Trustworthy Computing in a Multi-Core System Using Distributed Scheduling

D. McIntyre
Cleveland State University
Computer & Information Science
Cleveland, Ohio 44106, USA
d.mcintyre@csuohio.edu

F. Wolff, C. Papachristou, S. Bhunia
Case Western Reserve University
Electrical Engineering and Computer Science
Cleveland, Ohio 44106, USA
{fxw12,cap2,skb21}@case.edu

Abstract– *Hardware Trust is an emerging problem in semiconductor integrated circuit (IC) security due to widespread outsourcing and the stealthy nature of hardware Trojans. Conventional post-manufacturing testing, test generation algorithms and test coverage metrics cannot be readily extended to hardware Trojan detection. As a result there is a need to develop approaches that will ensure trusted in-field operation of ICs, and more generally trust in computing. We present a distributed software scheduling prototype, TADS (Trojan Aware Distributed Scheduling), to achieve a Trojan-activation tolerant trustworthy computing system in a multi-core processor potentially containing hardware Trojans. TADS is designed to be transparent to applications and can run on general purpose multicore PEs without modifications to the operating system or underlying hardware. TADS can, with high confidence, continue to correctly execute its specified queue of job subtasks in the presence of hardware Trojans in the multi-core PEs while learning the individual trustworthiness of the individual PEs. Specially crafted self-checking subtasks called bounty hunters are introduced to accelerate PE trust learning. Also, by learning and maintaining individual PE trustworthiness, the scheduler is able to achieve Trojan containment by scheduling subsequent job subtasks to PEs with high learned trust.*

I. Introduction

An issue has emerged in the trustworthiness of computing due to widespread outsourcing of the IC manufacturing processes to untrusted foundries in order to reduce cost. An adversary can potentially tamper a design in these fabrication facilities by the insertion of Hardware Trojans a maliciously inserted circuit which can be triggered under rare conditions and cause an error in the normal functionality of the original circuit [1], [2], [3]. With the rapid growth and complexity of ICs, a Trojan can be small enough to evade detection through traditional manufacturing tests [4], [5], [6]. These small Trojans are limited to flipping a few bits in the processor circuitry: arithmetic, instruction control logic, memory access control or bus logic. The Trojan in order to be clandestine, is triggered by a rare event or condition in the processors circuitry. The triggering is based on monitoring a few rare combination of signals in the processor circuitry. The rare event is crucial, otherwise it would be caught early during manufacturing test. Complex Trojans which match the sophistication of software Trojans will be detected at manufacturing test time, i.e. checking a login password, because the circuitry will occupy a large circuit area and consume additional power beyond manufacturing specifications and be caught early. Furthermore, the small Trojan can be time delayed, so that the destructive payload after triggering occurs many clock cycles later. The Trojan is randomly inserted in a few processor chips in order to further evade possible detection during manufacturing test time and when detected in field operation will appear as a random fault. This makes it difficult for detection by standard manufacturing sampling techniques or reverse engineering [7], [8].

Finally, the malicious Trojan circuitry is aware of fault tolerant logic in the processor, such as parity, error correction, triple modular redundancy majority-voting and encryption logic. The adversary has access to the processor layout at manufacturing and the Trojan temporarily deactivates the fault tolerant logic to avoid being detected or deliver the payload.

Due to widespread outsourcing and the stealthy nature of hardware Trojans there is a real need to develop approaches that will ensure trusted in-field operation of integrated circuits, and more generally bottom-line trust in computing. In particular, for applications with high trust computing requirements, measures must be taken to deal with ICs that potentially contain undetected malicious hardware during in-field use.

The advent of multi-core processing suggests its use in allowing parallel execution of the same functionality in order to verify correctness of results. Multi-core systems offer the additional benefit of concurrent redundancy so that as trust detection among the various cores are discovered, distributed software scheduling algorithms can be used to avoid low-trust cores. We use the multi-core platform in this paper, to propose a distributed software methodology, TADS (Trojan Aware Distributed Scheduling) to achieve a Trojan-aware trustworthy computing system in a multi-core system potentially containing hardware Trojans. We focus on the detection of faulty results arising from the activation of hardware Trojans and the high confidence subsequent computation of correct results.

We present a distributed scheduling prototype that can, with high confidence, continue to correctly perform its queue of general-purpose job subtasks in the potential presence of hardware Trojans in the multi-core PEs. Each PE contains identical job subtask scheduler code capable of managing subtask variant execution within the PE, including (if the PE is currently busy) flow of the subtask to a neighboring PE, variant result comparisons, and passing of correctly verified subtask results to output PEs of the multi-core. The distributed scheduler accomplishes the controlled scheduling and execution of variants of each subtask on different PEs and the subsequent comparison of results. When the results differ (a Trojan or transient fault has been detected), fault recovery is achieved by the scheduling of additional subtask variant executions and subsequent comparisons. By this process, the scheduler finally a) determines and maintains (learns) trust information in PEs that activated a Trojan, and b) determines with high confidence the correct value of the subtask and directs the result to output PEs. By learning and maintaining individual PE trustworthiness, the scheduler can achieve fault containment by scheduling subsequent job subtasks to PEs with high learned trust. TADS is a bottom-line approach that continues to execute subtask variants until result agreement is reached and therefore a high-confidence correct value is computed. Also the reason for the error (Trojan or transient fault) does not stop TADS from determining the correct value. However, we conservatively attribute each transient fault on a PE to be a Trojan (false-positive) in our multi-core learning process. Distinguishing Trojan from transient errors appears to be extremely difficult and until such work is done our conservative approach seems best.

Our proposed methodology is conceptually similar to approaches used in fault-tolerant systems [9], [10], [11]. However Trojans, unlike transient faults, are built into the logic of the multi-core, and can generate errors by circumventing detection logic. Also the existence of Trojans when detected can be learned (stored) and used by TADS to improve future scheduling of subtasks to PEs to achieve Trojan aware scheduling. Typically, reliable systems employ hardware techniques to address soft-errors, whereas TADS is a distributed software scheduler that bottom-line detects run-time errors (caused by either Trojans or transient errors) and runs on a general-purpose multi-core without modifications to either the operating system or underlying hardware.

A. Variants

TADS is based upon the scheduling and execution of two functionally equivalent variants of a subtask in two different PEs and the subsequent comparison of the results to determine hardware trust. The goal in producing a subtask variant is to produce a different variant code of the same subtask which exercises another different PE circuitry sufficiently differently that the likelihood of the original variant executing the same Trojan is extremely unlikely. Although the PEs may be heterogeneous, it is possible that they are identical. Variants can then be used to detect Trojans in identical PEs. A subtask variant can be any perturbation of the subtask that is functionally equivalent, i.e. for the same inputs produces the same outputs. Variants can be obtained by different instruction mixes obtained by combining both different instructions and different operands. For example, arithmetic subtract instruction (i.e. $A - B$) can be replaced with non-subtract instructions, i.e. A + negate(B).

B. Distributed Scheduler

The following is the rational behind the use of subtask variants in a multicore environment. See [12] for a more complete discussion. The property of rare Trojan activation suggests the unlikelihood of two functionally equivalent variant processes A and B, say, of a subtask both triggering the same Trojan. Therefore when two binary variants of a subtask are simultaneously executed on two different multicore PEs PE_A and PE_B, say, and their computed values V_A and V_B agree, it is highly unlikely that both executed the same Trojan (or different Trojan) and as a result obtained the same result (Fig. 1b).

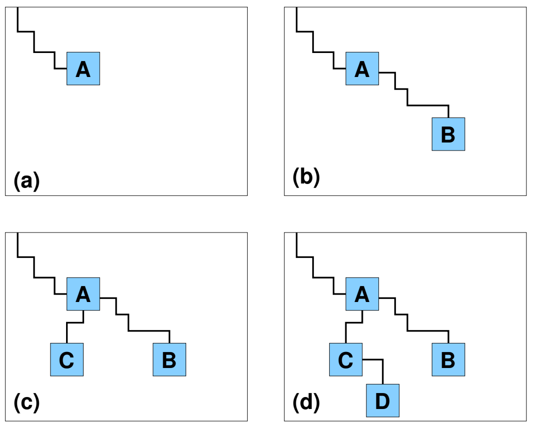

Figure 1. Example of placement of subtask variants

Thus it can be concluded with high confidence that $V_A = V_B$ is the correct value of the subtask and its value can be sent to the output PEs. Also if their computed values disagree, it is known with certainty that at least one of either PE_A or PE_B contained a Trojan (i.e. it is known with certainty that a Trojan has been detected). If they disagree, then the next step is to determine which of the PEs (or if both) contained a Trojan and reduce the trust level of the PE containing the Trojan. This can be accomplished by repeating the process and running an additional variant, C, say of the subtask (Fig. 1c). Then if $V_C = V_A$ (similar argument if $V_C = V_B$) then again it is highly unlikely that both variants C and A executed the same Trojan (or different Trojans) and as a result obtained the same result. Thus it is determined with high confidence, that $V_C = V_A$ is the correct value of the subtask and its value can be sent to the output PEs. Also if V_C is not equal to either V_B or V_A then it is known with certainty that at least two of the three PEs (PE_A, PE_B, PE_C) contained Trojans (this occurrence of two Trojans being activated for two variants of the same subtask would be extremely rare). It would be even rarer for all three PEs to contain Trojans. Clearly this process can theoretically be repeated several times to ultimately determine the PEs that contain Trojans and as a result adjust their trust levels (Fig. 1d). Of course the likelihood of having to repeat this process beyond the initial two variants A and B is almost zero. In reality, to have to repeat the process beyond variants A, B, and C will likely never happen. It also should be noted that eventually, frequently using only two variants, the variant execution process must end and a correct value for the subtask with high confidence computed.

C. Bounty Hunter

An accelerator was added to TADS to proactively detect hardware Trojans during in-field use at run time. Carefully crafted subtasks were designed to execute on PEs during idle time to ferret out existing Trojans. Such subtasks must be specially crafted to exercise the circuitry of a PE in an effort to both trigger a Trojan and then detect its existence. We call such independent special Trojan-searching subtasks bounty hunters. The ability of a bounty hunter subtask to detect a Trojan requires the subtask to know its correct values and after executing on any PE, to check the computed values against these known values (a simple example, though not nearly explorative enough might be a sorting algorithm with known input). If the values differ the bounty hunter would have detected a Trojan and as a result would reduce the trust of the PE.

II. Simulation Results

A java program was written to simulate trust determination and subtask scheduling. A combination of three jobs S15 with 15 subtasks, and two example DFGs, with 33 (DFG33) and 51 (DFG51) subtasks, respectively, generated by a random task graph generator were run all together on the three PE array sizes. PE array sizes of 4×4, 8×8 and 16×16 were simulated. Trojans were distributed as follows in the various PE arrays as indicated in Table I. We purposely did not randomly distribute the Trojans among the PEs in order to cause the maximum amount of Trojan activation in the system. For example, placing Trojans in rows 1 and $n - 1$ of the $n \times n$ PE array means that a subtask read by a PE in row 0 will (if it is busy) necessarily result in passing the subtask to either a row 1 or row n-1 neighboring PE which will contain a Trojan. Also, to further accentuate Trojan activation, the frequency of Trojan activation was set to 100% meaning that every time a subtask was executed on a PE containing a Trojan, the Trojan was activated causing incorrect results.

Table I
Location of Infected Trojans

$n \times n$	Trojan Distribution within the PE Array
4×4	All PEs in rows 1 and 3 contain Trojans
8×8	All PEs in rows 1 and 7 contain Trojans
16×16	All PEs in rows 1 and 15 contain Trojans

To demonstrate the effectiveness of TADS in learning PE trust and using the trust information to better schedule subsequent tasks to PEs with higher trust levels we simulated a slightly different version, TU (Trojan Unaware) which did not use this extra trust information. As a result TU will occasionally schedule subtasks to PEs whose trust is low whereas TADS will use known PE trust levels to avoid such assignments. Both versions used variants to detect the existence of Trojan activation and did any necessary additional scheduling of variants to finally correctly compute subtask values with high confidence. As a benchmark, we also ran a simple no-variant scheduler (NV), that did not do any execution of variants, trust determination, or Trojan avoidance. This scheduler merely routed subtasks to free PEs where they are executed (without knowing whether or not any Trojans were triggered). While this scheduler executed subtasks reasonably quickly (as there are no variants) there was no determination of validity of results and no attempts to correct them. Additionally, if there were any Trojans in the multi-core, the trustworthiness of the NV scheduler would diminish to the point where the system would become useless. Table II shows the time for completion of job S15 using TU, TADS and NV scheduling for various array sizes. The TU and TADS schedulers computed the correct values of all submitted jobs successfully despite frequent Trojan activation. TADS was run on each of the three pristine multi-cores (before bounty hunters were run). Then each of the $n \times n$ PE multi-cores was flooded with 3n bounty hunters causing PE trusts to be learned. Subsequently TADS was run on the multi-cores after trust was learned by running the bounty hunters. Comparing the results in Tables II it is clear that significant reductions of 30% in job completion times were achieved by TADS scheduling over TU scheduling since TU more often routed subtasks to PEs containing Trojans necessitating the execution of further variants. As expected the benchmark NV scheduler completion time was only 59 time cycles for all PE array sizes, 40% faster than schedulers TU and TADS. However the actual values computed for all jobs by NV were also incorrect and Trojans were not detected.

Table II
S15 Job Scheduling

PE Array Size	Job Completion Times	
	TU Unaware	TA (Trojan Aware) Before BH/After BH/NV
4×4	102	123 / 84 / 59
8×8	102	123 / 86 / 59
16×16	102	178 / 86 / 59

A combination of three jobs S15 with 15 subtasks, and two example DFGs, with 33 (DFG33) and 51 (DFG51) subtasks, respectively, generated by a random task graph generator were run all together on the three PE array sizes. Again, TADS was run both before, and after the bounty hunters were run. Table III illustrates TU, TADS and NV scheduling for the combination of three jobs. Again there is a significant decrease in job completion times (sometimes as large as 40%) over TU when TADS is used. Both schedulers TA and TU produced values that were correct with high confidence. Again as expected the benchmark NV scheduler completion time was significantly less than schedulers TU and TADS (ranging between 1 and 3 times faster) for all PE array sizes. However the actual values computed for all jobs by NV were also incorrect and Trojans were not detected.

III. Conclusions

We have presented a Trojan aware distributed scheduler prototype, TADS, that achieves trustworthy computing in a multi-core system potentially containing hardware Trojans. TADS is transparent to the applications running, and can run on general-purpose multi-core PEs without any modifications to either the operating system or underlying hardware. Also, by learning and maintaining individual

Table III
Combination Job Scheduling

	PE Array size	Job S15 Completion Time	Job DFG51 Completion Time	Job DFG33 Completion Time
TU	4×4	294	508	493
	8×8	204	350	336
	16×16	204	368	359
TADS	4×4	198 / 176 / 71	336 / 316 / 121	260 / 253 / 114
	8×8	140 / 93 / 59	218 / 172 / 111	188 / 166 / 109
	16×16	158 / 90 / 59	211 / 169 / 108	232 / 154 / 107

PE trustworthiness, TADS is able to achieve Trojan containment by scheduling subsequent job subtasks to PEs with high learned trust. Simulation results show that TADS can be an extremely effective software approach to Trustworthy computing in the presence of Hardware Trojans. However, due to the significant cost of variant management, TADS is primarily useful for those applications that require high trust computing in an age where Hardware Trust in ICs is becoming problematic, and do not have a real time component, or where absolute performance or overhead issues of area/power are not important. High confidence computing in multi-core environments can be achieved using distributed scheduling through subtask variant management. Unlike traditional fault tolerant variant techniques which focus on efficiently identifying transient errors, our approach identifies and locates persistent hardware Trojans in PEs and uses the learned PE trust to efficiently schedule future subtasks for Trojan avoidance.

References

[1] J. Markoff, “Old trick threatens the newest weapons,” *New York Times, Tuesday, October 27, section D1*, pp. 1,4, 2009.

[2] P. Marks, “Hardware trojans could sabotage microchips from within,” *New Scientist*, vol. 203, pp. 5–6, July 2009.

[3] S. Adee, “The hunt for the kill switch,” *IEEE Spectrum*, pp. 34–39, May 2008.

[4] F. Wolff, C. Papachristou, S. Bhunia, and R. Chakraborty, “Towards trojan-free trusted ics: problem analysis and detection scheme,” *Design, Automation, and Test in Europe (DATE’08)*, pp. 1362–1365, Mar. 2008.

[5] R. Rad, J. Plusquellic, and M. Tehranipoor, “A sensitivity analysis of power signal methods for detecting hardware trojans under real process and environmental conditions,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. PP, no. 99, pp. 1–1, Oct. 2009.

[6] D. Rai and J. Lach, “Performance of delay-based trojan detection techniques under parameter variations,” pp. 58–65, july 2009.

[7] R. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, “Mero: A statistical approach for hardware trojan detection,” *11th International Workshop Cryptographic Hardware and Embedded Systems (CHES’09). Lecture Notes in Computer Science 5747 (Springer)*, Sept. 2009.

[8] J. Kumagai, “Chip detectives,” *IEEE Spectrum*, vol. 37, no. 11, pp. 43–49, Nov. 2000.

[9] Y. Ma and H. Shou, “Efficient transient-fault tolerance for multithreaded processors using dual-thread execution,” *Intl. Conf. on Computer Design (ICCD’06)*, pp. 120–126, Oct. 2006.

[10] P. M. Wells, K. Chakraborty, and G. S. Sohi, “Adapting to intermittent faults in multicore systems,” in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, Mar. 2008, pp. 255–264.

[11] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, “Swift: software implemented fault tolerance,” *IEEE Intl. Symp. on Code Generation and Optimization (CGO’05)*, pp. 243–254, Mar. 2005.

[12] D. McIntyre, F. Wolff, C. Papachristou, S. Bhunia, and D. Weyer, “Dynamic evaluation of hardware trust,” *2nd IEEE International Workshop on Hardware-Oriented Security and Trust (HOST’09)*, pp. 20–27, July 2009.