Timing Channels through Shared Hardware Accelerators: Attacks and Protection

Tao Chen and Kai Wang
{tc466, kw296}@cornell.edu
Computer Systems Laboratory
Cornell University, Ithaca, NY 14853, USA

Abstract

Hardware specialization in the form of accelerators offer significant improvement in performance and energy efficiency compared to general-purpose processors. However, there is usually a lack of security guarantees by the existing hardware design methodologies. In this report, we show that shared hardware accelerators are vulnerable to timing channel attacks and can leak sensitive information. We demonstrate a covert channel attack on a real system consisting of a multicore processor and a shared hardware accelerator. We designed a communication protocol which allows two users of an accelerator to send arbitrary messages over the covert channel. To mitigate timing channel attacks, we propose a novel design automation approach that removes interference and allows hardware accelerators to be shared securely. Experimental results show that the proposed approach eliminates timing channels in shared hardware accelerators with small performance and area overhead.

1 Introduction

As Dennard scaling is reaching its limits, general-purpose processors can no longer benefit from CMOS technology improvements to achieve better performance and energy efficiency. On the other hand, transistor density continues to grow, which leads to the dark silicon problem: we don't have enough power budget to power on all transistors on a chip [6]. To overcome this problem, hardware specialization in the form of accelerators has emerged as a promising approach that offers orders of magnitude better performance and energy efficiency compared to general-purpose processors [2, 8, 10]. The result of this trend is that modern computing substrates are becoming increasingly heterogeneous, which often includes a multicore processor, GPGPUs, hardwired accelerators for specific tasks, and FPGA fabric for building reprogrammable accelerators [3].

Accelerators in a system often need to be shared among multiple users. The reason is twofold: First, as hardware accelerators become popular, the complexity and size of these accelerators continue to grow due to rapid scaling in application requirements. Moreover, hardware accelerators are often tightly integrated within a multi-core architecture and are required to service multiple users. The complexity of accelerators as well as the number of users to be serviced make it unrealistic to create a dedicated hardware accelerator for each user. Second, it is often unnecessary replicate accelerators for each user. Accelerators are a form of hardware specialization, which means they are only suitable for specific tasks, which in turn means they are not always needed by a user. Sharing accelerators increases their utilization and help amortize their cost.

However, as with many shared resources, sharing accelerators might introduce security issues since multiple users of the same accelerator could be mutually distrusting. As a result, the security of accelerators must be carefully considered.

Confidentiality is an important property of secure hardware because it prevents the disclosure of secret, sensitive, private, or proprietary information to unauthorized users or entities [7]. Unfortunately, shared resources can be exploited to leak information through timing channels, creating a breach of confidentiality of particular concern. A *timing channel* is a communication channel

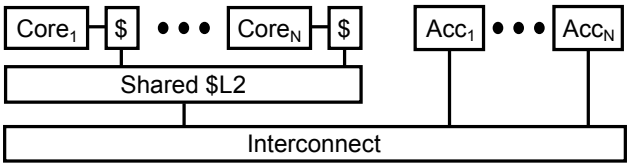


Figure 1: System setup: multi-core processor with shared hardware accelerators

based on timing information. A timing channel passes information using the speed at which things happen. Since a user of shared resources can affect the speed at which others users' requests are processed, shared resources can often be used as a timing channel. For example, Wang et al. [11] demonstrate that one adversary is able to covertly communicate with another adversary through a shared memory controller by dynamically scheduling its memory requests so to indirectly affect the memory access latency of the other adversary. The other adversary can deduce the exact sequence of the message sent even in the absence of an explicit communication channel through memory by design.

Shared hardware accelerators are similarly vulnerable to timing channel attacks because their resources are arbitrated among multiple users. The access latency of one user is therefore correlated with accesses from other users sharing the same accelerator. Inferring the timing of the accesses reveals secret or proprietary information about the design and allows adversaries to collude in staging sophisticated attacks. Such issue is especially concerning to the defense community where hardware accelerators are widely deployed in mission-critical applications.

Many of today's hardware accelerators are developed with a high-level design methodology for which high-level synthesis (HLS) plays an integral role in synthesizing and optimizing these hardware accelerators. While much of the emphasis has been placed on high performance, small area, and low power [4, 5], the security of these synthesized accelerators as well as the design methodology in general has been largely overlooked. Amid concerns over the global development process in which malicious hardware logic (Trojan) can be inserted unscrupulously into any system, the lack of security guarantee renders hardware accelerators extremely vulnerable to runtime hardware attacks.

In this report, we address the problem of timing channel attacks and their countermeasures on shared hardware accelerators. Our major contributions are:

1. To our best knowledge, we are the first to study timing channels through a shared hardware accelerator and to examine timing channels in the context of HLS.
2. We propose a realistic attack model and demonstrate a practical timing channel attack through a shared hardware accelerator on a real system. We designed a communication protocol that allows two users of an accelerator to send arbitrary messages over the covert channel.
3. We propose an efficient countermeasure that leverages HLS to mitigate the attack and present proofs on the effectiveness of these proposed countermeasures.

2 Attack Model

In this section, we discuss the attack model used in this report. We first show the system setup, identify the source of timing channels, then discuss two classes of timing channel attacks under this setup.

2.1 System Setup

Figure 1 shows the system setup we consider in this report, which consists of a multi-core processor with shared hardware accelerators. The processor cores are connected to the accelerator via an interconnection network. We assume each core runs a user program that can send requests to the shared hardware accelerator. The interconnection network has some internal logic to perform arbitration and coordinate accesses from different cores. A request starts processing in the accelerator after being granted access by the arbitration logic. When the accelerator finishes processing the request, it sends response back to the core. We assume the program running on that core has the capability to measure the time between sending the request and receiving the response. Also, we assume the accelerator can only process one request at a time. If a request is not granted access to the accelerator, either because the accelerator is busy, or because another competing request was granted access, the request is queued and competes for access next time when the accelerator becomes available. Thus, the request-to-response latency can be affected by whether other cores are requesting access to the accelerator or not. This interference between two requests is the source of timing channels.

In our setup, the N processor cores are divided into K security domains. A *security domain* contains one or more cores running user programs that trust each other. Different security domains can run programs that are mutually distrusting and want to keep secrets from programs in other security domains despite sharing the hardware accelerator. Thus, our goal is to prevent timing channels between two security domains.

2.2 Timing Channel Attacks

We consider two classes of timing channel attacks that can be carried out using a shared hardware accelerator.

Side Channel Attacks In side channel attacks, an attacker tries to obtain secret information, which the victim does not intend to leak [7]. For example, the victim may use an accelerator to speed up some cryptographic functions. The attacker deliberately creates contention by also sending requests to the accelerator, and measure its own request-to-response latency to figure out the access pattern of the victim, which could potentially correlate with secret owned by the victim.

Covert Channel Attacks In covert channel attacks, an insider (*Trojan*) wishes to leak some secret information to an outsider (*Spy*). However, they are not allowed to directly communicate with each other. In this scenario, they can use a covert channel to circumvent this limitation. For example, the *Trojan* can create contention patterns on the accelerator according to the information it wants to send, and the *Spy* can sense these patterns by measuring its own timing information when accessing the shared accelerator.

3 Implementing a Covert Channel Attack

In this section, we present an example attack in which the *Trojan* and *Spy* collude to send information over an accelerator covert channel. We will first show how interference affects access latency, then we introduce a communication procotol that enables arbitrary messages to be sent over the covert channel. After that, we will talk about some implementation issues such as clock synchronization. Finally, we will show an implementation of the covert channel attack on Zedboard.

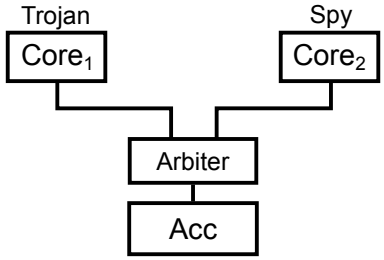


Figure 2: System setup for covert channel attack

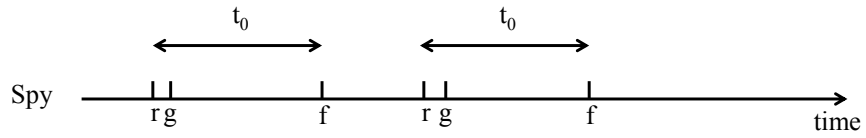


Figure 3: Spy Access Time Measurement without Contention. r: request, g: grant, f: finish.

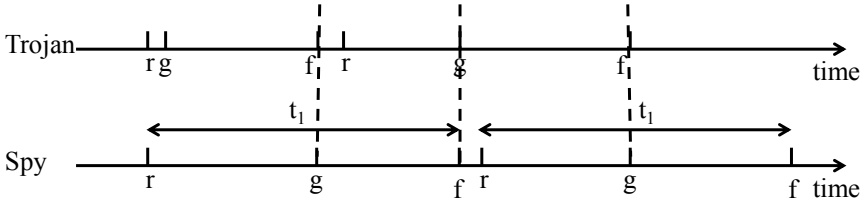


Figure 4: Spy Access Time Measurement with Contention. r: request, g: grant, f: finish.

3.1 Interference and Access Latency

Figure 2 shows a simplified system model with two processor cores and a shared accelerator. The arbitration logic in the interconnection network is modeled by an arbiter that connects the cores to the accelerator. The arbiter performs round-robin arbitration. The *Trojan* runs on Core 1 and the *Spy* runs on Core 2. The *Trojan* tries to send messages to the *Spy*. The *Trojan* does this by manipulating the number of requests it sends to the accelerator, which affects the the *Spy*'s access latency. The *Spy* will try to recover the message after collecting and analyzing its access latency. Explicit communication between two cores is not allowed so direct transmission is impossible. We will discuss how *Spy* observe the timing difference and use that to recover the message step by step.

In general, *Trojan* will send requests to the accelerator depending on the message content, while *Spy* will always be sending requests to sense contention pattern regardless of what *Trojan* does. Figure 3 shows *Spy*'s access latency when *Trojan* is not sending requests, i.e. no contention to the shared resource. We can see that whenever *Spy* sends a request, the arbiter will return a grant signal immediately because there is no competition for resource, and after a certain amount of time, the finish signal is sent by the accelerator to *Spy* through the arbiter. We denote the time elapsed from r to f as t_0. Figure 4, on the other hand, shows the access latency when *Trojan* is sending requests. We can see that *Trojan* first gets the grant and *Spy* has to wait until *Trojan* finishes, then grant is transferred to *Spy* and *Trojan* has to wait if it has another request. We denote the latency from r to f as t_1 in this case, and t_1 is clearly longer than t_0 because of the nonnegative waiting time caused by resource contention.

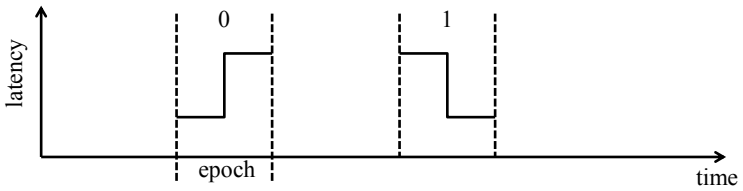


Figure 5: Manchester Codes for 0 and 1

3.2 Communication Protocol

Encoding one bit We could interpret the message as binary "0" or "1" whenever a t_0 or t_1 latency is detected respectively, but it is unreliable due to unsteady latency returned from the accelerator each time. In other words, the observed latency does not come with a clock signal, and the *Spy* wouldn't know where to start and end measuring. To solve this problem, we encode binary "0" or "1" using Manchester Coding [9]. Manchester Coding is self-clocking so we don't need to send a separate clock signal. Figure 5 shows the way Manchester Code encodes "0" and "1". For example, when *Trojan* has a bit to send, if the bit is "0", it will first not send any request for the first half of the defined period "epoch". In the second half of epoch, it will keep sending requests. In this case, the access latency recorded by the *Spy* will be first low then high, just like the Manchester code for "0", and "1" is encoded as first high then low.

Encoding characters and strings With the help of Manchester Coding, we are able to encode any binary string. But usually human communicate with natural languages, so we need a way to encode characters and strings. In our protocol, we added support for translating between ASCII and binary strings. For example, if the user input is the letter "y", it will first be translated to "01111001", and then the *Trojan* will send requests following Manchester Coding protocol as we described earlier. A message in our protocol is just an ASCII string, which is sent as a concatenation of ASCII characters using Manchester coding. Furthermore, in order to signal the begin and the end of a transmission to the *Spy*, we enforced an agreement on the *begin symbol* and the *end symbol* - 8 straight "0"'s signal the begin and 8 straight "1"'s signal the end of transmission (neither of them are used to encode printable ASCII characters).

3.3 Clock Synchronization

Given that *Spy* can now detect perfect low and high latencies, there is one very important issue to solve in order to correctly decode the message over a long period of time - clock synchronization. Note that the "clock" here refers to the clock of the latency signal, not the clock of the circuit (i.e. one clock period is one epoch). We can further break down clock synchronization into clock matching and clock skewing. Clock matching happens at the beginning of a message transmission. It is meant for the *Spy* to acquire clock from the encoded signal. In our protocol, *Spy*'s access time will always been low if *Trojan* is not sending any request. But when *Trojan* begins to send the begin symbol 8 "0"'s, *Spy* will detect a change in access latency from low to high and then from high to low and so on, but this wave can also be recognized as from high to low then from low to high and interpreted as "1"'s. This is due to the fact that *Spy* has no idea whether the first low is caused when *Trojan* is idle, or *Trojan* is active but not sending anything on purpose. Figure 6 also shows that the string of "0"s can be interpreted as a string of "1"'s if the clock offsets with half epoch. We solved this problem by taking advantage of a property of Manchester coding instead of creating a new protocol or sending extra clock information. In detail, before sending the start symbol, we let *Trojan* sends a clock synchronization message with intervening "0" and "1"'s, i.e. "01010101...". In this sequence, all low-to-high or high-to-low transitions are guaranteed to

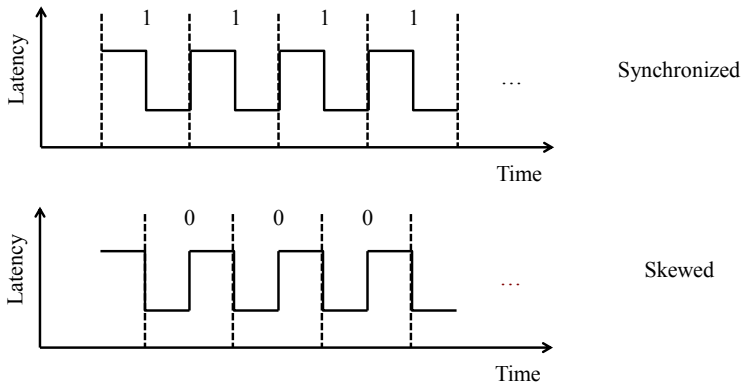


Figure 6: Different Interpretation When Clock Skewed

happen at approximately half of an epoch. With this clock synchronization message added to the beginning of each transmission, *Spy* can reliably acquire clock from the encoded signal and set its clock to match with *Trojan*'s clock.

With the clock matching technique, *Spy*'s clock is fairly accurate at the beginning, but over time, *Spy*'s clock may skew due to the fact that *Trojan*'s request-to-finish time may vary because of potential delay added to the accelerators like context switching in OS or hardware glitches. Thus *Trojan* may begin the next epoch earlier or later. This uncertainty will accumulate and cause the clock in *Spy* to skew. Therefore, we implemented a method to check for clock skewing and automatically adjust *Spy*'s clock. As we know that from clocking matching, if begin and end time of an epoch is synchronized at the very beginning, we can expect a transition at the middle of an epoch. However, if the transition happens either earlier or later than the midpoint, we know that the clock is skewed. When the amount of skew exceeds some threshold, we can adjust the clock boundary to shift the transition back to the midpoint of an epoch.

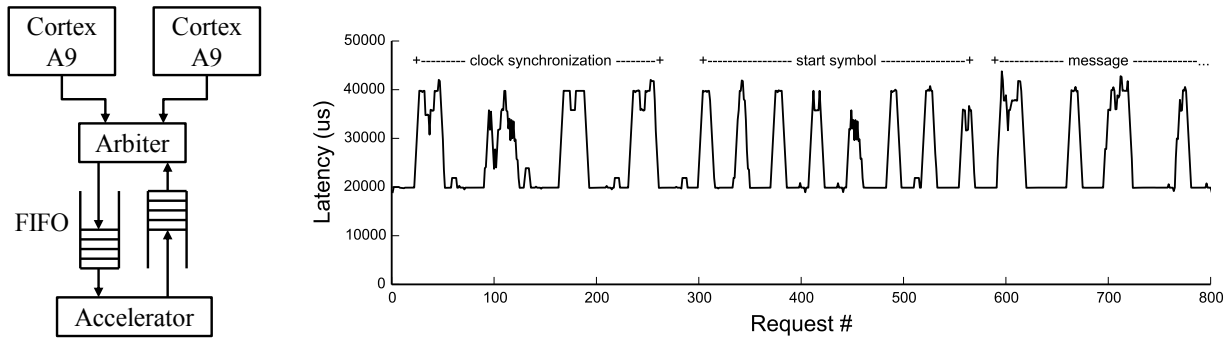
3.4 Zedboard Implementation

To demonstrate a practical covert channel attack on a real system, we implemented our covert channel design on Zedboard, which comes with a Xilinx Zynq-7000 All Programmable SoC containing a dual-core ARM Cortex-A9 processor and FPGA logic. We utilized a third-party infrastructure called Xillybus [1] to implement the interface between CPU cores and the accelerator, which is on FPGA logic. The Zedboard system setup is shown in Figure 7a. Xillybus provides two Linux devices at /dev/xillybus_read_8 and /dev/xillybus_write_8 to read data from and send data to the accelerator. The two cores can only access these devices in round-robin fashion.

We had some clock synchronization issues that caused the message interpretted by *Spy* to be incorrect. Those issues were mainly caused by the Linux OS background threads, which caused the access latency to vary unexpectedly. We had to tune our parameters (mainly the epoch duration) to achieve reliable communication. Figure 7b shows a segment of the latency observed by the *Spy*. We can clearly see phases of clock synchronization, transmitting the start symbol, and transmitting the actual message.

4 Protection Scheme

Based on the attack model in Section 2, the primary objective of any protection scheme is to eliminate the interference at the shared hardware accelerator between any two of the N cores. We define T_i, the *service interval* for core i, as the number of cycles between consecutive requests from



(a) (b)

Figure 7: Zedboard Implementation: (a) System Setup. (b) Spy Access Latency Measured on Zedboard.

core i. Assuming that the hardware accelerator incurs a constant latency of C cycles and is shared by N cores each of which belongs to a distinct one of K security domains (i.e. $K = N$), it follows from the problem setup that the shared accelerator achieves a service interval $T_i = N \cdot C$ cycles for each core. The accelerator is able to accept a new set of data input every C cycles. However, since it is shared and arbitrated among N cores, each core is only able to access the accelerator every $N \cdot C$ cycles in the worst case.

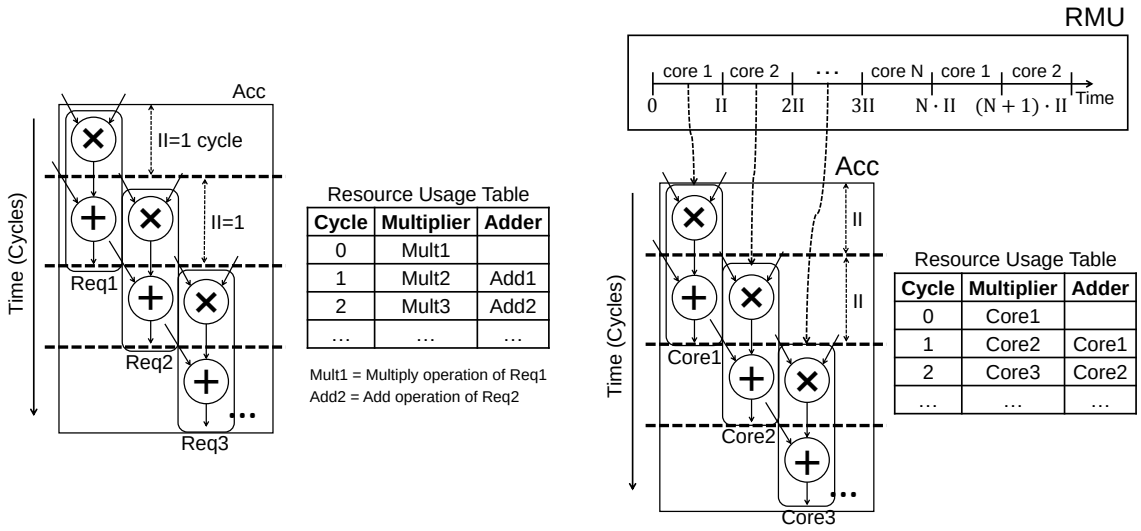
We further define L_i, the *service latency* for core i, as the number of cycles between core i's sending the request to the shared hardware accelerator to its receiving a response from the accelerator, also known as the request-to-response latency. If the accelerator is free when core i issues the request, then the service latency $L_i = C$ cycles for core i. If there are requests from k (for $k < N$) distinct cores with priorities higher than that of core i, then the round-robin policy would grant one request from each of the k cores before allowing core i's request to proceed, resulting in a service latency $L_i \geq k \cdot C + C = (k + 1) \cdot C$ cycles for core i. *Non-interference* between core i and core j at the shared accelerator is achieved if L_i is independent of requests from core j for the shared accelerator.

We would like to implement a shared hardware accelerator that executes requests from core i in the same manner regardless of whether there exist concurrent requests from the other cores. Let $p_{L_i}(l_i)$ denote the probability density function of the service latency for core i, and let $p_{Z_j}(z_j)$ denote the probability that the resource is occupied by core j. Non-interference implies that $p_{L_i|Z_j}(l_i|z_j) = p_{L_i}(l_i)$ for $i \neq j$ and $i, j = 1, 2, ..., N$.

4.1 Baseline Approach

To eliminate the interference among cores sharing the same hardware accelerator, a straightforward approach is to replicate the accelerator. By creating N copies of the same accelerator, one for each core, each core now has its own dedicated accelerator and no longer needs to compete for a shared resource. In this case, the service latency and service interval is identical for each core, where $T_i = L_i = C$ cycles. In fact, each core has exclusive access to its own dedicated accelerator, thus $p_{L_i|Z_j}(l_i|z_j) = p_{L_i}(l_i)$. As such, the accelerators satisfy the property of non-interference and are free of timing channels.

However, replication incurs significant area overhead proportional to the number of cores sharing the accelerator, which could quickly become unsustainable. It is necessary to intelligently optimize the design to achieve non-interference while minimizing the area overhead.



(a) (b)

Figure 8: Enhanced approach: (a) Resource Management Unit time-multiplexes requests to the pipelined accelerator. (b) Perform pipeline scheduling. The schedule of the accelerator is shown on the left with corresponding resource usage on the right.

4.2 Enhanced Approach

For the enhanced approach, we consider an HLS-based protection scheme that achieves non-interference with small area overhead. Specifically, we propose and address the following problem:

Given: (*1*) A data flow graph (DFG) representing the design to be synthesized. (*2*) The number of cores N that will be sharing the synthesized accelerator. (*3*) A target service interval C for each core. (*4*) A set of timing and resource constraints for the design.

Goal: Synthesize the shared hardware accelerator with the target service interval that satisfies the property of non-interference while minimizing the resource usage.

The baseline approach assigns each resource exclusively to one of the N cores. Thus much of the hardware may be idle at any given time since it cannot be shared across cores. To reduce this overhead, we can instead pipeline the accelerator to accept a new request every II cycles. Figure 8a shows the schedule of a pipelined accelerator with an initiation interval (II) of 1 cycle for the execution of three requests. The pipelined accelerator only requires one multiplier and one adder because these resources are shared among cores.

While it is desirable to reduce the area of the shared accelerator via pipelining and sharing, we still need to ensure non-interference. Since the synthesized pipeline is shared by N cores, we need some mechanism to assign requests from the N cores to the pipeline. To achieve this goal, we replace the round-robin arbiter with a resource management unit (RMU) that time-multiplexes the requests from different cores. Arbitration interference can be eliminated if only a specific core's request can be granted for a specific time slot. Specifically, we propose a time-multiplexing scheme in which time is divided into slots each of II cycles, during each of which the RMU can issue one request from a particular core. We say that this core is active during this time slot.

As shown in Figure 8b, the RMU manages incoming requests such that the cores take turn to issue requests to the accelerator. The length of each turn is II cycles. At the beginning of each slot, if the corresponding core has a pending request, the request is issued to the accelerator. If

Benchmark		Interval (cycles)	Clock Period	Resource Usage			
				Slice	**LUT**	**FF**	**DSP48**
`dir`	Duplication	6	8.89ns	336	1190	396	18
	Proposed	6	6.15ns	167 (-50%)	581 (-51%)	228 (-42%)	16 (-10%)
`lee`	Duplication	8	9.58ns	250	882	302	36
	Proposed	8	8.79ns	146 (-41%)	518 (-41%)	290 (-4%)	14 (-61%)
`pr`	Duplication	6	9.87ns	232	910	524	20
	Proposed	6	8.20ns	107 (-54%)	374 (-54%)	133 (-74%)	16 (-19%)
`wang`	Duplication	8	9.45ns	226	816	272	20
	Proposed	8	9.60ns	112 (-50%)	397 (-51%)	148 (-45%)	15 (-25%)
`mcm`	Duplication	4	9.54ns	338	860	472	52
	Proposed	4	8.24ns	121 (-64%)	273 (-68%)	231 (-51%)	42 (-19%)
`feig`	Duplication	8	9.67ns	3130	11468	4344	254
	Proposed	8	9.66ns	1765 (-43%)	6379 (-44%)	1787 (-59%)	130 (-49%)

Table 1: Performance and resource usage comparison

not, no request is issued. In general, core i is allowed to send requests to the accelerator at cycle $t = (i-1) \cdot II + j \cdot N \cdot II$ where $j = 0, 1, 2, ...$

It is evident that the number of cycles between the initiation and granting of a request depends solely on the time at which the request is initiated, and not associated with whether the accelerator is occupied. As a result, the service latency L_i for core i is the sum of the time between initiation and granting, and the actual latency of the accelerator C. Because C is constant, and L_i is also independent of Z_j, we know $p_{L_i|Z_j}(l_i|z_j) = p_{L_i}(l_i)$, which implies non-interference.

In addition to non-interference, it is important to synthesize the pipelined accelerator so that it achieves a service interval that is no worse than that of the baseline approach, which achieves a service interval $T_i = C$ cycles. The time-multiplexing scheme allows core i to be active every $N \cdot II$ cycles. In other words, core i is allowed to issue a new request every $N \cdot II$ cycles, and the pipelined accelerator is able to accept a new set of inputs from core i every $N \cdot II$ cycles. It follows that $T_i = C = N \cdot II$, and $II = \lfloor C/N \rfloor$. Based on this analysis, we must synthesize the pipeline targeting an $II = \lfloor C/N \rfloor$ cycles.

4.3 Evaluation of Protection Schemes

We evaluate our proposed protection scheme in terms of security guarantee, performance and resource usage. We use a set of computationally intensive accelerators as our benchmarks, which includes implementations of various digital signal processing algorithms. We use Vivado HLS 2014.2 to synthesize the behavioral description of these accelerators into RTL, then implement them on an FPGA. The target FPGA device is Zynq-7000. The system configuration is similar to the covert channel example in Figure 2. The Zynq-7000 has two ARM cores connected to the accelerator through the Resource Management Unit (RMU). The RMU is implemented in Verilog HDL, and coordinates accesses to the accelerator between the two cores, which belong to two security domains.

Security Evaluation Our protection scheme eliminates interference between requests from different security domains. If interference is eliminated, the response time a security domain sees should be independent of requests from other security domains. To test whether our protection scheme works, we again run the covert channel attack as in Section 3.

From our experiments, the *Spy*'s response latency no longer correlates with the access pattern of the *Trojan*. The only variation in response latency is from the misalignment of the request time and the start of an interval. The covert channel is cut off.

Date	Milestone	Contributor
11/07	Project idea - AES timing channel attacks	Tao
11/09	Project abstract	Tao, Kai
11/13	Researched on AES timing channel attacks	Tao, Kai
11/20	Topic changed to covert channel	Tao, Steve
11/30	Covert channel framework, transmission protocols, protection scheme created	Tao
12/01	Attack model and protection scheme tested, data collected	Kai
12/02	DAC paper submission	Tao, Steve
12/12	Project presentation	Tao, Kai
12/17	Project report	Tao, Kai

Table 2: Project Timeline

Performance and Resource Usage Table 1 shows the performance and resource usage comparison between duplication-based approach and our proposed protection scheme. During synthesis, the constraints are set such that two schemes achieve the same service interval. We set target clock period to 10ns. The clock period and resource usage numbers are obtained by feeding the RTL generated by high-level synthesis to Vivado for logic synthesis, placement and routing.

For each benchmark, the first row shows resource usage for the baseline protection scheme using on duplication, and the second row shows the resource usage of our proposed scheme based on pipelining and time multiplexing. We can see that compared to duplication-based approach, our scheme has much lower resource usage overhead, saving 50% slices, 50% LUTs, 45% FFs, and 30% DSP48E blocks on average. In fact, our approach only adds a small resource overhead to the unprotected design. This is mainly due to the sharing of resources between pipeline iterations.

We also evaluate the tradeoff between service interval and resource usage. Appendix A contains more details.

5 Project Management

Table 2 shows the timeline of our project. At first we planned to work on external timing channels on cryptographic hardware, where an adversary measures the execution time of the hardware to obtain secret information. However, after some initial evaluation, we concluded that most cryptographic hardware are not vulnerable to external timing channel attacks. After that, we changed our topic to timing channels through shared hardware, where contention between users is the source of timing channel.

We thank Steve Dai for his help in developing the project idea, providing insights during our discussions, and submitting a paper based on this project to DAC.

6 Conclusion

While hardware accelerators promise significant improvement in performance and energy efficiency, their security needs to be carefully examined to ensure a trustworthy system. In this report, we identify the issue of timing channels in shared hardware accelerators and demonstrate a practical attack that exploits the interference between processes of different security domains through a shared accelerator. We propose an enhanced protection scheme that leverages HLS to remove such interference to synthesize a design free of timing channel and prove the effectiveness of this countermeasure. Experiments demonstrates that our proposed scheme is able to efficiently achieve non-interference with low hardware overhead.

References

[1] Xillybus. `http://xillybus.com/`. Accessed: 2014-12-06.

[2] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: a Small-Footprint High-Throughput Accelerator For Ubiquitous Machine-Learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–284. ACM, 2014.

[3] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236. IEEE Computer Society, 2010.

[4] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, Apr. 2011.

[5] P. Coussy and A. Morawiec. *High-Level Synthesis*. Springer, 2010.

[6] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Int'l Symp. on Computer Architecture (ISCA)*, pages 365–376. IEEE, 2011.

[7] R. B. Lee. Security Basics for Computer Architects. *Synthesis Lectures on Computer Architecture*, 8(4):1–111, 2013.

[8] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *Int'l Symp. on Computer Architecture (ISCA)*, pages 13–24, 2014.

[9] A. S. Tanenbaum. *Computer Networks, 4th Edition*. Prentice Hall, 2003.

[10] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: Reducing the energy of mature computations. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218. ACM, 2010.

[11] Y. Wang, A. Ferraiuolo, and G. E. Suh. Timing Channel Protection for a Shared Memory Controller. *Int'l Symp. on High Performance Computer Architecture (HPCA)*, pages 225–236, 2014.

Appendix

A Tradeoff between Service Interval and Resource Usage

Here we study how different service interval requirement impacts resource usage of our proposed approach. Figure 9 shows the resource usage for `feig` when varying service interval from 5 to 15. We see that the general trend is when we reduce service interval, resource usage becomes higher for both the baseline duplication approach and our proposed approach. Nevertheless, our proposed approach always has much lower resource overhead compared to duplication. This demonstrates that our approach is flexible in terms of meeting different service rate requirements, while consistently providing security guarantees.

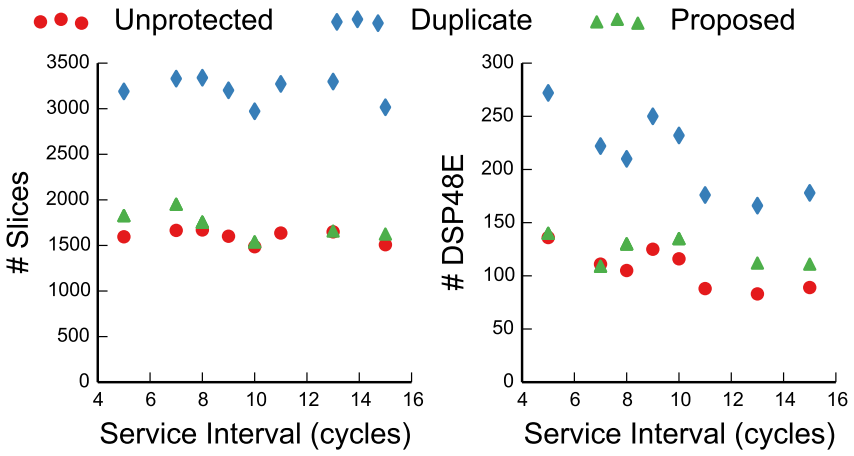


Figure 9: Resource usage when varying service interval for `feig`

This graph also shows slices usage grows slower than DSP48E usage as we reduce the service interval. This is mainly due to the fact that with a smaller service interval, more resources (e.g. multipliers, adders) becomes dedicated to meet the higher throughput requirement. While this increases the slices used to implement adders and other operators, it reduces the slices used to implement multiplexers. Thus the slices usage grows slower than the number of operators (e.g. DSP48E blocks).