Preventing Soft Errors and Hardware Trojans in RISC-V Cores

Edian B. Annink[a], Gerard Rauwerda[b], Edwin Hakkennes[b], Alessandra Menicucci[d]
Stefano Di Mascio[d,e], Gianluca Furano[e], Marco Ottavi[a,c]
[a]University of Twente, Enschede, the Netherlands, [b]Technolution B.V., Gouda, the Netherlands
[c]University of Rome Tor Vergata, Rome, Italy, [d]Delft University of Technology, Delft, the Netherlands
[e]European Space Agency, European Space Research and Technology Centre, Noordwijk, the Netherlands
edianannink@gmail.com, {gerard.rauwerda, edwin.hakkennes}@technolution.nl
{a.menicucci, s.dimascio}@tudelft.nl, gianluca.furano@esa.int, m.ottavi@utwente.nl

2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) | 978-1-6654-5938-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/DFT56152.2022.9962340

***Abstract*—Soft errors in embedded systems' memories like single-event upsets and multiple-bit upsets lead to data and instruction corruption. Therefore, devices deployed in harsh environments, such as space, use fault-tolerant processors or redundancy methods to ensure critical application dependability. Another rising concern in secure, critical space applications is the possible introduction of hardware Trojans in an untrusted phase of the manufacturing process. Besides environmental side-effects, an adversary that has injected a malicious mechanism e.g., in the processor or memory can trigger unwanted behavior or leak sensitive information. Techniques to prevent or mitigate hardware Trojans are important to ensure hardware security. Leveraging the openness of the RISC-V ISA, this paper introduces a novel solution to improve the security and dependability of softcores with a low area and latency overhead. The instruction validator which is the first part of this solution can effectively detect hardware Trojans and multiple-bit upsets in the instruction memory by checking instruction/address pairs using a Bloom filter probabilistic data structure. The second part of the solution is the proposal of an error correction code instruction memory using Hamming single-error correction to detect and correct single-event upsets. It has also been proven that the Hamming decoder improves the detection performance of the instruction validator.**

***Index Terms*—RISC-V, Hardware Security, Hardware Dependability, Hardware Trojans, Bloom Filters**

I. Introduction

The increase in capacity introduced by successive VLSI nodes resulted in more hardware per die and an increased complexity which, in turn, introduced negative side-effects such as security issues and more dependency on cell stability issues such as single-event upsets (SEUs) and multiple-bit upsets (MBUs). More hardware per area and smaller noise margins mean that SEUs and MBUs occur more frequently [1]. This trend, although mitigated, is visible also for space/critical embedded systems, where the use of larger and larger SRAM-based FPGAs, with processor soft cores and complex single-event effects (SEE) behavior is becoming the rule rather than the exception [2], [3].

A well-known hardware security issue is a hardware Trojan (HWT) which is a *malicious, intentional modification of a circuit design that results in undesired behavior when the circuit is deployed* [4]. HWTs may even lead to catastrophic system failures depending on the type of HWT [5]. The complexity of each step in the fabrication process of an integrated circuit (IC) or deployment of modern FPGAs makes it difficult to prevent HWTs. Reducing cost and a fast time to market (TTM) often forces research and development (R&D) departments to buy intellectual property circuits from other companies which increases the risk of HWTs even further [6].

978-1-6654-5938-9/22/$31.00 ©2022 IEEE

Important phenomena that affect hardware dependability are SEUs and MBUs. SEUs cause single-bit errors per word and MBUs cause double or more bit errors per word and both cause a *temporary change of memory contents or commands in an instruction stream* [7]. SEUs and MBUs in space originate from heavy ions coming from cosmic rays or high-energy protons coming from solar flares. SEUs and MBUs can also occur from secondary cosmic rays which can reach the Earth's surface. SEUs and MBUs often result in data corruption which may lead to system malfunctioning. While radiation hardening leads to fewer SEU and MBU cases in spacecraft, it is important to find other ways to mitigate or decrease SEUs and MBUs in digital circuits. Especially systems that can have a big impact on the environment and human lives such as space, missile, and avionics systems [7].

The spread of an open and free ISA like RISC-V already enabled a vast field of research activities for terrestrial applications (e.g. security, AI, etc.). RISC-V is quickly being adopted in the space sector [8] as a replacement for the aging SPARC ISA and opens strong opportunities to develop highly reliable architectures. Recent studies [9]–[11] show that the number of fault-tolerant RISC-V cores that prevent SEUs and MBUs is still limited. While this is still true, the number of researchers and the industry that is developing fault-tolerance solutions for RISC-V is growing. The first example of fault-tolerant RISC-V cores is the RISC-V core protected by triple modular redundancy (TMR) and Hamming codes based on the unprivileged specification proposed by Santos et al. [10]. A second example is the addition of ECC-protected memory to the out-of-order Rocket core *BOOM* by Berkeley proposed by Dörflinger et al. [9]. Ramos et al. [12] researched the impact of SEUs on multiple soft processors using SRAM-based FPGA implementations including the lowRISC SoC. SEUs were introduced using the Soft Error Mitigation (SEM) IP of Xilinx. The conclusion was that *Application Output Mismatches* caused by Silent Data Corruptions (SDC) and Hangs

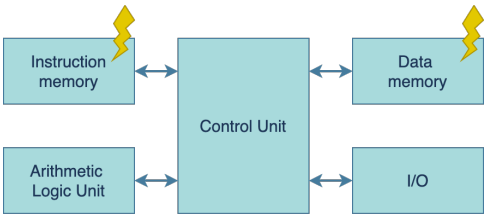

Fig. 1. Harvard architecture diagram

(infinite loop) were the most common faults besides hard faults (exception) and *Architecture Internal Failures* which means that the output is correct, but the internal state of the architecture is not. They ultimately claim that *fault-tolerant techniques should be applied to lowRISC if it is going to be used in space missions*. A study by A. E. Wilson et al. [13] tested the fault-tolerance of RISC-V softcores on Xilinx SRAM-based FPGAs. They have proven that while dependability is improved when using TMR, the dependability of the core is still limited by multiple factors including MBUs affecting two or three TMR domains. Another study by M. Ottavi et al. [14] investigates a signature-based checker that mitigates SEUs in a complex instruction set computer (CISC): The Intel 8051 8-bit microcontroller. This checker checks the control flow integrity by analyzing the signature that is created for every sequence of instructions before every program branch. This signature is generated by linear feedback shift registers (LFSR) and is compared with preloaded signatures. An error is raised if the signature does not exist. This checker provided an average of 98.86% coverage, a high level of protection against freezes, and a correlation of 50% between control flow errors and wrong computations.

The development of defense mechanisms against HWTs is relatively lagging according to a recent survey on RISC-V security regarding hardware and architecture [15]. This survey features multiple proposals that try to detect HWTs in RISC-V. Linscott et al. [16] focus on HWTs that are introduced in the fabrication process of silicon. The proposal is to mitigate HWTs by *mapping the security-critical portions of a processor design to a one-time programmable, LUT-free fabric*. This results in an area overhead of 27% when using the Rocket BOOM RISC-V core. Takahashi et al. [17] propose two detection methods based on machine learning and side-channel analysis. The methods were successful in detecting HWTs in PicoRV and Freedom RISC-V cores. The third proposal by Bolat et al. [18] introduces a protection architecture to detect HWTs in the instruction and data memory in RISC-V using a Bloom filter (BF). Hoque et al. [19] introduce a new HWT class that targets SRAM arrays. They conclude that these HWTs can evade *industry-standard post-manufacturing testing*. A study by A. Palumbo et al. [20] introduces a protection architecture by storing fragmented instruction/address pairs.

The number of HWT, SEU, and MBU countermeasures in RISC-V is still limited and results in a large overhead in terms of area and latency according to recent studies and surveys. A lot of work must yet be done to ensure that RISC-V-based ASICs and softcores are fault-tolerant and resistant to HWTs.

II. Theoretical Background

A. Fault model

It is unlikely that one solution covers all HWTs, SEUs, and MBUs. A fault model and threat model must be introduced to get an overview of the behavior of HWTs, SEUs, and MBUs. Consider the Harvard CPU architecture displayed in Fig. 1.

The instruction memory and data memory are most likely to be influenced by SEUs and MBUs as they take up the most area. The dependability of the instruction and data memory cannot be guaranteed in this case. In this work, we will focus on instruction memory.

B. Threat model

In [5] it is shown that different types of HWTs exist. The presented taxonomy shows that HWTs can be classified by looking at the *insertion phase*, *abstraction level*, *activation mechanism*, *effects*, and *location*. As mentioned before, this paper focuses on the presence of HWTs in the instruction memory. This is an important scope as HWTs could also be implemented in other components of the processor. For example, an HWT can modify the functionality of the registers or the arithmetic logic unit (ALU). This means that HWT detection located in the instruction memory can be bypassed as instructions are changed after the instruction fetch (IF) phase of the processor. Fig. 2 displays the proposed threat model of this type of HWT. This so-called instruction memory HWT is part of the class ($\in$) or is not part ($\notin$) of the class in each attribute.

An HWT that resides in instruction memory can be inserted in every phase. A requirement in the specification phase can be added that simplifies adding HWTs in a later phase. A possible example is that a third-party memory IP block is used in the design phase that injects malicious instructions into the instruction memory. Another example is that a developer can add malicious HDL code that implements an HWT in the instruction memory. In the worst case, an alternative photomask can be used to change or replace the instruction memory. If the testing phase is also modified to prevent the detection of this malicious change, an HWT can be introduced in the fabrication phase of the IC. The same is the case for assembly and packaging. If the memory is separated from the processor on a PCB, a malicious memory component can be used.

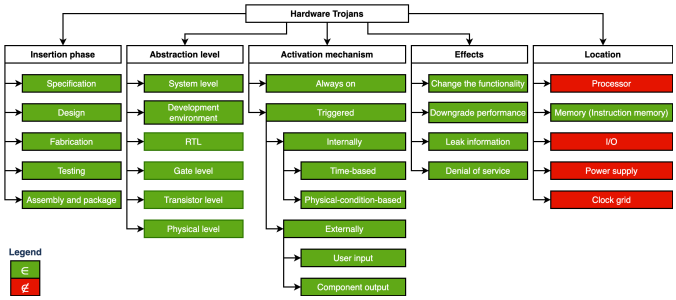

Fig. 2. Instruction memory HWTs classified by the HWT taxonomy [5]

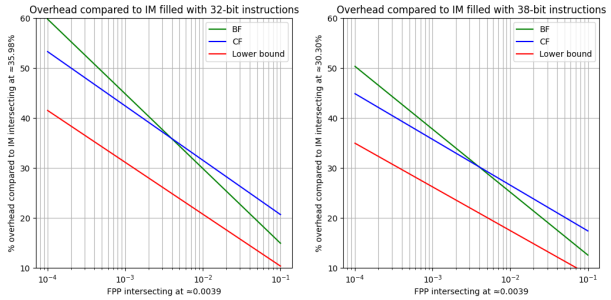

Fig. 3. CF and BF area overhead compared to IM

The HWT can also be implemented at every abstraction level as instruction memory can be maliciously modified or replaced on every level.

This specific HWT also supports every activation mechanism or *trigger mechanism*. The HWT can be always-on, internally triggered, and externally triggered.

All effects, also called *payloads*, are supported. The functionality can be changed such as changing instructions or injecting malicious instructions. Performance can be downgraded by spamming instructions or repeating instructions. Information can be leaked by injecting instructions that copy sensitive data to memory addresses that can be read by the adversary. Denial-of-Service (DoS) is also a possibility, e.g., completely disabling the instruction memory. No instructions can be written to the instruction memory and reading the instruction memory results in undefined signals which completely halts the pipeline.

This HWT only resides in the instruction memory; hence, the other locations are not part of the classification.

C. Probabilistic data structures

The Bloom filter (BF) and Cuckoo filter (CF) probabilistic data structures were analyzed.

The BF, introduced by Howard Bloom in 1970 [21], is the most used data structure that solves the *membership problem* for a dataset which is *a task to decide whether some element belongs to the dataset or not* [22]. The BF supports inserting and testing elements. The BF introduces a small percentage of errors which increases with the number of elements in the filter, also known as the false positive probability (FPP) (i.e. the BF returns that an element is part of the set while it was not inserted).

The CF proposed by B. Fan et al. in 2014 [23] also solves the *membership problem* like the BF. The CF supports deletion besides insertion and testing.

As the focus of this paper is embedded RISC-V cores, area overhead is an important criterion. A study by P. Reviriego et al. [24] showed that the BF false positive probability curve is lower than the CF FPP curve for a growing table occupancy. This means that in practice, the BF has a better false positive rate (FPR) with a growing table occupancy. While this is true, the CF consumes less memory than the BF with a maximum table occupancy of 95% and an FPP $\leq 0.39\%$. The CF and BF can both be evaluated by plotting the theoretical overhead compared to the instruction memory when varying the FPP. Fig. 3 displays the amount of area overhead for the CF and BF compared to the instruction memory (IM) for multiple configurations. The CF starts using fewer bits per item than the BF at an FPP threshold of 0.39%. This threshold corresponds to an overhead of 35.98% without using ECC and 30.30% when using a Hamming(38, 32) SEC code. An FPP threshold of 0.39% results in a large overhead of up to 35.98%. To conclude, the BF was chosen as the best candidate. An FPP this small is outside of the scope of this paper and results in an overhead that is unacceptable when considering strict memory requirements.

D. Non-cryptographic hash functions

Latency overhead must be minimized as checking the instruction/address pairs should not take multiple pipeline cycles to prevent major damage caused by faults or HWTs. For this reason, hash techniques that are used in network-based FPGA applications become relevant. Two studies from R. Dobai et al. show that while CRC is not designed as a hash function, it is often used in hardware applications. A CRC-based implementation was used for a hardware implementation on an FPGA that allows for fast lookups in dynamic packet filtering [25], [26] which is comparable to a fast lookup of instruction/address pairs concurrently to a RISC-V pipeline. Another study from M. J. Lyons et al. [27] evaluates the design of a BF for ultra-low-power systems by proposing a hardware accelerator for compressed BFs. In this hardware accelerator, the Multiply and Shift hash is used which was originally introduced by Dietzfelbinger et al. [28]. The MultiplyShift hash is a universal hashing scheme and can be computed using eq. 1.

$$h_a(x) = \left\lceil \frac{ax \mod 2^k}{2^{k-l}} \right\rceil, \quad \text{for } 0 \leq x, a < 2^k \qquad (1)$$

III. SEE AND HWT PROTECTION ARCHITECTURE

Fig. 4 displays the instruction validator design using the BF. The first step is to hash the instruction/address pair using the MultiplyShift or CRC-32C hash. The second step is testing the instruction/address hashes in the BF. As proposed in the study by Bolat et al. [18], each hash function can have its memory element that consists of a part of the total BF bit array. This separation allows for concurrently reading all the memory elements instead of reading the memory elements

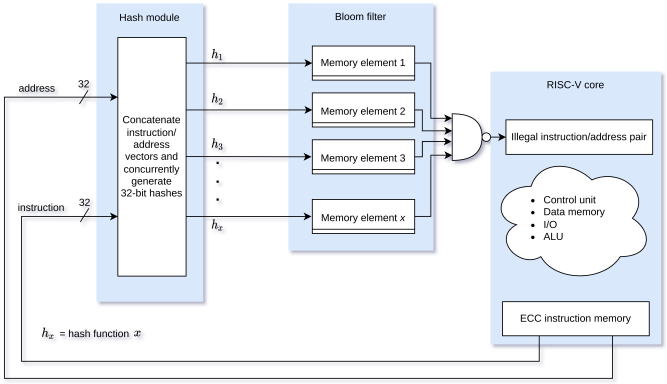

Fig. 4. High-level hardware design of the instruction validator and ECC instruction memory with the RISC-V core

Algorithm 1 m and k optimization with $\frac{m}{k}$ being a power of two

Require: $\epsilon \leq 1$
Require: $n \leq 2^{32}$
  $k_{opt} \leftarrow 0$
  $m_{opt} \leftarrow \infty$
  $\epsilon_{opt} \leftarrow 0$
  while $k \leq 7$ **do**
    while $x \leq 18$ **do**
      $p \leftarrow \left(1 - e^{-\frac{k \cdot n}{k \cdot 2^x}}\right)^k$
      if $p < (\epsilon \cdot 1.05) \ \wedge \ k \cdot 2^x < m_{opt}$ **then**
        $k_{opt} \leftarrow k$
        $m_{opt} \leftarrow k \cdot 2^x$
        $\epsilon_{opt} \leftarrow p$
      end if
    end while
  end while

sequentially. To prevent using modulo and use bit slicing, the m-k *optimization* algorithm and *rounding optimization* were introduced. To achieve this, both optimizations ensure that the individual bit array sizes are a power of two. The rounding optimization rounds the individual bit array size up to the next power of two. The m-k optimization algorithm displayed in Algorithm 1 computes the most optimal number of hash functions and bit array size with $\frac{m}{k}$ being a power of two based on specified n and ϵ. The minimum in this optimization is the lowest total bit array size while p is lower than $\epsilon \cdot 1.05$ which represents allowing a 5% deviation. The constraints for k and $2^x = m$ can be set accordingly which are 7 and 18 in this case. Variables k_{opt}, m_{opt} and ϵ_{opt} hold the most optimal k, m and ϵ after executing this algorithm.

A NAND gate can be connected to all outputs of the memory elements and becomes high when one of the memory elements tests negative when testing instruction/address pairs. The instruction validator validates all the instruction/address pairs and checks for MBUs on top of SEUs that the ECC instruction memory is handling. This means that detecting double errors using Hamming SEC-DED is redundant and Hamming SEC suffices and results in less area overhead.

IV. Experimental results

A. Simulation results

Fig. 5 displays the simulation setup. The Python module ***CO****routine based* ***CO****simulation* ***T****est****B****ench* (cocotb) [29] was used to simulate the instruction validator and ECC. The so-called *VHDL generator script* was developed which generates the instruction validator, RAM elements containing the individual BF bit arrays, the MultiplyShift or CRC-32C hash, and the instruction memory as shown in the test harness. The instruction memory is preloaded with Hamming SEC encoded instructions as displayed in Fig. 6. The saboteur before the Hamming decoder was used to inject SEUs and MBUs. The saboteur after the Hamming decoder was used to inject HWTs. The VHDL generator script was imported by an automation flow to simulate all different optimization/program configurations executing the following test cases:

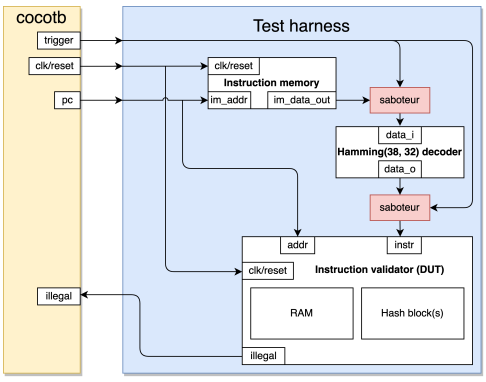

Fig. 5. Simulation setup

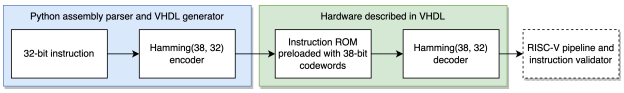

Fig. 6. ECC flow

1) Testing without injecting faults
2) Testing while injecting SEUs and MBUs
3) Testing while triggering different types of HWTs

All the test cases were executed using instruction/address pairs from the following benchmark programs provided by MiBench [30] compiled with the RISC-V GNU toolchain: Rijndael AES, Blowfish, Dijkstra, FFT, Patricia, SHA, and Quicksort.

Testing without faults verified the correct functioning of the ECC instruction memory and instruction validator. Running the test resulted in the illegal signal of the instruction validator staying low for all program/hash/optimization configurations.

Multiple *cocotb* methods were developed to test the instruction validator and Hamming decoder. All single-bit errors were successfully detected and corrected by the Hamming decoder. When injecting double- and triple-bit errors in 10 different runs, the FPR of all configurations using the MultiplyShift hash stayed below the set FPP of 0.05 with a small deviation. Observing the simulation waves in more detail resulted in an interesting finding. The Hamming decoder occasionally miscorrects bits when introducing MBUs as displayed in Table I which results in a better overall FPR for the instruction validator. This effect was proven by running all the MBU simulations without the Hamming decoder which resulted in an overall higher FPR. Another interesting observation was when applying a modulo with a power of two on the CRC-32C hash output to decrease area overhead, all tests failed. An additional analysis was executed on the CRC-32C hash by looking at single output bits. This resulted in the conclusion that the CRC-32C output could not be sliced and the complete hash output must be used to get a strong distribution and respect the set FPP.

A flow was developed that dynamically injected two HWT types. Each HWT type was injected with a probability of 0.1 per instruction fetch. The Dijkstra program was executed 10000 times while HWTs were injected in different parts of the program in each execution. The *injecting HWT* attack injected random instructions and became fully detected with a

Output	Codeword	Data bits
Instruction ROM	11111101011000100001001101100010010110	11111110110001000010010110000011
Saboteur	11111101011000100001001000110010110	11111110110001000010010110000011
Hamming decoder	11111101011000100001001001110001001011 0	11111110110001000010010111000011

TABLE I
Hamming decoder introducing an extra fault

Checker	LUTs	FFs	BRAM size (kbit)	F_{max} (MHz)	DSP Blocks
MultiplyShift with m-k	268	127	90	175	18
MultiplyShiftPipelined with m-k	272	94	90	175	15
Proposal in [18]	1539	89	64	106	0
Proposal in [20]	75	31	208	275	0

TABLE II
COMPARING SYNTHESIS RESULTS TO OTHER CHECKERS WITH DIJKSTRA AND $\epsilon = 0.01$

Component	Dynamic P (W)	LUTs	FFs	BRAMs	DSP blocks
FreNox RISC-V core	0.012	2363	1654	1	4
Instruction memory	0.019	0	0	4 (and 1 BRAM for check bits)	0
Data memory	0.013	38	0	4	0
ECC encoder	< 0.001	20	0	0	0
ECC decoder	0.009	97	0	0	0
Instruction validator	0.019	322	75	2.5	15
Overhead[1]	**63.6%**	**18.3%**	**4.5%**	**38.9%**	**375%**
Overhead[2]	**56%**	**11.2%**	**2.9%**	**38.9%**	**375%**

TABLE III
SYNTHESIS RESULTS OF FRENOX SOC-E

sequence length of four instructions with the CRC-32C hash. The MultiplyShift with m-k optimization which approximates the performance of CRC-32C detected all HWT attacks at a sequence length of five instructions. The MultiplyShift with rounding optimization was able to detect all HWT attacks at a sequence length of three instructions. However, this was the case as the rounding optimization increased the total bit array sizes which results in a better FPR. The *modifying HWT* attack modified instructions by executing an AND operation with a NOP instruction which has the same effect as skipping the instruction. This attack became fully detected at an instruction sequence length of three for all hash/optimization configurations.

B. Synthesis results

All instruction validator program/hash/optimization configurations generated by the VHDL generator script were synthesized for the Arty A7-100T containing the Xilinx Artix-7 XC7A100TCSG324-1 FPGA. The instruction validator with the CRC-32C hash failed the timing requirements of 100MHz because the modulo operation resulted in too large a slack. Nevertheless, using CRC-32C might still be useful for ASICs as multipliers needed by MultiplyShift consume a large amount of area in ASICs. Table II displays the synthesis results of the instruction validator with the MultiplyShift hash and the checkers introduced in [18] and [20]. The Sudoku Solver program synthesis results from [18] and [20] are displayed in the table. The Sudoku Solver program consists of 475 instructions which is similar to the number of instructions in the Dijkstra program which consists of 451 instructions. It can be observed that the amount of LUTs is higher than the proposal in [18] while being significantly lower than the proposal in [20]. The amount of FFs is higher than both proposals which are consumed by the instruction validator hash functions. The BRAM size is slightly bigger than [20] while being significantly lower than [18]. This has to do with the fact that the proposal in [18] allocates a fixed amount of memory to store instruction/address pairs. The F_{max} is between the proposal in [18] and [20]. The biggest difference between the instruction validator and both proposals is the amount of DSP blocks.

[1]Overhead of the instruction validator, ECC encoder/decoder and ECC check bits compared to the FreNox RISC-V core and the instruction/data memory

[2]Overhead of the instruction validator, ECC encoder/decoder and ECC check bits compared to the components in the table and other components part of the FreNox SoC-e

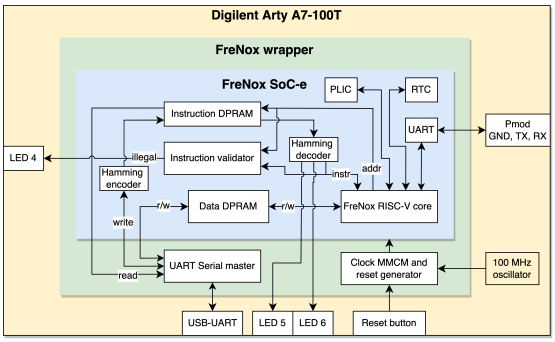

Fig. 7. Integration with FreNox

C. Implementation results

The instruction validator was integrated with the FreNox SoC-e; a SoC instantiating Technolution's RISC-V core called FreNox [31] and ECC instruction memory as displayed in Fig. 7. The FreNox SoC-e was synthesized with a separate Hamming SEC-DED encoder/decoder and the instruction validator was configured with the MultiplyShift hash and m-k optimization. The Hamming decoder being part of the pipeline critical path resulted in a frequency of 80 MHz, equal to a decrease of 20%. The instruction validator did not introduce any latency overhead. Compared to all components instantiated by the FreNox SoC-e, the instruction validator and Hamming encoder/decoder introduced a power overhead of 56%, a LUT overhead of 11.2%, and a FF overhead of 2.9% as displayed in Table III. A 1024 bit register file, 1460 32-bit instructions, a total bit array size of 10240 and an 8608 bit data memory resulted in a total area overhead of $\frac{10240+6\cdot1460}{32\cdot1460+1024+8608} = 33.7\%$ including the 6 check bits introduced by Hamming SEC encoding. A DSP block overhead of 375% was introduced which is acceptable as this makes up for just 7.9% of the available DSP blocks with the 4 DSP blocks consumed by the FreNox RISC-V core. Finally, the same tests that were introduced in the simulation were executed with FreNox. This again proved the functionality and advantages of both the ECC instruction memory and the instruction validator. All single-bit errors were detected and corrected by the Hamming decoder. Introducing double- and triple-bit faults resulted in hangs and crashes in the FreNox RISC-V core which were successfully detected by the instruction validator. An HWT attack was introduced in the QuickSort program based on the *modifying HWT* attack that overwrote a jump and link instruction. This resulted in the QuickSort algorithm subroutine being bypassed and failing to sort the array. While this attack only consisted of one instruction, it was also successfully detected by the instruction validator.

V. CONCLUSION

This paper introduced a novel solution to improve the security and dependability of RISC-V softcores with a low area and latency overhead. It has been proven that the proposed

instruction validator can effectively detect HWTs and MBUs in the instruction memory by checking instruction/address pairs using a BF probabilistic data structure. ECC instruction memory using Hamming SEC was proposed to detect and correct SEUs which improved the detection performance of the instruction validator besides error correction. An automation framework was developed to generate, simulate and synthesize the instruction validator for different configurations which presents the designer with different options based on the application requirements. Besides this automation framework, two BF optimizations were proposed that decrease the BF area overhead. To conclude, the instruction validator and ECC were successfully tested and integrated into the FreNox SoC-e with the FreNox RISC-V core on the Digilent Arty A7-100T development board using the Xilinx Artix-7 XC7A100TCSG324-1 FPGA. Integrating the instruction validator and ECC led to an area overhead of 33.7%. The introduction of ECC resulted in a maximum frequency reduction of 20%.

This proves that the instruction validator and ECC instruction memory are suitable to use for embedded RISC-V softcores with strict security and dependability requirements.

REFERENCES

[1] P. Nsengiyumva, D. R. Ball, J. S. Kauppila, N. Tam, M. McCurdy, W. T. Holman, M. L. Alles, B. L. Bhuva, and L. W. Massengill, "A comparison of the seu response of planar and finfet d flip-flops at advanced technology nodes," *IEEE Transactions on Nuclear Science*, vol. 63, no. 1, pp. 266–272, 2016.

[2] G. Furano and A. Menicucci, "Roadmap for on-board processing and data handling systems in space," in *Dependable Multicore Architectures at Nanoscale*. Springer, Cham, 2018, pp. 253–281.

[3] V. Vlagkoulis, A. Sari, J. Vrachnis, G. Antonopoulos, N. Segkos, M. Psarakis, A. Tavoularis, G. Furano, C. B. Polo, C. Poivey *et al.*, "Single event effects characterization of the programmable logic of xilinx zynq-7000 fpga using very/ultra high-energy heavy ions," *IEEE Transactions on Nuclear Science*, vol. 68, no. 1, pp. 36–45, 2020.

[4] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, May 2016. [Online]. Available: https://doi.org/10.1145/2906147

[5] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.

[6] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, 2011, pp. 67–70.

[7] E. Petersen, *Single event effects in aerospace*. IEEE Press, 2011.

[8] S. Di Mascio, A. Menicucci, G. Furano, C. Monteleone, and M. Ottavi, "The case for risc-v in space," in *Applications in Electronics Pervading Industry, Environment and Society*, S. Saponara and A. De Gloria, Eds. Cham: Springer International Publishing, 2019, pp. 319–325.

[9] A. Dörflinger, Y. Guan, S. Michalik, S. Michalik, J. Naghmouchi, and H. Michalik, "Ecc memory for fault tolerant risc-v processors," in *Architecture of Computing Systems – ARCS 2020*, A. Brinkmann, W. Karl, S. Lankes, S. Tomforde, T. Pionteck, and C. Trinitis, Eds. Cham: Springer International Publishing, 2020, pp. 44–55.

[10] D. A. Santos, L. M. Luza, C. A. Zeferino, L. Dilillo, and D. R. Melo, "A low-cost fault-tolerant risc-v processor for space systems," in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2020, pp. 1–5.

[11] W. Heida, "Towards a fault tolerant risc-v softcore," in *Master thesis*, 2016.

[12] A. Ramos, J. A. Maestro, and P. Reviriego, "Characterizing a risc-v sram-based fpga implementation against single event upsets using fault injection," *Microelectronics Reliability*, vol. 78, pp. 205–211, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271417304493

[13] A. E. Wilson and M. Wirthlin, "Neutron radiation testing of fault tolerant risc-v soft processor on xilinx sram-based fpgas," in *2019 IEEE Space Computing Conference (SCC)*, 2019, pp. 25–32.

[14] M. Ottavi, S. Pontarelli, A. Leandri, and A. Salsano, "Design and evaluation of a hardware on-line program-flow checker for embedded microcontrollers," in *2006 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2006, pp. 371–379.

[15] T. Lu, "A survey on RISC-V security: Hardware and architecture," *CoRR*, vol. abs/2107.04175, 2021. [Online]. Available: https://arxiv.org/abs/2107.04175

[16] T. Linscott, P. Ehrett, V. Bertacco, and T. Austin, "Swan: Mitigating hardware trojans with design ambiguity," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–7.

[17] J. Takahashi, K. Okabe, H. Itoh, X.-T. Ngo, S. Guilley, R.-R. Shrivastwa, M. Ahmed, and P. Lejoly, "Machine learning based hardware trojan detection using electromagnetic emanation," in *Information and Communications Security*, W. Meng, D. Gollmann, C. D. Jensen, and J. Zhou, Eds. Cham: Springer International Publishing, 2020, pp. 3–19.

[18] A. Bolat, L. Cassano, P. Reviriego, O. Ergin, and M. Ottavi, "A microprocessor protection architecture against hardware trojans in memories," in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2020, pp. 1–6.

[19] T. Hoque, X. Wang, A. Basak, R. Karam, and S. Bhunia, "Hardware trojan attacks in embedded memory," in *2018 IEEE 36th VLSI Test Symposium (VTS)*, 2018, pp. 1–6.

[20] A. Palumbo, L. Cassano, P. Reviriego, G. Bianchi, and M. Ottavi, "A lightweight security checking module to protect microprocessors against hardware trojan horses," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021, pp. 1–6.

[21] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, jul 1970. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/362686.362692

[22] A. Gakhov, *Probabilistic data structures and algorithms for big data applications*. Books on Demand, 2019.

[23] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 75–88. [Online]. Available: https://doi.org/10.1145/2674005.2674994

[24] P. Reviriego, J. Martínez, D. Larrabeiti, and S. Pontarelli, "Cuckoo filters and bloom filters: Comparison and application to packet classification," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2690–2701, 2020.

[25] R. Dobai and J. Korenek, "Evolution of non-cryptographic hash function pairs for fpga-based network applications," in *2015 IEEE Symposium Series on Computational Intelligence*, 2015, pp. 1214–1219.

[26] L. Kekely, M. Žádník, J. Matoušek, and J. Kořenek, "Fast lookup for dynamic packet filtering in fpga," in *17th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2014, pp. 219–222.

[27] M. J. Lyons and D. Brooks, "The design of a bloom filter hardware accelerator for ultra low power systems," in *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 371–376. [Online]. Available: https://doi.org/10.1145/1594233.1594330

[28] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, "A reliable randomized algorithm for the closest-pair problem," *Journal of Algorithms*, vol. 25, no. 1, pp. 19–51, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677497908737

[29] "cocotb — python verification framework." [Online]. Available: https://www.cocotb.org/

[30] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.

[31] "Frenox risc-v softcore," Jan 2022. [Online]. Available: https://www.technolution.com/advance/fpga-design-and-implementation/frenox-risc-v-softcore/?noredirect=en-GB