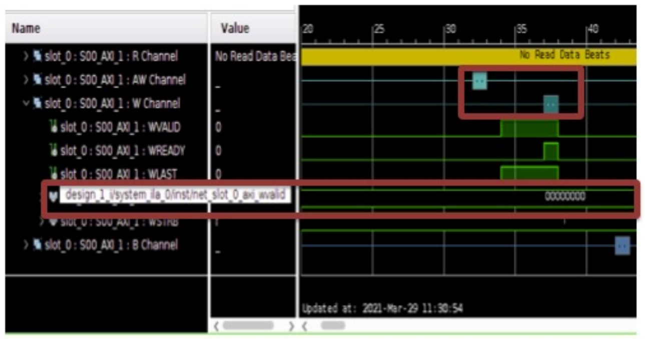

Name
Value
slot_0 : S00_AXI_1 : R Channel
No Read Data Beats
slot_0 : S00_AXI_1 : AW Channel
slot_0 : S00_AXI_1 : W Channel
slot_0 : S00_AXI_1 : WVALID
slot_0 : S00_AXI_1 : WREADY
slot_0 : S00_AXI_1 : WLAST
design_1_i/system_ila_0/inst/net_slot_0_axi_wvalid
00000000
slot_0 : S00_AXI_1 : WSTRB
slot_0 : S00_AXI_1 : B Channel
Updated at: 2021-Mar-29 11:30:54

Figure 8: Xilinx ILA display of internal TSNIC signals

5. PERFORMER USE CASES

As mentioned earlier, under the DARPA AMP program, we are collaborating with a number of other performers to develop HHM's. We have worked closely with Galois, Inc. and the University of Michigan to develop techniques for automatically inserting observation points into a patched binary. These observation points then transmit diagnostic information over the bus to the TSNIC for logging and analysis.

Dr. Tristan Ravitch and his team at Galois, Inc. and UCI, have developed a HHM that correlates the context of a binary's expected and actual execution with its interactions receiving and transmitting messages on the bus. The first part of this process begins with simulating the binary's execution using QEMU and feeding instructions into LLVM MCA to derive an estimated timing of the transmission of bus messages. This is then compared to the actual precise timing of bus transactions captured by our TSNIC platform. This process is depicted in Figure 9. The next phase of Galois' analysis is to instrument the binary to transmit the program counter and portions of the program call stack at points in the program where bus messages are received or transmitted. As well as developing a hardware module to capture these diagnostic messages and correlate them to that bus activity for the purpose of reasoning about the program state when critical system events are taking place. Both the precise timing and characterization of the program state will provide useful data for analyzing binary bugs and patches.

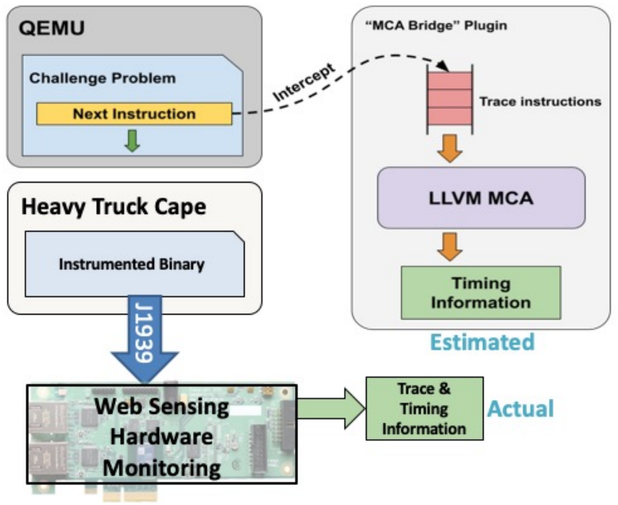


Figure 9: Galois, Inc. HHM Use Case

A second use case developed by Prof. Baris Kasikci and his team at the University of Michigan involves modules that will monitor binary variable values during execution. These values are then analyzed for invalid values, first-order statistics like min/max/avg, and other characteristics that will inform how the binary is performing. The University of Michigan team has a process, depicted in Figure 10, that generates candidate binary patches and then conducts symbolic program comparisons and non-interference proofs on those patches to determine if a patch should be revised or accepted. The run-time variable values that are collected using our hardware monitoring platform are fed into the symbolic program comparison step to help determine which patch should be accepted as valid.

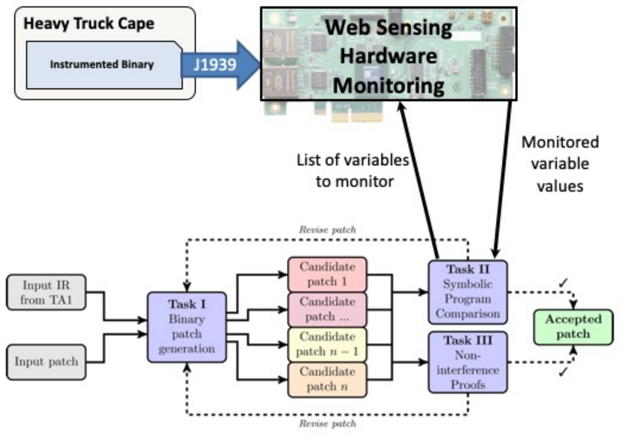


Figure 10: University of Michigan HHM Use Case

6. FUTURE WORK

We plan to develop additional capabilities to meet current and future industry needs. One prospective use case involves reasoning about monitored bus behavior for unsafe or incongruous behavior. In one of the AMP program challenge problems, the University of Michigan team discovered an unintentional stack smashing vulnerability. Though outside the scope of the original challenge problem, this represents a common, real-world vulnerability in the use of network software stacks. A code snippet of the vulnerability is shown in Figure 11. Here, the uninitialized stack variable is declared on line 1, potentially partially written to on line 3, and then fully read by the function called on line 11. This allows malformed J1939 packets or packets with improper data to be parsed and acted on by the ECU. We are developing HHMs to detect malformed packets, packets with suspicious extra data, and interpret payloads to ensure that the ECU does not take unsafe actions after receiving such a packet.

```
 1  struct can_frame cf;
 2
 3  nbytes = read(s, &cf, sizeof(struct can_frame));
 4  if (nbytes < 0 ) {
 5    perror("CAN read");
 6    return 1;
 7  }
 8
 9  if ( ((cf.can_id >> 8) & 0x7FFFF) ==
10       PGN_CruiseControlVehicleSpeed1 ){
11    rx_brake_routine(cf.data, bumper);
12  }
```

Figure 11: Stack smashing vulnerability code

Another desirable capability is support for DTrace[15] probe scripts in HHM development. DTrace is a mature, widely used standard for developing profiling hooks and understanding binary execution behavior. Very often in embedded systems, memory is a scarce resource; Instead of attempting to store all the DTrace data in system memory, it can instead be transmitted across the bus for collection and analysis by HHMs on the TSNIC.

7. CONCLUDING REMARKS

FPGA devices provide unique security and performance opportunities over existing COTS solutions for accurate real-time monitoring and interaction with critical control systems. When deployed on our Tactical Smart NIC (TSNIC), they can allow deployment, debugging, and validation of patched control system binaries while also presenting enhanced telemetry and security. DARPA is actively seeking transition opportunities for this technology within the heavy vehicle community.

8. ACKNOWLEDGEMENTS
This research is supported under DARPA's AMP program under contract N66001-20-C-4022. We would also like to thank the principal investigators and teams at Galois, Inc. and the University of Michigan for their collaboration and support under the DARPA AMP program.

1. REFERENCES

[1] Szydlowski, C., "CAN Specification 2.0: Protocol and Implementations," SAE Technical Paper 921603, 1992, https://doi.org/10.4271/921603

[2] SAE International. "SAE Truck and Bus Control & Communications Network Standards Manual". SAE Technical Paper, 2013. https://www.sae.org/publications/collections/content/j1939_dl/

[3] United States Department of Defense. "Digital Time Division Command/Response Multiplex Data Bus". MIL-STD 1553, 2018. https://quicksearch.dla.mil/qsDocDetails.aspx?ident_number=36973

[4] P. Murvay and B. Groza, "Security Shortcomings and Countermeasures for the SAE J1939 Commercial Vehicle Bus Protocol," in *IEEE Transactions on Vehicular Technology*, vol. 67, no. 5, pp. 4325-4339, May 2018, doi: 10.1109/TVT.2018.2795384.

[5] Cybersecurity & Infrastructure Security Agency. ICS Medical Advisory (ICSMA-19-274-01). 2020. https://us-cert.cisa.gov/ics/advisories/icsma-19-274-01

[6] IXXAT USB-to-CAN v2 Pro Data Logger https://www.ixxat.com/products/products-industrial/can-interfaces/usb-can-interfaces/usb-to-can-v2-professional

[7] CSS Electronics CAN Bus Logger with RTC timestamp. https://www.csselectronics.com/screen/product/can-bus-logger-canlogger2000

[8] NEOVI Fire 2 Vehicle Interface and Data Logger. https://store.intrepidcs.com/product/neovi-fire2

[9] Dahlstrom, J., Taylor, S., "Migrating an OS Scheduler into Tightly Coupled FPGA Logic to Increase Attacker Workload", In proceedings of MILCOM 2013, pp 986-991, Nov 2013.

[10] Dahlstrom, J., Taylor, S., "Hardware-Based Code Monitors on Hybrid, Processor-FPGA System-on-Chip Architectures", MILCOM 2015, pp 968-973, Oct 2015.

[11] Dahlstrom, J., Brock, J., Tenaw, M., Shaver, M., Taylor, S., "Hardening Containers for Cross-Domain Applications," MILCOM 2019 – 2019 IEEE Military Communications Conference (MILCOM), Norfolk, VA, USA, 2019, pp 1-6.

[12] Kleine-Budde, Marc. "The Official CAN API of the Linux Kernel", in proceedings of iCC, 2012.

[13] Stevens, R., Fenner, B., Rudoff, A., "Unix Network Programming Volume 1, Third Edition: The Sockets Networking API", Addison Wesley, 2003.

[14] Daily, J., "Heavy Truck Cape", Colorado State University, 2016, https://github.com/SystemsCyber/TruckCapeProjects

[15] Oracle DTrace Linux, 2013, https://docs.oracle.com/en/operating-systems/oracle-linux/dtrace-guide/dt_about.html