2015 IEEE Computer Society Annual Symposium on VLSI

*S*ecX: A Framework for Collecting Runtime Statistics for SoCs with Multiple Accelerators

Rajshekar Kalayappan and Smruti R. Sarangi
Department of Computer Science & Engineering, Indian Institute of Technology, New Delhi, India
{*rajshekark,srsarangi*}*@cse.iitd.ac.in*

***Abstract*—We are moving into an era where large SoCs will have a portfolio of different kinds of cores and accelerators. Many of these computational elements might be designed by third parties. In this setting, it is beneficial to collect accurate runtime information such that we can diagnose performance problems, verify and report correctness issues, and collect usage scenarios of third party hardware. This problem is non-trivial if we consider the possibility of defective or malicious elements in the chip. We design an architecture, *SecX*, which helps us collect various metrics of potential interest in a fair, reliable and secure fashion. These logs can subsequently be made available to users, IP vendors, and the system integrator through a trusted third party called the *auditor*. The performance (0.03%), power (1.04W) and area (0.32%) overheads of our scheme are minimal.**

I. Introduction

Designers are increasingly shifting towards incorporating more and more specialized accelerators on chip, instead of having many general purpose cores [1]. These accelerators can perform a variety of tasks such as encryption, compression, XML parsing, network packet processing, and regular expression matching. Such multi-core processors with a host of accelerators are already being manufactured by major processor vendors. Examples of such processors include Intel Ivybridge, IBM PowerEn [2], and IBM Power7.

In the future, we expect to see this trend continuing, and a section of the research community believes [2] that a lot of these accelerators will be designed by third parties. In this context, we note that there will be multiple parties involved in the process of designing an SOC – a *user* defines the system requirements, a *system integrator* designs a system that can meet the requirements by using its own designs and sourcing third party IPs from multiple *accelerator vendors*. It is expected that a heterogeneous system comprising of a myriad of such components will have a host of timing, correctness, security, and performance issues (as reported in [3, 4]). As a result, there is a need for a reporting (auditing) framework to collect information from processors in the field, and make it available to the various parties for offline analyses with user consent. Such a framework finds diverse applications in debugging, malware analysis, and reporting of usage scenarios as explained in Table I. Current approaches are either at a very high level such as performance counters, or are at a very low level such as scan chains and similar trace collection buffers. We desire a hardware solution that operates at the interface of the cores and the accelerators, and logs the interaction between these entities.

An added requirement of such a framework is that it has to be trusted by all the parties (users, system integrators, vendors), and should be tamper proof. This is important as all parties stand to gain from manipulation of logs. For instance, consider a scenario where the system integrator is in charge of the auditing. Now suppose a user's task fails to meet the deadline because of a poorly designed interconnect. The system integrator may instead fabricate logs that implicate some of the accelerators for the delay. This can have an adverse effect on the accelerator vendors from a business standpoint. Note that we assume a trusted foundry in the rest of the paper. There are works that focus on fabrication at untrusted foundries [5]; however, this is beyond the scope of this paper.

Fair, Tamper-Proof, Secure and Reliable Information Gathering: We design such a reporting framework called *S*ecX, that efficiently collects and disseminates useful runtime information to the concerned parties, in a manner that is fair to all concerned, that cannot be illegally influenced by anyone, and that does not disclose anything sensitive.

We define four metrics that capture the framework's requirements: QoS (quality of service), QoE (quality of environment), HoI (hash of input), and HoO (hash of output). The QoS metric refers to accelerator throughput or the duration of jobs, depending upon the context. The QoE metric describes the runtime environment perceived by an accelerator. This includes the memory latency, and available memory bandwidth for the accelerator. HoI is the hash of the input to an accelerator, and the HoO metric is the hash of the output. These four metrics are collected by a third party called the *auditor*, one who is trusted by all the parties (vendors, users, and system integrators). The *auditor* makes the logs available to the various parties on-demand, who benefit from them as shown in Table I. Note that the concept of a trusted third party is not unrealistic. Intel's Trusted Execution Technology (TXT) uses a secure cryptographic processor [7] provided by a trusted third party that both hardware and software developers trust.

The system integrator and the IP vendor incorporate *auditor*-provided macros in their designs. During runtime, the auditing hardware elements co-ordinate to fairly, securely and reliably log the four metrics for each task run on an accelerator. The logs are then transferred to the servers of the *auditor*. The different parties then request the *auditor* for the logs. Authentication and data integrity is ensured by *S*ecX. At the moment, we collect general architectural parameters; however our framework can capture very specific parameters also. In this paper, we look at the fundamental issue of guaranteeing the correctness and consistency of the collected information, and utilizing it gainfully (Table I) to find and report performance and correctness problems.

II. Related Work

The field of DFD (design for debug) architectures is very extensive. Over the last ten years researchers have proposed a lot of techniques to collect debugging information (in the form of logs), and analyze them for defects(representative papers [8, 9]). Chen et al. [10] create accelerator specific debug logs (in hardware) for both correctness and performance problems. Our work is different from this line of research as for us the collection of logs is difficult because of potentially malicious components in the system.

Another related line of research is the efforts made to detect and/or protect against malicious hardware. Hicks et al. [11] highlighted the problem of malicious hardware accelerators. Subse-

978-1-4799-8719-1/15 $31.00 © 2015 IEEE
DOI 10.1109/ISVLSI.2015.22

Table I
NEED FOR RUNTIME INFORMATION REGARDING DIFFERENT ON-CHIP COMPONENTS

Application	Description	Realization Through SecX
Improving Accelerator Designs	Accelerator designers will find usage scenarios useful, e.g., *typical* latencies and throughputs of the accelerator's accesses to the memory. They can use this to improve their pipeline designs – how best to overlap compute/communicate phases to reduce run time and/or power.	QoS and QoE help the *vendor* to study performance under different environment conditions
Analyzing System Throughput	System integrators will find extremely useful the breakdown of delays spent in different parts of the chip. For each task, we can know the time consumed at the cores, accelerators, memory system and the interconnect. This will aid in the identification of bottlenecks and improvement of future designs.	QoS and QoE help the *system integrator* to study performance of different components, both her own and externally sourced
Handling Correctness Issues	The accelerators may give incorrect results. Defects going undetected through post-silicon validation is well documented. The user will like a reporting framework to raise a concern with the vendor.	*User* detects functional bug (possibly offline); reports to *vendor*; uses HoI and HoO to corroborate her claim
Handling Timing Issues	An accelerator may take too long to complete a task. This, again, is something that cannot be caught during the validation phase for many accelerator classes. E.g., the run time of a linear programming accelerator is closely tied to the actual input. In a soft real time scenario, deterministic accelerator behavior is needed to meet deadlines.	*User* detects timing bug (possibly offline); reports to the *vendor*; uses QoS to corroborate her claim
Handling Security Issues	The accelerators can also possibly be deliberately malicious [6] – they can deliberately compute wrongly, compute slowly, disrupt the rest of the chip (DoS attacks) or perform data theft.	Computing wrongly/slowly is proven using QoS/QoE/HoI/HoO, thus discouraging accelerators; DoS and data theft are prevented by gateways (Section III-B)
Enforcing Service Level Agreements	What if the user falsely reports correctness or timing issues? What if the system integrator/accelerator vendor falsely blames the other for a missed deadline? This could damage the reputation of an accelerator vendor / system integrator and hamper her future in the industry.	*auditor* signed logs prevent the parties from making false claims

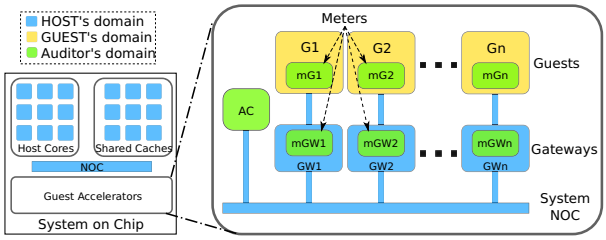

Figure 1. Hardware architecture

quently, there have been several proposals to detect and remove such hardware using a host of techniques such as security assertions, and pattern mining [12]. HybridOS [13] seeks to isolate defective or malicious accelerators at the OS level. Our contribution can enrich these techniques with high quality runtime information.

III. ARCHITECTURE

A. Hardware Overview

A SoC can be viewed as being composed of circuitry belonging to different domains – *host*, *guest* and *auditor*, as shown in Figure 1. The *host* domain refers to all components designed by the system integrator. The system integrator procures third-party accelerators from IP vendors in the form of hard macros. These third party accelerators constitute the *guest* domain. Each guest is allowed access to the rest of the chip only through host-maintained gateways, as shown in the figure. The auditor provides meters (in the form of hard macros) to both the accelerator vendor and the system integrator. The latter two integrate the meters into their designs as shown in Figure 1. The meters and the auditor-comptroller(AC) together form *S*ecX, and constitute the *auditor* domain.

As can be seen in Figure 1, for every accelerator in the system, there are two meters – one embedded in the guest design (mG) and the other in the host gateway (mGW). The need for dual-metering is explained in Section IV-B2. The two meters work together to collect the runtime statistics in a fair, secure and reliable manner. The collected logs are maintained by the central Auditor-Comptroller (AC).

B. Hardware Components

Auditor-Comptroller (AC): The Auditor-Comptroller (AC) is a small structure that handles the functionalities of receiving, validating, processing, storing and serving logs. Logs are maintained for each task run on each guest. The contents of each log are as

Table II
DESCRIPTION OF THE LOGS

Timestamp	16 B	job-id	8 B
guest-id	1 B	[up to 256 guests]	
job-type	1 B	[up to 256 job types]	
QoS.Latency	4 B	QoS.Throughput (B/s)	4 B
QoE	256 B	[$R \times nbins \times 4$ (R=4, nbins=16)]	
HoI	128 B	HoO	128 B
Checksum	32 B	[SHA-256]	
Total	**578 Bytes**	[R = No. of resources]	

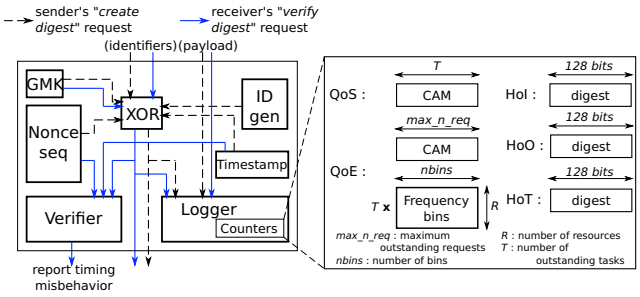

Figure 2. Structure of a meter

shown in Table II. Each log is tagged with a timestamp and a job-id. A hash-based message authentication code (HMAC) (based on 256-bit SHA) of the log is made to ensure integrity and authenticity. The AC has its own limited non-volatile storage to store the logs. Periodically, it encrypts and stores detailed auditor logs in the host's storage device. The ACs in all chips in the field are expected to periodically upload the logs to the auditor's servers. The various parties – system integrator, IP vendor and user – may contact the auditor to obtain the logs.

Meters: The schematic of a meter is shown in Figure 2. The meters are responsible for the collection of the QoS, QoE, HoI, and HoO metrics. We use a CAM array to store the starting time of each active task on the accelerator. When a task finishes, we access the CAM to find its starting time, and subsequently compute its duration. For QoE, we compute a distribution of resource access latencies using binning. Each bin is essentially a frequency count (4 bytes) of the number of accesses whose latencies fall in the particular bin. The number of bins, and the range of values that a particular bin corresponds to is accelerator/resource specific. *S*ecX imposes no restriction in this regard. Multiple outstanding resource accesses are possible – here again a CAM is employed to store the starting time of each resource access request. Additionally, the meter is capable of performing tabulation hashing to compute

HoI, HoO and HoT (hash of all traffic – see Section IV-B3). QoS, QoE, HoI, HoO are maintained only at mGW; HoT is computed at both the meters. The meters and the AC are capable of generating time-stamps, and have dedicated circuitry to perform different cryptographic operations (see Section IV-B5). The parameters and sizes of all the structures are shown in Table VI (in Section V).

Gateways: Gateways (GWs) are host-maintained circuits that form the interface for a guest to the rest of the chip. This allows the host to prevent any unauthorized access or signaling by the guest. Gateways essentially implement resource access lists. They also maintain a small TLB to translate the guest's memory requests (see Section IV-A).

IV. System Operation

A. Functional Layer

1) Job Creation: The accelerators expose a software API that allows users to access them. A function to access a guest has the following form:
int **issue***(int taskcode, void * input, void * output, int ip-length, int op-length)*

When the user executes this function, a host core sends a "job create" message to the guest containing the *taskcode* (task type, as the guest may be capable of performing more than one task), memory addresses of the input and the output, and the lengths of the input and the output segments.

After receiving this message, the gateway configures the access lists that it maintains, and it forwards the message to its meter *mGW*. mGW generates a job-id and records the job start time (required for the QoS metric), and initializes the frequency bins for the QoE metric. It also initializes the input and output hashes. The computation of the input/output hashes requires the mGW to be able to distinguish between guest accesses to the input data, output data and the temporary working area. We assume that the input and output memory regions cannot be used as temporary work-areas, and cannot be modified by other cores while the job is running (due to memory consistency issues). The mGW maintains a table of memory ranges for the input and output data of each active job. It accesses this table to find out if a memory request is reading an input, or writing to an output. It can then add the data read or written by the memory request to the HoI or HoO. Note that these are permutation independent hashes (order of accesses does not matter). Subsequently, the request (timestamped by the mGW) is forwarded to the guest, where its meter verifies that its current time is within τ cycles of the recorded job start time. If the time difference exceeds a threshold, then most likely there is a malicious delay, and the job is dropped.

2) Job Execution: During the execution of a job, the guest reads input data, performs computations on it. We assume the Rigel memory model [1] that enforces software based coherence at task boundaries. For security reasons, the gateway maintains a 32 entry 4-way associative TLB that maintains a portion of the host's TLB. This TLB is used to translate virtual addresses into physical addresses. If there is a TLB miss, then a request is sent to the host core. Now, when a guest wishes to read or write data, it sends a request to its meter (mG). It timestamps the message, and sends it to mGW. The mGW checks for malicious delays by comparing the timestamps, and then updates the HoO if it is a write request. The gateway then proceeds to construct a memory request by accessing its TLB. After the request returns (in the case of a read), mGW notes the time, computes the duration of the request, and updates HoI . Subsequently, the gateway forwards the response to the guest. mG follows the same protocol (same as while sending the message) to verify the timestamps, and check for malicious delays. Along with accessing memory, accelerators can access other resources as well (such as I/O devices). For all communication between the guest and the gateway, mG and mGW both individually maintain a permutation-independent hash of all traffic (*HoT*). These hashes are used to identify any data corruption in the communication.

3) Job Completion: The guest sends a job completion message, containing its computed HoT, to the GW once it is done with its computation. The GW forwards the message to the mGW, who compares its HoT against the one received to detect any data corruption. The mGW then computes the latency of the job, and the average throughput (output bytes/second). It compiles the various components of the QoS, QoE, and the input/output hashes to form the job log. It then creates a digest (SHA) of the log. The log, along with the digest is then sent to the AC via the GW. The AC, upon receipt of the log stores it in its local storage.

B. Security Layer

Owing to the fact that multiple parties are involved, a variety of misbehavior patterns and fairness issues are possible. The different parties can potentially project each other in bad light, which can have significant repercussions. The *S*ecX framework, which all parties trust, must prevent this. In this section, we will extend the functionality explained in the previous section with security and cryptographic aspects [14] to form a complete solution.

1) SecX Setup: We begin with explaining the setup of *S*ecX. When the chip is powered on for the first time (or maybe after burn-in tests), the AC needs to verify that all the meters in the system are genuine. We suggest the employment of Physically Unclonable Functions (PUFs) in the meters. A PUF typically uses some random phenomena such as lithographic variations to compute a unique key for a circuit. We assume that the fabrication facility is trusted, and it can record the PUF based keys for each meter after fabrication (see [15] for more details). To verify that the meters are genuine, the AC can aggregate the random keys, and send them to a remote server, which can validate the keys. After verification, the AC generates a Global Meter Key GMK (shared between all the *S*ecX hardware). It employs the technique of public PUFs suggested in [16] to securely distribute the GMK to all meters. Additionally, the GMK may be periodically updated to improve security. After receiving the GMK, each mGW sets up a nonce sequence (sequence of random values) and communicates this (securely, using the shared GMK) to its companion mG.

2) Handling Malicious Delays: We begin by explaining the need for a dual-metering scheme at the host-guest interface. Guest resource request/response messages cross the guest-host domain. If the latency is measured on the host side, the host can delay the sending of the response to the guest after the latency measurement, in a bid to affect the QoS. If the latency is measured on the guest side, the guest can delay the sending of the request after getting it timestamped (or delay the sending of the response to its meter), in a bid to affect the QoE. To guard against these, meters on both sides are required. The meter at the sending side timestamps the message, and the meter at the receiving side verifies that no undue delay was introduced. It is to the advantage of the sender to send the message to its meter as soon as possible for timestamping.

Similarly, it is to the advantage of the receiver to use the message as soon as possible after timestamp verification at its meter. Any delay induced in between timestamping at the sender, and verification at the receiver, by either party, is detected by the dual-meter system.

Only the meters can encrypt the timestamps because they have a unique key to perform the XOR based encryption (see Section IV-B5). This prevents parties from generating their own timestamps. It must be noted that this scheme only allows the detection of misbehavior. Identifying the guilty party is a harder problem, and is something we are pursuing. Table III shows the complete guest resource access protocol (functional and security aspects). This detection of malicious delays has a minimal effect on performance.

3) Handling Malicious Data Corruption: The host/guest can potentially provide the other with incorrect data, and accuse the latter of incorrect functionality. This is countered by having SecX maintain a hash of all traffic between the host and the guest (HoT). Just like in the case of malicious delays, computing the hash at only one of the meters proves insufficient. The solution is to maintain the HoT at both the host and guest meters, and compare the two HoTs at the time of job completion. A mismatch indicates a data corruption at some point of the host-guest communication. The HoT is not stored as part of the logs.

4) Authentication and Integrity: In our example system, the NoC is in the host's domain, allowing it access to all messages sent/received. This could compromise fairness, allowing for false accusations of a guest being defective. Thus, the logs, before transfer from the mGW to the AC, are encrypted. These encrypted logs are stored in the host's storage. SHA-based signatures, with a secret key known to all the auditor hardware is used in this case.

5) Cryptography Support: We use multiple types of encryption depending upon the security threats, and the time required for encryption and decryption. We use fast XOR based encryption for encrypting the timestamps of memory requests and responses, which are frequent messages. We use SHA encryption for a more infrequent class of messages related to the job completion protocol. Tabulation hashing is used to create digests of the input, output and all traffic, which is off the critical path. Lastly, for the AC's communication with the auditor's servers, we need to use one of the slowest (100+ cycles) yet strongest methods: public key encryption.

- **Simple XOR** : When messages move from one domain (host/guest) to the other, the two parties may induce delays to affect the QoS/QoE in their favor. To check this, a dual metering system, where their attached timestamps are checked (explained in Section IV-B2), is used. Now, these timestamps may be spoofed by the malicious party to escape detection. To protect against this, the timestamps have to be encrypted using a key private to the pair of meters. Also, these cross-domain messages are frequent. Therefore, the encryption mechanism must be light-weight. For this reason, a simple XOR-based encryption, along with nonce sequences is used.
- **SHA** : We employ SHA based encryption when integrity, authenticity and non-repudiation of messages is important. For each sender-receiver pair, a secret key is set up that is shared between the two parties. If party A sends a message to B, the message is attached with a digest of the message using the shared key. This digest allows B to verify that the sender was indeed A, and that the message was not tampered with. SecX employs SHA to sign logs.

Table IV
SIMULATION PARAMETERS

Parameter	Value	Parameter	Value
System Configuration			
Cores	24		
Accelerators	24	Accelerator Types	6
Technology	14 nm	Frequency	3.4 GHz
Shared Elements			
L3 cache	64MB (32 banks)	Main Memory Latency	200 cycles
NoC and Traffic			
Topology	2-D Torus	Routing Alg.	dyn XY routing
Flit size	16 bytes	Hop-latency	1 cycle
Router-Latency	3 cycles		
General Purpose Core Configuration			
Retire Width	4	Issue Width	6
Private L1 i-cache, d-cache			
Size	32 kB	Latency	4 cycles
Private L2 Unified Cache			
Size	256 kB	Latency	12 cycles
Cryptographic Circuitry			
XOR Encryption	4 cycles	SHA Encryption	41.04ns [17]

- **Tabulation Hashing** : Simple tabulation hashing is used by the meters to maintain hashes of the input/output/traffic of the accelerator. Apart from being simple to compute, this hashing technique has the desirable property of permutation-independence. The payload is broken down into fixed size tokens (8 bits). This token is used to index a precomputed secret table (4kB; known to the pair of meters only) to get a code (128 bits). This code is XORed with the current contents of the digest (128 bits) to give the new digest value.
- **RSA** : The communication between the AC and the auditor's servers is encrypted using RSA encryption.
- **Nonce Sequence** : The host/guest may perform a replay attack to induce a malicious delay. We avoid replay attacks by tagging messages with a nonce. The meter pairs share a secret nonce sequence.

V. EVALUATION : OVERHEAD OF AUDITING

In this section, we show that the performance, area and power overheads of SecX are minimal. We consider a system of 24 host cores and 24 guest accelerators with a 32-bank 64 MB LLC. The simulation parameters were derived from the designs of Intel Sandybridge, and IBM Power7 (see Table IV). A portfolio of 6 popular types of accelerators is considered – Table V compares their hardware and software implementations (scaled to $14nm$ using [24]). For the software timing study (Table V), single-threaded applications of the corresponding library were simulated using the Tejas simulator [25, 26]. To make the simulation more accurate, the relevant data was assumed to be in the LLC before-hand (no cold start misses to main memory). For the hardware timing study, we incorporated software implementations of the hardware accelerators in Tejas to give a heterogeneous chip comprising of general purpose cores and accelerators. We used Cadence tools to synthesize the control logic using the UMC $90nm$ standard cell library. Due scaling to $14nm$ was performed [24], and we used Cacti 5.3 [27] to estimate the area and delay of memory structures.

A. Performance Overhead

In this section, we present the performance overhead of performing secure, tamper-proof auditing. The utilization of the logs, be it to detect functional bugs or timing issues, is done offline.

We analyze the different stages of the auditing process and measure overheads. During the job issue phase, the initialization of the various counters and hashes is on the critical path. We estimate

Table III
GUEST RESOURCE REQUEST PROTOCOL

Resource Access			
Request			
(1)	**guest → mG**	$job\_id\|\|resource\_id\|\|payload$	|| is the concatenation operator
(2a)	**mG → guest**	$digest = XOR(GMK, job\_id\|\|req\_id \|\|resource\_id\|\|cur\_time\|\|nonce)$	GMK – key known to all third-party hardware A unique req_id is generated by mG nonce → next number in a secret sequence of numbers shared between mG and mGW
(2b)	**mG**		updates HoT (off critical path)
(3)	**guest → GW**	$digest\|\|payload$	
(4a)	**GW**		verifies if resource access is legal
(4b)	**GW → mGW**	$digest\|\|payload$	
(5)	**mGW**	$XOR(GMK, digest)$	decrypts and checks for timing misbehavior
(6a)	**mGW → GW**	req_id	
(6b)	**mGW**		records req_id, $send\_time$, $resource\_id$ (off critical path)
(6c)	**mGW**		updates HoT (off critical path)
(7)	**GW → resource**	$req\_id\|\|payload$	with a TLB access
Response			
(1)	**resource → GW**	$req\_id\|\|payload$	
(2)	**GW → mGW**	$req\_id\|\|payload$	
(3)	**mGW → GW**	$digest = XOR(GMK, req\_id\|\|cur\_time\|\|nonce)$	
(4a)	**GW → guest**	$digest\|\|payload$	
(4b)	**mGW**		uses $recv\_time$ to update QoE latency distribution (off critical path)
(4c)	**mGW**		updates HoI and HoO (off critical path)
(4d)	**mGW**		updates HoT (off critical path)
(5)	**guest → mG**	$digest\|\|payload$	
(6)	**mG**	$XOR(GMK, digest)$	decrypts and checks for timing misbehavior
(7a)	**mG → guest**	req_id	
(7a)	**mG**		updates HoT (off critical path)
(8)	**guest**		guest uses payload

Table V
ASIC V/S SOFTWARE IMPLEMENTATIONS OF ACCELERATORS

Task	**Details**	**Citation**	**Technology**	**Original**		**Scaled to 14nm**		**Software**	
				Latency	**Area**	**Latency**	**Area**	**Details**	**Latency**
FFT	complex 1024-point FFT	[18]	$0.18\ \mu m$	$3.2\ \mu s$	$0.686\ mm^2$	$2.3\ \mu s$	$0.024mm^2$	FFTW library	$32.72\ \mu s$
Sort	sorting 3969 longs	[19]	$0.13\ \mu m$	$32\ \mu s$	$2\ mm^2$	$17.01\ \mu s$	$0.07mm^2$	libstdc++6	$442.49\ \mu s$
JPEG	compressing 640×480 pixel bmp	[20]	$0.09\ \mu m$	$1.11\ ms$	$0.062\ mm^2$	$0.65\ ms$	$0.004mm^2$	Independent JPEG Group's library	$2.16\ ms$
MD5	512 byte input	[21]	$0.13\ \mu m$	$1.96\ \mu s$	$0.081\ mm^2$	$1.04\ \mu s$	$0.003mm^2$	OpenSSL/Crypto	$9.76\ \mu s$
AES	512 byte input	[22]	$0.13\ \mu m$	$1.25\ \mu s$	$0.015\ mm^2$	$0.753\ \mu s$	$0.0005mm^2$	library	$4.749\ \mu s$
RSA	128 byte input	[23]	$0.5\ \mu m$	$325\ ms$	$3\ mm^2$	$133.7\ ms$	$0.011mm^2$		$258\ ms$

this to be less than 15 cycles. During job execution, each resource access has to go through the dual-metering system. The critical path is the nonce retrieval (163 ps) and the XOR encryption (31 ps) at the sender's side, the transfer to the receiver (30 ps), and the XOR decryption at the receiver (31 ps). This would add a delay of 2 cycles (1 each for request/response) to an LLC access. When the job has completed, the logs are aggregated and signed using SHA. The logs are sent from the mGW to the AC, which stores it appropriately. These tasks can overlap with the host's utilization of the guest's output, and hence, are off the critical path.

Table V shows that software is roughly 2-20X slower on average. The additional delay of 2 cycles represents a worst-case overhead of 4% for a LLC read access (48 cycle, mean access time). However, full system simulations done using Tejas, revealed an average overhead of just 0.03%. The reason for this is as follows: the increase in resource access latency translates to a decrease in accelerator performance only if the parameters of the access (e.g., memory address for an LLC) depend upon the result of some computation. If there is no such dependency, then the access can be overlapped with another access or some computation, thus masking the auditing overhead. The nature of tasks typically accelerated are ones with statically determinable access patterns, allowing for significant overlap.

The NoC bandwidth overhead is minimal because the only usage is the transfer of logs from the meters to the AC. That is, per accelerated task, a mere 578 bytes (see Table II) is transferred.

B. Area Overhead

The overhead of auditing is dependent on the number of accelerators incorporated. The prime contributors to the area are the memory elements (volatile and non-volatile) that are required for the cryptographic operations and for storing/processing logs (in the case of the AC). The other significant contributors are the cryptographic circuits: SHA and RSA. Area estimates are obtained from ASIC implementations of published designs [17, 23]. Table VI gives the area breakup of the *S*ecX hardware. The total additional area is $1.267mm^2$, for a system with 24 guests. Assuming a reference die size of $400mm^2$, the area overhead is 0.32%.

C. Power Overhead

The peak power consumption of *S*ecX is estimated at 1.04W (TDP of 8-core Intel E5-2687W chip is 150W). Cacti [27] was used to estimate power for memory elements. References [29, 30, 31] were used for the major logic components. This estimate is assuming all 24 guests are operational, and the associated meters are working with a 100% duty cycle. In practice, the power consumed would be much lower as firstly, not all guests are simultaneously active, and secondly, the meters are inactive during the guest's compute phase.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented a mechanism for measuring useful metrics (QoS, QoE, HoI, HoO), an architecture for validating this information, and ultimately saving it in tamper proof logs. The

Table VI
AREA ESTIMATION OF AUXILIARY STRUCTURES

Component	Formula	Area Estimate	Details
Major Components of the Gateway Meter			
SHA		$0.003mm^2$	[17]
Memory for Tabulation Hashing Table	$2^8 \times 128bits$		
Memory for QoS	$T \times 8\ bytes$		T:number of tasks guest can run at the same time; R:number of different resources guest makes use of; $nbins$: number of bins in frequency binning
Memory for QoE	$R \times T \times nbins \times 4\ bytes$		
Memory for I/O/T hashes	$T \times 3 \times 1024\ bits$		
Memory for Nonce Sequence	$1024 \times 1\ bytes$		
Memory Total	Taking $R = 4, T = 4, nbins = 16$	$0.010mm^2$	8kB cache (Cacti [27])
Logic		$946.92\mu m^2$	
Sum		$0.014mm^2$	
Major Components of the Guest Meter			
Memory for Tabulation Hashing Table	$2^8 \times 128bits$		
Memory for HoT	$T \times 1024\ bits$		
Memory for Nonce Sequence	$1024 \times 1\ bytes$		
Memory Total	Taking $T = 4$	$0.010mm^2$	8kB cache (Cacti [27])
Logic		$296.98\mu m^2$	
Sum		$0.010mm^2$	
Major Components of the Auditor-Comptroller			
SHA		$0.003mm^2$	[17]
RSA		$0.014mm^2$	[23]
Memory for log processing		$0.010mm^2$	8kB cache (Cacti [27])
Non-volatile Memory for Audit Logs	$NJA \times 586\ bytes$; Taking $NJA = 7000$		NJA: number of jobs audited; refer Table II for log size
Non-volatile Memory Total		$0.664mm^2$	4MB 32nm RERAM [28]
Sum		$0.691mm^2$	
Total Area of Additional Hardware	Taking $num\_guests = 24$	$1.267mm^2$	
Area Overhead	Assuming a base chip area of $400mm^2$	0.32%	

performance (0.03%), power (1.04W) and area (0.32%) overheads are minimal. We wish to extend this work to consider an even larger set of possible attacks, and make minimal assumptions about trusted components.

REFERENCES

[1] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: An adaptive hybrid memory model for accelerators," *Micro*, 2011.

[2] R. Hou, L. Zhang, M. C. Huang, K. Wang, H. Franke, Y. Ge, and X. Chang, "Efficient data streaming with on-chip accelerators: Opportunities and challenges," in *HPCA*, 2011.

[3] M. Birnbaum and H. Sachs, "How vsia answers the soc dilemma," *Computer*, vol. 32, 1999.

[4] E. Haritan, H. Yagi, W. Wolf, T. Hattori, P. Paulin, A. Nohl, D. Wingard, and M. Muller, "Multicore design is the challenge! what is the solution?" in *DAC*, 2008.

[5] "Trusted foundry program." [Online]. Available: http://www.dmea.osd.mil/trustedic.html

[6] M. Hicks, M. Finnicum, S. T. King, M. Martin, and J. M. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *SP*, 2010.

[7] S. L. Kinney, *Trusted platform module basics: using TPM in embedded systems*. Newnes, 2006.

[8] B. Vermeulen and S. K. Goel, "Design for debug: catching design errors in digital chips," *IEEE Design & Test of Computers*, 2002.

[9] D. Josephson, "The good, the bad, and the ugly of silicon debug," in *DAC*, 2006.

[10] Y.-T. Chen, J. Cong, M. A. Ghodrat, M. Huang, C. Liu, B. Xiao, and Y. Zou, "Accelerator-rich cmps: From concept to real hardware," in *ICCD*, 2013.

[11] M. Hicks, M. Finnicum, S. King, M. Martin, and J. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *SP*, May 2010.

[12] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security checkers: Detecting processor malicious inclusions at runtime," in *HOST*, 2011.

[13] J. H. Kelm and S. S. Lumetta, "Hybridos: Runtime support for reconfigurable accelerators," in *FPGA*, 2008.

[14] S. William and W. Stallings, *Cryptography and Network Security, 5/E*. Pearson Education India, 2006.

[15] Y. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security." in *USENIX Security*, 2007.

[16] N. Beckmann and M. Potkonjak, "Hardware-based public-key cryptography with public physically unclonable functions," in *Information Hiding*, 2009.

[17] L. Dadda, M. Macchetti, and J. Owen, "The design of a high speed asic unit for the hash function sha-256 (384, 512)," in *DATE*, 2004.

[18] Y. Peng, "A parallel architecture for vlsi implementation of fft processor," in *ASIC*, 2003.

[19] N. Tabrizi and N. Bagherzadeh, "An asic design of a novel pipelined and parallel sorting accelerator for a multiprocessor-on-a-chip," in *ASIC*, 2005.

[20] M. Papadonikolakis, V. Pantazis, and A. Kakarountas, "Efficient high-performance asic implementation of jpeg-ls encoder," in *DATE*, 2007.

[21] A. Satoh and T. Inoue, "Asic hardware focused comparison for hash functions md5, ripemd-160, and shs," in *ITCC*, 2005.

[22] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen, "Design and implementation of low-area and low-power aes encryption hardware core," in *DSD*, 2006.

[23] Z. Keija, X. ke, W. Yang, and M. Hao, "A novel asic implementation of rsa algorithm," in *ASIC*, 2003.

[24] W. Huang, K. Rajamani, M. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," *Micro*, 2011.

[25] G. Malhotra, P. Aggarwal, A. Sagar, and S. R. Sarangi, "ParTejas: A parallel simulator for multicore processors," in *ISPASS*, 2014.

[26] S. Sarangi, R. Kalayappan, P. Kallurkar, and S. Goel, "Tejas simulator : Validation against hardware," 2015. [Online]. Available: http://arxiv.org/abs/1501.07420

[27] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," HP Laboratories, Tech. Rep., 2008.

[28] X. Dong, C. Xu, Y. Xie, and N. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *TCAD*, 2012.

[29] A. T. Tran and B. M. Baas, "Design of an energy-efficient 32-bit adder operating at subthreshold voltages in 45-nm cmos," in *ICCE*, 2010.

[30] A. Sajid, A. Nafees, and S. Rahman, "Design and implementation of low power 8-bit carry-look ahead adder using static cmos logic and adiabatic logic," in *IJITCS*, 2013.

[31] A. K. Pandey, R. A. Mishra, and R. K. Nagaria, "Leakage power analysis of domino xor gate," in *ISRN Electronics*, 2013.