Hermes: Secure Heterogeneous Multicore Architecture Design

Michel A. Kinsy, Shreeya Khadka, Mihailo Isakov and Anam Farrukh
Adaptive and Secure Computing Systems (ASCS) Laboratory
Department of Electrical and Computer Engineering
Boston University
Email: mkinsy@bu.edu

***Abstract*—The emergence of general-purpose system-on-chip (SoC) architectures has given rise to a number of significant security challenges. The current trend in SoC design is system-level integration of heterogeneous technologies consisting of a large number of processing elements such as programmable RISC cores, memory, DSPs, and accelerator function units/ASIC. These processing elements may come from different providers, and application executable code may have varying levels of trust. Some of the pressing architecture design questions are: (1) how to implement multi-level user-defined security; (2) how to optimally and securely share resources and data among processing elements. In this work, we develop a secure multicore architecture, named Hermes. It represents a new architectural framework that integrates multiple processing elements (called tenants) of secure and non-secure cores into the same chip design while (a) maintaining individual tenant security, (b) preventing data leakage and corruption, and (c) promoting collaboration among the tenants. The Hermes architecture is based on a programmable secure router interface and a trust-aware routing algorithm. With 17% hardware overhead, it enables the implementation of processing-element-oblivious secure multicore systems with a programmable distributed group key management scheme.**

I. Introduction

Despite the usefulness of abstraction in computer architecture, it fails to establish robust security guarantees across the different layers [1]. Furthermore, in most systems-on-chips, with heterogeneous processing elements, most programs share resources with little to no protection between them.

A. Security Problem

Cybersecurity research generally deals with software security vulnerabilities with a major assumption of a secure underlying architecture. Hardware security primitives, on the other hand (e.g., trusted platform modules (TPM), physical unclonable functions (PUF)) are scarcely used to build secure systems. Prevalent use of commercial off-the-shelf (COTS) software and hardware solutions for general-purpose system-on-chip (SoC) architectures is giving rise to a new set of security challenges. In a multicore smartphone chip for example, complex runtime interactions between processors running untrusted applications can sometimes circumvent the built-in security guards and access memory blocks in encrypted sections of the phone. These systems are vulnerable to a variety of software-centric attacks, such as spyware, trojans and viruses as well as hardware-centric attacks such as channel attacks and hardware trojans [2].

In System-on-chip (SoC) design with system-level integration of large numbers of heterogeneous processing elements, the cores are connected together using a variety of fabric interconnect technologies. Network-on-chip (NoC) has emerged as the *de facto* communication fabric in large multicore architectures [3]. Unfortunately, since all the direct interactions between processing elements occur in the NoC, it presents a major attack surface. In conventional multicore heterogeneous systems, processing elements or tenants (i.e. IP cores) and software may have varying levels of trust. For example in current SoC-based mobile phones, many concurrently running *apps* significantly impact each other's execution and result in potentially harmful consequences for the user [4].

B. Threat Models

Computing system attacks can be classified as: (1) software (virus) attacks, (2) invasive attacks using micro-probing techniques, and (3) side channels attacks, where valuable system information is inferred from runtime power consumption, timing information, electromagnetic signatures and I/O activities. NoC-based heterogeneous multicore systems can be vulnerable to all three types of attacks [5].

The Hermes security model is designed to guard against the following attack scenarios:

1) On-Chip Denial of Service (OC-DoS) attacks: system performance is degraded by injecting a deluge of useless packets into the network.
2) Virtual Channel (VC) attacks: shared VCs can be plowed, allowing malicious flows to build their packet contents out of other flows' residual data.
3) Physical Memory attacks: traditional security features built in the Memory Management Unit (MMU) or the Direct Memory Access (DMA) are bypassed or the address space layout randomization (ASLR) protection is circumvented.

Therefore, it is important to design an integration framework that supports multi-level user-defined security protocols, isolates hardware subsystems and code, and maintains optimal sharing of computing resources and data amongst the tenants. In this paper, we propose and develop such a framework where different trust level cores can be integrated onto the same chip. The key contributions of this work are (1) a processing-element-oblivious secure network interface architecture, (2) a programmable, efficient and distributed group key management algorithm, and (3) a hardware-supported security-aware on-chip routing.

II. Related Work

Many hardware-based security techniques have been proposed in recent years [6]. For most multicore systems-on-chip, a secure core at each node is unnecessary. In these systems, mixed criticality or multi-tenant/multi-trust computation is often the design choice [7]. Different solutions have been proposed at different design abstraction levels, from gate-level description [8] to system virtualization [9]. Wassel *et al* implemented a non-interfering scheme for secure NoC in SurfNoC [10]. Sajeesh and Kapoor have highlighted in [11] some of the advantages of implementing security policies at the network interface level in NoC based systems for secure communication among such IP cores. In [12], Porquet *et al* introduce a solution for co-hosting different protection domains or compartments on the same shared memory multiprocessor SoC using a NoC architecture. Our proposed framework addresses both the hardware and software components of multi-tenant execution. It allows system designers to define and enforce execution communication rules for both secure and non-secure cores or software at the on-chip communication layer. Previously proposed secure processors [1] are still supported in our design framework since the security protocols are not bound by the processor core type. The design of our group key management scheme is informed by the model of attacks highlighted in [13]. In this model, if a secure processor core is used at a processing site, the system designer can bypass the network interface security module. In such cases, the traffic coming from the processor is treated as a non-secure communication from the point of view of the on-chip network security protocol. In [13], an Authenticated Key Exchange amongst a group is explored. Using group keys allows a message to be sent to multiple recipients without having to pay the cost of encrypting the data multiple times.

III. Security Policy

Hermes is a secure multicore computing architecture framework. It reduces the system attack surface by creating a virtualization layer that isolates compute threads based on system and user defined trust levels and security policies.

A. Process Isolation via Hardware Virtualization

In current SoCs, the way to schedule tasks to processing nodes, generally makes it difficult to reason about the run-time interactions between functions of different trust levels, especially in the absence of hardware-level support. This procedure often leads to ad-hoc execution modes where trusted or untrusted software could be running on both trusted or untrusted hardware. Figure 1 shows a set of applications with mixed security being mapped onto a mixed security hardware. **Hermes** achieves both hardware and software views of secure processing by grouping processors into physical zones called *wards* and virtual logical zones called *islands*.

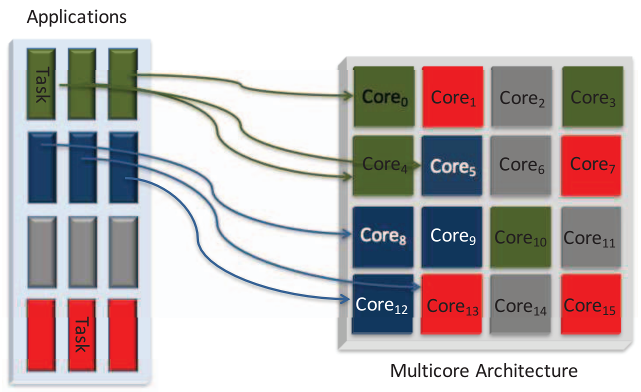
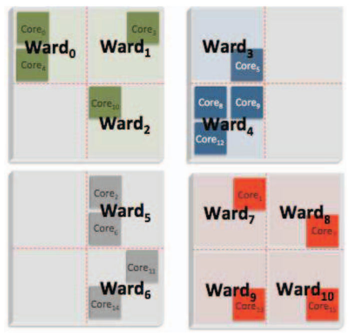


Fig. 1. Trusted/untrusted applications running on trusted/untrusted cores. Different trust levels are illustrated by different colors (e.g., red represents the least trusted program or core).

First, the on-chip processing elements are divided into *wards* that are identified and formed at system integration time, based on IP or processing element provenance. *Wards* are created to help negotiate the security keys used to create the *islands* in a trusted manner. Because the security level is inherent to the IP origin, the *wards* remain constant throughout the chip's lifetime. The chip is divided into physical quadrants, and, within each quadrant, nodes of the same security level make up one *ward*. Hardware security is divided into highly trusted, trusted, untrusted, and unknown levels. Each *ward* has a representative node/module, called the *anchor* node, specified and selected during system integration stage. The anchor node has a table containing the reachability and security information of the other *anchor* nodes and nodes in the same *ward*. Figure 2 shows the *ward* grouping of the illustrative mixed security heterogeneous architecture presented in Figure 1. The anchoring of *wards* provides a simple method of node discovery and key distribution without requiring a full list of node keys at each node.

Second, the processing nodes of Hermes are virtually grouped into logical zones called *islands* based on their static or runtime security characteristics. These nodes can be either secure processors or non-secure processors, as well as any combination of hardware processing units with varying levels of trust. Figure 3 shows an illustration of a virtually partitioned view in the baseline multicore chip, based on trust level. Kernel or user applications are assigned a security abstraction: trusted and untrusted, that creates four basic island security levels namely: (1) trusted hardware and software; (2) trusted hardware and untrusted software; (3) untrusted hardware and trusted software; and (4) untrusted hardware and software. Each node is assigned to an island based on the applications running on them and their IP trust credentials. Islands allow keys to be managed at runtime to maintain isolation, as tasks are moved between the processing units.

One of the key innovations of **Hermes** architecture is that the physical *wards* and the virtual *islands* are decoupled, hence making the node placement decisions independent of the processing cores' security needs. This allows for efficient on-chip routing and node grouping.



Ward	Anchor	Members
0	$Core_0$	$Core_4$
1	$Core_3$	
2	$Core_{10}$	
3	$Core_5$	
4	$Core_8$	$Core_9$ $Core_{12}$
5	$Core_2$	$Core_6$
6	$Core_{11}$	$Core_{14}$
7	$Core_1$	
8	$Core_7$	
9	$Core_{13}$	
10	$Core_{15}$	

Fig. 2. Illustrative case of *ward* groupings with associated *ward* table.

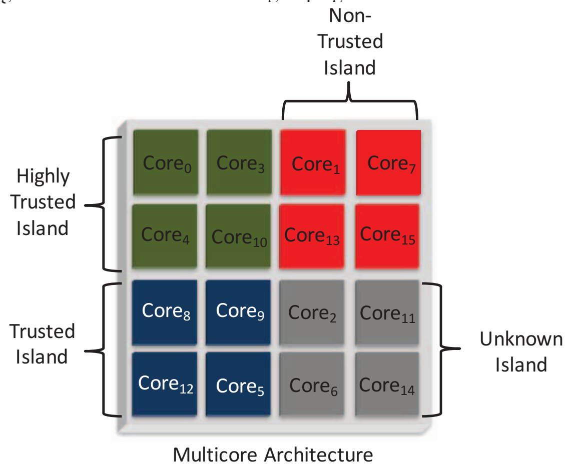


Fig. 3. Hardware virtualization through trusted, untrusted and unknown island partitioning.

B. Enhanced Programmable Memory Access Management

In multicore systems, the implementation of on-chip security policies is typically handled by the memory management unit (MMU) through (1) application memory management, (2) operating system memory management, and (3) hardware memory management. The MMU is placed between the processing element and the memory subsystem where it translates virtual addresses into physical addresses and performs access right validations. All of the memory management safeguards, however, stop at the processing node boundary.

The **Hermes** framework extends the MMU protections and security policies beyond the node boundary into the NoC layer while still applying conventional system memory management techniques at the node-level. The **Hermes** architecture, is an interconnected network of nodes where the physical memory is distributed. A portion of the total on-chip memory is allocated to each processor node in a *Non-uniform Memory Access* (NUMA) style. Figure 4 shows the new enhanced system node organization. The MMU not only provides local memory protection guarantees for processing requests, serviced at the node-level but also guarantees their secure routing when requests need to traverse the on-chip network by sending processor *id* and access code *ac* along with the messages.

On a ***load*** or ***store*** miss at the processing element, the higher order bits in the address are used to locate the physical location of the memory block being addressed. During packetization at the source, the processing element *id* is added to the address and/or data with an access code (shorter version of island key). The packet is then encrypted using the island key or the destination master key. At the remote node, the security layer checks that the process or processor making the request belongs to the appropriate island. After depacketization, the local router sends the source PE *id*, the address, the *access code* (AC), and the data if it is a write operation, to the memory module. The lower bits of the address are used to index into the *Access Code Table* (ACT) to check that the PE is part of the island authorized to access the memory block. In parallel to this, memory boundaries and access code are fetched from *Base Table* (BT) to verify that the PE protection key matches the current access code value associated with the memory block being accessed. The AC, in addition to the Base and Limit registers, helps preserve forward and backward secrecy by being updated on changes in island membership , even over the same memory range. Figure 5 illustrates the memory access control logic.

IV. Security Mechanisms: Distributed Island Key Management Approach

The **Hermes** system uses an efficient dynamic key management protocol to manage and isolate various trust levels with low hardware-overhead. The protocol generates island keys based on the processing elements' and applications' trust levels.

For this type of distributed key management system, there are three general approaches for handling the distribution: (1) store the full list of public keys at each node, (2) store only neighbor's key at each node and (3) divide the nodes into clusters, where one node in the cluster stores the public keys of the other clusters, while remaining nodes in the cluster only store the lead node's public key. For practicality and security purposes, we implement the third approach: a distributed and coarse-grained method, where nodes only store certain public keys.

During application mapping, initial public keys are created using the Diffie-Hellman protocol. As the application data is placed onto the nodes, public keys are stored on *anchor* nodes. Each *anchor* node has a table containing the public key list of the other *anchor* nodes and the nodes in its *ward*. The notation for node keys is as follows: the public key of a node i is denoted K_{P_i}, the private key is K_{Pr_i}, and an island v's key is represented as K_{G_v}. Public requests and responses are denoted $rq_{K_{P_i}}$ and $rp_{K_{P_i}}$, respectively. The process of dynamically creating, expanding and contracting islands happens through the *join* and *leave* operations.

The *join* operation protocol is used to add a node to an island or to start an island. An island is a set of nodes with access to a particular data block. When a new node needs access to a data block, it first must obtain the public key of an island member (referred to as its sponsor) and join the island before requesting access. The sponsoring node will verify the security level of the requesting node and, if the security is

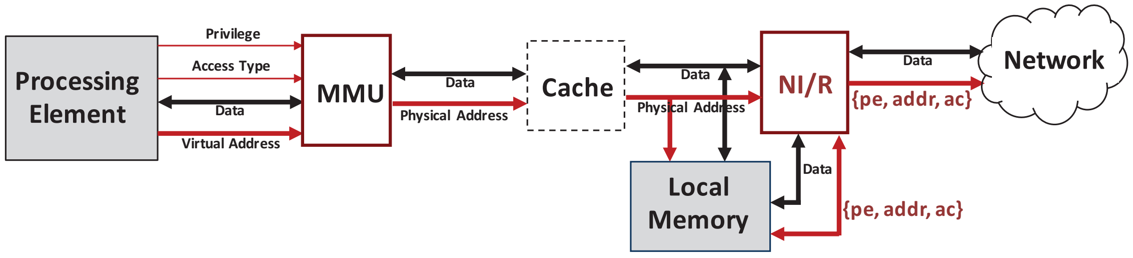
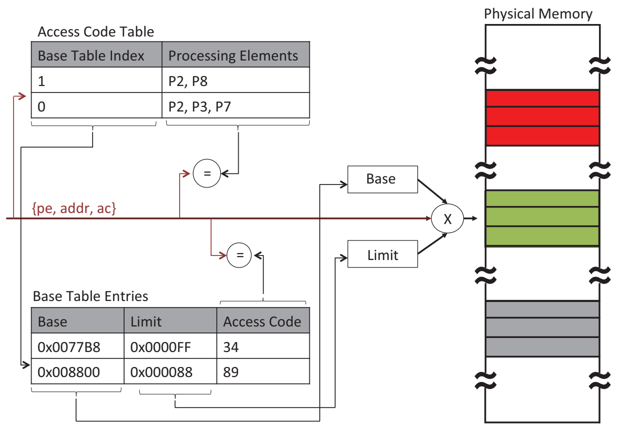


Fig. 4. Programmable enhanced access control at the node boundary through the router network interface.



Base Table Index	Processing Elements
1	P2, P8
0	P2, P3, P7

Base	Limit	Access Code
0x0077B8	0x0000FF	34
0x008800	0x000088	89

Fig. 5. Access key managed memory zones.

sufficient, will initiate the key update process and provide the new node with the island key. The full protocol is described below, assuming requesting node i is in anchor node x's ward and the sponsor node j is in anchor node y's ward.

1) Node i sends an encrypted message to *anchor* node x requesting node j's public key: $E_{K_{P_x}}(E_{K_{Pv_i}}(M_{ix}(rq_{K_{P_j}})))$.
2) *Anchor* node x sends an encrypted message to *anchor* node y requesting node j's public key for node i including i's public key: $E_{K_{P_y}}(E_{K_{Pv_x}}(M_{xy}(rq_{K_{P_j}}, N_i, K_{P_i})))$.
3) *Anchor* node y sends an encrypted message to node i using i's public key containing node j's public key for node i: $E_{K_{P_i}}(E_{K_{Pv_y}}(M_{yi}(rp_{K_{P_j}})))$.
4) Node i then sends an encrypted message to node j using j's public key requesting to join the island assigned to the memory block at j: $E_{K_{P_j}}(E_{K_{Pv_i}}(M_{ij}(rq_{K_{G_v}})))$.
5) Node j verifies node i's access code embedded in the key request message to determine i's trust level. If there is no island, node j creates a symmetric key and sends it to i, $E_{K_{P_i}}(E_{K_{Pv_j}}(M_{ji}(rp_{K_{G_v}})))$. If there is an existing island, node j creates a new island key using a one-way function f such that $K_{G'_v} = f(K_{G_v})$. Node j sends the new island key $K_{G'_v}$ to both i and j sponsors for the island to enable the propagation of updates. Node j also marks its island table to reflect i as a dependent node. When sponsor key update reply comes back then j sends i the new key.
6) When node i receives the message, it updates its island table to make j as its sponsor for the particular island.

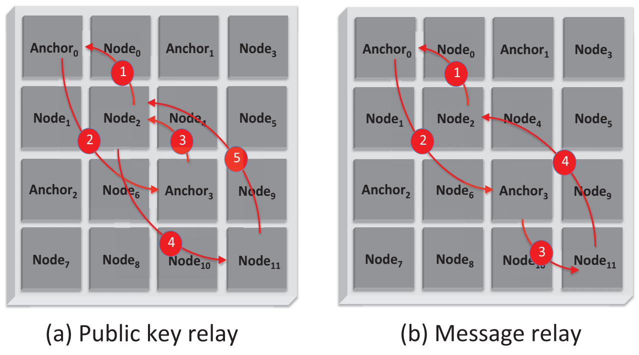


Fig. 6. The two forms of *Join* operation protocol.

Figure 6 shows an illustration of the *join* operation. To cut down the number of messages for establishing or joining a new island, we add the message relay capability, where if two *anchor* nodes are at the same trust level, then the second *anchor* can directly send the island key request to the node in its *ward*, as shown Figure 6(b).

The *leave operation* protocol is as follows:

1) Node i sends to node j, its sponsor for an island, a leave request. j sends the new island key $K_{G'_v}$ to its node sponsor of the island for propagated updates. When the reply from sponsor comes back, then node j removes i from its island table and sends a reply to i.
2) When node i receives the message, it updates its island table for the particular island.

Islands dynamically change when certain execution events occur. For example, when a cache-miss involves a remote access; and the processing node making the cache request is not part of the *island* holding access key to the memory block of interest. This event will lead to an *island join operation*. Dynamic task and thread scheduling, re-scheduling and load re-balancing may also activate idle nodes and create *join operations*. When tasks or threads finish or exit the system, these events may trigger *leave operations*.

Both *join* and *leave operations* may lead to a new island topology. In such cases, the routing connectivity graph of the island needs to be rebuilt and associated routing tables will need to be updated. During the re-keying process, network virtual channels are also reset to prevent VC plowing.

V. Hermes Architecture Support

The design methodology behind the **Hermes** architecture is to provide hardware-supported mechanisms for user-defined or software-defined security rules and their enforcement in heterogeneous multicore systems-on-chip. It effectively decouples the security or trust levels management of processing cores from the integrated SoC.

A. Hardware-Support Implementation

The hardware template depicted in figure 7, can be described as the high-level node module of the **Hermes** architecture. Each processing node has (1) a processor socket to support integration of third party processing IPs, (2) a network interface module supporting our novel secure computing protocol, and (3) a virtual channel router. A major contribution of this work is the complete decoupling of the system level fine-grained security management framework from the processing elements (cores or tenants) and executing software security level. Any processing unit executing any type of software can be installed in the processor socket. Similarly, our security and trust models are oblivious to the on-chip network router microarchitecture.

The hardware modification is constrained to the *Network Interface* (NI) module. The NI is responsible for converting data traffic coming from the local processor and cache subsystem into packets/flits that can be routed inside the network, and for reconstructing packets/flits into data traffic at the opposite side when exiting the NoC. The new network interface has two datapaths: one encrypted and one bypass. The encrypted datapath's functional block description is provided in subsection V-B. The *Bypass path* through the NI module enables the disabling or power-gating of the encryption function at a given core site.

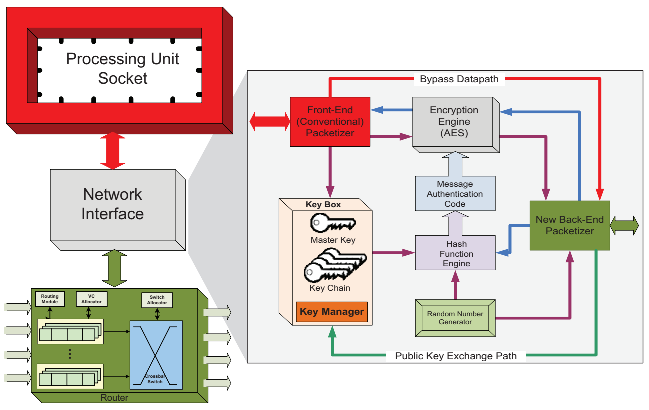


Fig. 7. Hermes Architecture. A new Network Interface is the key component of Hermes. All of the security features of the system are independent of the processing unit.

B. Communication Protocol

Hermes is an encryption-based secure architecture. Its communication protocol is as follows:

1) Local processor unit generates message traffic consisting of memory load and store operations, cache coherency messages and inter-core communication traffic.
2) The Front-End Packetizer converts the processor produced data traffic into packets that can be used for communication with the *Network Interface* (NI).
3) The appropriate communication key is selected by the *Key Manager* in the NI. All of the keys are stored in the *Key Box*, which also contains the *Key Manager* function block. There is a single master key per processing site stored in the *Key Box*. The set of all the other keys is referred to as the *Key Chain*.
4) The message authentication code (MAC) is generated by feeding the key and one random number into the *Hash Function Engine*. MAC is used as session encryption key, so even the same core-pair can have multiple distinctive communication sessions.
5) The processor generated packets are fed into the *Encryption Engine* with the MAC to be encrypted. We implement AES key encryption algorithm for the actual encoding of packets, given its low hardware logic cost. The *Encryption Engine* uses the MAC as a key to encrypt the data using the AES algorithm.
6) The encrypted packet and a second random number are re-packetized by the Back-End Packetizer
7) On the receiving side, packets are first depacketized into encrypted packets and random numbers. The random number is used with the communication key to generate the MAC used to decrypt the packet. The blue directional edges in Figure 7 show the return side data-flow.

C. Trust-Aware On-Chip Routing Algorithm

The on-chip routing is also aware of the logical security islands and tries to either prohibit or limit the traversal of zones by non-member generated traffic. Algorithm 1 describes the added routing function.

D. Illustrative Example

Figures 8 and 9 show the case where the data placement for an application task and read and write operations dynamically create a security group. The initial placement is shown in Step 1. The effect of a read-only request from PE 2 is presented in Step 2. A read/write request from PE 4 causes the group to expand (Step 3). In Step 4, an attempt to read/write by PE 6 through PE 2 fails (red edge), since PE 2 cannot be a sponsor. Sponsorship consists of authorizing other threads/tasks/processes to read or write a copy of data without informing the initial owner. Consequently, PE 6 had to join the security group through PE 1 (Step 5). In Step 6, PE 8 is able to join the group through PE 6 and bypass PE 1's sponsorship.

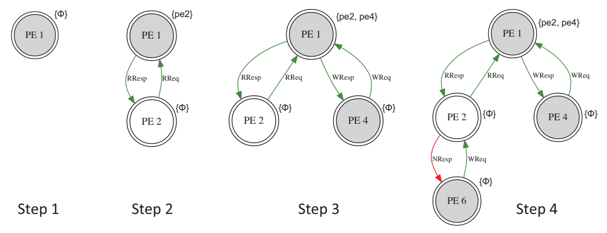


Fig. 8. Group-forming example (steps 1 through 4).

Algorithm 1: Trust-aware on-chip routing algorithm

```
1  Objective
2  Minimize intersections across all routing path sets among islands;
3  A system with a list of processing elements $P = \{p_1, p_2, ..., p_n\}$ ;
4  With the following corresponding list of routers $R = \{r_1, r_2, ..., r_n\}$ ;
5  Find a set of routes $S = \{S(R_1), S(R_2), ..., S(R_n)\}$ ;
6  Such that $\forall\ p_i \in P,\ S(R_i) = \{r_u, ..., r_v\}$ with $1 \leq u < v \leq n$ while
     minimizing $\forall (i, j)\ S(R_i) \cap S(R_j)$. ;
7  The association of a processing element $p_i$ to a router $r_j$ is denoted
     $p_i \triangleright r_j$ (a $p_i$ runtime trust classification depends on the IP core trust
     level and the security of the program running on the core. ;
8  $\forall\ p_i \in P,\quad S(R_i) = \phi$;
9  for $i \in [1, n]$ do
10     for $j \in [1, n]$ do
11         if $(p_i \triangleright r_j)$ then
12             $S(R_i) = S(R_i) \cup \{r_j\}$
13         end
14     end
15 end
16 while $(\forall\ p_i \in P,\ |S(R_i)| > 1$ and $\forall\ (i, j)\ S(R_i) \cap S(R_j) \neq \phi)$ do
17     if $(\exists\ (p_i, p_j)\ |\ S(R_i) \cap S(R_j) \neq \phi)$ then
18         if $((|S(R_i)| > 1) \wedge (|S(R_j)| > 1))$ then
19             $S(R_i) = \begin{cases} S(R_i) - \{S(R_{min}) \cap S(R_i)\}\ where \\ S(R_i) \subsetneq \{S(R_{min}) \cap S(R_i)\} \\ \{r_e\}\ for\ any\ r_e \in S(R_i)\ otherwise \end{cases}$
20         end
21     end
22 end
```

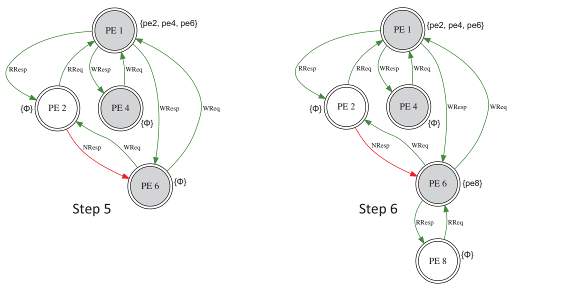


Fig. 9. Group-forming example (step 5 and 6).

VI. System Performance Evaluation

The effectiveness of the protection provided by the **Hermes** architecture to the attack models outlined in Section I-B is measured through flows and processes isolation. Strict isolation of flows and processes based on trust levels effectively creates control access to shared resources: (1) shared memory regions in the network, i.e., virtual channels at the router, and (2) distributed shared main memory modules. The degree of process isolation provided by **Hermes** inversely corresponds to the computing system attack surface, where higher isolation means smaller attack surface.

For the system performance and evaluation, an 8×8 2-D mesh topology design is implemented on a Xilinx Virtex7-XC7VX980T FPGA device. The router has 4 virtual channels and 8 slots per virtual channel. The *Heracles* [14] RTL simulator is used for all the experiments. *Heracles*' injector cores are used to create network and memory traffic. Table I shows the FPGA synthesis results. In the table, BA stands for baseline architecture - *Heracles*' 7-stage in-order RISC processor, AES is the 128-bit version and KS stands for key storage unit. The hardware overhead to fully implement the security features of **Hermes** architecture is only 17%.

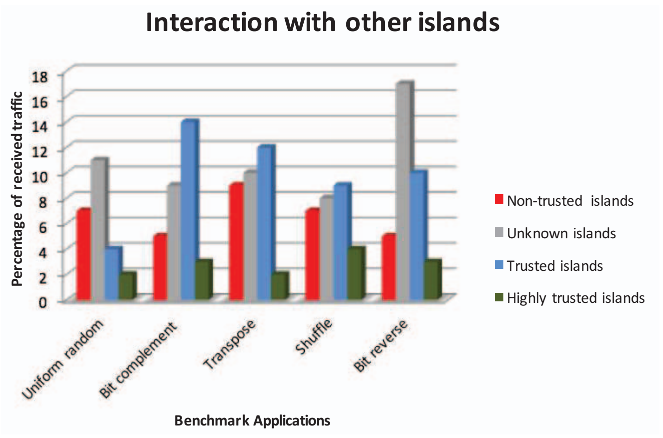


Fig. 10. Percentage of interaction with other islands for different traffic classifications.

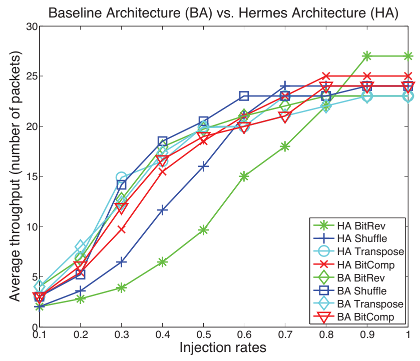


Fig. 11. Throughput per benchmark for the baseline and **Hermes** architecture.

The trust level per core is randomly assigned. The synthetic benchmarks, *Uniform random*, *Bit complement* (BitComp), *Shuffle*, *Transpose*, and *Bit reverse* (BitRev) are used for the various evaluations. Figure 10 shows the interactions among the different traffic classifications. For example in the *Uniform random* benchmark less than 2% of the highly secure traffic is interacting (i.e., sharing physical link or buffer space or memory block) with other types of traffic. The limited interaction reduces the attack surface and ensures a greater security.

Figures 12 and 11 show throughput and latency results of all the synthetic benchmarks on the baseline and **Hermes** architectures. The graphs are color-coded to match the trust-level (e.g., HA BitRev green means that the *Bit reverse* traffic is run in the highly secure mode). Overall, the proposed secure architecture shows no significant performance penalty. In some cases, we actually see a performance improvement at higher injection rates. This improvement occurs because the **Hermes** architecture tries to confine the traffic to their trust classification islands, which decreases path diversity and throughput for low injection rates but reduces *head-of-line-blocking* at high injection rates.

Resource	BA	BA + AES	BA + AES + KS	Hermes
Regs	370829	400498	411626	446782
LUT	321582	347309	356956	376251
Percentage	-	8	11	17

TABLE I
FPGA IMPLEMENTATION RESOURCE UTILIZATION

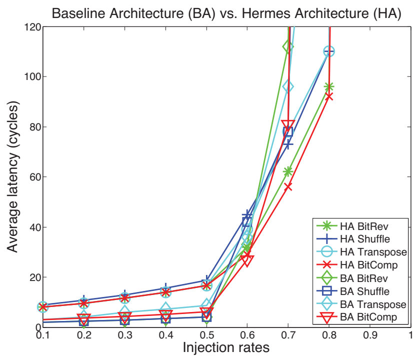


Fig. 12. Average latency per benchmark for the baseline and **Hermes** architecture.

In addition to the synthetic benchmarks, applications from the SPLASH-2 benchmark suite are used to evaluate the Hermes network performance and security enforcement. These applications are simulated as traces. The Graphite [15] distributed x86 multicore simulator is used to generate the traces. Figure 13 shows the throughput results for the **Hermes** architecture on the SPLASH-2 benchmarks as fractions of throughput on the baseline architecture. The performance decline is only 1% to 9% across all the benchmarks when compared to the non-secure baseline architecture.

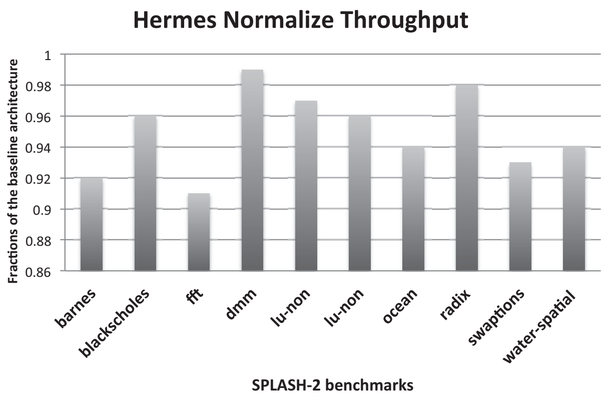


Fig. 13. Throughput per benchmark in SPLASH-2 suite for **Hermes** architecture.

VII. Conlusion

To solve the problem of mixed security of software and hardware components we develop: (1) **Hermes**, a generalized multi-tenant, multicore computer architecture with virtual logical zones to enforce trust levels, and (2) a dynamic key management protocol that lends itself well to the architecture for secure efficient heterogeneous computing. **Hermes** isolates flows and processes based on their trust levels, effectively creating system-level control access to shared resources: (a) shared memory regions in the network, i.e., virtual channels at the router, and (b) distributed shared main memory modules. The architecture monitors processing traffic to verify their compliance to the trust level security policy in effect, in the various sub-parts of the system. The **Hermes** framework is currently limited to offline classification of programs and processing elements. Also the set of rules governing the mapping of programs to cores must be defined beforehand. The architecture has no runtime learning and classification of threats or trust level re-assignments.

Acknowledgment

We sincerely thank Joseph Mohr and Brian Gravelle for their help in exploring earlier versions of the work. We also thank the anonymous reviewers for their helpful feedback.

References

[1] S. Chhabra, Y. Solihin, R. Lal, and M. Hoekstra, “Transactions on computational science vii,” M. L. Gavrilova and C. J. K. Tan, Eds. Springer-Verlag, 2010, ch. An Analysis of Secure Processor Architectures, pp. 101–121.

[2] M. Tehranipoor and F. Koushanfar, “A survey of hardware trojan taxonomy and detection,” *Design Test of Computers, IEEE*, vol. 27, no. 1, pp. 10–25, Jan 2010.

[3] N. E. Jerger and L.-S. Peh, “On-chip networks,” *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–141, 2009.

[4] Q. A. Chen, Z. Qian, and Z. M. Mao, “Peeking into your app without actually seeing it: Ui state inference and novel android attacks,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC’14. Berkeley, CA, USA: USENIX Association, 2014, pp. 1037–1052. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671291

[5] L. Fiorin, C. Silvano, and M. Sami, “Security aspects in networks-on-chips: Overview and proposals for secure implementations,” in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, Aug 2007, pp. 539–542.

[6] X. Wang, M. Tehranipoor, and J. Plusquellic, “Detecting malicious inclusions in secure hardware: Challenges and solutions,” in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, Jun 2008, pp. 15–19.

[7] J. Oberg, T. Sherwood, and R. Kastner, “Eliminating timing information flows in a mix-trusted system-on-chip,” *Design Test, IEEE*, vol. 30, no. 2, pp. 55–62, Apr 2013.

[8] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, “Complete information flow tracking from the gates up,” *SIGARCH Comput. Archit. News*, vol. 37, no. 1, pp. 109–120, Mar. 2009.

[9] J.-Y. Hwang, S. bum Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, “Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones,” in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, Jan 2008, pp. 257–261.

[10] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, “Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. New York, NY, USA: ACM, 2013, pp. 583–594.

[11] K. Sajeesh and H. K. Kapoor, “An authenticated encryption based security framework for noc architectures,” in *Proceedings of the 2011 International Symposium on Electronic System Design*, ser. ISED ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 134–139. [Online]. Available: http://dx.doi.org/10.1109/ISED.2011.17

[12] J. Porquet, A. Greiner, and C. Schwarz, “Noc-mpu: A secure architecture for flexible co-hosting on shared memory mpsocs,” *Design, Automation and Test in Europe Conference and Exhibition*, no. undefined, pp. 1–4, 2011.

[13] J. Katz and J. S. Shin, “Modeling insider attacks on group key-exchange protocols,” in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS ’05. New York, NY, USA: ACM, 2005, pp. 180–189.

[14] M. A. Kinsy, M. Pellauer, and S. Devadas, “Heracles: A tool for fast rtl-based design space exploration of multicore processors,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’13. New York, NY, USA: ACM, 2013, pp. 125–134.

[15] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, “Graphite: A distributed parallel simulator for multicores,” in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, Jan 2010, pp. 1–12.