2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)

SoCINT: Resilient System-on-Chip via Dynamic Intrusion Detection

Amr Sayed-Ahmed[1] Jawad Haj-Yahya[2,3] Anupam Chattopadhyay[2]
[1] SyoSil ApS, Copenhagen, Denmark [2]Nanyang Technological University (NTU), Singapore
[3]Institute of Microelectronics, Agency for Science, Technology and Research (ASTAR), Singapore
anupam@ntu.edu.sg

***Abstract*—Modern multicore System-on-Chips (SoCs) are regularly designed with third-party Intellectual Properties (IPs) and software tools to manage the complexity and development cost. This approach naturally introduces major security concerns, especially for those SoCs used in critical applications and cyberinfrastructure. Despite approaches like split manufacturing, security testing and hardware metering, this remains an open and challenging problem.**

In this work, we propose a dynamic intrusion detection approach to address the security challenge. The proposed runtime system (SoCINT) systematically gathers information about untrusted IPs and strictly enforces the access policies. SoCINT surpasses the-state-of-the-art monitoring systems by supporting hardware tracing, for more robust analysis, together with providing smart counterintelligence strategies. SoCINT is implemented in an open source processor running on a commercial FPGA platform. The evaluation results validate our claims by demonstrating resilience against attacks exploiting erroneous or malicious IPs.

I. Introduction

Modern complex *System-on-Chips* (SoCs) are architected by integrating designed and verified hardware blocks, commonly known as *Intellectual Properties* (IPs). The stringent time-to-market constraints along with the various engineering challenges faced during design makes it a common practice for SoC designers to rely on IPs provided by an ecosystem of third-party vendors and suppliers. Security becomes a major concern in such an approach. IP suppliers can easily introduce a security backdoor in the design through injecting malicious logic, often referred as a *hardware Trojan* [1]. Even in cases where there are no malicious intentions by the IP provider, the aggressive optimization requirements and the wide range of the verification space for IPs may result in bugs and vulnerabilities inadvertently left into the design, which can be exploited by an adversary to bring the system down [2], [3]. An attacker can potentially misuse the privileges of an untrusted IP, eventually failing the system during critical operations or get access to sensitive information.

The ideal solution to handle this situation is to ensure through formal verification techniques, e.g., model checking, that the resulting SoC design is trustworthy and free from security vulnerabilities. However, in practice, the complexity of SoC is far beyond capabilities of the state-of-the-art verification tools [4], [5]. The other solution, investigated in this work, is to deploy a dedicated subsystem to continously observe the behavior of the IPs and ensure adherence to the access policies outlined at the early design phase. In case of anomalous behavior, a blocking mechanism is activated. To achieve this goal, in this paper, we design a hardware system, named **SoCINT**, which acts as an intelligence agency for SoC designs.

A powerful intelligence system has to trace every act performed by the opponent together with relating the act with its time of occurrence, e.g., opponent O has done an act A_1 at time t_1 and another act A_2 at t_2. SoCINT applies the same idea by observing the sequential states of an untrusted IP within a limited time frame, comparing this captured trace of states with the expected trace provided by a security policy. In case of an inconsistency between the two traces, a warning is announced. SoCINT operation phases are as following:

1) *Gathering*: at every time slot t (circuit clock), SoCINT gathers current *state*, which are the values of state variables, corresponding to the state of an IP.
2) *Monitoring*: based on a given security policy and the current state variables, predict the variable states after k time slots and compare with the corresponding observed state values.
3) *Warning*: issue warning in case of a mismatched result.

The aforementioned process, akin to concurrent error detection (CED) schemes, is, however, not sufficient to block active security attacks. We further propose two counterintelligence strategies. The first strategy is used for IPs that display anomalous behavior but are needed for critical operations of the system. So we want the IP to keep operating even after we are suspect that it is a threat. The essence of this strategy is to provide the attacking IP with false information, purporting it to have a successful attack. This strategy might be a risky choice since attacked IP might perform other types of attacks that cannot be detected by SoCINT. Because of that, SoCINT also supports a more secure strategy which *kills* the attacked IP through disconnecting it from the network of SoC. In summary, SoCINT conducts one of two types of counterintelligence strategies when a malicious IP tries to compromise sensitive data:

1) *Disinformation*: provide the attacking IP with wrong data.
2) *Disconnection*: simply block the attacking IP from accessing SoC resources.

Further details of our proposal are organised in the rest of this paper as follows. Section II presents the assumptions of the threat model. Section III gives a detailed overview of SoCINT and section IV discusses the hardware design of the proposed system. Section V evaluates a prototype for SoCINT. The related works are discussed in section VI and the work is summarised in section VII.

2380-6923/19/$31.00 ©2019 IEEE
DOI 10.1109/VLSID.2019.00080

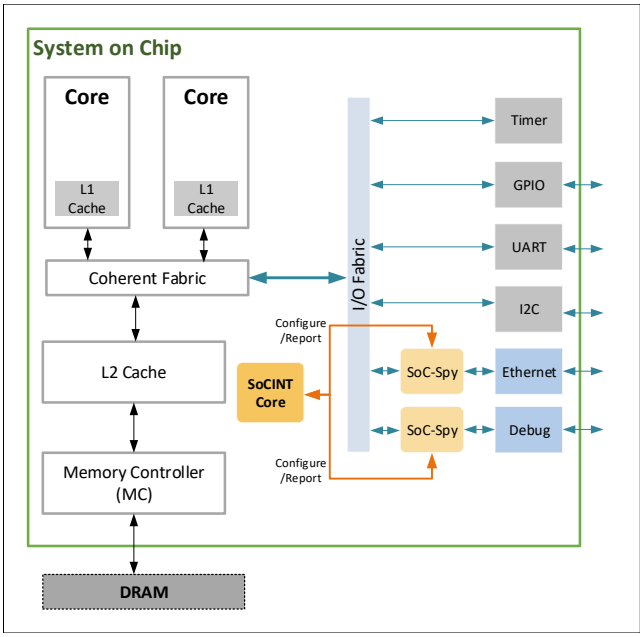

Fig. 1. SoCINT integration framework

II. Threat Model

A. Life-cycle Assumption

The first assumption is that we are the last ones who access the SoC design as well as the proposed SoCINT. This implies that the specification, IP cores, hardware tools, and the fabric configuration are within the trusted computing base.

B. Architectural Assumptions

The focus of this work is on SoCs which integrate microprocessors and peripheral IPs connected by a *Network-on-Chip* (NoC). We assume that some of the IPs are given the privileges to read/write registers of other IPs and/or a direct memory access to the physical memory.

C. Attack Model

The attacker can modify the untrusted IPs by inserting hardware Trojans that can impact the system in an adverse manner. Another scenario is that the attacker is aware of a vulnerability that can be utilized to build a system-level attack. The attacker is further aware of the design of at least one IP including the privileges given to IP and the commands that IP uses to communicate with other SoC IPs. The attacker can induct such Trojans through software that runs on one of SoC or external microprocessor that can communicate with this IP.

III. SoCINT Overview

SoCINT is a hardware intelligence system for monitoring activities of untrusted IPs together with blocking security attacks that exploit privileges given to such IPs. SoCINT enforces a security policy which is a combination of invariants to ensure that untrusted IPs cannot be exploited for orchestrating security attacks. Proving formally that the invariants of such a policy hold for SoC IPs requires a proof across all possible traces which is an intractable task [4], [5]. To bypass this daunting complexity of the verification task, SoCINT enforces the policy through a consistent comparison between the behavior of observed IPs and invariants of the security policy.

In fact, the idea of integrating monitoring systems with SoCs has been proposed before by [6], [7], however, such systems support only *non-transition invariant*, i.e., invariant that depends only on a single state. For instance, the invariant $Globally(Addr \neq 10001)$ is non-transition since it models that the value of the address $Addr$ at the current state is not equal to the constant 10001, independently from the previous or the next values of $Addr$ and other state variables. Indeed, a robust security policy shall include more powerful invariants that model transition relations between states, named *transition invariants*. For example, $Globally(Addr_1 = next(Addr_2))$ is a transition invariant since it relates the current value of the state variable $Addr_1$ with the value of $Addr_2$ at the next state. Verifying such a transition invariant in hardware requires from the system to gather partial *traces* about the states of the SoC (state variables) during the run-time and then compare the gathered trace wrt. the invariant. Indeed, it is unpractical to store all states of the system, because of that, SoCINT considers only partial traces with a length of k.

More precisely, we define a state s to be determined by the values of state variables $\{v_1, v_2, \cdots, v_n\}$. A trace is a sequence of accessible states $s_1, s_2, \cdots, s_n$ that appear in some hardware execution. A transition invariant is a relation T on the hardware states such that for every partial trace $s_i, s_{i+1}, \cdots, s_{i+k}$ the pair of states (s_i, s_{i+k}) is an element of T. In summary, SoCINT supports both types of invariants which empowers a robust security policy to be enforced.

The simplest way to implement an invariant is to generate a hardware module for it through one of the high-level synthesis tools [8]. It is important to note that, such a solution does not give the resilience needed to respond to new threats, where one or more elements of the invariant might be updated even after the fabrication of the chip. Thus, we design SoCINT to be reconfigurable and adaptive to new invariants and security policies. The configuration data updates control registers which reconfigure the invariants based on the new security policy. The configuration is not limited to the invariants; it also controls which counterintelligence is enabled. In case of disinformation strategy, it also provides a pattern used for deforming the sensitive data.

The following key principles guide the design of SoCINT:

1) Protection mechanism against hardware Trojans and vulnerabilities of IPs.
2) Supporting security policies that are modeled as transition invariants.
3) Resilience to block the newly discovered threats even after fabricating the chip.
4) Providing Counterintelligence based on the function of the attacking IP.
5) Negligible overhead that does not influence the communication process between the IPs.

IV. Design

The proposed intelligence system provides a single locus of control for each SoC IP named SoC-Spy. Moreover, SoCINT deploys a lightweight processor core running a simple low-level software (*integrity kernel*) to configure those SoC-Spies.

The principles of SoCINT can be leveraged to handle a broad scope of security problems. In this work, we base the design of SoCINT on the widely-used AMBA AXI4 NoC port standard, providing a coarser and more trustworthy level of memory protection. In the modern processors, such a task is performed through *I/O memory management unit* IO-MMU, however, implementations of the IO-MMU do not fully protect SoC from DMA attacks [9]. IO-MMU protection works at the granularity of pages. That is, mapped data — e.g., network packets— is usually allocated by the standard kernel into multiple allocations from the same page, while other locations in the page may save sensitive data. Thus, the untrusted IP mapped to such a page can compromise the sensitive data. Moreover, mapping and unmapping IPs to pages are expensive processes, e.g., unmapping a page requires about 2000 cycles on an Intel Sandy Bridge machine [10].

We claim that SoCINT can offer a more robust solution to this problem with a much lower overhead. That is, SoCINT can enforce the IP to access at time t only a determined address, say d_0. If the IP tries at time t to access another address that is not d_0, then SoCINT will treat this as malicious behavior. Relating the time with the event of accessing the memory forces a more robust security policy than what is provided by IO-MMU. Further, SoCINT is implemented in hardware and operates in parallel to the SoC, so it does not cause any additional latency.

As depicted in Figure 1, SoCINT framework consists of a light-weight processor served as integrity core (SoCINT-Core) which is connected to a so called SoC-Spy for every IP that has master and slave interfaces with the NoC. Note that IPs with master ports have the privileges to access a wide range of memory addresses (physical RAM) and the registers of other I/O IPs. In response to commands received through the slave interface, the IP should read from the specified memory location by the command. This rule assumes that the IP will always obey the received commands literally. A malicious IP could easily bypass this assumption by acting without receiving any command or accessing a different location than what is stated in the received command.

To detect and block such untrusted IPs from acting maliciously, we interpose a SoC-Spy between each IP's ports (master and slave) and the NoC bus. The distributed nature of the SoC-Spies enables a tailored sub-policy for each spy based on the commands that flow through interfaces being regulated. In the following, we introduce the design of the central unit of the SoCINT (SoC-Spy), showing how it provides the required intelligence and counterintelligence activities.

A. Intelligence of SoC-Spy

The spy provides the three types of intelligence mainly through the following hardware blocks: 1) *state gathering*, 2) *policy monitoring*, and 3) *warning*.

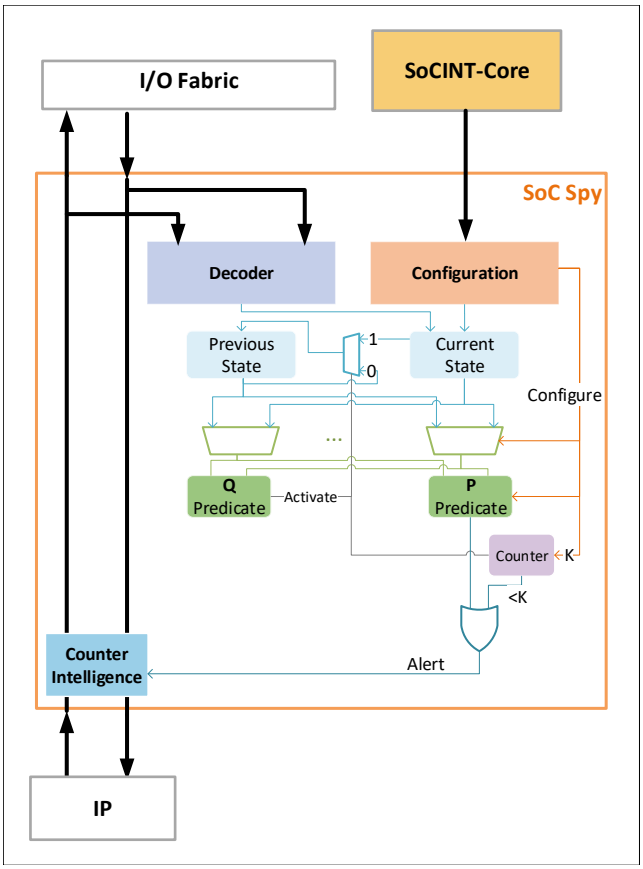

Fig. 2. SoC-Spy structure

1) State Gathering: This block is responsible for collecting and feeding the desired state to the monitoring block. We determine at the design time, based on the purpose of SoC-Spy, which state variables will be watched and stored in *state registers*. As shown in Figure 2, the state variables are chosen based on the command patterns that are exchanged between the IP under observation and other SoC IPs. The unit captures the required variables through a *Finite State Machine* (FSM) decoder for the commands that are received or sent through the ports (master and slave) of the IP under supervision. Since such selection of variables cannot be changed after fabrication, the gathering block watches a large variety of variables beyond what is needed at the moment of designing the system. Such redundancy allows the addition of new invariants to the security policy in response to vulnerabilities discovered post-deployment. The state registers also include constant values that are determined by the configuration data and needed by the policy monitoring block.

2) Policy Monitoring: The monitoring block is responsible for comparing the behavior of IP as per the security policy. SoCINT considers the implication form of invariants: $\Box(Q(v_1, \cdots, v_n) \rightarrow \Diamond_{\sim k} P(v_1, \cdots, v_n))$[1]. Q and P are n-

[1]$\Box$ refers to globally, $\Diamond_{\sim k}$ is for bounded eventually by k, and $\rightarrow$ is the implication operator

ary predicates that depend on the state variables v_i with $i = 1, \cdots, n$ which have been collected and saved in the state registers. Note that when the configuration data sets ($P = False$), the implication form turns out into $\Box(\neg Q(v_1, \cdots, v_n))$, which is the second invariant form needed to model the policy.

The invariants are implemented as hardware through integrating three types of units:

1) *Routing* the state registers to the inputs of predicates. The Routing block is implemented as a series of MUXes. For each of the MUX, the input lines are the available state registers, and the select line is driven by the configuration data.
2) *Predicate* units are composed of comparators and logic modules. Each comparator is given two inputs generated from the routing MUXes and configured through the configuration data by one of the comparison operations $\{=, \neq, <, \leq, >, \geq\}$. The logic module gets its couple of inputs from outputs of comparators or other logic modules to apply one of the configured logic operations $\{\neg, \vee, \wedge\}$. Such modularity allows building n-ary predicates. In practice, n is typically a small number.
3) *Assert* unit implements the implication operator $Q \rightarrow \Diamond_{\sim k} P$ between two predicates. The implication operator may run across over several clocks cycles. The antecedent Q is always a combinational proposition that is calculated at time t while P is a sequential proposition with its truth values captured within the time-frame from t to $t + k$. The value of k is also provided by the integrity kernel as configuration data. If it is ever the case that Q is true while P is always false within its time-frame, then the assertion is triggered to false. The triggering values of the assertions are the second type of the spy's intelligence. To design the hardware implementation of this concept, the truth value of Q enables the storing of the current state into another set of registers named *previous state*. It also activates a counter which runs for k clock cycles. The output of the counter is just one bit that is only false when the value of the counter reaches to k, otherwise, the output bit is true. Besides that, the predicate P gets its inputs from the registers of the saved previous state and the current state, providing its output to a logic block with OR operation. The second input of this OR block is the counter output bit. As shown in Figure 2, these three units (P, the counter, and OR operation) build the function $\Diamond_{\sim k} P$ that is required by the invariant.

The proposed modularity of the configurable invariants makes it possible for us to provide a tool that takes a number of invariants and automatically generates a hardware implementation for them. However, the current version of SoCINT requires building manually the hardware invariants from the aforementioned components.

3) Warning: It is a Merge block which issues a violation flag when at least one of the assertions is false. The violation flag is the intelligence provided by the block.

B. Counterintelligence of SoC-Spy

In case of an active intelligence warning, the SoC-Spy is configured by a control bit to activate one of the following counterintelligence strategies:

1) Disinformation: The block that is responsible of this strategy prevents the IP from accessing (read or write) the desired memory location if it contradicts the security policy (alert was asserted). During the activation of the counterintelligence, it captures the address that the IP tries to access and applying a deforming operation on it. The deformation is done by changing the address to new address within the allowed range of the IP.

2) Disconnection: Based on the warning flag, the counterintelligence part of the SoC-Spy forwards or blocks the outputs of the related IP. This blocking part is implemented as a multiplexer that takes the warning flag as a select line and chooses between forwarding the master output of the IP or disconnecting it.

C. Integrity Kernel

As depicted in Figure 1, SoC-Spies are connected through separated interconnection buses with the isolated lightweight processor (integrity core). The configuration data is provided by the low-level software (integrity kernel) running on the integrity core, which is stored in the local memory of the core. At the initialization of the processor, the configuration data is sent to the SoC-Spies. Through this mechanism, the invariants are configured, and portions of the configuration data are fed into the corresponding registers of the hardware spy. The integrity core can update the configuration data to enforce new policy rules prior to issuing the resume command. It sends a resume command over the policy configuration interconnect to flush all the existing policy rules and insert all new rules. It is useful to physically separate the integrity core and the whole SoC-INT from the surrounding logic so that it is protected from security attacks that exploit the sharing of resources. Further, the system can be independently analyzed without scalability problems.

V. Evaluation

This section describes the FPGA-based implementation of SoCINT and how we test the defending mechanism of the system against software attacks exploiting erroneous or malicious IPs. Lastly, the section looks at the hardware overhead of SoCINT.

A. Implementation

We use a Xilinx VC707 evaluation board, which includes a Virtex-7 FPGA, to implement a prototype for a system integrating SoCINT with LEON3 [11]. LEON3 is an open source SoC with strong adoption as a research prototype and has been used in industry as well. We choose the version of LEON3 design tailored for Xilinx VC707 board, which consists of the following IPs: 1) Four cores of LEON3 SPARC V8 Processor, 2) AHB Debug UART, 3) JTAG Debug Link, 4) GR Ethernet MAC, 5) LEON2 Memory Controller, 6) AHB/APB Bridge, 7) Single-port AHB SRAM module, 8) Generic UART, 9) Multi-processor Interrupt Ctrl, 10) Modular Timer Unit, 11) AMBA Wrapper for OC I2C-master, 12) General Purpose I/O port, and finally 13) SPI Controller. This SoC design is a representative mid-range system available in market currently.

For the integrity core of SoCINT, we deploy MicroBlaze architecture processor since it is well supported by Xilinx tools. The integrity core is very lightweight; it has no cache, MMU, or complex optional instructions. Further, the core leverages only its local memory which is a 16KB on-chip RAM. This local memory is thus inaccessible from other IPs in the system. The SoC-spies and their interfaces with the integrity core and LEON3 IPs are written in VHDL. We connect each SoC-Spy to the integrity core through AXI4-Stream interface which consists of a pair of uni-directional point-to-point data streaming channels. The AXI4-Stream interfaces follow AMBA 4 AXI4-Stream Protocol Specification.

We interpose SoC-Spy for each IP that has both master and slave interfaces. The IPs of the used LEON3 that have this characteristic are the AHB Debug UART and the GR Ethernet MAC. So, indeed, the implemented SoCINT consists of two SoC-Spies in addition to the integrity core.

B. Combined Software/Hardware Attack

A malicious IP that has the privilege of bus-master access could be exploited to perform powerful attacks. We evaluate the defending mechanism of SoCINT against such attacks by injecting a hardware Trojan in one of the privileged IPs in LEON3 design. We select the Ethernet MAC, since 1) it has all privileges to access different locations in the physical memory or registers of I/O peripherals and 2) it is connected to the outside world, which means that it can be utilized to attack the SoC remotely. The Ethernet design uses the IP and UDP protocols from the TCP/IP protocol-suite.

The Trojan injected in the Ethernet IP has two typical parts: trigger and payload. The Trojan always inspects the 7-bit unused field of the received UDP/IP packets. As specified by the UDP protocol, the unused field is a part of a 32-bit control field for applications, and it can be used as wanted. The trigger of the Trojan is activated when there are four successive packets that their 7-bit unused fields build a value matches a specific "trigger" value. The probability to activate this 28-bit trigger value through testing is equal to 2^{-28}. Further, the chosen trigger value is unlikely to appear in a regular sequence of four packets. On the other hand, the payload has the same debugging capabilities of an Ethernet debugging unit [12] which empowers an attacker to build a software application to compromise remotely the data of all registers and physical memory locations. The designed attack is orchestrated on software that runs on a remote computer. The application sends specified AHB commands to the payload of the Trojan within the application layer of UDP/IP packets. On the other side, the payload is extracted from the packets and then processes these malicious commands. When the processing is finished the payload formats a reply packet with the stolen data and then sets the MAC to transmit it. To constrain the Ethernet IP in such a way that this attack fails, it is necessary to update the policy of the deployed SoC-Spy only to permit accesses that are consistent with the commands received from the slave interface with NoC as the SoC designated. By adding this invariant, the proposed SoCINT is capable of blocking the attack mentioned above once the Trojan activates it.

Figure 3 demonstrates the sequence of events occurring at the SoCINT system at run-time. First during the system boot

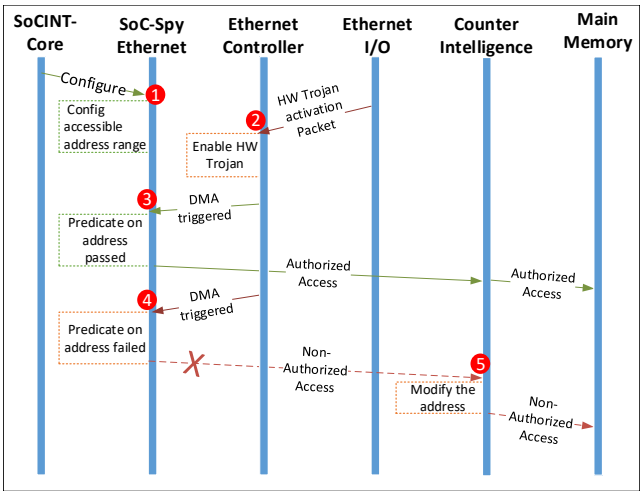

Fig. 3. Running example of the SoCINT system. An Ethernet IP with HW-Trojan. At some stage the HW Trojan is activated and attempts to access memory area with sensitive data.

❶ the SoCINT core configures the SoC-spies, whereas the spy dedicated for the Ethernet IP is configured with the memory address range that the Ethernet IP is allowed to access. Later ❷ the attacker sends a packet over the Ethernet port that activates the HW-Trojan. The Ethernet IP behaves normally ❸ for some period (does memory accesses to its allocated memory space), the SoC-spy monitors these accesses and the invariants are passing. While at some stage ❹ the IP attempts to access memory space from its permitted memory range. The SoC-Spy monitors the access and if an invariant fails, and *Alert* is generated and passed to the the *Counter-Intelligence* unit before the actual memory access executed. At its turn ❺, the *Counter-Intelligence* unit activates the disinformation strategy through modifying the address of the memory access, with a new address within the allowed range of the IP, in order to mislead the attacker of the system.

C. Cost Evaluation

SoCINT deploys additional hardware resources which are not only proportional to the number of SoC-Spies being integrated but also the additional features to coordinate the operations. We synthesized three distinct designs using Vivado 2017.1 tool provided by Xilinx to estimate the exact resource usage. The first instance is devoid of any SoCINT functionality. The second adds the integrity core connected to one SoC-spy, while the third deploys two SoC-Spies. The second and third instance shares the same SoC-Spy which is designed for the Ethernet IP. In addition to this spy, the third instance utilizes another spy for observing the debugging UART, wheres each spy enforces three invariants. All designs consist of processor cores running at 200MHz. The FPGA hardware resource utilization is listed in Table I. The first column of the table provides the design name. The second column gives the estimated power. The remaining columns show the number of look-up tables (LUTs), registers, slices, and on-chip RAMs utilized (actual number and the percentage

TABLE I
FPGA RESOURCE UTILIZATION OF DESIGNS.

Designs	Power (W)	LUTs	Registers	Slices	RAMs
No SoCINT	3.047	52,961 (17.44%)	36,363 (5.9%)	17,295 (22.8%)	113 (10.9%)
SoCINT (one SoC-Spy)	3.119	57,467 (18.9%)	41,401 (6.88%)	19,335 (25.1%)	142 (13.9%)
SoCINT (two Soc-Spies)	3.138	58,832 (19.37%)	43,064 (7.3%)	20,113 (26.05%)	151 (14.6%)

among FPGA resources) by the designs. It can be observed that the overhead area introduced by SoCINT varies between 2% to 4%. Similarly, the power overhead is within the range of 2% to 3% compared to the baseline design.

Moreover, SoCINT imposes a negligible latency overhead since it adds to the SoC fabric only a multiplexer and XOR operation, each having two inputs.

VI. Related Work

Security issues originating from the integration of third-party IP providers is an well-recognised problem. Diverse solutions to this threat have been proposed in the literature, such as, IP watermarking [13], security-aware formal verification and testing [14], and recently, a blockchain-based IC trust management [15]. While an independent survey of SoC trust management is of signifant interest, due to the space limitations, we restrict the following discussion to the works closely matching our approach.

The most relevant run-time security checking systems for SoCs are reported in [6], [7]. Those systems are limited by simple predicates, which do not support comparisons over time frames. Also, they did not discuss or present any counterintelligence capabilities. In addition, these systems are configured only with static security policies that do not change over time. On the other hand, our detection system can update the security policy during the run-time through monitoring the flow of the commands that are sent by the trusted IPs of the SoC.

Another direction of work relies on a design paradigm for facilitating the acquisition of trustworthy IPs through "proof-carrying code" (PCC) [16]. There, a set of security related properties are formulated, and a formal proof of these properties is crafted by the designer. Any undesired modification to the IP is likely to violate the proofs. In this approach, it is assumed that all attacks and data leakage can be predicted at design time, which is clearly shown to be insufficient in view of novel attack vectors like [2]. Along the same line of research, the concept of self-verifying ASICs is promoted in [17]. This work restricted itself to ASIC design and did not show ready adoption for heterogeneous programmable platforms. Our proposition, in contrast, can be quickly assimilated in an industrial SoC.

VII. Conclusion

This work introduces a novel hardware-based defense mechanism for security of SoCs. SoCINT is integrated with SoC through NoC fabric to detect run-time failures of a pre-defined security policy. What distinguishes SoCINT over other monitoring systems is its support of transition invariants, thereby enabling powerful security invariants. Moreover, detection of security failures trigger one of the counterintelligence activities to defeat active attacks. The configurable nature of SoCINT provides SoC designers with a tool to respond to newly discovered vulnerabilities. Experimental studies show that SoCINT is effective at detecting and recovering from system-level security attacks and incurs very low performance overhead.

Acknowledgement

This research is supported by National Research Foundation: NRF-BICSAF project (Project ID: NRF2016NCR-NCR001-006) and National Cybersecurity R&D Program (Project ID: NRF2014NCR-NCR001-30).

References

[1] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE design & test of computers*, vol. 27, no. 1, 2010.

[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[4] J. Rajendran, A. k. Kanuparthi, M. Zahran, S. Addepalli, G. Ormazabal, and R. Karri, "Securing processors against insider attacks: A circuit-microarchitecture co-design approach," *IEEE Design & Test*, vol. 30, no. 2, pp. 35–44, 2013.

[5] A. Das, G. Memik, J. Zambreno, and A. Choudhary, "Detecting/preventing information leakage on the memory bus due to malicious hardware," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 2010, pp. 861–866.

[6] M. LeMay and C. Gunter, "Network-on-chip firewall: Countering defective and malicious system-on-chip hardware," in *Logic, Rewriting, and Concurrency*. Springer, 2015, pp. 404–426.

[7] A. Basak, S. Bhunia, T. Tkacik, and S. Ray, "Security assurance for system-on-chip designs with untrusted ips," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1515–1528, 2017.

[8] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage," in *Hardware-Oriented Security and Trust (HOST), IEEE International Symposium on*. IEEE, 2012, pp. 49–54.

[9] A. Markuze, A. Morrison, and D. Tsafrir, "True iommu protection from dma attacks: When copy is faster than zero copy," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 249–262, 2016.

[10] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir, "Utilizing the iommu scalably," in *USENIX Annual Technical Conference*, 2015, pp. 549–562.

[11] J. Gaisler, "The leon processor user's manual," *Gaisler research Google Scholar*, 2001.

[12] M. Isomäki, *Processor Debugging Through Ethernet*. Chalmers tekniska högskola, 2005.

[13] A. T. Abdel-Hamid, S. Tahar, and E. M. Aboulhamid, "A survey on ip watermarking techniques," *Des. Autom. Embedded Syst.*, vol. 9, no. 3, pp. 211–227, Sep. 2004.

[14] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Automatic RTL-to-formal code converter for IP security formal verification," in *17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, Dec 2016, pp. 35–38.

[15] S. Bose, M. Raikwar, D. Mukhopadhyay, A. Chattopadhyay, and K.-Y. Lam, "Blic: A blockchain protocol for manufacturing and supply chain management of ics," in *IEEE International Conference on Blockchain*, 2018.

[16] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.

[17] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish, "Verifiable asics," in *IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 759–778.