CIDPro: Custom Instructions for Dynamic Program Diversification

Thinh Hung Pham*, Alexander Fell*, Arnab Kumar Biswas†, Siew-Kei Lam*, and Nandeesha Veeranna*
*Nanyang Technological University, Singapore
Email: {pham_ht, afell, vnandeesha}@ntu.edu.sg, siewkei_lam@pmail.ntu.edu.sg
†University of South Brittany, France
arnab-kumar.biswas@univ-ubs.fr

***Abstract*—Timing side-channel attacks pose a major threat to embedded systems due to their ease of accessibility. We propose CIDPro, a framework that relies on dynamic program diversification to mitigate timing side-channel leakage. The proposed framework integrates the widely used LLVM compiler infrastructure and the increasingly popular RISC-V FPGA soft-processor. The compiler automatically generates custom instructions in the security critical segments of the program, and the instructions execute on the RISC-V custom co-processor to produce diversified timing characteristics on each execution instance. CIDPro has been implemented on the Zynq7000 XC7Z020 FPGA device to study the performance overhead and security trade-offs. Experimental results show that our solution can achieve 80% and 86% timing side-channel capacity reduction for two benchmarks with an acceptable performance overhead compared to existing solutions. In addition, the proposed method incurs only a negligible hardware area overhead of 1% slices of the entire RISC-V system.**

I. Introduction

Embedded systems have become an integral part of our lives, and hence it is essential they are secure. Unfortunately, the cryptographic schemes in embedded systems are deployed in unforeseen adversarial settings where keys can be compromised through side-channel attacks. Such attacks have been reported on embedded devices [1] where the attacker extracts secret information from a victim program by observing a physical phenomenon, e.g. execution time and power consumption, during its execution [2]. In this paper, we consider the information leakage wherein the secret information of a victim program is unintentionally leaked to the attacker via the timing channel. For example, a modular exponentiation function (*modExp*) used in RSA decryption (as shown in Algorithm 1), induces information leakage in execution time due to the absence of an else-branch, while the condition of the if-branch depends on the secret key k. The attacker can observe the execution time of the victim program and establish a correlation between the observation and hypothesis of the secret information such as a cryptographic key. In this paper, we show that existing software countermeasures for mitigating timing leakage are ineffective especially when the cryptographic algorithms execute in bare metal mode or with an embedded OS (Operating System), which is common in embedded systems.

To overcome the limitations of the existing software countermeasures, we propose a framework called *CIDPro* that relies on dynamic hardware diversification to vary the timing characteristics of the program during execution. *CIDPro* is built upon the widely used LLVM compiler infrastructure [3] and targets the Rocket core [4] which is based on the RISC-V ISA. The framework is able to reduce timing side-channel leakage significantly by utilizing hardware diversification modules invoked by custom instructions that are inserted in the original source code.

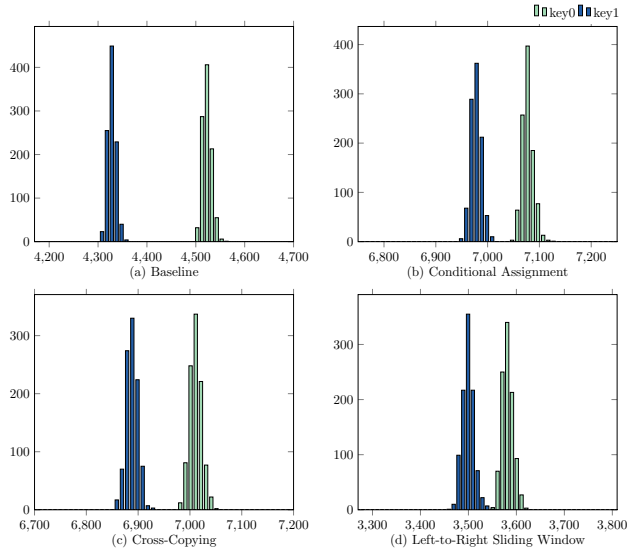


Fig. 1: Timing histogram of (a) a baseline program, (b) after transformation using cross-copying, (c) using conditional assignment and (d) using left-to-right sliding window. In each plot, the X-axis and Y-axis show the execution time in clock cycles and number of instances with that execution time, respectively.

The paper is organized as follows: Section II discusses existing countermeasures for timing leakage reduction, while *CIDPro* is presented in Section III. In Section IV, we provide a detailed discussion of the experimental results and Section V concludes the paper.

II. Related Work

A typical countermeasure against timing side-channel attacks is to apply program transformations such that the crit-

ical functions (e.g. *modExp*) complete in constant time. Two representative program transformation techniques are *cross-copying* [5] and *conditional assignment* [6]. In cross-copying, an else-branch is added with a dummy task that mimics the execution pattern in the if-branch to equalize the execution time of both branches. In conditional assignment, the condition is directly encoded into the branches using bit masks and bitwise logical operators.

The countermeasures, mentioned above, suffer from several limitations: First, the program transformations are typically performed on high-level programs. As such, compiler optimizations may inadvertently reduce the effectiveness of the countermeasures, causing timing side-channel vulnerabilities at the micro-architecture level. In addition, even though the conditional assignment removes critical conditions by flattening and converting them into primitive arithmetic and bitwise instructions, certain instructions (especially multiplications, divisions, etc.) may cause timing side-channel leakage due to variable clock cycle requirements based on the operand values. Second, both the existing techniques cannot minimize the hamming weight of different keys. Third, both the existing techniques result in performance degradation as additional instructions are introduced to equalize the execution times.

Algorithm 1 Modular Exponentiation

Input: data y, private key k of length d, integer N
Output: $y^k \mod N$

```
1: procedure MODEXP(y, k, N)
2:     r ← 1
3:     for i ← 1 to d do
4:         if (k mod 2 == 1) then        ▷ Critical Condition
5:             r ← (r × y) mod N
6:         y ← (y × y) mod N
7:         k ← k ≫ 1
8:     return r mod N
```

The sliding window technique [7] has been previously presented to reduce the effect of the hamming weight difference of keys on the execution time of the *modExp* function. This technique not only improves the performance but also reduces the timing-channel leakage. However, the sliding window technique is not able to eliminate the information leakage. In particular, the residual timing leakage is notable on embedded systems with a low noise runtime environment.

These limitations of the existing approaches are highlighted in Figures 1(b)-(d), which show the timing histograms of the RSA algorithm (that includes the *modExp* function) when it is executed with two different keys in the absence of an OS, i.e. in a bare metal environment that is common in embedded systems [8]. The timing characteristics corresponding to the two different keys are clearly distinguishable with the existing approaches and hence, they do not provide effective countermeasures against timing side-channel attacks. As such, executing functions in constant time cannot be easily achieved in software, particularly for embedded systems.

Timing side-channel attacks to extract not only the Hamming weight but also the value of fixed Diffie-Hellman exponents and RSA keys, have been reported in [9]. To mitigate the attacks, the authors presented blinding techniques to randomize the execution time. However, the blinding technique requires a significant amount of computations, which poses a heavy burden for lightweight processors in embedded systems.

Hardware solutions usually require substantial changes to the processor architecture that impacts all programs running on the processor and also incur performance overhead on non-security critical programs. A secure processor that can protect against side-channel attacks using masking and hiding techniques, is proposed in [10]. Besides an independent data path to implement the masking scheme, a pipeline randomizer adds non-deterministic dummy control and data signals to the processor data path. Another secure processor called Ascend, is presented in [11] that relies on an ORAM controller to obfuscate the address bus. However, information leakage cannot be prevented due to on-chip resource sharing. An instruction shuffler is proposed in [12] to shuffle independent instructions randomly for protection against side-channel attacks.

In this paper, we propose the *CIDPro* framework as a countermeasure against timing side-channel attacks especially in low noise embedded systems by utilizing both software and hardware methods with minimal resource overhead.

III. Proposed Solution

In this section, we describe the proposed *CIDPro* framework and its hardware implementation that utilizes custom instructions to effectively reduce timing side-channel leakage with low hardware cost and acceptable performance overhead.

A. CIDPro Framework

The premise of our framework is hardware diversification using custom instructions to execute diversified (time-varying) instructions (DIs) as shown in Figure 2.

The original source code of the program is given to the framework, which consists of two parts: 1) An LLVM pass transforming the source code into an assembly file, and 2) custom instruction generation of a set of m diversified instructions $DI = \{DI_1, DI_2, \ldots, DI_m\}$. Each $DI_i \in DI$ with $1 \leq i \leq m$ performs an arithmetic operation such as an ADD, MUL, etc, which is usually found in the source code of cryptographic primitives. Further, each $II_i^j \in DI_i$ represents a diversified version of the same operation with $1 \leq j \leq n$. Therefore every instance II_i^j provides the same functionality (i.e. $f(II_i^1) = f(II_i^2) = \cdots = f(II_i^n)$), but exhibits different execution time characteristics.

For each DI_i, a corresponding hardware module $CoPr(DI_i)$ is implemented in a hardware description language (HDL). These modules are integrated as custom instructions (CI) in the Secured Processor together with a Diversity Control Unit and a Pseudo Random Number Generator (PRNG). At runtime, the Diversity Control Unit selects the corresponding DI_i based on the CI, while the random number determines the II_i^j to be executed. Hence with every

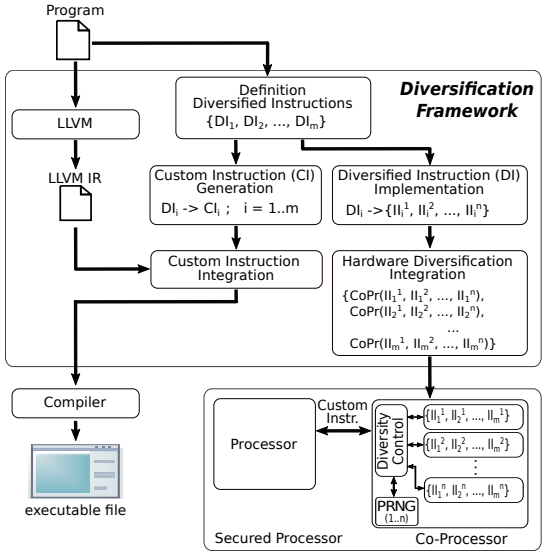


Fig. 2: Proposed *CIDPro* Framework

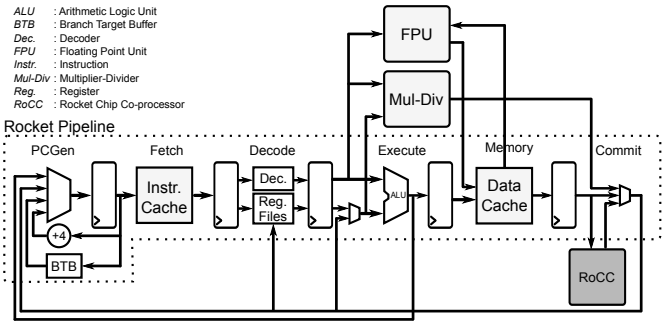


Fig. 3: Rocket Chip Micro-Architecture

invocation of a CI, a different II_i^j is selected, resulting in non-deterministic execution times of the program that reduces timing side-channel leakage.

The Custom Instruction Integration Unit facilitates the interfacing of LLVM and the set of DIs. An LLVM pass automatically identifies the arithmetic operations in the LLVM IR (Intermediate Representation) that exist in DI and that are supported by the Co-Processor. The corresponding CIs are then integrated into the original source code.

Unlike existing hardware countermeasures, the proposed *CIDPro* framework requires minimal changes to the processor architecture by utilizing a dedicated co-processor to execute CIs. This leads to low resource overhead and design efforts. In addition, the framework avoids negative side effects on non-security critical programs that run in the same environment as the security critical programs, since the former will be executed on the base processor unaffected. Furthermore, a developer does not need to write programs in a new language or with security in mind.

The CIs are kept as private information and are automatically inserted into the security critical programs to mitigate the information leakage via timing side-channels. We assume that our proposed framework is not available to the attacker

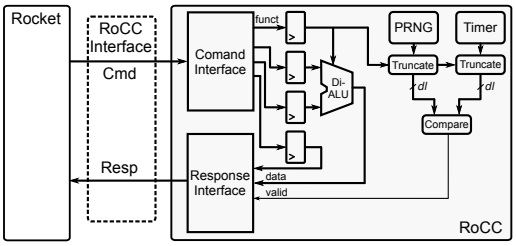


Fig. 4: Hardware Diversification on Co-processor

and we also do not consider hardware probing or reverse engineering attacks through which the attacker can obtain details of the diversified hardware implementation. Finally, it is worth mentioning that while we have only considered single operations as custom instructions in this paper, the framework can be easily extended to generate time-varying custom instructions that consist of multiple operations.

B. Hardware Implementation

The Secured Processor shown in Figure 2 is a RISC-V architecture that includes a Rocket core, a custom defined Rocket Chip Co-Processor (RoCC), L1 caches and a Floating Point Unit (FPU) [4]. The RoCC, which executes the CIs, is a tightly integrated extension to the processor pipeline as shown in Figure 3. This enables the RoCC to stall the entire pipeline until the CI has completed its execution. In our work, the Secured Processor is implemented on the Zynq7000 XC7Z020 FPGA device.

The Co-Processor utilizes the integrated DSPs on the FPGA fabric to support the operations required by the CIs. Instead of implementing multiple versions of the same instruction for each $II_i^j \in DI_i$ with varying execution times, only one hardware module is implemented for $f(II_i^j)$ for all elements in DI_i. To emulate the different execution times of II_i^j, the PRNG is connected to a comparator that triggers a *valid* signal when the random value matches the output of the timer. This indicates the completion of the operation on the Co-Processor and the valid result is passed to the Rocket core (refer to Figure 4).

The *funct* signal determines the operation to be executed on the Diversifying ALU (Di-ALU) and also allows the developer to limit the range of numbers generated by the PRNG. A higher range for the random value results in a larger reduction in the timing side-channel leakage. However, this also culminates in a longer overall execution time for the program, since the pipeline of the Rocket Chip stalls for long periods. Hence a trade-off between channel leakage and execution time should be considered by setting an appropriate diversification level dl. The level specifies the number $n = 2^{dl}$ of diversified instructions of which the execution times are varied from 1 to n clock cycles. $dl = 0$ means no diversification. This trade-off is discussed in Section IV-C.

IV. Results and Discussion

This section first describes the benchmark programs and the metrics used in our evaluations. This is followed by the

investigation of the trade-off between performance overhead and information leakage reduction of the *CIDPro* framework in a low-noise runtime environment (i.e. bare-metal mode) with respect to varying diversification levels. Finally, the results of channel capacity reduction are presented to show the effectiveness of *CIDPro* compared to existing solutions in both low-noise and OS environments (i.e. Linux mode).

A. Benchmark Programs

To compare the effectiveness of the proposed solution with existing methods, benchmark programs that can be implemented on embedded systems (like RISC-V based systems) that contain timing side-channel vulnerabilities are required. In our experiments, we use the RSA modular exponentiation (*modExp*) to encrypt or decrypt a message [13] from the benchmark suite introduced in [14] and modular multiplication (*mulMod16*) from the IDEA cipher [15]. Since these programs originate from practical cryptographic algorithms with different degrees of sophistication, they are meaningful candidates for the evaluations.

B. Experimental Setup

A timing side-channel is a communication channel created by unintentional information leakage by a victim program. Here the input is the set of values given to a victim program and the output is the timing observations of the program by an attacker. Like any other communication channel, timing side-channel can be measured by Shannon's channel capacity [16] which represents the tight upper bound on the information transmission rate using that channel. We use a command-line tool called LeakiEst [17] to estimate the side-channel capacity from observations of program execution times. Benchmark programs are executed on the Rocket chip system, and the execution time is extracted through the performance counters integrated in the Rocket chip. We collect 1000 samples for each of the two secret inputs for each benchmark program to obtain the leakage channel capacity using LeakiEst, which uses the iterative Blahut-Arimoto algorithm [18], [19] to estimate the channel capacity.

We perform a distinguishing experiment by running each program with two distinct secret input values. Here the security concern is that the secret keys can be inferred from the execution time based on the bit patterns of the keys. We compare our proposed solution with two existing and widely used solutions, i.e. cross-copying [6] and conditional assignment [5]. In addition, we evaluated the sliding window technique for the modular exponentiation benchmark [7], which is widely used in cryptographic libraries [20] such as *Libgcrypt*. Left-to-right (LR) windowed form with a window size of 3 is considered in our experiments.

C. Performance overhead and security trade-offs

As mentioned in Section III-B, increasing the diversification level in *CIDPro* increases randomness in the execution times of programs leading to a reduction in information leakage via timing side-channels. Figure 5 shows the timing histogram of

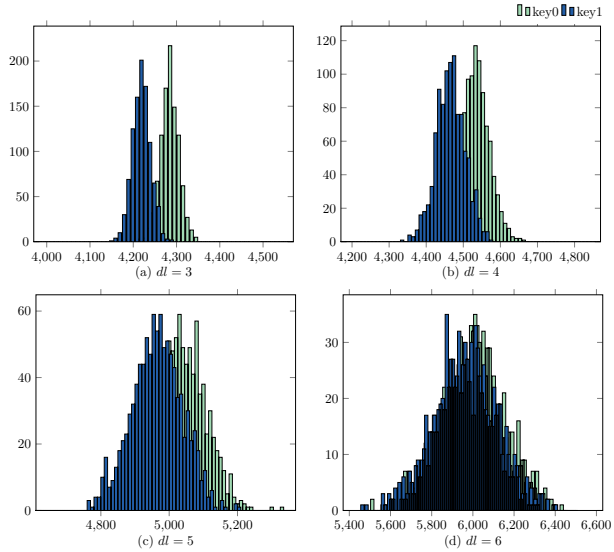
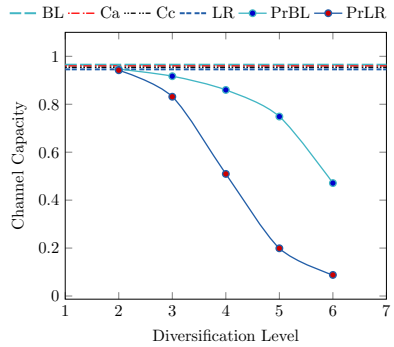


Fig. 5: Timing histogram of *CIDPro* for the *modExp* benchmark with different diversification levels. In each plot, the X-axis and Y-axis show the execution time in clock cycles and number of instances with that execution time, respectively.

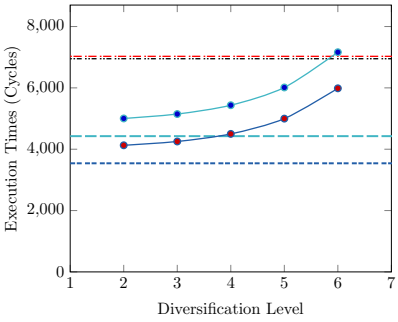
the LR version of *modExp* that utilizes the *CIDPro* framework with different diversification levels. As can be observed, if dl increases, the distributions for two different keys (i.e. key0, key1) become indistinguishable which implies effective mitigation of information leakage via timing side-channels.

Figure 6 shows the channel capacities and the average execution times of the *modExp* benchmark using the *CIDPro* framework with varying diversification levels compared to existing solutions. *BL*, *Ca*, *Cc* and *LR* denote the baseline program without countermeasures, and the program with the existing countermeasures i.e. conditional assignment, cross-copying, and left-to-right sliding window, respectively. *PrBL* and *PrLR* represent the *BL* and *LR* programs that are modified by *CIDPro*. As can be observed in Figure 6(a), the channel capacities of *BL*, *Ca*, *Cc* and *LR* are almost equal to one, which indicates a high timing information leakage. It is evident that the timing leakage reduces significantly for *PrBL* and *PrLR* with increasing diversification levels. In particular, *PrLR* exhibits a higher reduction in channel capacity compared to *PrBL* similar to the capacity comparison between *LR* and *BL* (refer to Figure 1). Figure 6(b) shows that the execution times of *Ca* and *Cc* are notably higher compared to the baseline program, while *LR* has the lowest execution time among all the techniques. Although *PrBL* and *PrLR* result in an increased execution time compared to *BL* and *LR*, they are still faster than *Ca* and *Cc* for almost all the diversification levels considered. The increment in execution times of *PrBL* and *PrLR* in the range $2 \leq dl \leq 5$ is relatively small compared to those for $dl > 5$.

Figure 7 displays the result of the same experiment with the *mulMod16* benchmark. While *Ca* results in a reduction in channel capacity, it has a significant increase in execution

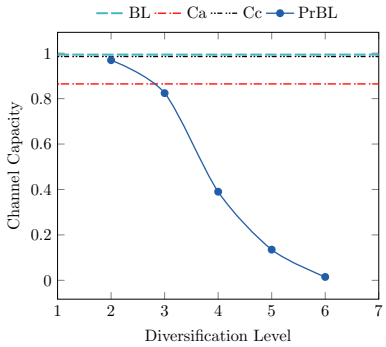


(a) Channel capacity

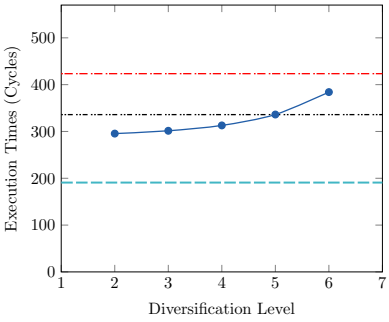


(b) Average execution time

Fig. 6: Impact of various diversification levels on the *modExp* benchmark in comparison to existing solutions



(a) Channel capacity



(b) Average execution time

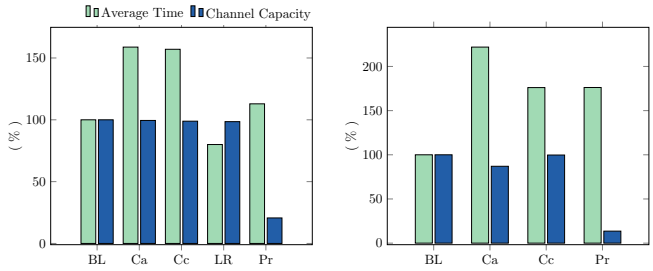
Fig. 7: Impact of various diversification levels on the *mulMod16* benchmark in comparison to existing solutions

time compared to the baseline program. *Cc* exhibits a better performance than *Ca*, but it is not able to reduce channel capacity in low-noise environments. *PrBL* has a significant reduction in channel capacity. While the channel capacity reduces considerably with an increase in the diversification level, this adversely influences the execution time as expected. For $dl \leq 5$, the overall execution time for *PrBL* remains lower than *Cc* and *Ca*.

D. Evaluation Results

Figure 8 compares the increased performance cost and the leakage reduction of the various countermeasures with respect to the baseline for the *modExp* and *mulMod16* benchmarks in bare-metal mode. The performance cost and the leakage reduction are evaluated in terms of the average execution time and the channel capacity, respectively. *Pr* denotes the programs that utilize the proposed *CIDPro* with $dl = 5$ on *LR* for the *modExp* and on *BL* for the *mulMod16* benchmark.

As can be observed in the figure, the existing solutions have a negligible reduction in channel capacity, while *Pr* achieves a significant reduction. The channel capacity of *Pr* is reduced to 20% and 14% for the *modExp* and *mulMod16* benchmarks, respectively. Existing solutions and *Pr* result in additional average execution time with the exception of *LR* in the *modExp* benchmark. The execution time increases due to the insertion of dummy instructions to balance branches in security critical conditions for existing methods and the long variable execution times of CIs. For the *modExp* benchmark,



(a) *modExp* (b) *mulMod16*

Fig. 8: Impact on the average execution time and channel capacity of *CIDPro* in comparison to existing solutions and the baseline.

Ca and *Cc* increase the execution time by over 50% compared to *BL*, while the proposed solution requires only 13% additional execution time.

Ca and *Cc* in the *mulMod16* benchmark result in a significant increase in execution time of 120% and 75% respectively compared to *BL*. Even though the proposed *CIDPro* results in a similar execution time for *mulMod16* as *Cc*, it achieves a significant reduction in channel capacity. It is worth mentioning that the timing cost in our proposed solution just applies to the critical functions that use the CIs to mitigate timing side-channel leakage. The execution time of the normal functions remains unaffected.

Figure 9 reports the results of the binaries executed in a

TABLE I: Hardware Resource Utilization

	RocketTile	CPU	FPU	RoCC
Slices	6612	1331	3458	84
DSPs	34	4	20	10

Linux OS environment on the Rocket chip. The Linux OS is very constrained with minimum effects on the programs and all executable files are compiled and linked statically. Therefore the timing noise level due to different auxiliary processes is expected to be lower in contrast to a fully functional OS. This means that the noise incurred by the Linux OS affects the information leakage only marginally. As it can be observed in Figure 9, the existing solutions (*Ca* and *Cc*) are not able to achieve notable reductions in channel capacity. The results show that *Pr* effectively mitigates the timing side-channel with channel capacity below 0.2 and 0.12 for the *modExp* and *mulMod16* benchmarks, respectively.

E. Hardware Resource Utilization

Table I reports the hardware resource utilization of RoCC that implements the hardware diversification as part of *CIDPro* compared to the typical Rocket chip system (*RocketTile*) that includes a processor core (CPU) and a floating point unit (FPU). The hardware resource utilization is reported in terms of the number of slices and DSP blocks for FPGA implementation on the Zynq7000 FPGA device. The area overhead of RoCC is negligible (i.e. 1% of slices) compared to the entire Rocket chip system.

V. Conclusion

We have proposed *CIDPro*, a framework consisting of an LLVM compiler pass and hardware diversification to minimize timing side-channel leakage of cryptographic programs that run on soft-processors in embedded systems. The proposed framework utilizes custom instructions to realize the hardware diversification without changing the base processor architecture and incurring only 1% area overhead for the co-processor. Experimental results targeting low noise runtime environments on RISC-V based system demonstrated that our framework achieves 80% and 86% timing side-channel capacity reduction compared to existing solutions. In addition, we show that the proposed solution leads to the best trade-off in timing leakage reduction and performance overhead compared to the existing countermeasures.

Acknowledgment

The research described in this paper has been supported by the National Research Foundation, Singapore under grant number NRF2016NCR-NCR001-006.

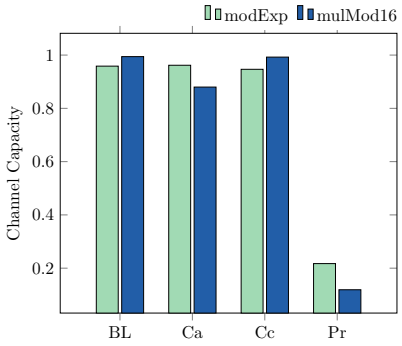


Fig. 9: Comparing channel capacity of the proposed solution and the existing solutions running on Linux OS environment.

References

[1] J. A. Ambrose, R. G. Ragel, D. Jayasinghe, T. Li, and S. Parameswaran, "Side channel attacks in embedded systems: A tale of hostilities and deterrence," in *16th International Symposium on Quality Electronic Design*, Mar. 2015, pp. 452–459.

[2] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, pp. 1–27, 2016.

[3] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection*, 2015, pp. 3–9.

[4] K. A. et al., "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[5] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks," in *Information Security and Cryptology*. Springer Berlin Heidelberg, 2006, pp. 156–168.

[6] J. Agat, "Transforming out Timing Leaks," in *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000, pp. 40–53.

[7] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[8] M. Melkonian, "Get by Without an RTOS," *Embedded Systems Programming*, pp. 146–164, Sep. 2000.

[9] P. C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. CRYPTO '96: Springer, 1996, pp. 104–113.

[10] F. Bruguier, P. Benoit, L. Torres, L. Barthe, M. Bourree, and V. Lomne, "Cost-effective design strategies for securing embedded processors," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 1, pp. 60–72, Jan 2016.

[11] L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "Design and implementation of the ascend secure processor," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[12] A. G. Bayrak, N. Velickovic, P. Ienne, and W. Burleson, "An architecture-independent instruction shuffler to protect against side-channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 20:1–20:19, Jan. 2012.

[13] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[14] H. Mantel and A. Starostin, "Transforming Out Timing Leaks, More or Less," in *European Symposium on Research in Computer Security*. Springer, Cham, 2015, pp. 447–467.

[15] X. Lai, "On the design and security of block ciphers," ETH Zurich, PhD Thesis, 1992.

[16] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, July 1948.

[17] T. Chothia, Y. Kawamoto, and C. Novakovic, *A Tool for Estimating Information Leakage*. CAV 2013: Springer, 2013, pp. 690–695.

[18] R. Blahut, "Computation of channel capacity and rate-distortion functions," *IEEE Transactions on Information Theory*, vol. 18, no. 4, pp. 460–473, Jul 1972.

[19] S. Arimoto, "An algorithm for computing the capacity of arbitrary discrete memoryless channels," *IEEE Transactions on Information Theory*, vol. 18, no. 1, pp. 14–20, Jan 1972.

[20] D. J. Bernstein, J. Breitner, D. Genkin, L. G. Bruinderink, N. Heninger, T. Lange *et al.*, "Sliding right into disaster: Left-to-right sliding windows leak," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 555–576.