Providing Accountability in Heterogeneous Systems-on-Chip

RAJSHEKAR KALAYAPPAN, Indian Institute of Technology Dharwad, India and Indian Institute of Technology Delhi, India
SMRUTI R. SARANGI, Indian Institute of Technology Delhi, India

When modern systems-on-chip (SoCs), containing designs from different organizations, miscompute or underperform in the field, discerning the responsible component is a non-trivial task. A perfectly accountable system is one in which the on-chip component at fault is always unambiguously detected. The achievement of *accountability* can be greatly aided by the collection of run-time information that captures the events in the system that led to the error. Such information collection must be fair and impartial to all parties. In this paper, we prove that logging messages communicated between components from different organizations is sufficient to provide accountability, provided the logs are authentic. We then construct a solution based on this premise, with an on-chip trusted auditing system to authenticate the logs. We present a thorough design of the auditing system, and demonstrate that its performance overhead is a mere 0.49%, and its area overhead is a mere 0.194% (in a heterogeneous 48 core, 400mm^2 chip). We also demonstrate the viability of this solution using three representative bugs found in popular commercial SoCs.

CCS Concepts: • **Hardware → System on a chip**; **On-chip resource management**; **Transaction-level verification**; **Bug detection, localization and diagnosis**; *Hard and soft IP*;

Additional Key Words and Phrases: accountability, auditing, SoC, third-party IPs, accelerators, heterogeneous processors

ACM Reference Format:
Rajshekar Kalayappan and Smruti R. Sarangi. 2017. Providing Accountability in Heterogeneous Systems-on-Chip. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2017), 23 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Researchers[1] believe that the systems of the future will have numerous third party IP (3PIP) circuits that perform specialized tasks. Industry is already moving in this direction. IBM has recently decided to offer its Power 9 architecture to system integrators (SIs) such that they can create a large SoC using multiple Power 9 cores, and other components (cores and accelerators from third parties). Other IBM processors such as IBM Power 7 and Power 8, and Power En already incorporate a lot of third party accelerators for performing different kinds of tasks such as regular expression matching, cryptography, compression, XML parsing, and network packet processing. In the domain of desktops and rack servers, Intel Ivybridge and Skylake processors incorporate accelerators

[1]New Paper, Not an Extension of a Conference Paper

Authors' addresses: Rajshekar Kalayappan, Indian Institute of Technology Dharwad, Dharwad, Karnataka, 580011, India, Indian Institute of Technology Delhi, New Delhi, Delhi, 110016, India, rajshekar.k@iitdh.ac.in; Smruti R. Sarangi, Indian Institute of Technology Delhi, New Delhi, Delhi, 110016, India, srsarangi@cse.iitd.ac.in.

1539-9087/2017/1-ART1 $$15.00
https://doi.org/0000001.0000001

for processing JPEG images and VP-8/9 video decoding. ARM processors stand out in this space, because they are arguably the most popular IPs that are licensed to SIs. This trend of "IP reuse" is prudent because designing cores and high performance accelerators are very specialized tasks, and often a single company lacks the know-how and sometimes the intellectual property rights to design all of these modules. Because of area, performance, cost, and time-to-market constraints most companies find it prudent to integrate various third party IPs and create a large SoC. We refer to such 3PIP containing SoCs as heterogeneous SoCs.

The verification of such heterogeneous SoCs is a daunting task [29]. Even with homogeneous SoCs (SoCs entirely designed by the same organization), many bugs escape validation and affect the target application in the field (see the errata document published by Blue Gecko [5]). With heterogeneous SoCs, the problem is further exacerbated. Industry giants such as Texas Instruments [1], Xilinx [7], NXP Semiconductors [2], and Microsemi [6] have all published lengthy errata documents listing various bugs in their commercially available heterogeneous SoCs. The problem classes include bugs in the SI's modules, 3PIP modules such as *Universal Asynchronous Receiver-Transmitter* (UART) controllers not conforming to published standards, other published bugs in the 3PIP modules (such as documented bugs in ARM cores), and cases where the 3PIP module's behavior is ambiguous or unspecified. Thus faulty designs aside, since the different modules are not designed by a single organization, a new source of bugs has to be dealt with – incomplete or ambiguous requirement specification by the SI, and incomplete or ambiguous functionality specification by the 3PIP vendor. There is little standardization in the industry in these aspects [4]. This leads to many corner cases not being clearly understood. Such integration bugs are harder to catch during post-silicon validation [29]. For example, Texas Instruments has published an errata document [1] describing a bug in its integration of an ARM Cortex core in the CC2538 SoC. Furthermore, we also have to deal with the possibility of malicious cores and accelerators [8].

Since verifying and validating heterogeneous SoCs is never *complete*, many bugs make their way to the field. A heterogeneous SoC once commissioned, may therefore produce the wrong results, or may take too long to produce the results. The customer, whose application has suffered as a result, expects to be compensated. As discussed, the malfunction could be the fault of any of the 3PIP vendors or/and the SI. The organizations found responsible for designing the faulty modules, or the SI found responsible for incorrect integration, must then duly compensate the customer. Note that the responsible organizations also suffer a loss in their credibility in the market. Naturally, every organization would like to evade being held responsible for such a malfunction.

This brings us to our problem statement – the achievement of *accountable* heterogeneous SoCs. An accountable SoC is one in which the cause of the malfunctioning can be *accurately* narrowed down to one (or more) defective on-chip components. Accuracy here implies that the faulty components, and only the faulty components, are unambiguously implicated.

The problem of identifying malfunctioning on-chip components in the field is well established – various aspects of it have been studied in the literature. Let us first consider the most common assumption, which is that the SI is completely trusted by all the 3PIP vendors. There are three sub-classes of solutions to tackle this problem. The first class of solutions are from the reliability and Hardware Trojan detection areas. At run-time, SI-designed test circuitry detect if the 3PIPs are not conforming to the expected functional and timing requirements [8, 20]. The second is that the SI tries to deterministically reproduce the bug by simulating the system in a controlled environment with more visibility into the system. We need the user's inputs, and a method of reproducing the bug. This can be done for small circuits such as small micro-controllers or DSPs; however, given the complexity of modern manycore hardware and parallel software, it is impractical to reproduce the bug deterministically. The third approach is to log information in the chip's target environment itself that makes the reproduction of the bug easier [15, 21, 33]. The SI adds on-chip debug hardware that logs the values of critical internal signals, in an effort to capture the state of the system before it failed. In this paper, we concern ourselves with this approach of online logging to aid debugging.

Now, the assumption that the SI is trustworthy cannot be realistically made. The SI is after all another business organization. The possibility of the SI being malicious to further its own interests has been recognized by researchers such as Guin et al. [16]. The IEEE Design Automation Standards Committee has even developed a standard, IEEE 1735 [3], which prescribes how IP vendors can protect their IP design from being maliciously stolen or modified by the SI. Thus, while employing online logging to aid debugging, having SI designed loggers is not fair to the 3PIPs [21]. The SI stands to gain by tampering the logs and shifting the responsibility for its own bugs to the 3PIPs (either unknowingly or maliciously). By the same argument, designating the 3PIPs to do the logging is unfair to the SI. Therefore, a logging solution is required that is fair to all parties involved.

The problem we wish to solve is similar to the one of malicious contractors in the cloud domain [24]. A contractor who performs outsourced compute jobs may maliciously do so in an approximate fashion, so as to maximize its own profits. The customer would like to be able to detect such malice. This is similar to the chip malfunctioning in the field and producing incorrect results. Both problems can be solved by increasing the visibility into the functioning of the compute element (contractor or chip). An aspect of our proposed solution for achieving accountable SoCs is inspired by the idea of *inner state hashes* (see Section 6), used to detect malicious contractors [24].

The aim of this work is to introduce accountability as, not just a desirable, but a necessary property of modern heterogeneous SoCs, and also to introduce one approach of providing this property. We first formally define the problem of accountability, and then prove that authentic logging of messages exchanged by on-chip components designed by different organizations is sufficient to guarantee accountability. We construct a solution based on this premise, and present a thorough design of it, and show its efficacy in the face of some real bug scenarios derived from the errata documents of popular commercial SoCs. We also demonstrate that accountability can be achieved with modest overheads. We believe that our proposed solution is just the tip of the iceberg, and will serve to inspire other researchers. We propose a generic approach to accountability that functions rather efficiently in most cases. A massive scope for further research exists to realize economical accountability solutions under different conditions and constraints.

2 RELATED WORK

2.1 Runtime Solutions for Creating Trustworthy Systems

Abramovici et al. [8] recognized that attempting to detect Hardware Trojans during the pre-silicon verification stage or the post-silicon validation stage, results in incomplete coverage. The main reason for this is that Trojans may be trigger-based. The triggers are typically extremely rare events. The Trojans consequently escape the verification and validation stages. Hence, run-time solutions are required that detect when a 3PIP is not behaving according to specifications. Such solutions are effectively accountability solutions, if it can be assumed that the SI is trustworthy. Similarly, reliability techniques (comprehensive survey by Kalayappan et al. [20]) that are employed to detect hard and soft errors can be used as accountability solutions as well, again under the assumption that the SI is trustworthy.

Vermeulen et al. [15, 33] recognized that to debug modern complex SoCs, it is required that they be observed in their target environment, running their target applications. These works focus on the NoC. They recognize that such information collection can be done at the granularity of transactions between components, rather than adopting a per-cycle approach. This approach not only reduces the sizes of the traces, it also provides the necessary abstraction for the traces to be related to the software running on the SoC. We also subscribe to this strategy of logging information at the

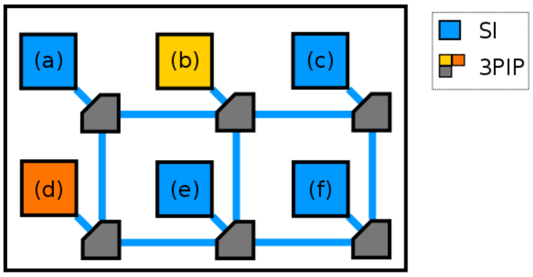

Fig. 1. Model of a 3PIP-containing SoC

granularity of transactions for the purpose of providing accountability. This line of proposals also assumes that the SI is trustworthy and the collected logs are untampered.

Cloud computing enables the owner of a job to outsource it to a contractor, who computes it in exchange for a fee. Now, the contractor may perform an approximate computation to reduce operational costs and therefore increase profits. To detect such malicious computations, simply verifying the final result is insufficient. Instead, looking at the intermediate results as well gives greater insight into the computation done by the contractor, enabling detection of any malice [24]. We adopt a similar approach by basing our offline analysis on the intermediate results produced by each on-chip module as part of executing the customer's application. Additionally, adopting the idea of *inner state hashes*, proposed by Kupcu [24], helps reduce the size of the logs that need to be collected at runtime (see Section 6).

2.2 Tamper Proof Hardware for Collecting Logs

Kalayappan et al. [21] present a scheme to reliably log the interactions between the SI circuitry and third party accelerators. They assume that the SI and the accelerator vendors do not trust each other. They employ trusted third party meters to do the logging. The reliable logs thus collected can be used for a variety of purposes including debugging, analyzing performance bottlenecks, and investigating violations in security.

3 ACCOUNTABILITY: THE PROBLEM AND A SOLUTION PARADIGM

3.1 Model of a Heterogeneous SoC

We begin by describing a generic model of a heterogeneous SoC containing IPs from many different vendors. We focus on NoC based SoCs in this work. Figure 1 shows a descriptive example. The convention followed is that components designed by the same vendor are given the same color. The vendor who integrates the different components is the system integrator (SI). All on-chip circuitry designed by the SI are termed the *host* circuitry (H). In Figure 1, all components in blue constitute H. All other components other than the host are termed as *guests*. In Figure 1, components (b) and (d) are guests, designed by two different vendors. Let the set of all guests be $\mathbb{G}$. Note that the guests cannot communicate with each other directly. Any communication between two guests has to pass through the host. The network-on-chip (or NoC), responsible for providing the network interface to each component as well as routing packets towards their destination, may be part of the host circuitry or may be externally procured, that is, a guest. In Figure 1, the NoC, depicted in gray, is a guest.

When the SI decided to incorporate a particular guest G, it would have agreed upon the conditions of its use with the guest vendor. First, the nature of the inputs to the guest, $IpCond_G$, are decided upon. For example, in the case of a JPEG accelerator, the possible dimensions of the input image may be agreed upon by the SI and the guest vendor. Second, the nature of the output of the guest, $OpCond_G$, is negotiated. This could be the actual result of a computation, as is the case in, say, a cryptographic accelerator. It could also be a property of the result. For example, in the case of a compression accelerator, the desired compression ratio may be agreed upon. Third, the quality of service (QoS) – that is, the guest's latency / throughput while performing a job – $QoSCond_G$ is agreed upon. Fourth, the quality of the environment (QoE) in which the guest operates, $QoECond_G$, is agreed upon. QoE refers to the latency / throughput of H servicing resource requests (e.g., last-level cache requests) made by G.

3.2 The Problem of Accountability

In performing the task given by the user of the SoC, H carries out some computations of its own, and can request the services of the on-chip guests for others. We term such requests as *jobs.*

Definition 3.1. It is possible that the result returned to the user is either erroneous, or took too long to be produced. An `accountable` heterogeneous SoC is defined as one in which the cause of the error or the delay can be unambiguously isolated to the host and/or one or more of the guests.

To do this isolation, we propose to analyze each of the constituent jobs performed by the guests. The exercise may find one or more of the guests guilty. Alternatively, all guests may be found to have performed as per specification, thereby implicating the host as guilty of the errorneous task execution.

Let the input provided to some guest G as part of some job j be Ip_j, the output be Op_j, the quality of service provided be QoS_j, and the quality of environment be QoE_j.

Definition 3.2. If X is a measurement and Y is a negotiated condition, $X \lhd Y$ denotes that X satisfies Y. $X \ntriangleleft Y$ denotes that X does not meet the agreed upon condition Y.

Axiom 1. *If the dispute is regarding an incorrect task result, a guest G may be held responsible if and only if the latter performed a job j such that*
$Op_j \ntriangleleft Op_G$ *and* $Ip_j \lhd Ip_G$.

Axiom 2. *If the dispute is regarding a task result taking too long to be computed, a guest G may be held responsible if and only if the latter performed a job j such that*
$QoS_j \ntriangleleft QoS_G$ *and* $QoE_j \lhd QoE_G$.

Axiom 3. *If no guests can be held responsible, then the SI must take responsibility.*

3.3 Accountability through Logging

Our approach to providing accountability is through having the host log events during the SoC's operation. This can be accomplished by one of several methods: have dedicated structures on chip, reuse trace buffers used for debugging (DFD), or use regular physical memory. Subsequently, the logs are stored in an off-chip disk owned by the user. In the event of incorrect functioning of the chip, the user provides the logs to the SI. The SI then analyzes the recorded logs offline to ascertain the guilty component(s).

3.3.1 *High Level Logging and Auditing.*

As per the execution model described, to provide accountability, for each job j executed by a guest G, $\forall G \in \mathbb{G}$, four logs are required: (i) $IpLog_j$: consisting of all communication from H to G, (ii) $OpLog_j$: consisting of all communication from G to H, (iii) $QoSLog_j$: recording the QoS provided, (iv) $QoELog_j$: recording the QoE provided. These four logs constitute the *high level logs.*

During an off-chip investigation, these four logs are analyzed according to the aforementioned negotiated conditions.

It bears re-iterating that the vendors do not always trust each other, and having one's design held responsible for an SoC's incorrect functioning is detrimental to a vendor's credibility in the market. Consequently, a component may tamper with the logs to hide a bug in its design or shift the blame towards another component. To ensure that the recorded logs are not tampered with, we propose to employ an on-chip auditing system, trusted by all vendors, that certifies events. The certificates are stored along with the logs. A log is authentic only if it has an accompanying certificate. If, during offline analysis, the SI claims that a guest was the cause of the incorrect behavior, it must prove it using the certified logs.

Definition 3.3. The high level auditing system provides four certificates for each job j:
$IpCert_j$ certifying $IpLog_j = Ip_j$, $OpCert_j$ certifying $OpLog_j = Op_j$,
$QoSCert_j$ certifying $QoSLog_j = QoS_j$, $QoECert_j$ certifying $QoELog_j = QoE_j$.

Theorem 3.4. *A heterogeneous SoC is accountable if and only if the four logs – $IpLog_j$, $OpLog_j$, $QoSLog_j$, and $QoELog_j$, and the four certificates to authenticate the collected logs – $IpCert_j$, $OpCert_j$, $QoSCert_j$, and $QoECert_j$, are available, $\forall j \in$ jobs performed by any guest $G \in \mathbb{G}$.*

Proof: By Definition 3.3, the four certificates ensure that the four logs collected by the host – $IpLog_j$, $OpLog_j$, $QoSLog_j$, and $QoELog_j$ – correspond to what actually transpired during the in-field execution, that is – Ip_j, Op_j, QoS_j, and QoE_j. By Axioms 1, 2, and 3, the four measurements – Ip_j, Op_j, QoS_j, and QoE_j – are necessary and sufficient to provide accountability.

3.3.2 Low Level Logging and Auditing. As stated in Theorem 3.4, the high level logs directly correspond to accountability. However, practically collecting them in hardware, with minimal overhead, is difficult. We propose to have the loggers incorporated on-chip by the SI collect *low level logs*. For each message that is communicated between the host and a guest (recall that two guests cannot directly communicate with each other), (i) the contents of the message, (ii) the sender and receiver IDs, and (iii) the time of transfer, are logged.

Theorem 3.5. *The high level logs can be derived offline by aggregating low level logs.*

Proof: *IpLog* and *OpLog* can be derived by aggregating the message contents in the low level logs. $IpLog_j$ ($OpLog_j$) can be derived by aggregating the contents of all messages sent from H to G (from G to H) as part of a job j. Similarly, *QoSLog* and *QoELog* can be derived by aggregating the times of transfer. $QoSLog_j$ can be derived using the job request and the job response messages. $QoELog_j$ can be derived using the resource request and the resource response messages.

To ensure that the low level logs are not tampered, a low level auditing system, trusted by all parties, is required on-chip. This system serves to perform audit at the level of messages transferred from a sender to a receiver. The requirements of such a system is formally defined in Section 4, followed by an intuitive construction of a solution. We then apply this solution in the SoC model in Section 5, and present a detailed architecture.

4 AUDITING MESSAGES IN A SENDER-RECEIVER PARADIGM

4.1 The Problem

Consider two mutually distrusting nodes S and R connected by a single link that is assumed to be non-faulty. S (the sender) performs some computation, and sends the result as a message to R (the receiver). The message serves as an input to R, who begins computing after receiving the

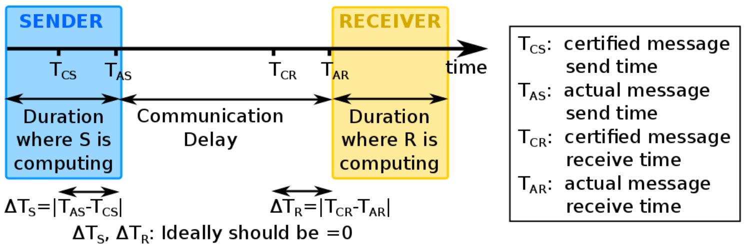
SENDER
RECEIVER
time
T_CS
T_AS
T_CR
T_AR
Duration where S is computing
Communication Delay
Duration where R is computing
ΔT_S=|T_AS-T_CS|
ΔT_R=|T_CR-T_AR|
ΔT_S, ΔT_R: Ideally should be =0
T_CS: certified message send time
T_AS: actual message send time
T_CR: certified message receive time
T_AR: actual message receive time

Fig. 2. Candidate solution: sender as auditor

message. Let us consider a single message transmission. Let the sender send message MSG_S at time T_S. Let the receive get the message MSG_R at time T_R. If the receiver did not get any message, then $MSG_R = T_R = \phi$, and if the sender did not send any message, then $MSG_S = T_S = \phi$.

The auditing system has to produce a certificate $\mathbb{C}$, that consists of (i) S_C, the certified sender, (ii) R_C, the certified receiver, (iii) MSG_C, the certified message contents, and (iv) T_C, the certified time of communication.

Definition 4.1. For the auditing system to be sound, it must guarantee the following four properties:

(1) **Non-Repudiation**: $S_C = S$ AND $R_C = R \Rightarrow$ The actual sender and receiver are certified.
(2) **Integrity**: $MSG_S = MSG_R = MSG_C \Rightarrow$ The message was received and certified as sent.
(3) **Timeliness**: $|T_S - T_C| < \tau$ AND $|T_R - T_C| < \tau$, where τ is a predefined (small) positive constant $\Rightarrow$ The actual (within a small margin of error) time of transfer is certified, and no undue delay is induced between the sending and receipt.
(4) **Atomicity**: Either $(((MSG_S = MSG_R = MSG_C) \neq \phi) \wedge ((T_S \approx T_R \approx T_C) \neq \phi))$ OR $((MSG_S = MSG_R = MSG_C = \phi) \wedge (T_S = T_R = T_C = \phi))$. The message is either sent and received, and certified and logged (and the corresponding times are approximately the same) or not sent/received/certified and logged at all.

We derive these properties from non-repudiation research [23] that essentially deals with providing certificates to the sender and receiver of a message that allow them to prove to an adjudicator that they sent and received the message respectively.

We now intuitively construct a sound auditing system.

4.2 Naive Solutions

4.2.1 The Parties Themselves do the Auditing. A possible solution is to have one party do the auditing. Without loss of generality, let us assume the sender, S, does the auditing. In this strategy, non-repudiation cannot be guaranteed as the sender may produce a certificate for a different S_C ($\neq S$) and R_C ($\neq R$). Integrity cannot be guaranteed as the sender may produce the certificate for a message different from that which was sent, $MSG_C \neq MSG_S$. Timeliness cannot be guaranteed either. The certified times and actual times of sending/receiving the message must ideally be the same (see Figure 2), i.e., $T_{CS} = T_{AS}$ and $T_{CR} = T_{AR}$, to ensure timeliness. However, the sender may choose to make the certified time of sending much lower than the actual time of sending $T_{CS} << T_{AS}$, thus making it seem like it finished its work/computation much earlier. The sender may choose to make the claimed time of receipt much lower than the actual time of receipt ($T_{CR} << T_{AR}$), thus making it seem like R took longer to finish its work. Atomicity cannot be guaranteed either as S may not produce certificates for messages it did send, and may produce certificates for messages it did not send to R.

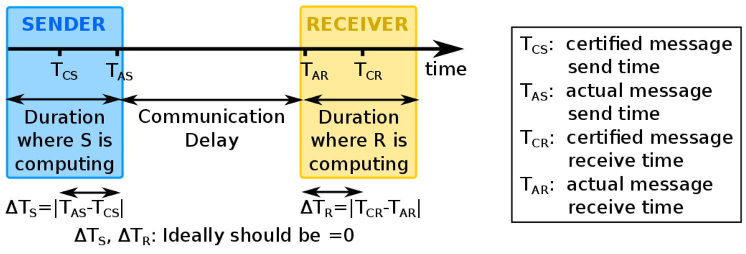
SENDER
RECEIVER
T_CS T_AS
T_AR T_CR
time
Duration where S is computing
Communication Delay
Duration where R is computing
ΔT_S=|T_AS-T_CS|
ΔT_R=|T_CR-T_AR|
ΔT_S, ΔT_R: Ideally should be =0
T_CS: certified message send time
T_AS: actual message send time
T_CR: certified message receive time
T_AR: actual message receive time

Fig. 3. Candidate solution: parties audit themselves

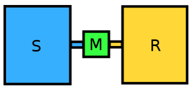

Fig. 4. Candidate solution: TTP as a Mail Server

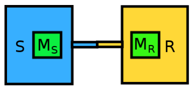

Fig. 5. Candidate solution: DuoMeter

Another possible solution is to have each party audit itself. S and R both generate certificates $\mathbb{C}_S$ and $\mathbb{C}_R$ respectively. Clearly, there is no way to ensure that the certificates are coherent. $\mathbb{C}_S$ and $\mathbb{C}_R$ may certify different senders and receivers violating non-repudiation, and have different message contents thus violating integrity. Figure 3 describes the issues with timeliness. $\mathbb{C}_S$ may certify $T_{CS} << T_{AS}$ to make it appear that S finished its computation much earlier than it actually did. $\mathbb{C}_R$ may certify $T_{CR} >> T_{AR}$ to make it appear that R spent lesser time computing than it actually did. $\mathbb{C}_S$ and $\mathbb{C}_R$ may not be produced at all, or $\mathbb{C}_S$ or $\mathbb{C}_R$ may be produced for messages that were not sent, violating atomicity.

Thus, having the parties audit themselves is not a sound auditing system.

4.2.2 Trusted Third Party(TTP) providing a Mail Service. Let us assume a Trusted Third Party (TTP) organization that is trusted by all parties – the SI and all the 3PIP vendors. This TTP organization designs meters that perform the necessary audit. Figure 4 shows how the meter can be employed in a “mail-server” configuration. S sends the message MSG to M at time t, requesting it to forward it to R. Aside from forwarding the message, M produces a certificate $\mathbb{C}$ with $S_C = S$, $R_C = R$ (non-repudiation), $MSG_C = MSG$ (integrity), and $T_C = t$ (timeliness). $\mathbb{C}$ is essentially a cryptographic hash of (S_C, R_C, MSG_C, T_C), made using a secret key known only to M, the trusted third party. Thus, during an offline dispute, the TTP organization can verify if the certificates are genuine.

Additionally, M produces $\mathbb{C}$ for every message that is sent to it by S, and also forwards the message to R. It does not produce $\mathbb{C}$ for messages it did not receive. However, it is possible that $\mathbb{C}$ (and the corresponding log) is discarded by S of R while resolving a dispute offline. To counter this, we propose to employ a cryptographic system whose state (crypto-state) *changes with every round*. The details of the encryption scheme we employ in our proposed implementation is given in Section 5.2.2. In brief, the cryptographic key changes in a certain predefined way after each round of encryption. Consequently, if the certificate of the k^{th} message, $\mathbb{C}^k$, is discarded during an offline dispute, this can be detected. It will not be possible for any of the parties to read the $(k+1)^{th}$ message or verify $\mathbb{C}^{k+1}$ without processing $\mathbb{C}^k$.

It is difficult to employ this solution in SoCs where all vendors distrust each other, because it requires M to have direct (and reliable) links to both the sender and the receiver. The TTP provides meter designs to the SI and 3PIP vendors in the form of soft IPs. The latter embed the meters in their designs, thereby, making it impossible for a meter to have direct links to multiple parties.

4.3 DuoMeter: Solution using Embedded Meters

In the *DuoMeter* scheme, trusted third party (TTP) meters are embedded at both S and R (see Figure 5). S sends the message MSG_S intended for R to the TTP meter (M_S) embedded in it at time T_S. M_S encrypts (S, R, MSG_S, T_S) using a secret key it shares with M_R (receiver side trusted meter), and sends the encrypted message to S. M_S also produces a certificate $\mathbb{C}_S$, a hash of (S, R, MSG_S, T_S) produced using the same secret key. S then sends the encrypted message to R, who forwards it to M_R at time T_R, claiming that it is a message from S. The latter decrypts the message MSG_R and sends it to R, who consumes it. M_S also produces a certificate $\mathbb{C}_R$, a hash of (S, R, MSG_R, T_R).
Non-repudiation: When M_R decrypts the message, it verifies that (i) the sender ID in the message matches the sender ID claimed by R, and (ii) the receiver ID in the message matches the ID of the node it is embedded in.
Integrity: The message sent from M_S to M_R (through S and R) is accompanied by a *checksum*, which is essentially $\mathbb{C}_S$. If S or R tamper with the message after it is certified by M_S, the verification of the checksum at M_R will fail. However, it is not possible to identify which of S or R or both modified the message. This is similar to the *last-link* problem encountered in accountability solutions for the Internet [9]. We term this attack as an *irrational integrity attack*. We use the word "irrational" because S and R have no immediate gain from performing such an attack [24]. S cannot hide a wrong computation by it by modifying the message after the $\mathbb{C}_S$ was produced because $\mathbb{C}_S$ was produced on the message given by S to M_S. Similarly, modifying the message does not help R evade a wrong computation. R has no way of knowing at the time of communication (before it sends to M_R) whether the input will cause it to compute wrongly since the message is unreadable (encrypted). The only motivation for S and R to perform such an attack is to bring down the performance of the system as a whole through the penalties paid for recovering from the attack.
Timeliness: Recall that ($T_{CS} < T_{AS}$) is advantageous to S since this shortens the certified duration of S's computation. Similarly ($T_{CR} > T_{AR}$) is advantageous to R since this shortens the certified duration of R's computation. However, since S cannot modify the message after it is certified by M_S, S is forced to complete all its work before the $\mathbb{C}_S$ is produced. Since the message is encrypted, it has to be decrypted by M_R before R can use the message. Thus R is forced to begin its work only after $\mathbb{C}_R$ is produced. Thus, neither S nor R can subvert the timeliness property to their advantage. However, they may still perform an *irrational timeliness attack* similar to the irrational integrity attack. The only effect of this will be to bring down the performance of the entire system. Now, since the message received at M_R contains the time of sending T_S as well, M_R can use this to check if any undue delay was induced by T_S, i.e., if $T_R - T_S > \tau$ (τ is a predefined constant). Thus, just like the *irrational integrity attack*, an *irrational timeliness attack* can be detected, but it is not possible to find out who had inserted the delay (S or R).
Atomicity: It is possible that S or R drops the message after the $\mathbb{C}_S$ is issued. This scenario can be detected by the round-based crypto-system described in Section 4.2.2. All further communication between S and R will fail if a message is dropped. Thus, an *irrational atomicity attack* can be detected. This attack is an *irrational* one as well – neither S nor R serve to gain anything from doing this. And just like the other two irrational attacks, the *irrational atomicity attack* can be detected but it cannot be ascertained who the guilty party was. An attack where both the log and message are discarded is also countered through employing a round-based crypto-system as described in Section 4.2.2.

Thus, the redundant metering solution is a sound auditing system. The identities of the sender and the receiver, the contents of the message, the time of sending and the time of receiving can be accurately certified. Atomicity is also guaranteed. All types of irrational attacks can be detected, but the guilty party cannot be ascertained.

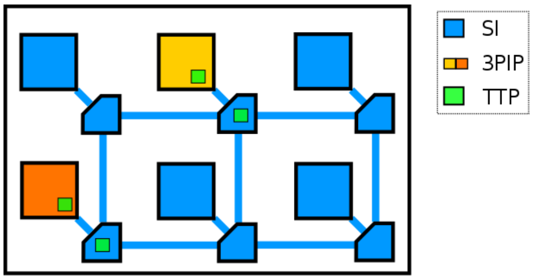

Fig. 6. Authentic Low-Level Logs with the DuoMeter Auditing Scheme

The two certificates $\mathbb{C}_S$ and $\mathbb{C}_R$ are in a sense redundant since they certify the same features of the message transfer ($T_S \approx T_R$, if timeliness is not violated). However, having the certificates generated at both S and R makes the storing of the certificates along with the logs in SoCs easier, as discussed ahead in Section 5.2.1.

5 AUDITING IN 3PIP-CONTAINING SOCS

5.1 Implications of a Sound Audit

THEOREM 5.1. *If a* sound *audit can be provided at every on-chip interface between two mutually untrusting parties, then accountability can be provided.*

Proof: Low-level logs are collected by SI-designed loggers. These can be used to achieve high-level logs, by Theorem 3.5. As mentioned previously, two guests never directly communicate with each other. However, the guests communicate with the host. If the host is trusted by the guests, then there is no on-chip interface between two mutually untrusting parties. If the host is not trusted, and if a sound audit can be provided at every on-chip interface between the host and a guest, then

(i) the integrity, non-repudiation, and atomicity properties are necessary and sufficient to certify the content of every message exchanged by H and G for a job j done by the latter, and hence to provide $IpCert_j$ and $OpCert_j$,

(ii) the timeliness, non-repudiation, and atomicity properties are necessary and sufficient to certify the times of transfer of every message exchanged by H and G for a job j done by the latter, and hence to provide $QoSCert_j$ and $QoECert_j$.

Since all four high-level certificates can be produced, and high level logs are collected (through collecting low level logs, Theorem 3.5), accountability can be achieved, by Theorem 3.4.

5.2 Authentic Low-Level Logs with the DuoMeter Auditing Scheme

The TTP provides designs of the meter to the SI and the 3PIP vendors. The design of the meter is discussed in Section 5.2.2. The SI and the 3PIP vendors embed the meters in their own designs, so as to implement the DuoMeter auditing scheme at each host-guest interface, as shown in Figure 6, thereby achieving accountability as per Theorem 5.1. The protocol followed by the host and the guest circuitry to send a message to the other is discussed in Section 5.2.1. The overheads introduced by the auditing process in discussed in Section 5.3.

5.2.1 Message sending protocol. The protocol is described in terms of "sender S" and "receiver R", rather than "host" and "guest", as the protocol is symmetric for both host-to-guest and guest-to-host communication. As required by the DuoMeter scheme, S has a TTP meter embedded in it, referred to as M_S. Similarly, R has a meter embedded in it, referred to as M_R. The two meters share a secret key state K_{S-R}. The secret is embedded by the TTP at the time of design.

Table 1. Message sending protocol

		At the Sender	
(S1)	$S \to M_S$	$\odot$PT	PT: plain text (the message)
(S2a)	M_S	$state\_replay = XOR_{K_{TTP}}(K_{S-R}\|\|CTR_{S-R})$	key state is saved for replay support (Section 5.2.2)
(S2b)	M_S	$CT = E_{S-R}(PT\|\|T_S)$	CT, checksum, encoded state computed simultaneously
(S2c)	M_S	$checksum = \mathbb{C}_S\_hash = H_{S-R}(PT\|\|T_S)$	
(S3a)	$M_S \to S$	$\odot MSG_S = CT\|\|checksum$	
		$\odot T_S$	T_S: time of message sending
		$\odot \mathbb{C}_S = \mathbb{C}_S\_hash\|\|state_{replay}$	
(S3b)	M_S		updates K_{S-R} and CTR_{S-R}
(S4a)	$S \to R$	$\odot$M	
(S4b)	S		S stores <SEND,PT,R,T_S,$\mathbb{C}_S$>
		At the Receiver	
(R1)	$R \to M_R$	$\odot MSG_R = CT\|\|checksum_{recv}$	
(R2a)	M_R	$state\_replay = XOR_{K_{TTP}}(K_{S-R}\|\|CTR_{S-R})$	key state is saved for replay support (Section 5.2.2)
(R2b)	M_R	$K_{temp} = K_{S-R}; CTR_{temp} = CTR_{S-R}$	temporary key state used during hashing
(R2c)	M_R	$D_{S-R}(CT) \Rightarrow (PT, T_S)$	
(R3a)	$M_R \to R$	$\odot$PT	
(R3b)	M_R		updates K_{S-R} and CTR_{S-R}
(R4a)	R		uses PT
(R4b)	M_R	$checksum_{comp} = \mathbb{C}_R\_hash = H_{temp}(PT\|\|T_S)$	
(R5a)	M_R	$(checksum_{recv} == checksum_{comp})$?	integrity/atomicity check
(R5b)	M_R	$(T_R - T_S \le \tau)$?	timing check (τ is a predetermined constant)
(R6)	$M_R \to R$	$\odot T_S$	
		$\odot \mathbb{C}_R = \mathbb{C}_R\_hash\|\|state_{replay}$	
(R7)	R		R stores <RECV,PT,S,T_S,$\mathbb{C}_R$>

'S': sender; 'R': receiver; 'M_S': meter at sender; 'M_R': meter at receiver;
'$||$': concatenation; subscript $S - R$: the key state shared between M_S and M_R is used;
K_{TTP}: key shared between meter and the parent TTP organization
Steps with the same numbered prefix can execute simultaneously. For example, steps (S2a), (S2b) and (S2c) can execute simultaneously. Similarly, steps (S3a) and (S3b) can execute simultaneously.

Table 1 describes the protocol followed to send a message from one on-chip component to another.

S sends the message to be sent, PT, to the TTP meter embedded in it, M_S. M_S first prepares a signed record of its current cryptographic state as *state_replay*. This signing is done with a secret key shared between M_S and the TTP, K_{TTP}. M_S computes a hash of the message contents PT and the time of sending T_S (current time at M_S), as $\mathbb{C}_S\_hash$. M_S prepares the cipher text CT by encrypting PT and T_S. The encryption and the hashing are done using the shared key K_{S-R}. M_S responds to S with the message ready to be transferred, MSG_S – a concatenation of CT and $\mathbb{C}_S\_hash$. $\mathbb{C}_S\_hash$ serves as a checksum for integrity checking at the receiver. M_S also sends the certified time of sending T_S. M_S also sends the certificate of message sending $\mathbb{C}_S$ – a concatenation of $\mathbb{C}_S\_hash$ and *state_replay*. S then sends M to R. If S is the host, it logs the message transfer as <SEND, PT, R, T_S, $\mathbb{C}_S$>.

R cannot begin using the message MSG_R, as it is encrypted. So it is forced to send it to M_R. M_R first prepares a signed record of its current cryptographic state as *state_replay*, just as done in M_S. M_R decrypts the message to obtain the original message PT, and the time of sending T_S. M_R sends PT to R, who can now begin processing the message.

M_R computes a hash of the decrypted message contents PT and T_S, as $\mathbb{C}_R\_hash$. If $\mathbb{C}_R\_hash$ does not match the received checksum, then an integrity or atomicity error is inferred. If the time of message receipt at R (current time at M_R), T_R, is much later than T_S, then a timing error is inferred. In the case of an error, the certificate of message receipt $\mathbb{C}_R$ sent by M_R to R attests the same. If there was no error, M_R sends R a concatenation of $\mathbb{C}_R\_hash$ and *state_replay* as the certificate of message receipt $\mathbb{C}_R$. M_R also sends the time of sending T_S to R (note that T_S was used to compute $\mathbb{C}_R\_hash$; also, if there was no timing error, then $T_R \approx T_S$). If R is the host, it logs the message transfer as <RECV, PT, S, T_S, $\mathbb{C}_R$>.

In the event of an error, a choice of operational policy exists. A simple mechanism is to have the meter shut down until the TTP decides that the dispute is resolved offline, and remotely signals the pair of meters to reset their cryptographic states to a common value. This serves as a deterrent to the parties from performing irrational attacks, as the chip's downtime reflects poorly on all parties.

5.2.2 Design of the Meter. As evident from the protocol, the solution utilizes cryptographic techniques to both provide certificates and counter misbehavior attacks by the communicating parties. Public key cryptography cannot be employed because hardware implementations take too long to perform encryption/decryption. Additionally, they have a large area overhead as well (Keija et al.[22] describes a hardware implementation of the RSA algorithm that occupies $0.014mm^2$ at 14nm and takes millions of cycles to encrypt a 1024-bit block). So we base our design on the PRESENT block cipher [12], a symmetric technique.

Though the employment of cryptographic techniques in hardware constitutes an area overhead (albeit meager, as elaborated in Section 5.3.2), we believe the achievement of the desirable property of accountability justifies the added cost. There are a wide range of works [10, 25, 26] that employ cryptography in SoCs to enable protocols that require different security properties such as authenticity, privacy, and integrity.

Block Cipher. The PRESENT block-cipher is used to perform encryption. It works with an 80-bit shared secret key. It takes 32 cycles to encrypt a 64-bit block. The parallelized implementation offers a throughput of 1 block per cycle. It is extremely lightweight as well – the parallel implementation occupies approximately $0.0065mm^2$ at 14nm technology (duly scaled according to [18]). PRESENT is also highly resilient to cryptanalysis attacks. One of the reasons for its high resilience is that the 80-bit key is updated after each round of encryption/decryption. In each round, the key is left-rotated by 61 bits. The last five bits are then XOR-ed with the contents of a 5-bit counter that is updated every round. The first four bits are passed through a substitution box (S-Box). It is imperative for the "key state", that is the key value and the counter value, to be the same at the sender and the receiver, for each communicated block of data. This essentially means that the same number of blocks have to be decrypted as were encrypted. This property has two important uses: first, ensuring the atomicity guarantee when messages are exchanged (see Section 4.3), and second, ensuring that the SI does not drop logs (see Section 4.2.2).

Due to its superior performance, light weight and high resilience, the PRESENT cipher has been incorporated in the international standard for lightweight cryptographic methods by the International Organization for Standardization and the International Electrotechnical Commission.

Hash Function. The PRESENT block cipher can also be made to function as a one-way hash function. The design follows the Davis-Meyer mode and is termed DM-PRESENT [12]. The hash function is used to sign the $\mathbb{C}_S$ and $\mathbb{C}_R$ certificates, as well as produce checksums. The hash function has the same throughput and latency as the base PRESENT block cipher.

Replay Support. Each TTP meter also shares a secret 85-bit key with its parent organization. This key is used to encrypt the key state by a simple XOR operation. The encrypted state is also part of the $\mathbb{C}_S$ and $\mathbb{C}_R$ certificates issued to S and R. This is required during the offline resolution of a dispute. To verify the authenticity of a produced certificate, the TTP organization first decrypts the encrypted key state using the secret 85-bit key. It then uses the key state to compute the hash function of the claimed plain text and timestamp. The computed hash must match that in the produced certificate.

Structure of the Meter. Figure 7 describes the block diagram of the meter. It has two modules of the PRESENT block cipher, one for encryption/decryption and the other for hashing. The modules

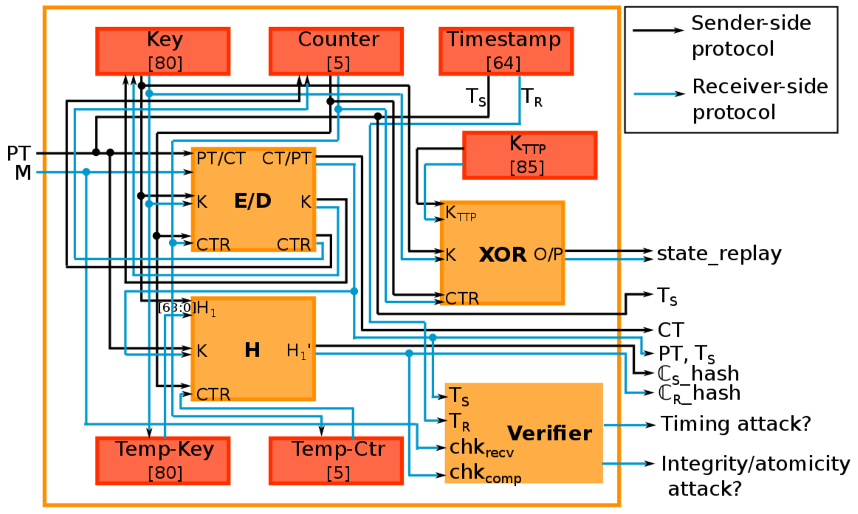

Fig. 7. Meter design: Structural

Table 2. Meter area estimation at 14nm

Component	Area Estimate
E/D block cipher	$6500.000\mu m^2$
H block cipher	$6500.000\mu m^2$
Logic	$2699.452\mu m^2$
Verifier	$119.737\mu m^2$
K_{TTP}	$86.995\mu m^2$
Key register	$81.840\mu m^2$
Temp-Key register	$81.840\mu m^2$
Timestamp	$65.472\mu m^2$
XOR	$51.144\mu m^2$
Counter register	$5.115\mu m^2$
Temp-Counter register	$5.115\mu m^2$
Total area	$16196.710\mu m^2$

contain the 5-bit counters required by the protocol. An 80-bit register contains the key, while another 80-bit register *Temp-Key* holds a copy of the key to be used for hashing (see step (R2b) in Table 1). Similarly, a 5-bit register holds the counter value, and another 5-bit register *Temp-Ctr* holds a copy to be used for hashing. An 85-bit register K_{TTP} holds the secret key shared with the TTP parent organization. An 85-bit wide XOR gate is used to encrypt the key state for replay support. A 64-bit counter is used to provide the timestamp. A verifier module checks for irrational integrity, timing and atomicity attacks. Note that the non-repudiation property is not explicitly checked because in this design for every *H-G* interface, there is a unique pair of meters. A message signed at the source meter cannot be understood by any meter other than the source's pair meter, who has the required key for decryption. Only a message that originated at the source can be decrypted at the source's pair meter, again for the same reasons. The TTP organization can uniquely identify meters by K_{TTP}, allowing for any attack on non-repudiation to be detected during the offline investigation. The estimated area for a meter, synthesized with a 90nm ASIC library and duly scaled to 14nm according to [18], is $0.016mm^2$ (see Table 2).

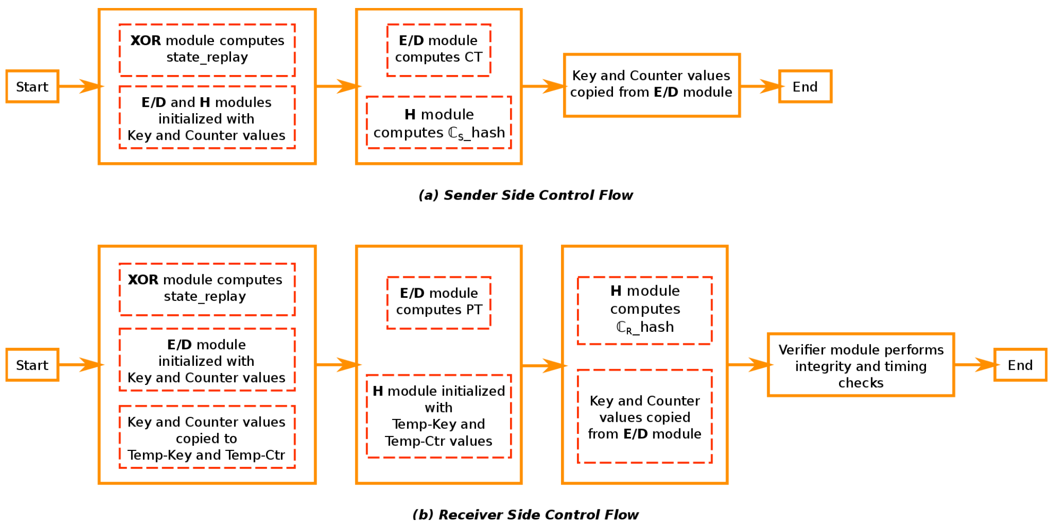
Start
XOR module computes state_replay
E/D and H modules initialized with Key and Counter values
E/D module computes CT
H module computes $\mathbb{C}_S$_hash
Key and Counter values copied from E/D module
End
(a) Sender Side Control Flow
Start
XOR module computes state_replay
E/D module initialized with Key and Counter values
Key and Counter values copied to Temp-Key and Temp-Ctr
E/D module computes PT
H module initialized with Temp-Key and Temp-Ctr values
H module computes $\mathbb{C}_R$_hash
Key and Counter values copied from E/D module
Verifier module performs integrity and timing checks
End
(b) Receiver Side Control Flow

Fig. 8. Meter design: Control

Meter Operation. The flow of control in the proposed meter design is elaborated in Figure 8. The boxes with dashed lines are sub-tasks within a state. The sub-tasks are executed in parallel. All sub-tasks complete before moving to the next state.

Tampering and Other Issues. The TTP meter design needs to be protected against tampering by the parties: the SI and the third party vendors. This is an established field of research, with many solutions proposed [16], and even standards established [3]. These are essentially based on the principle of *logic obfuscation*. We will discuss one proposal: "FORTIS" [16]. The meter IPs are accompanied by a cryptographic digest of the entire IP in their headers. The digest is computed using a secret key shared between the TTP and the EDA companies, whose tools are employed by the SI and the third party vendors (the EDA companies are also deemed trustworthy). When the SI or third party vendor attempts to synthesize her design, the EDA tool computes the digest for the embedded TTP meters. If a meter has been modified, the computed digest does not match that in the IP header. The synthesis process then aborts.

A related issue is that the meters need to be embedded as prescribed by the TTP. Kalayappan et al. [21] have proposed a method to ensure this.

A third issue is that of clock synchrony between the meter pairs, which again is an established field of research [14].

5.3 Evaluating the Overheads of the Auditing Process

5.3.1 Performance Overhead of Auditing. Let us consider the sending of a 64-bit message. At the sender side, the encryption (step (S2b) in Table 1) and providing of the $\mathbb{C}_S$ (steps (S2a) and (S2c)) are done in parallel. This takes 33 cycles. Taking into account the communication with the meter (steps (S1) and (S3a)), the latency is 35 cycles. At the receiver side, the decryption takes 33 cycles (step (R2c)). The receiver can begin using the payload after it has been decrypted. Again, the communication with the meter (steps (R1) and (R3a)) accounts for 2 cycles, giving a latency of 35 cycles. Thus, the overall latency is 70 cycles for a 64-bit block. The checking for attacks,

Table 3. Simulation parameters

Parameter	Value	Parameter	Value
Cores Accelerators Technology	24 24 14 nm	Microarchitecture Accelerator Types	Intel Sandybridge 6
L3 cache	32MB (32 banks) 30 cycle latency	Main Memory Latency	200 cycles
Topology Flit size Router-Latency	2-D Torus (10 × 10) 8 bytes 3 cycles	Routing Alg. Hop-latency	dyn XY routing 1 cycle

Table 4. Accelerator types and simulated workloads

Accelerator Type	ASIC Implementation	Workload Simulated
FFT	[28]	4096-point FFT
Sort	[32]	Sort 3969 words
JPEG compression	[27]	Compress 640×480 pixel image
MD5 hashing	[31]	Hash 1 MB input
AES encryption	[17]	Encrypt 1 MB input
RSA encryption	[22]	Encrypt 1 kb input

and the production of the $\mathbb{C}_R$ certificate constitute another 35 cycles, including 1 cycle for the communication (step (R7)). But these 35 cycles are off the critical path.

However, the effect of this latency is almost completely masked in a real SoC. An SoC was simulated using the cycle accurate Tejas simulator [30]. Table 3 gives the details of the simulated chip. It contains 24 3PIP accelerators – 4 accelerators of each type listed in Table 4. All other components of the chip such as general purpose cores, shared caches and the NoC are designed by the host. Every communication between the host and an accelerator has to be audited. Each accelerator was made to perform a task, as described in Table 4. The auditing resulted in a mere 0.49% average overhead in terms of the time taken for an accelerator to complete its job. The reason for this miniscule overhead is threefold: (i) the auditing system is fully pipelined, allowing a throughput of 64 bits/cycle, (ii) computation time accounts for a significant fraction of the runtime of an accelerator, (iii) there is considerable overlap between the computation and the communication, effectively masking the overhead induced by auditing. The tasks that are typically accelerated have statically determinable access patterns, and are thus amenable to prefetching, further reducing the overhead.

Additionally, the latency of 70 cycles is in the same order as that of regular packet transfer in modern SoCs. For example, in the considered chip, with a 10 × 10 layout of routers, a flit sent from one end of the chip to the other takes 80 cycles. This is in the base system without any auditing being performed. Thus, we believe designing a real time system accommodating the audit latency does not stretch the designer a great deal. Especially when the throughput of the auditing system is the same as that of the NoC. Thus, we believe the constraint induced in designing SoCs with cores tightly integrated to push the chip at its limit in terms of performance, is not too large.

Regarding real-time systems, no great complication is introduced by the auditing process. This is because there is no additional source of non-determinism being introduced – the number of cycles taken to audit a 64-bit flit is always *exactly* 70 cycles, and the throughput is always *exactly* 64 bits / cycle. Since there is no non-determinism, the designer can provide hard real time guarantees.

Since the proposed meter design offers a throughput of only 64 bits per cycle, the audit system can slow communication down in chips with NoCs offering a greater bandwidth. This problem can be alleviated by using a more powerful meter. Suppose the NoC link bandwidth is 128 bits per cycle. Then our meters can be provided with four PRESENT modules instead of two. Two modules perform encryption/decryption, and work on alternate input blocks. Similarly, two modules perform hashing, and work on alternate blocks, and XOR their outputs to produce the final hash. This gives us a meter capable of auditing at the rate of 128 bits per cycle.

5.3.2 Area and Power Overhead of Auditing. Considering the chip described in Table 3, a total of 48 meters are required – a pair of meters at each host-guest interface. Assuming a reference die size of $400mm^2$, the meters constitute an area overhead of 0.194%. Note that the DuoMeter approach scales linearly with increasing number of third party components. For every third party component, exactly two meters have to be placed at its interface with the host. In terms of power, $0.99\mu W$ is consumed for auditing each 64-bit transfer. The PRESENT cipher modules consume the majority of the energy. Even if all the 3PIPs are simulataneously communicating, the power consumed will be $24\mu W$, which is next to negligible.

6 ECONOMICAL LOGGING

A certified logging of every message exchanged between modules designed by two mutually untrusting parties guarantees accountability. Thus, in principle, every kind of malfunction and malicious attack can be detected. However, this requires infinite logging bandwidth and storage space, so as to not have any performance or storage overhead (note that the overhead of generating the required certificates is negligible, as demonstrated in Section 5.3.1). Fortunately, this is a well-studied problem in the domain of post-silicon validation. We shall use similar techniques.

The different techniques of reducing the logging result in different performance and storage overheads, and may constrain which malfunctions may be detected. Let us discuss these overheads. We envision design-for-debug (DfD) structures being re-used [19] to store the logs and certificates on-chip. When the buffers fill up, they may be written to off-chip disks. This writing constitutes two overheads: storage – the space occupied in the disks, and time – the time to save the logs (the off-chip writing consumes resources that could have otherwise been used for performing the chip's primary functionality). Reducing the logging reduces both the storage and the time overheads. The TTP meters need to be more sophisticated to support these techniques (discussed later in this Section), adding to the on-chip space overhead.

A technique must be chosen depending upon the module being covered, and the kind of accountability desired. We will briefly describe the different techniques here. We will also demonstrate the usage of some of the techniques while detecting real malfunctions derived from the errata documents of real SoCs in Section 7. Thoroughly evaluating all the techniques is beyond the scope of this paper.

Temporal Summarization: In its simplest form, it involves performing the dumping only when the chip malfunctions. If a log needs to be written and the buffer is full, the oldest entry is over-written. This results in the most recent messages exchanged before the malfunction being available for the offline dispute resolution, which may be sufficient in many cases [13].

On-chip monitors may be employed [13, 15] that turn the logging on and off based on the occurrences of *events*. Domain-specific rules may be encoded in these monitors that observe relevant signals to detect events of interest.

A sampling-based approach may be employed to log only some of the jobs performed by a module, rather than logging them all. This may satisfy scenarios where a statistical study of the quality of operation is desired.

Spatial Summarization: Not all bits of the message may be relevant for debugging. Domain knowledge can help prune the fields of less interest to reduce the size of the logs [13].

Compression: Again, as done in post-silicon validation [11], compression techniques may be employed to reduce the size of the logs before dumping them off-chip. Chandran et al. [13] have demonstrated that the techniques of summarization and compression together can reduce the logging by up to 80%.

Hashing: The entire message contents, though useful while recreating the malfunction offline, are not always necessary. Just the hash of the message contents is sufficient. Our approach is inspired by the idea of *inner state hashes*, proposed to detect malicious contractors in the cloud domain [24].

We will consider a simplified model of execution on an SoC for the purpose of explaining the proposal. The simplifications are *not* necessary for the proposal to work. Let the SoC be composed of N modules $M_1, M_2, ..., M_N$. Let a module M_i perform functionality F_i, accepting one or more inputs and producing one or more outputs. The customer's application that runs on the SoC is a task-graph, where each task is one of the aforementioned functionalities. The outputs of a parent node form the inputs of its children nodes.

We will also restrict this discussion to only functionality-related malfunctions, and not timing-related ones, though the methodology can be applied for timing-related malfunctions as well.

At run-time, as part of the accountability framework, the inputs and outputs of each module are hashed by Trusted Third Party (TTP) circuitry, and the hashes are logged. Kupcu [24] terms these "*Inner State Hashes*", as they capture intermediate results of the application, rather than the just the final result. The hashes don't require much storage or off-chip bandwidth as they are small, possibly 64-bit long for each datum (input or output).

If the SoC malfunctions, the following procedure is followed to detect the responsible module offline.

- *Set-up:* The customer submits the application (code and data), and the logged hashes to the arbitrating authority. The system integrator (SI) submits the SoC design. The 3PIP vendors submit hashes of their designs. These are used to verify if the 3PIP designs are incorporated without any tampering [16].
- *Localizing the Faulty Module:* Each node in the task-graph is then re-executed and verified, in topological order, observing dependencies [24], as follows:
  - Let the functionality of the current node be F_{cur}, and the module performing the functionality be M_{cur}. Let the set of inputs to the node be $\{i_1, i_2, ...\}$. Note that the inputs have been verified to be correct − no error has yet been detected up till this point.
  - The customer, the SI, and the designer of M_{cur} (if different from the SI), agree upon what the ideal result of executing $F_{cur}(i_1, i_2, ...)$ should be: $\{o_1^{ideal}, o_2^{ideal}, ...\}$.
  - $F_{cur}(i_1, i_2, ...)$ is then computed using the design for M_{cur}, either in a simulator, or by writing the design on an FPGA. Let the result be $\{o_1^{offline}, o_2^{offline}, ...\}$. These results are then compared with the ideal result. That is, $\forall i, o_i^{offline} == o_i^{ideal}$?. If the equality does not hold for even one output, then there is a deterministic bug in M_{cur} that could have caused the chip's malfunction. The offline analysis ends for this node, marks M_{cur} guilty, and also does not explore the children of this node in the task-graph. If the equality holds for all outputs of M_{cur}, then no deterministic bug exists in M_{cur} that could have caused the malfunction. The analysis continues.
  - A hash of the ideal outputs is computed as H_{ideal}. Let the hash of the outputs of M_{cur} that was logged at runtime, that is the *inner state hash*, be $H_{runtime}$. As proposed by Kupcu [24], the hashes are compared for equality: $H_{ideal} == H_{runtime}$? If the hashes are equal, then

no non-deterministic bug was triggered at runtime. However, if the hashes are not equal, then some non-deterministic bug was triggered at runtime, that could have caused the malfunction. The offline analysis marks M_{cur} guilty, and does not proceed to analyze the children of this node in the task-graph.

As an example, if we employ such a scheme with a JPEG compression accelerator, the entire input raw image (640×480 pixels) of 7372800 bits need not be logged. A 64-bit hash of the image will suffice, reducing the size of the log to a mere 0.0008%.

Histograms: The transfer time of each and every message may also not always be required. Application-specific mechanisms of aggregating this information are possible. For example, let us again consider the input to a third party JPEG compression accelerator. The input raw image (640×480 pixels) arrives at the accelerator as multiple messages (1800 messages, assuming bandwidth of 64 bytes). Instead of saving the transfer time of each message, a histogram of the transfer times may be sufficient to describe the QoE enjoyed by the accelerator [21]. Assuming a histogram of 16 buckets, each 11-bits wide (to accommodate the maximum of 1800), the time logs can be reduced to a mere 1.04%.

Note that the techniques to reduce the logging have to be implemented with support from the TTP meters. This ensures that they do not affect the authenticity of the logs in any way. For example, let us consider employing histograms to capture the QoS of a JPEG accelerator, and the QoE of the host. The TTP meters provide a sound audit for each message, guaranteeing non-repudiation, integrity, timeliness, and atomicity of the message transfer. However, instead of providing a certificate, or hash, for the transfer time of each message, the meter in the host maintains two histograms – one for the guest's QoS, and one for the host's QoE [21], and provides these histograms along with certificates at the end. Offline dispute resolutions now operate in terms of the histograms rather than individual requests and responses, which may suffice for many scenarios. The meters consequently need to be more sophisticated, adding to the area overhead. They do not add to the performance overhead however, as the techniques function off the critical path.

7 OFFLINE FAULT LOCALIZATION

We demonstrate the viability of our proposal using some bugs and defects, as is the standard practice in the field of post silicon validation and security [13]. We consider three bugs, deriving our examples from the published errata documents of widely used commercial SoCs [1, 2, 6].

7.1 Bug I

Microsemi's (SI) SmartFusion2 M2S050 SoC [6] has an embedded ARM Cortex-M3 core (3PIP). The ARM core makes memory requests to the SI-provided cache controller through two buses: *I-bus* for instructions, and *D-bus* for data. Now if the I-bus and the D-bus are concurrently accessed, then an erroneous value may be returned.

Let us elaborate our scenario with certain notations and assumptions (see Figure 9). Let us denote the ARM core as $\mathcal{M}_a$. Let us assume that an SI-provided controller module ($\mathcal{M}_c$) directs the different modules on chip. $\mathcal{M}_c$ directs $\mathcal{M}_a$ to execute a certain program present in a certain address range in memory, $\mathcal{A}_p$. $\mathcal{M}_a$ gets its inputs also from memory at the address range, $\mathcal{A}_i$. $\mathcal{M}_a$ writes the result of its execution, which we will assume is a single word, at $\mathcal{A}_o$. $\mathcal{M}_a$ makes many accesses to the memory through the SI-provided cache controller, which we will denote as $\mathcal{M}_f$ (the faulting module), for both instructions and data. To keep the discussion simple, let us assume that the only communication that $\mathcal{M}_a$ makes with the rest of the chip as part of its execution of the specified program are accesses to the memory through $\mathcal{M}_f$. Once $\mathcal{M}_a$ completes its task and

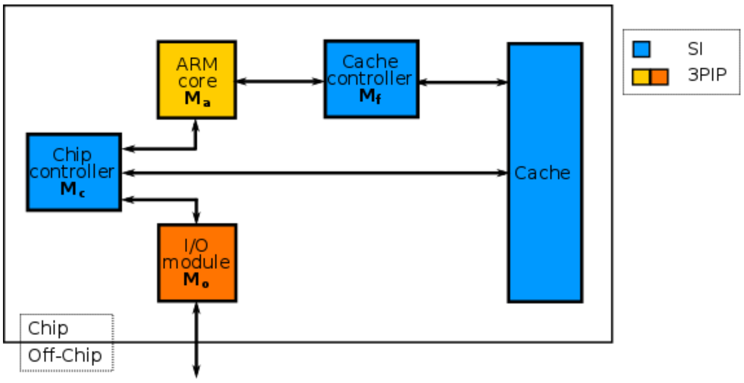

Fig. 9. Bug I Scenario

writes the result to $\mathcal{A}_o$, it signals $\mathcal{M}_c$. Let us assume that $\mathcal{M}_a$'s output is to be consumed by the on-chip module $\mathcal{M}_o$ (3PIP) that then communicates the result off-chip. Once $\mathcal{M}_c$ receives the task completion signal from $\mathcal{M}_a$, it reads the word at $\mathcal{A}_o$, and directs $\mathcal{M}_o$ to communicate the word off-chip. $\mathcal{M}_o$ encodes the data according to the off-chip channel and sends it.

Offline Detection with Complete Logging: Beginning with the incorrect result of the chip, we work backwards.

- We first analyze the functioning of $\mathcal{M}_o$. All input messages to $\mathcal{M}_o$, and output messages from $\mathcal{M}_o$ are logged. Thus, the directions and the datum word received from $\mathcal{M}_c$, and the encoded word sent on the channel are available. We verify whether the encoded output data corresponds to the input provided, with respect to the directions given. Since the verification succeeded, we can conclude that $\mathcal{M}_o$ is not to blame for the chip's incorrect output.
- We then verify if $\mathcal{M}_c$ gave the right directions to $\mathcal{M}_o$. We also verify if the datum word communicated by $\mathcal{M}_c$ to $\mathcal{M}_o$ is the same that was written by $\mathcal{M}_a$ at the end of its execution. We can do this by looking up the last write request made by $\mathcal{M}_a$ to $\mathcal{A}_o$. Since the verifications succeed, we conclude that this phase of $\mathcal{M}_c$'s operation did not cause the chip's incorrect output.
- We then verify $\mathcal{M}_a$'s functioning. We find that executing the program at $\mathcal{A}_p$, with the inputs at $\mathcal{A}_i$, gives a result different from that stored by $\mathcal{M}_a$ at $\mathcal{A}_o$ at the end of its execution. Thus, $\mathcal{M}_a$ *may* be the module responsible for the faulty output.
  $\mathcal{M}_a$ communicates with $\mathcal{M}_f$ as part of its operation. We then verify this communication. The logs contain all requests made by $\mathcal{M}_a$, and all responses sent by $\mathcal{M}_f$. We verify if all read requests made by $\mathcal{M}_a$ were serviced with the latest values at those addresses. For all addresses that are written to by $\mathcal{M}_a$, we verify that subsequent read requests returned the value last written by $\mathcal{M}_a$. During this verification, we find that for some requests, invalid values were returned by $\mathcal{M}_f$. Hence, we cannot hold $\mathcal{M}_a$ responsible for the chip's incorrect output. Instead, we conclude that $\mathcal{M}_f$ is the module responsible.

Logging using Inner State Hashes: If we follow the method of logging only *Inner State Hashes*, as detailed in Section 6, the amount of storage required to save logs can be greatly reduced. For each task, and for each pair of communicating modules, only one 64-bit hash needs to be saved. Thus, one hash each is logged for:

- Output messages of $\mathcal{M}_o$

- Input messages (data and control) to $\mathcal{M}_o$ from $\mathcal{M}_c$
- Output messages of $\mathcal{M}_a$ to $\mathcal{M}_c$, signaling the completion of the task
- Cache requests made by $\mathcal{M}_a$ to $\mathcal{M}_f$ as part of executing the program
- Cache responses from $\mathcal{M}_f$ to $\mathcal{M}_a$
- Input messages (control) to $\mathcal{M}_a$ from $\mathcal{M}_c$ to start the former's execution of the program

In total, 384 bits need to be stored to enable detection of this bug. An additional 85 bits per hash needs to be stored as replay support (Section 6.2.3 in the main document). The total amounts to 894 bits.

7.2 Bug II

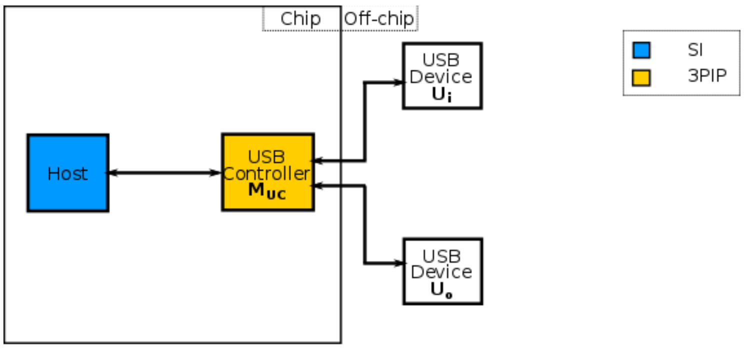

Fig. 10. Bug II Scenario

Texas Instruments' (SI) CC2538 SoC [1] has an embedded USB controller, which we assume for this discussion to be third-party provided (3PIP). Now if this controller is servicing a transfer of packets of type DATA1 *to* device *A*, and the host wishes to read from another device *B*, the controller *may* begin sending NAK packets to other communicating USB devices.

Let us elaborate our scenario with certain notations and assumptions (see Figure 10). Let us denote the faulting USB controller module as $\mathcal{M}_{uc}$. The customer has an array of USB devices, that are accessed by the *host* (the SoC), through $\mathcal{M}_{uc}$. Let the host request for a data transfer from a USB device $\mathcal{U}_i$. Let a USB device $\mathcal{U}_o$ have a data transfer to it be prematurely terminated due to the spurious NAK packets emanating from $\mathcal{M}_{uc}$.

Offline Detection with Complete Logging: The analysis begins from the disruption of the transfer to $\mathcal{U}_o$, as recognized by the customer.

- The first module analyzed is $\mathcal{M}_{uc}$, as it is the chip's interface to the USB devices. Analysis of the logs reveals the NAK packet that terminated the USB transfer to $\mathcal{U}_o$ (note that NAK packets are handshake packets, which are logged completely). Since the logs reveal that the NAK packets originated from $\mathcal{M}_{uc}$, it *may* be the module responsible for the chip's malfunction.
- All input messages to $\mathcal{M}_{uc}$ are analyzed to see if any input packet could have caused it to send the spurious NAK packet. The payload of data packets is not required for this analysis. Finding no such messages, we conclude that $\mathcal{M}_{uc}$ is the module responsible for the premature termination of the USB transfer to $\mathcal{U}_o$.

Logging using Summarization: Let us assume that the "spatial summarization" and the "temporal summarization" techniques are used to reduce the logging of $\mathcal{M}_{us}$'s communication. All control packets (token, handshake, and special) to/from $\mathcal{M}_{us}$ are logged. For data packets, only the header information is stored, and not the payload. Let an on-chip trace buffer of 1kB be used to store the logs.

Table 5 shows the number of bits required to store the log of each packet type.

Table 5. Logging different packets related to the USB protocol

Stream Description	Log	Certificate	Total (bits)
Stream to $\mathcal{U}_o$	DATA1 packet 4-bit PID, 7-bit addr	64 + 85	160
Request to read from $\mathcal{U}_i$	IN packet 4-bit PID, 7-bit addr, 4-bit ENDP	64 + 85	164
NAK disrupting stream to $\mathcal{U}_o$	NAK packet 4-bit PID, 7-bit addr	64 + 85	160

The collected logs are stored in the trace buffer. If the trace buffer is found full, the oldest entry is cleared to make space for the new entry (temporal summarization). When a malfunction is detected, the contents of the trace buffer is dumped off-chip for analysis. Thus, at the time of the malfunction, the trace buffer contains logs of the (at least) 49 most recent packets processed by $\mathcal{M}_{us}$ (considering a 1kB trace buffer). Any packet that could have caused $\mathcal{M}_{uc}$ to produce a legal NAK must be present among the logs. It is highly unlikely that such a packet was received at $\mathcal{M}_{uc}$ more than 49 packets ago.

7.3 Bug III

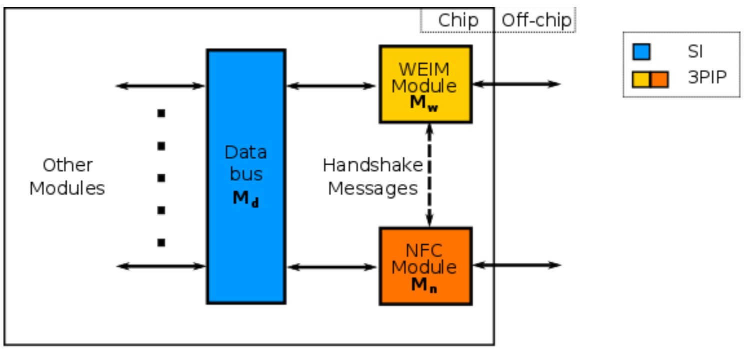

Fig. 11. Bug III Scenario

NXP's (SI) MCIMX31 SoC has a bug [2] associated with the Wireless External Interface Module (WEIM) (3PIP), the NAND Flash Controller (NFC) (3PIP), and the data bus (SI). The WEIM and the NFC have a special handshake logic allowing them to share the data bus. This logic can stall making the bus inaccessible to all parties, if the WEIM access starts at a particular state of the NFC's state machine.

Let us elaborate our scenario with certain notations and assumptions (see Figure 11). Let the WEIM module be denoted by $\mathcal{M}_w$. Let the NFC module be denoted by $\mathcal{M}_n$. Let the data bus be denoted by $\mathcal{M}_d$. $\mathcal{M}_n$ and $\mathcal{M}_w$ communicate with the external off-chip world. They also communicate to the other modules on chip through $\mathcal{M}_d$. Let $\mathcal{M}_w$ and $\mathcal{M}_n$ access $\mathcal{M}_d$ in the fault inducing way, as described above. This is externally observed as stalled and/or inaccessible NAND Flash, wireless and possibly other interfaces as well.

Offline Detection with Complete Logging: We begin with the non-responsive NAND Flash and wireless interfaces.

- We analyze the messages input to $\mathcal{M}_n$ and $\mathcal{M}_w$ off the chip, and find that they abruptly stop.
- We then analyze the exchanges between $\mathcal{M}_n$ and $\mathcal{M}_w$ and the data bus $\mathcal{M}_d$. We find that the handshake protocol between $\mathcal{M}_n$ and $\mathcal{M}_w$ had begun, but did not end.
- We analyze other messages received by $\mathcal{M}_d$ from other modules, and find them in order. We thus, localize the fault to these modules: $\mathcal{M}_n$, $\mathcal{M}_w$, and $\mathcal{M}_d$. The messages exchanged between them that led to the stall, including the timing information, are available. The scenario can therefore be recreated in an offline simulation.

Logging using Summarization: Let us assume the "temporal summarization" and "spatial summarization" techniques are used to reduce the amount of logging. Let us assume that there are separate trace buffers of size 1kB each associated with $\mathcal{M}_n$, $\mathcal{M}_w$, and $\mathcal{M}_d$, named $\mathcal{T}_n$, $\mathcal{T}_w$, and $\mathcal{T}_d$ respectively. Control packets are completely stored, while only the header information is stored for data packets. Let us assume that 8 bits are sufficient to store the required information in all cases. Further, timestamps are stored as 8-bit fields.

Thus, with 16-bit message logs, each requiring 149-bit certificates (including replay support), $\mathcal{T}_d$ can store the last 49 control messages sent to/from $\mathcal{M}_d$. This is sufficient to demonstrate that the bug is localized to the modules $\mathcal{M}_d$, $\mathcal{M}_n$, and $\mathcal{M}_w$. Enough information is available to conclude that the handshake protocol did not complete. Older messages are not required. Similarly, the last 49 messages received at each of $\mathcal{M}_n$ and $\mathcal{M}_w$ are available in the trace buffers $\mathcal{T}_n$ and $\mathcal{T}_w$ respectively. This is sufficient to recreate the bug during offline simulation for further analysis.

8 CONCLUSION

Providing accountability in SoCs containing components from different vendors is of great industrial importance. In this work, we proved that accountability can be provided if authentic logs of every message exchanged between on-chip components designed by different organizations are available for offline analysis. We then devised a solution based on this insight, by employing a trusted on-chip auditing system that authenticates the logs. We then presented a thorough design of this solution, and demonstrated that its overheads are quite modest. We also showed how the overhead of logging can be reduced by employing techniques from the post-silicon validation domain. We also demonstrated the viability of our proposal using bug scenarios encountered in commercial SoCs. We believe that this work can serve as a starting point for a vast number of ideas to achieve accountability using different approaches, under different constraints, achieving various degrees of accountability, and incurring different costs in terms of performance, chip area, and off-chip storage.

REFERENCES

[1] 2013. CC2538 Errata Note. http://www.ti.com/lit/er/swrz045a/swrz045a.pdf.
[2] 2013. MCIMX31 and MCIMX31L Chip Errata. http://www.nxp.com/assets/documents/data/en/errata/MCIMX31CE.pdf.
[3] 2014. IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP). https://standards.ieee.org/findstds/standard/1735-2014.html.
[4] 2014. SoC Integration Mistakes. http://semiengineering.com/experts-at-the-table-soc-integration-mistakes/.

[5] 2016. Blue Gecko SoC (EFR32BG1) Errata. http://www.silabs.com/Support%20Documents/RegisteredDocs/efr32bg1-errata.pdf.

[6] 2016. ER0195 Errata SmartFusion2 M2S050 (T,TS). https://www.microsemi.com/document-portal/doc_view/135069-er0195-smartfusion2-soc-m2s050-t-ts-errata.

[7] 2016. Xilinx Zynq-7000 AP SoC Production Errata. https://www.xilinx.com/support/documentation/errata/en247.pdf.

[8] Miron Abramovici and Paul Bradley. 2009. Integrated Circuit Security: New Threats and Solutions. In *CSIIRW*.

[9] Katerina Argyraki, Petros Maniatis, Olga Irzak, Subramanian Ashish, and Scott Shenker. 2007. Loss and delay accountability for the Internet. In *ICNP*.

[10] Jerry Backer, David Hely, and Ramesh Karri. 2016. Secure and Flexible Trace-Based Debugging of Systems-on-Chip. *ACM TODAES* (2016).

[11] Kanad Basu and Prabhat Mishra. 2011. Efficient trace data compression using statically selected dictionary. In *IEEE VTS*.

[12] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. 2007. PRESENT: An ultra-lightweight block cipher. In *CHES*.

[13] Sandeep Chandran, Preeti Ranjan Panda, Smruti R Sarangi, Ayan Bhattacharyya, Deepak Chauhan, and Sharad Kumar. 2017. Managing Trace Summaries to Minimize Stalls During Postsilicon Validation. *IEEE TVLSI* (2017).

[14] B. Couillard. 2002. Method and apparatus for synchronizing real-time clocks of time stamping cryptographic modules.

[15] Kees Goossens, Bart Vermeulen, Remco Van Steeden, and Martijn Bennebroek. 2007. Transaction-based communication-centric debug. In *NOCS*.

[16] Ujjwal Guin, Qihang Shi, Domenic Forte, and Mark M. Tehranipoor. 2016. FORTIS: A Comprehensive Solution for Establishing Forward Trust for Protecting IPs and ICs. *ACM Trans. Des. Autom. Electron. Syst.* (2016).

[17] Panu Hamalainen, Timo Alho, Marko Hannikainen, and Timo D Hamalainen. 2006. Design and implementation of low-area and low-power AES encryption hardware core. In *DSD*.

[18] Wei Huang, K. Rajamani, M.R. Stan, and K. Skadron. 2011. Scaling with Design Constraints: Predicting the Future of Big Chips. *Micro* (2011).

[19] Neetu Jindal, Preeti Ranjan Panda, and Smruti R. Sarangi. 2017. Reusing trace buffers to enhance cache performance. In *DATE*.

[20] Rajshekar Kalayappan and Smruti R Sarangi. 2013. A survey of checker architectures. *ACM CSUR* (2013).

[21] Rajshekar Kalayappan and Smruti R Sarangi. 2015. SecX: A Framework for Collecting Runtime Statistics for SoCs with Multiple Accelerators. In *ISVLSI*.

[22] Zhu Keija, Xu ke, Wang Yang, and Min Hao. 2003. A novel ASIC implementation of RSA algorithm. In *ASIC*.

[23] Steve Kremer, Olivier Markowitch, and Jianying Zhou. 2002. An intensive survey of fair non-repudiation protocols. *Computer communications* (2002).

[24] A. Kupcu. 2015. Incentivized Outsourced Computation Resistant to Malicious Contractors. *IEEE Transactions on Dependable and Secure Computing* (2015).

[25] Chung-Wei Lin, Bowen Zheng, Qi Zhu, and Alberto Sangiovanni-Vincentelli. 2015. Security-Aware Design Methodology and Optimization for Automotive Systems. *ACM TODAES* (2015).

[26] Philipp Mundhenk, Andrew Paverd, Artur Mrowca, Sebastian Steinhorst, Martin Lukasiewycz, Suhaib A. Fahmy, and Samarjit Chakraborty. 2017. Security in Automotive Networks: Lightweight Authentication and Authorization. *ACM TODAES* (2017).

[27] M. Papadonikolakis, V. Pantazis, and A.P. Kakarountas. 2007. Efficient High-Performance ASIC Implementation of JPEG-LS Encoder. In *DATE*.

[28] Yongjun Peng. 2003. A parallel architecture for VLSI implementation of FFT processor. In *ASIC*.

[29] Resve Saleh, Steve Wilton, Shahriar Mirabbasi, Alan Hu, Mark Greenstreet, Guy Lemieux, Partha Pratim Pande, Cristian Grecu, and Andre Ivanov. 2006. System-on-chip: Reuse and integration. *Proc. IEEE* (2006).

[30] Smruti R Sarangi, Rajshekar Kalayappan, Prathmesh Kallurkar, Seep Goel, and Eldhose Peter. 2015. Tejas: A java based versatile micro-architectural simulator. In *PATMOS*.

[31] A. Satoh and T. Inoue. 2005. ASIC hardware focused comparison for hash functions MD5, RIPEMD-160, and SHS. In *ITCC*.

[32] N. Tabrizi and N. Bagherzadeh. 2005. An ASIC design of a novel pipelined and parallel sorting accelerator for a multiprocessor-on-a-chip. In *ASIC*.

[33] Bart Vermeulen. 2008. Functional debug techniques for embedded systems. *IEEE Design & Test of Computers* 25, 3 (2008), 208–215.

Received September 2017; revised March 2009; accepted June 2009