2022 NDIA MICHIGAN CHAPTER
GROUND VEHICLE SYSTEMS ENGINEERING
AND TECHNOLOGY SYMPOSIUM
CGS / CYBER SECURITY OF GROUND SYSTEMS SESSION
AUGUST 16-18, 2022 - NOVI, MICHIGAN

The Tactical Smart NIC

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

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

ABSTRACT

This paper describes a novel network security appliance -- the Tactical Smart Network Interface Card (TSNIC) – that leverages state-of-the-art Field Programmable Gate Array (FPGA) technologies to continuously maintain the integrity of tactical missions. The Smart NIC appears as an all-hardware "bump-in-the-wire" along any network segment or attached to an industry standard bus interface providing infrastructure defense for ground vehicles. It can be custom configured to provide encryption, protocol and file format validation, and/or protocol encapsulation. These capabilities are achieved by several innovations: high-level synthesis (HLS) for rapid circuit development, automated parser generation to adapt to mission requirements, and a hardware nano-marshal to dynamically adapt defensive posture in the face of changing threat profiles.

Citation: J. Dahlstrom, S. Padnos, J. Brock, and S. Taylor, "The Tactical Smart NIC," In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 16-18, 2022.

1. INTRODUCTION

Many organizations handle sensitive data and files relating to military missions, trade secrets, intellectual property, private personal data, and/or classified projects [1]. Traditionally, these organizations have been well advised to implement an "airgap" that physically disconnects computers containing sensitive information from any connection to the Internet, to protect against theft. Unfortunately, airgaps come with substantial cost in productivity and assume that the relevant staff, with the expertise to handle sensitive information, is co-located. Airgaps are increasingly impractical given the need to connect critical systems, such as industrial plant, to cloud-based analytic platforms (e.g., Google Analytics) or support condition-based maintenance of DoD vehicles [2,3,4].

Ground vehicles increasingly rely upon standard networking technologies to link embedded control systems with sensors, actuators, and human machine interfaces through industry standard buses -- CAN, J1939, MIL-STD-1553, USB -- and other communications interfaces – PCIe, Gigabit Ethernet (GigE), and OpenVPX. In many

instances, vehicles are periodically inter-connected with military installations to provide mission parameters, or to effect maintenance and upgrades. Network connected personal devices – phones, tablets, and laptops – are increasingly being used to manage and interact with these systems. Though network boundary protections generally separate installations from the Internet, there are many threat vectors that circumvent such protections: for example, unintended connections, insiders, zero-day exploits, supply chain interdiction, and persistent implants [5].

This paper explores the capabilities of the Tactical Smart Network Interface Card (TSNIC) -- a network appliance technology under development in the DARPA AMP program. In a previous GVSETS article [6], we explored its use for ground vehicle patch analysis. This paper explores its more general cyber security capabilities in support infrastructure defense and condition-based maintenance. In this application, the TSNIC provides a *hardware barrier* between network segments that continuously validates mission traffic. Consequently, it acts to constrain the attack surface *behind* conventional boundary defenses, such as firewalls and intrusion detection systems, hardening the attached systems against cyber threats.

2. DESIGN PHILOSOPHY

The Smart NIC is shown in Figure 1. It consists, by design, of a *single* FPGA chip interfacing directly to GigE and PCIe interfaces on the left and bottom, and industry standard buses via daughter cards attached to the ribbon connector on the right. OpenVPX is accommodated via a simple PCIe adaptor or a variant of the board with an alternate connector. In consequence, the appliance forms a "bump-in the-wire" between any pair of the available connections, with the FPGA forming a bridge for all communication. Consequently, the FPGA can monitor and interact with all systems attached to its interfaces and can act to store and forward traffic between them.

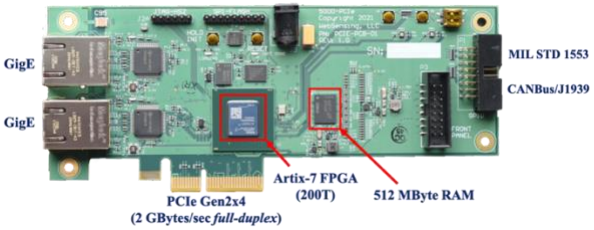

Figure 1. Tactical Smart NIC

As an *all-hardware* appliance, the Smart NIC offers several key security advantages: All sensitive data -- encryption keys, buses, and algorithmic functionality -- is strictly hidden within the security perimeter afforded by the FPGA chip-boundary [7,8,9,10,11], mitigating reverse engineering in the event that a TSNIC is lost or captured in the field; No software is present in the device, thereby mitigating malicious implants and zero-day attacks; Either the PCIe or one of the GigE connections can be used as an out-of-band channel to adapt the device to alternate mission profiles, augment its internal functions, or upgrade the device; Extensive anti-tamper and circuit destruction techniques have been developed to enhance its resilience.

For versatility, all circuits resident in the Smart NIC are developed using a rapid prototyping technology termed High Level Synthesis (HLS). This process allows algorithm specifications to be designed and tested in C, C++, or System-C. The working code is then automatically transformed into a standalone, reusable, *hardware block*. These reusable circuit plugins can be directly integrated into a static circuit design in the FPGA. Alternatively, the block can be treated as a *container*. Using a technique known as *partial reconfiguration*, the FPGA can then be partitioned into segments and containers can be dynamically loaded into a partition then linked into the overall function of the device on-the-fly. To manage this process, we have developed a thin, hypervisor-like hardware layer termed a *Nanomarshal* [12].

3. CRYPO ACCELLERATION PLUGIN

In its most primitive form, the Smart NIC can be used purely as a crypto accelerator by placing encryption and decryption blocks between two of its interfaces. By default, the appliance provides encryption/decryption blocks implementing the Advanced Encryption Standard (AES-256). These plugin blocks have been certified under the NIST Cryptographic Algorithm Validation Program (CAVP).

Our initial rendering of AES into HLS, a straightforward transliteration of the standard algorithm, was discouraging, performing at only 5Mb/sec [13]. Subsequently, we standardized on a 16-byte wide AXI-stream implementation that forms our current baseline, and explored a variety of optimizations and algorithmic alternatives, based on the FIPS standard. The optimizations did not involve low-level HDL circuit design; instead, they were achieved automatically using *pragmas* available within the HLS process.

The results are summarized in Table 1 which defines the trade-space that can be applied, based on application performance requirements. This trade-space allows circuit performance -- bandwidth (BW), maximum clock frequency (Fmax), and latency (LAT) in clock cycles/128bit block – to be traded for FPGA resources -- Block RAM (BR), Flip-flops (FF), and Lookup-Tables (LUT).

TABLE I. Implementation Trade-Space

Method	**BW** Mb/s	**Fmax** MHz	**LAT** cy/blk	**BR**	**FF**	**LUT**
Baseline	12	174	1880	4	611	1707
Loop Unrolling	68	131	246	16	809	1740
Array Partitioning	723	147	26	0	539	45036
Function In-lining	822	167	26	0	658	3437
Cyclic Partitioning	1103	224	26	0	542	23065
Pipelining	9216	144	2	0	557	44945

TABLE I highlights 5 primary optimization techniques: *Loop Unrolling* parallelizes loops that do not contain data dependencies between each iteration of the loop. *Array Partitioning* removes the serialization bottlenecks caused by parallel data accesses that must queue for BRAM interfaces. This optimization is achieved by partitioning C-arrays into register sets that can be accessed concurrently. *Function In-lining* yields the hardware equivalent of inline C-functions in which function bodies are directly replicated at each call site. *Cyclic Partitioning* optimizes the logic depth used to achieve a single round of the AES algorithm and thereby increases Fmax. *Pipelining* is a standard technique in which data dependency analysis is used to allow overlapping of operations.

These studies led to the discovery of a "sweet spot" in the trade-space: Without Cyclic Partitioning and Pipelining (last 2 rows) – the implementation achieves close to Gigabit line speed -- 0.82 G/sec -- with only ~2.5% LUT utilization on the Artix 200T FPGA device used by default on the TSNIC.

Adding Cyclic Partitioning increases performance to 1.1Gb/sec – exceeding line speed -- but at heavy cost in LUT recourses (~20% of the Artix 200T FPGA), while pipelining again doubles LUT use. Unfortunately, the AES "Cyclic Block Chaining" (CBC) mode of operation has a data dependency that limits pipelining. However, "Counter" mode relieves this dependency and yields a 9-fold speedup.

Note that generally two instances of the encryption block are required: one for encryption and one for concurrent decryption (with similar resource requirements). A wide variety of FPGAs with differing resources are available for use on the TSNIC. The baseline unit uses the Artix-7 -- the smallest family that supports partial reconfiguration.

4. PARSER PLUGINS

The TSNIC is concerned with monitoring the flow of traffic across its interfaces and *validating* both that messages adhere to an industry standard protocol and that message content is valid in the context of a tactical mission. Generally, its action on detecting a valid message is to allow it to pass; conversely, its action on detecting invalid data is to drop the message -- mitigating potential exploitation -- and/or generate an alert. To achieve message validation, the TSNIC uses custom *parsing engines* that lay across its communication paths.

Parsing is the general process of taking an input stream of symbols and understanding their format (syntax) and meaning (semantics). For example, compilers such as GCC use a parser to validate that a computer program, written in some programming language such as C/C++/Java/Fortran is written correctly (i.e. is syntactically valid), and to understand its structure (i.e. its semantics) for the purpose of machine code generation and optimization.

Parsers are tools that apply a collection of formal *grammar rules*, defining some input language, to determine if the input adheres to the rules. For example, the following 3-rule grammar G defines a language in which a stream of symbols is valid only if it begins with the character ‘a’, ends with ‘c’, and contains one or more intervening ‘b’ characters:

G : ‘a’ Bs ‘c’ ;
Bs: ‘b’ | Bs ‘b’ ;

The “or” symbol | designates an alternative definition for the rule defining “Bs”. This grammar accepts as valid the input streams *abc*, *abbc* and *abbbc* etc, but rejects any other stream, e.g. *a, ac, aaa, ccc, adx, abbbx,* etc. Individual characters such as ‘a’ are *terminal symbols* that must be present in the input stream; all other symbols are non-terminals representing intermediate structural elements. For binary grammars, hexadecimal terminal values can also be used (e.g. ‘\xFF’ represents a single byte value corresponding to 255 in decimal).

Parser generators are tools that take a grammar as input and automatically generate a program that implements the associated parser. The most mature and widely used generator is Bison which accepts two primary classes of grammar: Generalized Look-Ahead (GLR) and the more restrictive Left-to-Right Look Ahead (LALR). Both classes of grammar are expressed in Backus-Naur Form (BNF), used above to define the grammar G. Under the DARPA SafeDocs program, new tools are being developed based formal methods. One of the most mature is the Hammer *combinator* library which provides a collection of well-defined base parsers and methods to combine them to build more complex parsers. The resulting parsers are provably correct by construction. The Hammer library provides a collection of backends that allow different classes of grammar to be implemented, including GLR and LALR.

Though GLR grammars are more general, LALR grammars are sufficient for validating a wide variety of protocol and file formats and can be realized with a *push-down automaton* – a finite state machine employing a single stack to store symbols while parsing the input stream. The state machine relies on two core operations *shift* – involving saving a symbol from the input onto the stack and *reduce* – involving the application of a grammar rule to detect a structure in the input and reduce the symbols on the stack. The state-machine is generic and common to all grammars, however, the order in which shift and reduce operations are applied to the input is based on a collection of *parsing tables* derived from the grammar (usually referred to as action/goto tables [14]). These tables map the current state of the automaton, to a next state based on the symbol read from the input stream.

The TSNIC can employ any LALR grammar written in either Bison or Hammer. This is achieved through a fully automated compilation process outlined in Figure 2. Using Bison, the input grammar -- defined in BNF -- is fed directly into the standard Bison parser generator (leftmost brown path). For Hammer, an equivalent parser is defined using pre-existing combinators in C and linked to the Hammer library (blue path). These tools can both produce a set of parsing tables expressed in a common machine-readable XML format. A conversion tool – *xml2h* – is used to convert the xml parsing tables into a C-header file (pda.h) containing a two-dimensional C-array. Rows in the array correspond to *states* in the automaton, while columns designate terminal and non-terminal symbols encountered when reading the input stream. Entries in the array designate shift or reduce actions applied in each state. Consequently, the C-array provides a complete definition of how the push-down automaton should operate to validate any particular grammar.

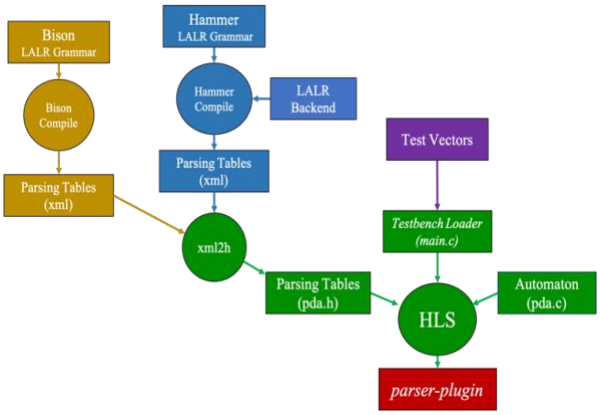

Figure 2. Parser Automation Process

The C-array is combined with generic LALR *automaton* (pda.c) [14] and *testbench* code (main.c) to produce a runnable C-program implementing the parser. This parser is validated using a set of representative test vectors files (purple) to ensure that the parser operates correctly. Since the parser is a well-structure C-program it can then be fed directly into High-Level Synthesis (HLS) to produce a *hardware* implementation of the parser that can be loaded onto the TSNIC as a plugin. The hardware parser-plugin is validated using hardware-software co-simulation employing the same test-vectors and testbench code used to validate the software version of the parser.

Unfortunately, though the conventional C-array is a convenient conceptual framework to consider, it is impractical since it contains many states that cannot in practice be reached. Consequently, a highly optimized alternative representation is used that removes much of the sparse structure. These optimizations use similar techniques to those employed internally by Bison and are described in detail in [15]. To illustrate how these concepts map in practice, Table 2 characterizes the size of a variety parsers taken from the Open-Source Parser Experimentation Repository [16] in terms of their rule and state set size. The *Size* column shows the size of the unoptimized parser-plugin in *Kilobytes* using conventional parsing tables which would typically be mapped to BRAM resources in the FPGA; The *Opt* column shows the optimized size in Kilobytes. The *Result* (Res) column shows the improvement; typically, a compaction more than 75% is achieved. For the Artix 200T, with 1.46Mbytes of BRAM, a full JSON parser would consume ~2% of the BRAM resources.

TABLE 2. Parser BRAM resources

Parser	Rules	States	Size	Opt	Res
json	191	229	74	13.3	82%
com	880	854	481	69.5	85%
resp	271	279	151	6.3	96%
json w/ unicode	325	689	407	30.3	92%

5. ENCAPSULATION PLUGIN

The previous sections have described plugin circuit blocks that operate on Ethernet packets and provide AES Encrypt/Decrypt or

validation. The TSNIC also employs a generic parse-encrypt-encapsulate pipeline to reformulate a packet to hide its content within a standard IPSec Encapsulating Security Payload (ESP) frame. Figure 3 illustrates how this process operates between any two communication ports IN and OUT.

Incoming Ethernet frames follow the green path through the TSNIC with their payloads temporarily stored in a data buffer accessible to multiple plugins. An internal random number generator is used to prepend an 8-byte random value to the frame to add entropy into short messages; A 2-byte length field is also appended to the frame. The new resulting frame forms an *enhanced payload*. The enhanced payload and the associated Ethernet/IP/Protocol header from the original frame are treated separately. The enhanced payload is concurrently encrypted and/or parsed using the associated plugins (c.f. Sections III and IV). The encryption plugin simply encrypts the entire enhanced payload. If the parser validates the packet, a signal (OK) causes the encrypted enhanced payload to be assembled into an IPSec ESP packet using the header information from the original packet. If the resulting payload is larger than a single frame, the payload is sent in two IPSec ESP packets, using the ESP sequence numbering to label the packets for decoding at the receiver.

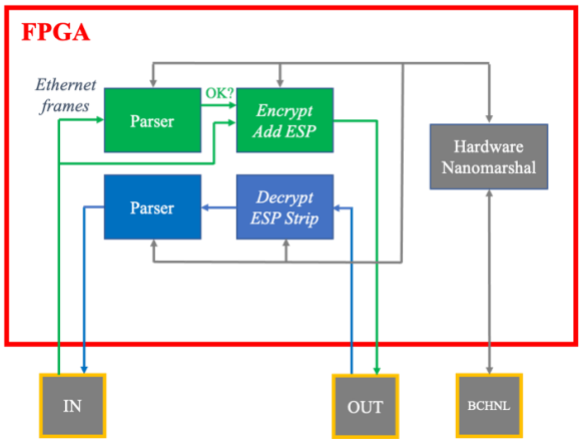

Figure 3. TSNIC Datapath

At the receiver, only incoming IPSec ESP packets are accepted by the TSNIC via the blue path in Figure 3. The encrypted enhanced payload is stripped out of the ESP packet and the header information is also separated out. The encrypted payload is decrypted using the AES plugin. If parsing of the resulting payload succeeds, the header information and decrypted payload are used to re-construct the original source ethernet frame which is then passed to the receiver.

Any parsing algorithm specific to a particular tactical mission can be used in the encapsulation process. The parse-encrypt-encapsulate pipeline can be instantiated across any communication path within the TSNIC between two interfaces: PCIe to GigE, GigE to GigE, and GigE or PCIe to CAN/J1939/1553. This is made possible because all communication paths within the device are rendered into the standard AXI-stream representation, which can be consistently buffered with FIFO's and passed between plugins at will.

6. PROTOCOL HANDLING OPTIONS

All the plugins described thus far operate seamlessly on UDP traffic: Since there is no coordination between sender and receiver, simply dropping packets if they fail to decrypt or parse is a highly effective control mechanism.

For TCP and high-level protocols, simply dropping packets offers several challenges and options. TCP attempts to reliably deliver an entire message, broken into multiple packets, within a single session. To parse large files and message transfers, it is therefore necessary for parsing engines to keep track of the beginning and end of each message, using information contained in the packet headers, and continue to parse across breaks in the transfer that result in multiple Ethernet frames. To parse large files, it is possible to either parse the stream on the fly, and close the session if parsing fails, or store

and forward messages in their entirety. Recall that the TSNIC provides a large on-board RAM, outside the FPGA chip boundary, to facilitate store and forward options. This RAM can be viewed as an extension of the FPGA trust-boundary provided that all information contained within it is encrypted and decrypted *within* the trust boundary.

TCP is extremely belligerent, as one would expect, in attempting to deliver a message. If any single packet in a large transfer is dropped, TCP will continually timeout and resend that packet in futile, continually intercepted and dropped, reattempts to deliver the entire message. Only after exhaustive attempts, will the connection eventually timeout and close. Unfortunately, this has several implications: the session can remain open for several ***minutes*****,** causing any service receiving packets to waste resources while the client engages in repeated attempts at retransmission, consuming bandwidth. To avoid these delays and overheads, when dropping a packet, it is possible to generate, in hardware, a TCP RST message to the receiver informing it to close the connection, thereby allowing it to free resources and proceed immediately. This technique is commonly used by firewalls to close problematic connections.

There are two options on how to handle the sender: either to generate an "alert" message -- assuming that the client is legitimate with its channel intermittently compromised -- or keep quiet deliberately forcing the client to waste resources. The former is most appropriate to embedded situations where the client is being tampered with through some pre-installed implant; the latter is valuable in a case where there is Internet connectivity, and it is desirable not to disclose detection information to the client.

To ensure that an RST message cannot be abused as a malicious attack, there are harsh constraints on its use: it must carry a legitimate sequence number within an existing session and therefore lie within an existing sequence. There are multiple ways to achieve this. For example, to repurpose the existing packet (i.e., the packet being dropped) as the RST message or, to generate a completely new RST message in hardware, copying only the needed information fields to cause the RST to operate correctly from the existing message. To use the first technique, the TCP payload associated with the packet being dropped is truncated and removed, its Ethernet- and IP-header remains unchanged, IP and TCP checksums are regenerated, and the FCS is regenerated – all in hardware. This has the benefit that IP-header options may exist and remain intact, while TCP-options are rendered safe. This option therefore allows IP-security options to be used, if they are desired, however, it presents an opportunity to inject malformed options. The alternative is to construct a completely new TCP RST packet in hardware, taking only the necessary fields to affect the reset operation from the packet being dropped. This higher-level of assurance is more secure and eliminates IP and TCP options to render them safe. Obviously, there are a host of intermediate alternatives between these extremes.

One further option that we have employed is to completely avoid TCP and only allow UDP traffic. Obviously, this has the disadvantage that in general there is no assurance of delivery for the sender and large messages must be segmented in some other manner; Generally, these attributes would be handled by the TCP SYN/ACK handshake. However, it is possible to arrange an alternative hardware signaling – that we term a *turnstile* [11] – to notify a sender that outgoing data and files have been received intact. Unlike TCP this mechanism does not require *any* communication to transition from the receiver to the sender. It allows the

TSNIC to operate as a diode or gateway, protecting a sensitive network or mitigating information leaks, but yet provide reliable file and data delivery.

7. HARDWARE NANOMARSHAL

In a previous paper we have described a *software Nanomarshal* technology developed to support cross-domain applications [13]. That technology operates on System-on-Chip devices, such as the Xilinx Zynq and UltraScale MPSoC, that combine multiple processors, with on-chip FPGA, and a diverse variety of peripheral interfaces. The software Nanomarshal provides the ability for a container, with a designated identifier (id), to be *created*, *started*, *interrogated*, *stopped*, and *destroyed* at runtime using the Open Containers Initiative (OCI) compliant management interface.

Taking these ideas one step further, the TSNIC employs a *hardware nano-marshal* that allows *hardware containers* -- developed and validated in C and rendered into hardware through HLS -- to be instantiated and managed from within the FPGA as illustrated by the grey box in Figure 3.

Recall that all communication interfaces inside the TSNIC are treated uniformly as AXI-streams. The Nanomarshal manages the FPGA as a collection of *partitions*. Partitions are set in place astride each of the input and output interfaces, as illustrated in Figure 3, to form a generic harness. Using an advanced technique termed *partial reconfiguration*, the Nanomarshal can dynamically insert and delete hardware containers into these partitions, tearing down live streams and connecting streams into the new containers on-the-fly. The plugins described in previous sections are all encapsulated as containers, using an automated HLS workflow, allowing them to be either present, or absent i.e., replaced by a container containing just an AXI-stream *wire*. For example, when used simply for encryption acceleration, the parsing engine is absent; when used solely for data validation, the encryption and encapsulation engines are absent.

A container may wrap a working hardware implementation of a particular algorithm with additional information. For example, the AES encryption plugin, when wrapped as a container, includes an AES-256 encryption key in a format that includes offline key-expansion to optimize performance. Re-keying can thus be achieved, through partial reconfiguration, by dynamically replacing the encryption container.

Though currently the Nanomarshal operates with predefined containers set within its harness, it is possible to employ either an unused PCIe or GigE interface to form an out-of-band backchannel (BCHNL) as illustrated in Figure 3. This channel is completely separated from normal traffic flow traversing the TSNIC and used only to manage the internals of the device from a separate air-gapped network. This addition is the focus of our existing work and would allow the TSNIC to function as a High-Assurance Guard (HAG), able to adaptively adjust its behavior to perceived threat level. These ideas build upon a Simple Network Management Protocol (SNMP) backchannel already available on our other products.

8. SHAPING THE ATTACK SURFACE

One unfortunate aspect of modern network protections is that they tend to be focused on the boundary of an installation, as illustrated conceptually in Figure 5(a) as a circle around a protected network.

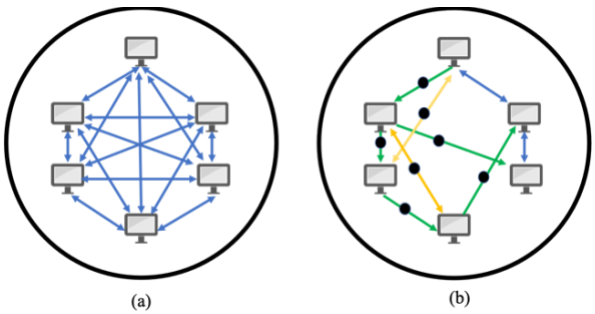

Figure 4. Shaping the Attack Surface

Recall that there are many methods to breach boundary protections: supply chain interdiction, unanticipated connections, transient connections, trusted insiders, zero-day exploits, and persistent implants to name but a few. Once behind the boundary, all connected systems form an attack surface that is directly accessible and vulnerable, moreover, communication is often in the clear without encryption. Even when link-level encryption is used, exploits may transition, in encrypted form, to unprotected software stacks on the other side of encryptors.

For embedded systems on military vehicles, there are few locations that can serve as such a boundary anyway, nowhere to host complex intrusion detection systems, and no time to act upon intrusions using conventional CERT-like investigations: All defense must, of necessity, be automatic and immediate. Consequently, hardening the attack surface to make intrusions difficult to perpetrate within the timescales of a tactical mission, is a more practical alternative.

Figure 5(b) shows how to achieve this conceptually by placing TSNIC appliances at the endpoints on selected links (black dots) and using them to impose validated one-way (green) [9] or bi-directional (gold) traffic flow. This results in an attack surface shaped to protect high-value assets. Traffic over these links is continuously verified through parsing and a multiplicity of encryption keys can be imposed, on differing time-schedules, via an out-of-band backchannel.

9. VIRTUAL ISOLATED NETWORKS

When multiple TSNIC's are connected in matched sets, they transparently form a *hardware overlay* -- shown in Figure 5 – that we term a Virtual Isolated Network (VIN) [10]. A VIN allows any group of devices, computers, or networks to inter-operate over the Internet, while being completely isolated from the rest of the Internet. Attached systems can be anywhere in the world, connected to any network, so long as there is one wired connection into each TSNIC from the Internet. Once connected into a VIN, each system can only communicate with other systems within the same VIN – effectively creating a "virtual air-gap" around the VIN that mitigates malicious intrusion and continuously validates traffic.

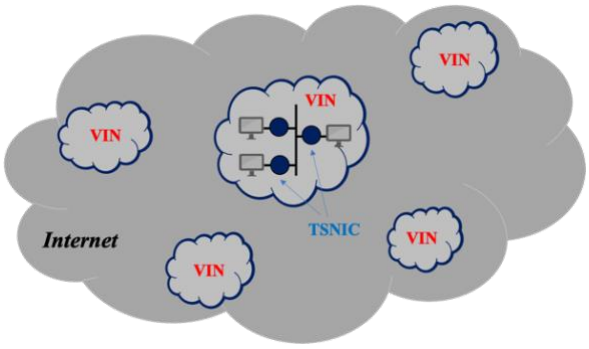

Figure 5. Virtual Isolated Network (VIN)

Since no communication from the VIN to other hosts on the Internet is possible, a VIN is the appropriate location to house valuable data that must be shared within it: cloud-based databases, intellectual property, industrial manufacturing data, maintenance data, or private personal information (PPI). In consequence, the TSNIC provides a distributed alternative to accomplish the goals for which an air gap was conceived.

Obviously, a collection of TSNIC devices forming a VIN is not as inherently secure as an air-gapped network: There is still a connection to the Internet and users rely on the *hardware base-of-trust* provided by the TSNIC logic for security. However, the VIN addresses the needs of organizations that seek the middle ground between a completely air-gapped network and one that is directly connected to the Internet without protections – a middle ground that can reap the benefit of cloud-based services with lower risk.

10. CONCLUDING REMARKS

The TSNIC's military utility is to harden *any* network segment with real-time hardware verification and protection – both within military vehicles and more broadly, military installations. In a previous paper [6],

we have also described how the TSNIC is used to directly monitor ground vehicle buses -- akin to a logic analyzer. In that use case, the device is directly attached to the J1939 bus and monitors a patched, instrumented control system binary, providing high-resolution timing and detailed analysis of execution traces.

As an all-hardware device, the TSNIC raises the barrier to intrusion, increases attacker-workload, and mitigates reverse engineering. It is not a replacement for boundary defenses but rather complements these protections and can be used anywhere in the network hierarchy, shaping the attack surface over which an adversary must operate.

Internally, the TSNIC employs reusable network interface and encryption components but employs custom parsing elements realized automatically though High-Level Synthesis. This combines the flexibility of software with the security and performance of hardware. Hardware Nanomarshal technology, built around partial reconfiguration, extends the flexibility of FPGA-based security allowing on-the-fly changes to security posture in reaction to perceived threats.

REFERENCES

[1] The IP Commission Report, The National Bureau of Asian Research, 2013, updated in 2017 and 2021.

[2] C. Adams, “HUMS Technology”, Aviation Today, May 2012. https://www.aviationtoday.com/2012/05/01/hums-technology/

[3] M.C. Carter, “Post Implementation CBM Benefit Analysis – U. S. Army AH-64D Apache Helicopter Main Transmission Accessory Sprag Clutch Endurance Project”, Annual Conference of the Prognostics and Health Management Society 2013.

[4] P. Shanthakumaran, “Usage Based Fatigue Damage Calculation for AH-64 Apache Dynamic Components”, The American Helicopter Society 66th Annual Forum, Phoenix, Arizona, May 11-13, 2010.

[5] D. Kushner, “The Real Story of STUXNET “, IEEE Spectrum, Feb 2013.

[6] 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.

[7] Jason Dahlstrom and Stephen Taylor, "System-on-Chip Data Security Appliance and Methods of Operating the Same", U.S. Patent 10,148,761, Dec 4 2018.

[8] Jason Dahlstrom and Stephen Taylor, "Endpoints for Performing Distributed Sensing and Control and Methods of Operating the Same", US Patent 10,440,121 , Oct 8 2019.

[9] Jason Dahlstrom and Stephen Taylor, "System-on-Chip Data Security Appliance and Methods of Operating the Same", U.S. Patent 10,389,817, Aug 20 2019. *Diode Continuation Patent.*

[10] Jason Dahlstrom and Stephen Taylor, "System-On-Chip Data Security Appliance Encryption Device and Methods of Operating the Same", U.S. Patent 10,616,344, Apr 7 2020. *VIN Continuation Patent.*

[11] Jason Dahlstrom and Stephen Taylor, "Hardware Turnstile", U.S. Patent 10,938,913, Mar 2, 2021.

[12] Jason Dahlstrom, James Brock, Mekedem Tenaw, Matthew Shaver and Stephen Taylor, "Hardening Containers for Cross-Domain Applications," MILCOM 2019, Norfolk, VA, USA, 2019, pp. 1-6.

[13] B. Nicholas, SSL Hardware Hiding: Increasing the Security of OpenSSL Through Tightly Coupled FPGA Hardware, M.Sc. Thesis, Thayer School of Engineering at Dartmouth, Nov. 2017.

[14] A.V. Aho, R. Sethi, J.D.Ullman, “Compilers”, Addison Wesley, 1988.

[15] S. K. Popuri, “Understanding C parsers generated by GNU Bison”, Sept 2006. https://www.cs.uic.edu/~spopuri/cparser.htm

[16] Parser Experimentation Library: Thayer School of Engineering at Dartmouth College. https://github.com/lvln/thayer_parsers.