HYDRA: a multi-core RISC-V with cryptographically useful modes of operation

BEN MARSHALL, DAN PAGE, THINH PHAM, MAX WHALE, Department of Computer Science, University of Bristol, UK

Relative to other architectural options, multi-core processor designs often represent an efficient, flexible solution for general-purpose workloads. Based on the premise that multi-core processors are generally viable as an implementation platform for cryptography, this paper investigates specific instances which harness the concept of Composable Lightweight Processors (CLPs). We first identify several cryptographically useful composition modes, then implement and evaluate support for them in a RISC-V based proof-of-concept dubbed HYDRA: while retaining characteristics which stem from a generic multi-core design, HYDRA can also operate in modes specifically designed to address domain-specific challenges relating to efficiency and security.

CCS Concepts: • **Computer systems organization → Multicore architectures**; • **Security and privacy → Cryptography**; **Hardware attacks and countermeasures**;

ACM Reference Format:
Ben Marshall, Dan Page, Thinh Pham, Max Whale. 2022. HYDRA: a multi-core RISC-V with cryptographically useful modes of operation. 1, 1 (May 2022), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

From single- toward multi-core processor designs. The term multi-core[1] can be applied to concrete processor designs which vary, for example, in terms of 1) the number of cores, 2) the type of cores, i.e., whether they exhibit general- or special-purpose capabilities, and/or form a homogeneous (or symmetric) or heterogeneous (or asymmetric) whole, and 3) how cores are interconnected, either via dedicated communication fabric and/or shared resources such as caches. Set within a broad, diverse design space for parallel computing technologies more generally, the trend from single- toward multi-core [BDM09] processors can be rationalised by two points; both relate to the challenge of scaling instruction throughput in the face of Moore's Law. First, single-core designs are typically limited by Instruction Level Parallelism (ILP): the effectiveness of a super-scalar pipeline, for example, is limited by ILP, irrespective of any increased transistor budget stemming from a decrease in feature size. In contrast, multi-core designs "spend" that budget on support for Thread Level Parallelism (TLP) which often matches demand so can be effectively harnessed at an application level. Second, and in part as a result of their reliance of ILP, single-core designs often rely on incremental improvements, e.g., to clock frequency, which are not sustainable from a physical perspective. In contrast, multi-core designs adopt a more radical architectural reorganisation which is, therefore, less reliant on such improvements.

Implementation options for cryptographic workloads. Implementations of cryptography often have a central role to play within any use-case deemed security-critical, e.g., those which involve the computation, storage, or communication of identity-, location-, or finance-related data. The task of producing such an implementation can be challenging, however, in the sense it will often 1) involve computationally intensive, somewhat niche functionality, 2) need to satisfy a range of efficiency-related quality metrics such as high-throughput, low-latency, low-footprint, power-efficiency, and high-assurance, and, at the same time, 3) form a central target in what is a complex, evolving attack surface. A rich body of literature, capturing the field of cryptographic engineering, has explored techniques which attempt to address such challenges; as one may expect, selection from the large design space of options depends on making appropriate trade-offs based on the use-case (versus there being a single "best" option for all use-cases).

Particularly within embedded use-cases, some form of System on Chip (SoC) is common. The idea is to combine a single general-purpose processor core with multiple special-purpose (cryptographic) IP cores: doing so represents a trade-off which maximises efficiency and security in the IP cores, as a result of their domain specificity. A multi-core processor could be seen as representing the opposite trade-off, namely one which maximises flexibility (and thus generality), so represents a viable alternative to SoCs for cryptographic workloads. At least two forms of argumentation[2] support this claim. First, flexibility can help mitigate challenges related to utilisation. Whenever cryptographic functionality is invoked, pertinent efficiency- and security-related quality metrics must be satisfied, but, depending on the use-case, the frequency of invocation (resp. the period of execution) may be limited. For example, networked communication using TLS [Res18] means invocation of AES [fip01] or RSA [RSA78] at all will depend on use of an associated cipher suite, but, even then, could on a per-packet (for AES) or even per-session (for RSA) basis. An IP core for AES or RSA is likely to be far from fully utilised; a multi-core processor can support other workloads when unused for AES or RSA. Second, flexibility

[1] We immediately face, and largely ignore, an issue around this terminology: an n-core processor might be termed single-core if $n = 1$, multi-core if $2 \leq n \leq 8$, or many-core if $n \geq 10$, noting the thresholds involved are somewhat arbitrary.

[2] We note that similar points support the use of multi-core processors within Digital Signal Processing (DSP), where, e.g., utilisation of and agility with respect to media codecs is often important.

Author's address: Ben Marshall, Dan Page, Thinh Pham, Max Whale, {ben.marshall, daniel.page,th.pham,mw17440}@bristol.ac.uk, Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol, UK, BS8 1UB.

can help mitigate challenges related to agility. Specifically, any design, standardisation, and implementation decision may have a long life-span, implying a need to cater for future (and so unknown) capabilities with respect to the attack landscape. An IP core for AES or RSA is likely to be fixed-function; a multi-core processor permits agility, since an implementation can more easily refined (e.g., with a suitable countermeasure) or replaced (e.g., with an alternative underlying primitive).

Remit and organisation. Based on the premise that multi-core processors are generally viable as an implementation platform for cryptography, this paper investigates specific instances which harness the concept of Composable Lightweight Processors (CLPs) due to Kim et al. [KSG+07]. At a high level, CLPs offer architectural flexibility in the sense that the resources associated with p processor cores can be operate in either a conventional (i.e., segregated) or composed (i.e., aggregated) manner. Crucially, this configuration can be selected dynamically, e.g., to support the workload at hand. We use the term composition mode to describe the high level functionality offered by the composed cores; support for a given composition mode requires a lower level design that specifies, e.g., how micro-architectural resources are structured and managed. In [KSG+07], and also in related work prior to it, the majority of composition modes considered are motivated by parallel computation. We aim to extend this remit by identifying several cryptographically useful composition modes, then implementing and evaluating support for them in a RISC-V based proof-of-concept dubbed HYDRA[3].

2 DESIGN

2.1 Concept

Consider a set $P = \{P_0, P_1, \ldots, P_{p-1}\}$ of p atomic processor cores. In a standard multi-core processor, each $P_j \in P$ operates in a conventional manner; this implies $|P|$ system cores are available.

However, a composition configuration C for that multi-core processor will partition P. If $|C(P)_i| = 1$, then the atomic core $P_j \in C(P)_i$ operates in a conventional manner. If $|C(P)_i| > 1$, however, then each atomic core $P_j \in C(P)_i$ operates in a composed manner, i.e., as a single composed system core. Exactly one atomic core, denoted $\overline{P_j}$, has a distinguished role as the "controller" or primary (versus secondary) core within said composed core. Put together, use of C will imply $|C(P)|$ system cores are available: the number of system cores is now equal to the number of conventional cores plus the number of composed cores. For example, and without loss of generality, consider a set

$$P = \{P_0, P_1, P_2, P_3\}$$

and a composition configuration

$$C(P) = \{\{\overline{P_0}, P_1\}, \{P_2\}, \{P_3\}\}.$$

In this case, $|C(P)| = 3 < 4 = |P|$ system cores are available: $C(P)_0 = \{\overline{P_0}, P_1\}$ operates in a composed manner, i.e., as a single composed system core formed from P_0 and P_1 (noting P_0 is the primary core in this case), whereas $C(P)_1 = \{P_2\}$ and $C(P)_2 = \{P_3\}$ each operate in a conventional manner.

[3] The implementation is available, under an open-source license, via https://github.com/scarv/hydra.

Each composed system core (independently) operates in a cryptographically useful composition mode, which is also selected using C: we outline the rationale for and high-level semantics of each such mode in the following sub-sections.

2.1.1 Mode 1: wide data-path mode. Consider an ISA with a b-bit word size, meaning, e.g., it includes 1) a b-bit general-purpose register file, and 2) instructions which naively support operations using b-bit operands. Lee et al. [LYS04] observe that cryptographic workloads often involve operations which can be described as having Multi-word Operands, Multi-word Results (MOMR): this means the operands of a given operation are some $n > b$ bits, and are therefore represented using $w = \lceil n/b \rceil$ words. Such operations are not supported naively in the ISA, meaning they must be implemented using a higher-level algorithm; the (asymptotic) efficiency of such algorithms often depends strongly on w. For example, typical algorithms for multiple-precision addition [MvOV96, Algorithm 14.7] have an $O(w)$ execution time, for example, whereas typical algorithms for multiple-precision multiplication [MvOV96, Algorithm 14.12] or modular multiplication [MvOV96, Algorithm 14.36] have an $O(w^2)$ execution time. Put simply then, support for a larger b and thus smaller w will render such operations more efficient.

RISC-V can be viewed as supporting a *statically* scalable word size, in the sense that 32-bit (RV32), 64-bit (RV64), and 128-bit (RV128) variants of the ISA exist. In this composition mode, HYDRA supports a *dynamically* scalable word size: in essence, the b-bit data-paths within c atomic cores used to form a composed core are combined to form a single $(c \cdot b)$-bit data-path. Consider, for example, a composed core formed from $c = 2$ atomic cores, each of which supports RV32 meaning $b = 32$. RV32 instructions executed by this composed core are overloaded such that they support $c \cdot b = 2 \cdot 32 = 64$ bit operands, via the resulting wider data-path. It is vital to note that the composed core does *not* support RV64, however; the overloading is not suggestive of multi-ISA [CHC+20] functionality, for example.

2.1.2 Mode 2: SIMD compute mode. In this composition mode, HYDRA supports a Single Instruction, Multiple Data (SIMD) execution model. That is, in a given execution cycle, each of the c atomic cores used to form a composed core will execute the *same* instruction; they do so using data read from and written to *different* (i.e., their own, local) register files. If each atomic core has a b-bit data-path, then the composed core can be viewed as processing c-element, or $(c \cdot b)$-bit vectors comprised of b-bit sub-words. On one hand, this suggests the composed core does not naively support sub-words of less than b bits: if $b = 32$, for example, 8- or 16-bit sub-words often *are* support by more traditional, dedicated SIMD extensions. On the other hand, however, it has a *somewhat* scalable number of sub-words depending on c (the value of which is upper-bounded by the total number of atomic cores); although clearly not analogous to the standard RISC-V vector extension, it *could* be viewed as a light-weight alternative for (very) specific use-cases.

Within the context of cryptography, the composition mode is intended to support (at least) two use-cases. First, it can represent a performance-enhancing mechanism. For example, it supports a cryptographic implementation technique termed word-slicing (see, e.g., [BKP21, Section 2.2.2]). Second, it can represent a security-enhancing mechanism. For example, it can be harnessed to realise

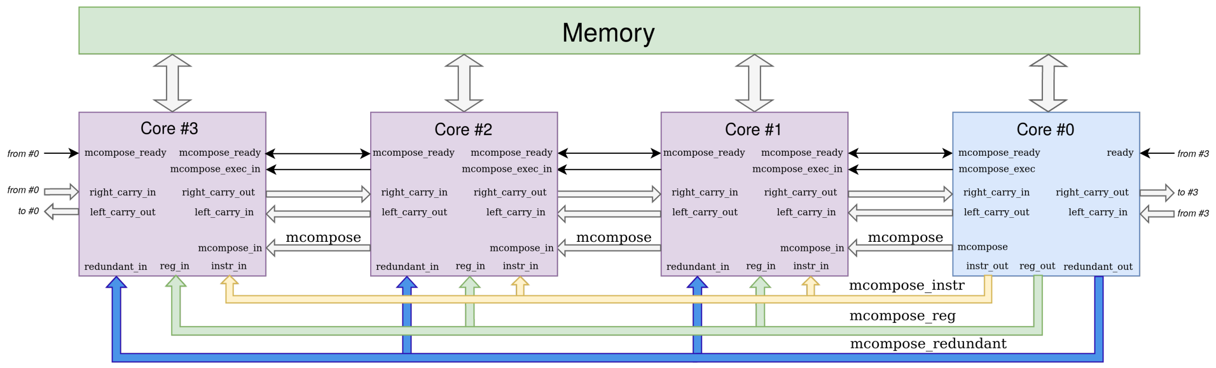

Fig. 1. The architecture diagram of the composed mode, showing an example 4-core system.

"balanced" forms of execution which act to mitigate certain forms of side-channel attack (see, e.g., [API08]).

2.1.3 Mode 3: SIMD redundant mode. In this composition mode, HYDRA supports a redundant or "lock step" execution model. That is, in a given execution cycle, each of the *c* atomic cores used to form a composed core will execute the *same* instruction. Although they do so using data read from and written to *different* (i.e., their own, local) register files, the data and therefore computational behaviour across all atomic cores is expected to be the *same*; a per-cycle test of that fact is performed, and, if it fails an exception is raised. This is a well established approach within the context of both safety and security, as a means of detecting transient faults.

Within the context of cryptography, the composition mode is intended as a countermeasure against intentional, i.e., are adversarially injected, faults and therefore fault attacks (see, e.g., [JT12, BBKN12, KSV13]); once a fault is detected, we expect the resulting exception to be handled via an suitable, security-conscious response mechanism (see, e.g., [YGD+16, YDG+19]). Intuitively, the goal is to raise the difficult of mounting a successful attack (simultaneously injecting faults with the same effect in multiple cores, i.e., a faults which are undetected by the test, is significantly more difficult for the attacker) using a mechanism which composes with other, e.g., algorithmic countermeasures.

2.2 Realisation

Following the design concept above, a realisation approach for a CLPs system is presented in this subsection. Figure 1 depicts the architecture diagram of a CLPs system. Generally, the CLPs system can be built based on a conventional multi-core system, in which each core having its ID (i.e., *mhartid*) can possibly run independently with its own or sharing data to exploit the advantages of thread-level parallelism. To provide the enhanced features of the HYDRA composed modes, the CLPs system adds a required interface between cores, and corresponding additional logic and states, e.g., Control Status Registers (CSR), in each core. Basically, the additional parts form two main supporting mechanisms: composable configuration and composed computations.

The composable configuration is to configure, enter and exist from a specific composed mode. It adds two custom CSRs specifying the composed mode and the number of composed cores. These CSR are configured by the primary core, *Core*#0 (see Figure 1), and the configured information is passed to the secondary cores through the *mcompose* bus. if the core number is set greater than one, the primary core enters to a composed mode and the secondary cores, of which ID is less than the core number are asked to join the composed mode, otherwise, the cores exist from a composed mode.

The composed computations basically require the composed cores executing the same instruction. To do so in a composed mode, the primary core fetches an instruction in memory and broadcasts it on the *mcompose_instr* bus, allowing the secondary cores to decode this instruction instead of fetching their own from memory. In addition, the primary core will orchestrate the composed execution using *mcompose_exec* and *ready* signals. *mcompose_exec* synchronously triggers the secondary cores to begin decoding the instruction, and *mcompose_ready* ensures that all cores are ready before an instruction begins execution. The instruction executes the computations on composed words possibly stored in each core's architectural states (i.e., general-purpose registers). Depending on a specific composed mode, the composed computation is handled accordingly;

In SIMD compute mode, the composed words are viewed as vectors of independent elements, each of which is handled by a composed core. To load and store a vector in memory, the primary core needs to broadcast the address storing the vector to the secondary cores via the *mcompose_reg* bus.

In wide data-path mode, the composed words are viewed as vectors of dependent elements. The dependency presents the order of element bits in a n-word-wide value. The composed computation requires passing the carry flags of each partial computation across the composed cores. That can be done by using the *left_carry* and the *right_carry* signals.

In SIMD redundant mode, the composed words are viewed as vectors of the same value elements. That allows the same operation independently to compute the same values on each composed core. So computational behaviour across the cores is assumed to be the same. To detect a fault occurred, a per-cycle test of that fact is performed, and, if it fails an exception is raised. An exhausted test on every state in the cores would ideally detect any fault that happened. But that results in a highly increased area overhead due to a large number of comparisons and connections across the cores. For a lightweight solution, we adopt an efficient fault test. We consider a fault can be a control-flow fault or a data fault. For the former, we check the program counter value of the cores, and for the latter, we check the write-back value at each core. These values are broadcasted on the *mcompose_redundant* bus for the fault detection.

3 IMPLEMENTATION

3.1 Hardware

In this section, a composable 4-core system implemented on a FPGA platform is presented as the proof of concept. We choose the PicoRV32[4] core as the target core for the composable system. The PicoRV32 is a lightweight open-source RISC-V core and also highly configurable, optionally supporting multiplication (*M*) instruction extension. The original PicoRV32 is modified to support running in a multi-core system with composition ability. Particularly, we add 1) the *mhartid* CSR to provide an ID number to identify a core in the multi-core system, 2) necessary interface signals and corresponding functional logics to support composed executions, mentioned in the above Section. We also implement an interconnection and a simple bus arbitration, based on a round-robin fashion, which allows the cores to possibly access the same memory. Sharing memory between cores is necessary to perform composed operations.

To produce an experimental platform which permits evaluation of, e.g., hardware overhead and cycle-accurate execution latency, we make use of the SASEBO-GIII [HKSS12]: this includes two FPGAs, namely a Xilinx Kintex-7 (model `xc7k160tfbg676`) target FPGA, and a Xilinx Spartan-6 (model `xc6slx45`) support FPGA. We use the former exclusively, synthesising the composable system for it using Xilinx Vivado 2018.2; default synthesis settings are used, with no effort adopted in synthesis or post-implementation optimisation. The FPGA uses a 200 MHz external clock input, which is adjusted into a 50 MHz internal clock signal for use by the host core itself.

We also implement a baseline (conventional) 4-core system employing the original PicoRV32 for the hardware overhead comparison. Table 1 reports the number of Lookup-Tables (LUTs) and FlipFlops (FFs) of the two multi-core systems and theirs based cores. In addition, their longest delay path (in ns) is also reported. To make the PicoRV32 core composable, an raised overhead of 32% and 15% (resp. 32% and 15%) in LUTs and FFS, respectively, in a primary core (resp. secondary cores) versus the orignial core is required. That leads to the total overhead of the composable system increases by 14% and 8% of the LUTs and FFs, respectively, compared to the conventional system. Additionally, we notice that the longest delay path of the composable system is longer by 32% compared to the conventional system. The delay is caused by passing the *carry* bit between the cores. Considering the PicoRV32 is a size-optimised core, the increased overheads seem reasonable, and suggests composed functionality could be considered as a lightweight solution for embedded systems.

3.2 Adapting software

One of the main benefits of composed modes is that it is generally simple to adapt software to make use of the composed modes in an ad-hoc manner. The model used for composed functions assumes that the primary core was not composed before entering the function and ensured it was not composed upon exit, and the secondary cores are ready for composition. The required adaptation can basically be viewed as three main changes in the code: a) using *csrw* instructions to enter and to exit from a composed mode with a specific required number of cores, b) changes relating to the fact that the effective word size is varied according to number of composed cores rather than fixed to 4-byte words of a single core, and c) changes relating to memory access that each memory access instruction will invoke an memory operation for each composed core with increased memory address. For example, Figure 2 shows the composed versions of a multi-precision addition function. The fist two instructions configure for the wide data-path mode and number of cores required for the composed mode. The third instruction derives the effective word size for the composed mode. The main body and loop code of the function is mostly the same code as the function for single core, except for the increment of pointers (i.e., *a_addr*) accordingly to the effective computed word size. That reduces the required number of iterations in the loop, hence increases the speed of the computation. Finally, before returning from the function, an *csrw* instruction is used to exit from the composed mode.

[4]https://github.com/cliffordwolf/picorv32

Table 1. Comparision of hardware overheads of conventional 4-core system vs composable alternative.

	LUTs	FFs	Longest delay	
Original core	1425 (1.00×)	968 (1.00×)	–	–
Primary core	1881 (1.32×)	1109 (1.15×)	–	–
Secondary core	1897 (1.33×)	1059 (1.09×)	–	–
Conventional system	10964 (1.00×)	4181 (1.00×)	5.324	(1.00×)
Composable system	12522 (1.14×)	4517 (1.08×)	7.096	(1.33×)

```
//input:  int *a_addr, int *b_addr,
//        int n_bytes, int n_cores
//output: int *r_addr
mp_add_com:
  csrwi mcompose_mode, mcompose_wide
  csrw  mcompose_reg,  n_cores
  slli bytes_per_word, n_cores, 2

  li   carry, 0
  add  addr_end, n_bytes, a_addr
mp_add_comp_loop:
  beq  a_addr,  addr_end, mp_add_end
  lw   a_value, 0(a_addr)
  lw   b_value, 0(b_addr)
  add  r_value, a_value,  carry
  sltu carry,   r_value,  a_value
  add  r_value, r_value,  b_value
  sltu t_carry, r_value,  b_value
  or   carry,   carry,    t_carry
  sw   r_value, 0(r_addr)
  add  a_addr,  a_addr,   bytes_per_word
  add  b_addr,  b_addr,   bytes_per_word
  add  r_addr,  r_addr,   bytes_per_word
  j    mp_add_comp_loop
mp_add_end:

  csrw mcompose_reg, zero
ret
```

Fig. 2. The composed versions of multi-precision addition implementation.

Table 2. Cycle and instruction counts compared across the single core, 2-core and 4-core systems (plus overhead versus baseline in parentheses).

1024-bit Operations	Metric	Single Core	Composed Systems 2 Cores	Composed Systems 4 Cores
Addition	Instructions	431 (1.00×)	228 (1.89×)	124 (3.48×)
	Cycles	2023 (1.00×)	1225 (1.65×)	843 (2.40×)
Multiplication	Instructions	15822 (1.00×)	4083 (3.88×)	1091 (14.50×)
	Cycles	179395 (1.00×)	72490 (2.47×)	32086 (5.59×)
ModExp	Instructions	57395054 (1.00×)	15019653 (3.82×)	4144180 (13.85×)
	Cycles	594838395 (1.00×)	238293765 (2.50×)	106391562 (5.59×)

Table 3. Comparison of ChaCha20 encryption performance for different message sizes (plus overhead versus baseline in parentheses).

Message size	Metric	OpenSSL	Single Core	Composed Systems 2 Cores	Composed Systems 4 Cores	128 bit Vector
64 bytes	Instructions	2825 (1.00×)	1765 (1.60×)	1199 (2.36×)	659 (4.29×)	607 (4.65×)
	Cycles	25073 (1.00×)	17603 (1.42×)	11905 (2.11×)	7609 (3.30×)	
128 bytes	Instructions	5555 (1.00×)	3483 (1.59×)	2345 (2.37×)	1285 (4.32×)	1182 (4.67×)
	Cycles	49705 (1.00×)	34910 (1.42×)	23415 (2.12×)	14959 (3.32×)	
256 bytes	Instructions	11015 (1.00×)	6919 (1.59×)	4637 (2.38×)	2537 (4.34×)	2332 (4.72×)
	Cycles	98969 (1.00×)	69524 (1.42×)	46435 (2.13×)	29659 (3.34×)	
512 bytes	Instructions	21935 (1.00×)	13791 (1.59×)	9221 (2.38×)	5041 (4.35×)	4632 (4.74×)
	Cycles	197497 (1.00×)	138752 (1.42×)	92475 (2.14×)	59059 (3.34×)	
1024 bytes	Instructions	43775 (1.00×)	27535 (1.59×)	18389 (2.38×)	10049 (4.36×)	9232 (4.74×)
	Cycles	394553 (1.00×)	277208 (1.42×)	184555 (2.14×)	117859 (3.35×)	

Table 4. Comparison of results of unprotected and protected AES encryption against control flow and data fault injections.

	Control flow fault case				Data fault case			
Implementations	Passed	Failed	Broken	Detected	Passed	Failed	Broken	Detected
Unprotected AES	30	62	8	–	42	29	29	–
Protected AES	12	0	2	87	46	0	0	54

4 EVALUATION

4.1 Benchmarks

To evaluate the effectiveness of the HYDRA composed modes for cryptographic workloads, our benchmarks include primitive functions of widely used cryptographic algorithms:

(1) A set of three multi-precision algorithms, namely, addition, multiplication, and modular exponentiation (*modExp*), which are widely used in asymmetric cryptosystems (like RSA), were considered. A number size of 1024-bits was used for these operations throughout the experiments. We apply the Coarsely Integrated Operand Scanning (CIOS) Montgomery algorithm [KAK96] (i.e. Montgomery reduction and multiplication) for the *modExp* function.

(2) ChaCha20, an ARX based stream cipher, is deployed in many application domains [Ber08]. The performance of Chacha20 block function can be typically accelerated by using vector instructions [MPP21]. We use this function to evaluate the SIMD compute mode of HYDRA.

(3) AES-128, referring to a 128-bit key variant, is known vulnerable to fault-injection attacks (see, e.g., [TMA11]). The AES encryption function is implemented and executed in the redundant mode to evaluate the mitigation against fault attacks.

4.2 Evalation results

Table 2 and Table 3 present the performance evaluation of the muti-precision algorithm set running in the wide data-path mode and the ChaCha20 encryption running in the SIMD compute mode, respectively, in terms of retired instruction and clock cycle counts. The system is flexibly configured to compose two and four cores in the composed modes for the performance comparison to the single core execution. One could note that the cycle counts are considerably larger than the instruction counts. That is mainly because PicoRV32 has three non-pipelined stages of execution, of which stage requiring at least one clock cycle, which results in the minimum cycles per instruction of 3 cycles. Memory operations and taken branches will take longer cycle counts depending on memory access latency.

As can seen in Table 2, the instruction reduction ratios gained by using the wide data-path mode in the modExp and the multiplication are significantly better than that in the addition. This is because the addition performs one operation per word, requiring $O(n)$ operations for n-word-wide values, while the multiplication requires $O(n^2)$ operations. So the number of multiplication (resp. addition) operations requires with N composed cores N^2 times (resp. N times) lesser than with a single core. However, the cycle count reduction from using the composed systems versus the single core is not good as the instruction reduction in the modExp and multiplication algorithms. We recognise that the contention of memory access between cores when executing the memory instructions in the algorithms increases the latency of the instructions, hence, limiting the cycle count reduction. On the other hand, Table 3 shows the SIMD compute mode of the composed system also gains considerable better performance versus the single core system. It speeds up the ChaCha20 encryption more than 2× and 3× compared to the baseline OpenSSL implementation. The instruction reduction in the composed system of 4 cores is comparable to the results of using 128 bit RISC-V vector extension reported in [MPP21].

To evaluate fault-attack mitigation, we model that the fault attacks can inject a fault in the data flow and the control flow of functions. We use the simulation-based evaluation to simulate the fault injections causing a wrong value in the used registers (resp., the program counter) in the data fault attacks (resp., the control flow fault attacks). For each experiment, we repeat 100 times of the AES encryption. And after each encryption, the encrypted data is compared to the reference value to check if the function returns the correct result. During the encryption duration, a fault is randomly injected, which can possibly cause the following results: a) *Passed* means the injected fault does not affect encrypted data; b) *Failed* means wrong encrypted data due to the injected fault is returned. This is a critical case because it can be exploited by an adversary for a successful fault attack; c) *Broken* means the function is broken and can not return encrypted data; d) *Detected* means the occurring fault is detected in the HYDRA redundant mode. Table 4 reports fault rates for the AES encryption executing in the unprotected single core and in the protected HYDRA redundant mode. Interestingly, we can see that even if a fault is injected every time of the 100 AES encryptions, only 62 times (resp. 29 times) of control flow faults (resp. data faults) return wrong encrypted data, which is needed by a fault-injection attack. Remarkably, with the protection by using the HYDRA redundant mode, no wrong encrypted data is returned. All the possible faults inducing wrong encrypted data is detected.

5 CONCLUSION

In this paper, we have presented HYDRA, a proof-of-concept multi-core processor. Harnessing the concept of CLPs to support cryptographically composition modes, the design and implementation of HYDRA emphasises flexibility. For example, our preliminary results suggest that HYDRA can offer the general-purpose benefits of a multi-core processor; at the same time, it can allow specialisation, when needed, to address pertinent efficiency or security challenges.

As a proof-of-concept, a wide range of future work seems interesting. A non-exhaustive list of examples includes:

- We have largely ignored system-level challenges, e.g., how a HYDRA-like architecture is managed by or interacts with a kernel; this and similar challenges need to be explored and addressed, in part to evaluate any result with respect to more realistic workloads.
- An obvious drawback of the wide data-path mode is the increased critical path, which may necessitate a slower clock frequency. It makes sense to explore whether/how this can be addressed (e.g., using a dynamic clock frequency scaling), to minimise or avoid impact on other modes (i.e., execute wide data-path at clock frequency x and all others at some $y > x$).
- By default, and aligning with the area-optimised remit, PicoRV32 uses a bit-serial hardware multiplier. If a combinatorial alternative were used instead, Karatsuba-like [KO63] techniques seem compelling: intuitively, additional control logic could combine the multiple, smaller per-core products into one larger product.

ACKNOWLEDGEMENTS

This work has been supported in part by EPSRC via grant EP/R012288/1, under the RISE (http://www.ukrise.org) programme.

REFERENCES

[API08] J.A. Ambrose, S. Parameswaran, and A. Ignjatovic. MUTE-AES: A multiprocessor architecture to prevent power analysis based side channel attack of the AES algorithm. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 678–684, 2008.

[BBKN12] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[BDM09] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009.

[Ber08] Daniel J. Bernstein. Chacha, a variant of salsa20. In *Workshop Record of SASC: The State of the Art of Stream Ciphers*, 2008.

[BKP21] J.W. Bos, T. Kleinjung, and D. Page. Efficient modular multiplication. Cryptology ePrint Archive, Report 2021/1151, 2021.

[CHC+20] Y. Cheng, L. Huang, Y. Cui, S. Ma, Y. Wang, and B. Sui. Efficient multiple-ISA embedded processor core design based on RISC-V. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2020.

[fip01] Advanced Encryption Standard (AES). National Institute of Standards and Technology, Federal Information Processing Standard (FIPS) 197, 2001.

[HKSS12] Y. Hori, T. Katashita, A. Sasaki, and A. Satoh. SASEBO-GIII: A hardware security evaluation board equipped with a 28-nm FPGA. In *IEEE Global Conference on Consumer Electronics*, pages 657–660, 2012. https://doi.org/10.1109/GCCE.2012.6379944.

[JT12] M. Joye and M. Tunstall, editors. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012.

[KAK96] C. Kaya Koc, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Physics-Doklady*, 7:595–596, 1963.

[KSG+07] C. Kim, S. Sethumadhavan, M.S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S.W. Keckler. Composable lightweight processors. In *International Symposium on Microarchitecture (MICRO)*, pages 381–394, 2007.

[KSV13] D. Karaklajić, J.-M. Schmidt, and I. Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, 2013.

[LYS04] R.B. Lee, X. Yang, and Z. Shi. Validating word-oriented processors for bit and multi-word operations. In *Annual Computer Security Applications Conference (ACSAC)*, pages 473–488, 2004.

[MPP21] B. Marshall, D. Page, and T. H. Pham. A lightweight ise for chacha on risc-v. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 25–32, 2021.

[MvOV96] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC, 1996.

[Res18] E. Rescorla. The Transport Layer Security (TLS) protocol version 1.3. Internet Engineering Task Force Request for Comments (RFC) 8446, 2018.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM (CACM)*, 21(2):120–126, 1978.

[TMA11] M. Tunstall, D. Mukhopadhyay, and S. Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *Workshop on Information Security Theory and Practice (WISTP)*, LNCS 6633, pages 224–233. Springer-Verlag, 2011.

[YDG+19] B. Yuce, C. Deshpande, M. Ghodrati, A. Bendre, L. Nazhandali, and P. Schaumont. A secure exception mode for fault-attack-resistant processing. *IEEE Transactions on Dependable and Secure Computing*, 16(3):388–401, 2019.

[YGD+16] B. Yuce, N.F. Ghalaty, C. Deshpande, C. Patrick, L. Nazhandali, and P. Schaumont. FAME: Fault-attack aware microprocessor extensions for hardware fault detection and software fault response. In *Hardware and Architectural Support for Security and Privacy (HASP)*, pages 8:1–8:8, 2016.