Aftab: a risc-v implementation with configurable gateways for security

AFTAB: A RISC-V Implementation with Configurable Gateways for Security

Maryam Rajabalipanah[1], Mahboobe Sadeghipour Roodsari[1], Zahra Jahanpeima[1], Gianluca Roascio[2], Paolo Prinetto[2], Zainalabedin Navabi[1]
[1] School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Tehran, Iran
{ m.rajabalipanah, mahboube.roudsari, jahanpeima, navabi}@ut.ac.ir
[2] Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy
{gianluca.roascio, paolo.prinetto}@polito.it

***Abstract*— A processor plays an important role in the security of an entire embedded system. There are two reasons for this. One is that a processor is a general-purpose machine, the program of which can be altered for ill-intended purposes. The other factor that adds to the vulnerability of the embedded system is that an embedded processor uses memory addressing for all its instructions and data. This creates a wide-open gateway in and out of a processor by which secure data can be read, unintended data and instructions can be injected in the processor, and the processor can be made to perform unwanted tasks and operations. The remedy we have planned for this is securing the memory gateways and separating them from the rest of the processor architecture. This is to say that we design configurable gateways for instruction and data read and write operations that can be configured to prevent various forms of attacks coming from the processor's memory. These configurable gateway architectures are applied to a RISC-V architecture that we have implemented at the RT Level.**

This paper discusses our implementation of RISC-V architecture that we refer to as AFTAB. The paper emphasizes on the memory gateways of this processor and shows its interfaces with the configuration part of the processor and the architecture of the read and write gateways. After a general presentation of security and threads, we show how AFTAB gateways can be designed and configured for certain types of attacks. All works presented in this paper have been described at the RT Level and synthesized. The synthesis results will be presented.

Keywords—Processor, RISC-V, Configurable Gateways, Security, Buffer Overflow, Hardware Implementation.

I. INTRODUCTION

According to the recent progress of electrical systems and their capability to perform diverse applications, they are susceptible to several security threats. Also, with the recent proliferation of embedded devices, there is a growing need to develop a security framework to protect such devices. Processors that play the role of the heart of a digital system, are one of the most critical components in an embedded system that attract the attention of attackers. There are several attack scenarios such as return-oriented programming (ROP), code reuse, buffer overflow, code injection, memory safety violation, and pointer corruption in the processors. These attacks play a significant role in the system components, wherein an adversary can control the whole system by exploiting vulnerabilities in the system.

There are many techniques for protecting systems against some of the attacks. Stack canaries are words placed on the stack to detect return address overwriting that can happen by attackers [1]. Control flow integrity (CFI) is another technique that guarantees the program follows CFG in the runtime [2]. Address Space Layout Randomization (ASLR) prevents memory corruption with randomization address space positions of stack, code, and other data areas [3].

Accordingly, the processor designers put their efforts into designing hardware that is secured against attackers. Several works have been proposed to solve the security issues in ARM [4], Intel [5], [6], and RISC-V processors [7-9].

RISC-V is an open-source Instruction Set Architecture developed by the University of California, Berkeley [10]. Owing to being simple, free access, and open-source ISA that allows anyone to design and customize it for a specific target application, RISC-V is becoming more and more popular. Its application domains include IoT, machine learning, signal processing, real-time embedded systems [11-14]. This also enhances the sensitivity of the system running on RISC-V against the attacks, regardless of time. The processor's vulnerabilities can be exploited at boot time and/or at run time. RISC-V being open-source provides an environment to implement different levels of security by adding hardware-based security solutions for a wide range of attacks.

We exploit the popularity of RISC-V and present our RISC-V based processor, called AFTAB (**A** **F**ine **T**urin **A**rchitectural **B**eing) meaning sunshine in the Persian language. The AFTAB architecture includes a core and interfacing gateways. The gateways wrap around the core and control and monitor its incoming and outcoming data from/to the memory and all memory-mapped devices. This paper focuses on the buffer-overflow attack in Return-Oriented Programming (ROP) as an instance of vulnerability that is directly related to the memory [15]. Not only the addresses that are the entry points defined by compiler/developer but also any executable address can be targeted by the jump and return instructions. This is the basic of what attackers use. The resolution of this problem in the AFTAB architecture is to involve a lightweight hardware-based shadow stack to work against such attacks. Since the gateways are internal to the architecture of AFTAB, they cannot be attacked as easily as the memory that connects to AFTAB by external busses. This makes our AFTAB architecture a secure implementation of RISC-V.

The rest of the paper is organized as follows. Section II briefly introduces RISC-V ISA and reviews RV32IM's instructions and user-accessible registers. At the end of this section, the AFTAB implementation is presented. Section III illustrates our configurable gateways and their advantages. Section IV explores the RISC-V security concerns. Section V describes our remedy for RISC-V protection against ROP attacks. Section VI discusses the experimental results in terms of hardware resources. Section VII presents conclusions and a brief review of our overall work.

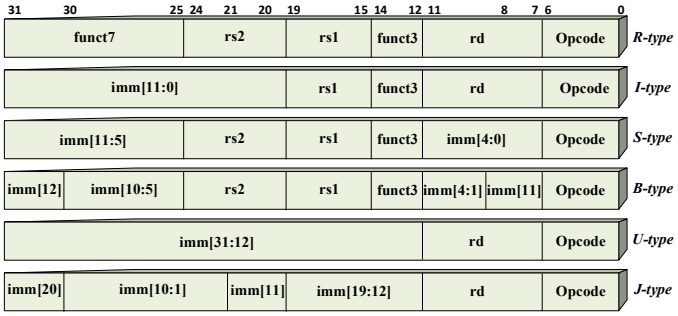

Fig. 1. RISC-V Instruction Formats

II. AFTAB Design of RISC-V

A. RISC-V

RISC-V is the fifth-generation simplified instruction set architecture developed by UC Berkeley and is currently supported by the RISC-V foundation [10], [16]. It is based on reduced instruction set computer (RISC) principles with several extensions for 32-bit, 64-bit, and 128-bit instruction widths. It has a fixed base integer ISA that includes three primary variants (RV32I, RV64I, and RV128I) providing 32-bit, 64-bit, and 128-bit address spaces respectively [17]. All RISC-V processor implementations must support one of the three mentioned standard base ISA. Additionally, the standard bases can work with the standard extensions without contradiction. Some of the widely used extensions are referred to as M for Integer Multiplication and Division, A for Atomic Instructions, and C for Compressed Instructions.

This paper presents our AFTAB secure implementation for the RV32IM, i.e., 32-bit instruction width architecture supporting base integer ISA with its M-extensions.

B. AFTAB Programmer's View

1) Instruction Set

RISC-V instructions are 32 bits in length, and there are six core instruction formats (R/I/U/S/B/J) as shown in Fig. 1. Letters R, I, S, B, U, and J are used for instruction types that correspond to register-register, short immediate and loads, stores, conditional branches, long immediate and unconditional branch operations, respectively. In Fig. 1, rs, rd, funct and imm respectively refer to source register, destination register, function, and immediate value. The combination of funct7 and funct3 with opcode determines what operation to perform.

RISC-V instructions are categorized into arithmetic, data transfer, logical, shift, and conditional/unconditional branch instructions. Arithmetic and logical instructions have either three or two register operands, and a sign-extended 12-bit immediate, denoted by the I-type. Shift instructions use the I-type or R-type format and can be done right/left, logical/arithmetic, and register/immediate.

Data transfer instructions, i.e., load and store, transfer a value between the registers and memory. Loads are encoded in I-type format, and stores are S-type. These instructions can transfer signed/unsigned, byte or half-word, or word value.

The JAL and JALR instructions specify unconditional jumps and use the J-type format. The target address can be specified relative to the program counter (JAL) or as an absolute address in a register (JALR). On the other hand, branch instructions denote conditional jumps based on a comparison. Their first two operands are registers of which the values are compared. The third operand specifies the destination address relative to the program counter. Branches use the B-type instruction format.

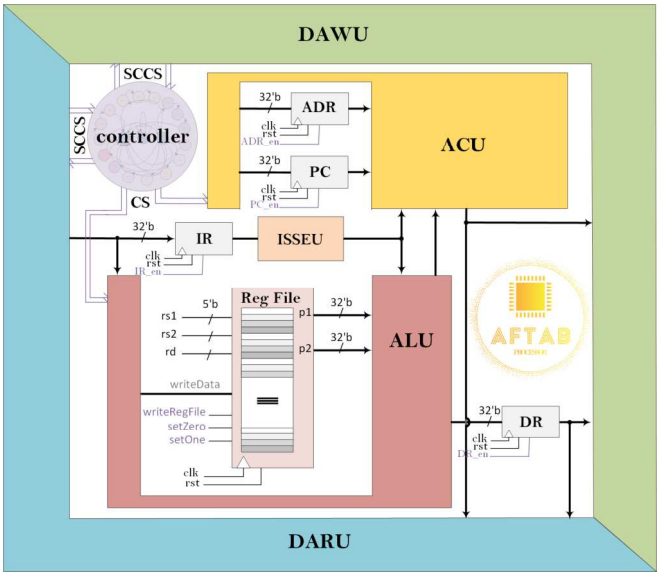

Fig. 2. Abstract RTL of AFTAB Processor

There are also multiplication and division instructions in RV32IM that use the R-type format. In these instructions, the content of one register is multiplied/divided by the contents of another register and the result is placed in the destination register. Multiplication/division can be signed or unsigned operations.

2) User Accessible Registers

The RISC-V has 32 general-purpose registers. X0, the first register, is hardwired to zero, writes to it is ignored, and is always read as 0. The other registers are introduced as follow: x1 (return address), x2 (stack pointer), x3 (global pointer), x4 (thread pointer), x5 (alternate return address), x6-7 (temporary registers), x8 (frame pointer), x9 (saved register), x10-17 (function argument/return value), x18-27 (saved registers), x28-31 (temporary registers).

C. AFTAB RTL Architecture

This part presents the architecture, configuration, and operation of our 32-bit AFTAB implementation of RISC-V. AFTAB communicates with the memory with standard memory accessing handshaking signals through a 32-bit address bus and 32-bit bi-directional data bus.

The description of the AFTAB RTL architecture is divided into three main parts: Datapath, Controller, and Gateways. First, we depict the overall Datapath and its main components. Next, we illustrate the controller and its contribution to the Datapath, and finally, we demonstrate the Gateways.

1) Overall Datapath

In the AFTAB's Datapath, shown in Fig. 2, the user-accessible registers are included in a register-file of sixteen 32-bit registers, each of which can be used as the source or destination of an instruction. This register-file has a multi-port structure, the interface of which is provided by three 4-bit ports for addressing the register-file, a 32-bit *writeData*, and two 32-bit *P1*, *P2* for the read data. The access to the contents of two source registers is simultaneous. The *writeRegFile*, *setZero*, and *setOne* are the input ports that will be issued for

writing the *writeData*, value one, and value zero to the destination register. The register-file is encircled by a unit called arithmetic and logical units (ALU) which is mainly responsible for performing arithmetic, shift, logical, multiply and divide operations.

The AFTAB's Datapath also includes four standard registers, i.e., PC, IR, ADR, DR, that are mainly used for memory data transfer and memory addressing. The program counter register (PC) keeps the address of the next instruction and the instruction register (IR) stores the newly fetched instruction. The address register, i.e., ADR, refers to the memory location for reading, or the address to which DR is to be written. The task of address calculation is handled by the address calculation unit (ACU), which encompasses the ADR and PC registers and provides them with the proper inputs.

As discussed in Section II, a fetched instruction consists of several fields based on the format of the instructions. All the instruction formats except register/register have an immediate value that is spread in various parts of an instruction (separate immediate fields). When an instruction is to be executed, the appropriate immediate value must be constructed from the immediate fields. Hence, in our architecture, a unit called Immediate Selection and Sign Extension Unit (ISSEU), which is sitting next to IR in Fig. 2 is appointed to form the immediate value according to the currently fetched instruction format.

2) Controller (Overall view)

The AFTAB's controller controls the flow of data in the AFTAB's Datapath by providing both an appropriate number of states and proper control signals. Here we concentrate on the main states of our controller.

When an instruction is to be fetched, the first state (fetch) opens a path in the Datapath from the PC register to the memory and from the memory to IR in order to read the instruction from the memory and store it in IR. The next state (decode) specifies the instruction's operation by checking the opcode of the instruction and leads the controller to the corresponded execution states. There are sets of executing states for each instruction's operation that achieve the required actions for running the operation in our Datapath and returning to the fetch state.

The AFTAB's controller is also responsible for providing security check control signals related to security solutions, an example for which is proposed in Section IV.

3) Gateways

The data transfer instructions require access to the memory. We develop our design by providing gateways to facilitate data transfer proceedings. The gateways, called DARU and DAWU (Data Adjustment and Read/Write Unit), sit outside the AFTAB's core, and act as an interface between the core and external devices. They take control of the bi-directional transfer of data from the processor to the outside, and vice versa.

Beyond the above task, the gateways establish a platform to improve the performance of the AFTAB processor. For example, many hardware solution techniques can be implemented in these gateways for security. In the following section, we introduce the other benefits that can be gained by our proposed gateways. Section V describes the shadow stack hardware solution located in the gateways.

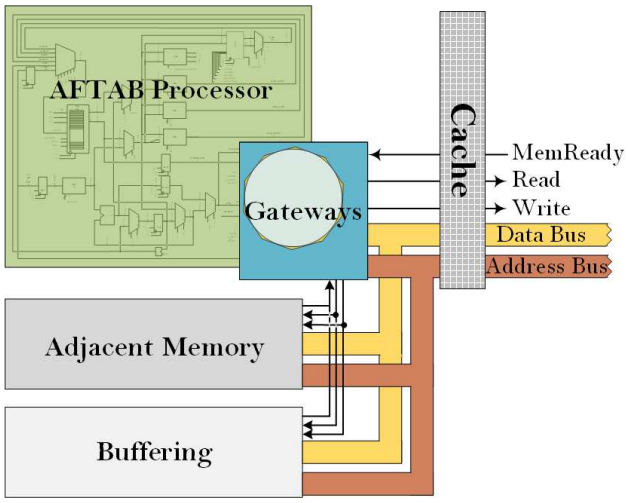

Fig. 3. The Visionary View of Processor Side of an Embedded System

III. Configurable Gateways

AFTAB gateways provide configurable mechanisms for memory access. These hardware units are separate modules from the core of AFTAB. This way, handling memory-mapped operations and providing alternatives and remapping memory accesses only go into modules that are dedicated for such purposes. Memory map operations that benefit from such dedicated hardware units include accessing various types of memories, cache handling, IO devices, interrupt, and stack handling. In general, gateways provide memory access for instructions related to memory access or instructions that use memory-mapped registers, IO devices, adjacent memory, closely coupled memory, etc.

The following parts explain more impacts of these configurable gateways and their architectures.

A. Impact of Gateways

Our gateways facilitate and accelerate access to the memory. Some benefits of DARU and DAWU gateways in the processor are listed as below:

- Data buffering and adjustment: The gateways control the sequence and the number of transferred bytes in load and store instructions.
- Endian handling: These gateways are capable of handling big/little-endian data format.
- Limited handling of most handshaking and arbitration tasks for small systems.
- Limited preliminary address decoding for system memory and devices.
- Gateways can potentially handle near-memory processing.

B. AFTAB Core Interfacing

DARU and DAWU gateways are interfaces between the AFTAB processor and its memory. These gateways are separated from the core and perform all the tasks mentioned in the previous part. DARU and DAWU have their independent FSMs for buffering, handshaking, and internal

register controls. Processor *Read/Write* control signals are directly connected to the gateways, and the gateways also manage bus driving for the processor.

C. Bus Utilization Switching

When a read or write operation is to occur, the processor issues the *Read* or *Write* control signal to inform the target device. The target device can be physically near the processor like the adjacent memory and the buffering unit, shown in Fig. 3, or away from the processor like the main memory. In traditional bus structures, bus arbitration and memory handshaking that are used for DRAM and slow communications are bypassed by the bus structure for adjacent memory and IO devices. This bypassing is usually done by the external bus structure using proper address decoding logic structures.

In our design of AFTAB, as shown in Fig. 3, the gateways sit between the processor's core logic and the external system bus structure. The gateways intercept memory access signals and the address bus. If the request is for DRAM and shared devices on the system bus, the signals will be passed on to the system bus and the bus handles arbitration and any necessary handshaking. On the other hand, if the gateways decide that the request is for the processor's local SRAM for stack handling or adjacent memory accessing, the system bus will not be involved and DARU and DAWU directly handle the memory of IO accessing.

As shown in Fig. 4, the gateways are equipped with switches on *Read* and *Write* control signals to provide a local path. The local path connects the processor and the nearby devices. The decision to switch to the main shared bus or the local bus is made in DARU and DAWU using the address decoding unit and a configuration table. In this configuration, access to the local devices takes fewer clock cycles and because it is concentrated in gateways, there is a better control, thus more security in what the processor accesses.

D. Gateway Configurable Architecture

The structures of the gateways are shown in Fig. 4. Registers, a configuration table, address decoding, switches, and an FSM, are the most important units of DARU and DAWU.

The registers temporarily store the transferred data for the read and write operations. A counter also keeps track of the number of transferred bytes. The configuration table contains the base addresses of the memory-mapped segments. When the processor is starting up, the addresses are loaded into this configuration table.

Address decoding unit receives the AFTAB core's *Read/Write* control signals and address, and distributes them to the corresponding target device. Address decoding unit uses the configuration table to see where the coming address refers. If the incoming address refers to a location of the main memory, the switches will be connected to the right-hand side *Read/Write*. However, if the address refers to a location of adjacent memory, then the switches will be connected to the *Read/Write* signals on the lower side of the gateway switches in Fig. 4.

The FSMs of DARU and DAWU start data transferring when receiving *Read/Write* and end it after receiving *MemReady*. The FSMs also control the flow of data.

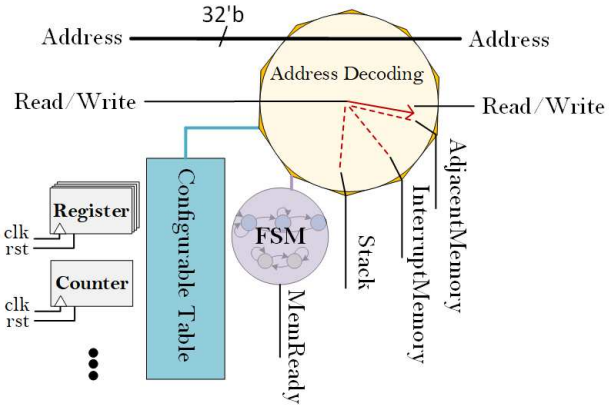

Fig. 4. Gateways Structure

IV. RISC-V Security Concerns

The issue of data security and protection has recently intercepted a primary interest at all levels of digital application design. The hardware level is not excluded, since it is now widely accepted that it is subject to numerous vulnerabilities and specific attacks [18]. Moreover, being at the base of a computing system, its compromise can mean the compromise of the overlying services, even if they have adequate software protections [19]. For this reason, the world of research and industry is moving more and more in the direction of finding the most suitable technologies to create hardware components that meet *security-by-design* requirements.

In particular, in the context of processor architecture design, the challenge is to be able to make them resilient to families of very powerful binary attacks, which inject malicious executions directly at the machine-code level. This is the case of Code Injection and Code Reuse Attacks [20-22], including the famous Return-Oriented Programming (ROP) paradigm [15]. Thanks to these exploits, it is possible to make a victim processor execute an arbitrary sequence of instructions to perform any kind of malicious action.

These types of attacks are made possible starting from source code vulnerabilities allowing memory corruption [23], and in particular, code pointers corruption, such as the return addresses of functions that are redirected to other areas at the attacker's will. Although the attack domain is software, applying defenses at this level may not be enough. For example, the randomization of code addresses to make it more difficult to locate targets [24] is too weak and can be easily brute-forced [25], or code static analysis to remove vulnerabilities [26] is based on the recognition of known patterns, and hardly gets full coverage.

A promising direction is about blocking these types of attacks at the hardware level. Since these are based on the knowledge of parameters directly managed by the hardware, such as machine instructions and memory addresses, the defenses can also be architected by abstracting from the details on higher-level services. Furthermore, users are not required to adopt any particular strategy for the design and use of software, since they rely on hardware architectures already protected by design. The issue of the hard portability and applicability of these solutions in legacy systems can be solved by the current trend, which sees an increment in the use of soft architectures such as RISC-V, synthesizable on hardware-reconfigurable devices.

The literature presents numerous ideas on hardware-based solutions, ranging from the simplest replication of sensitive

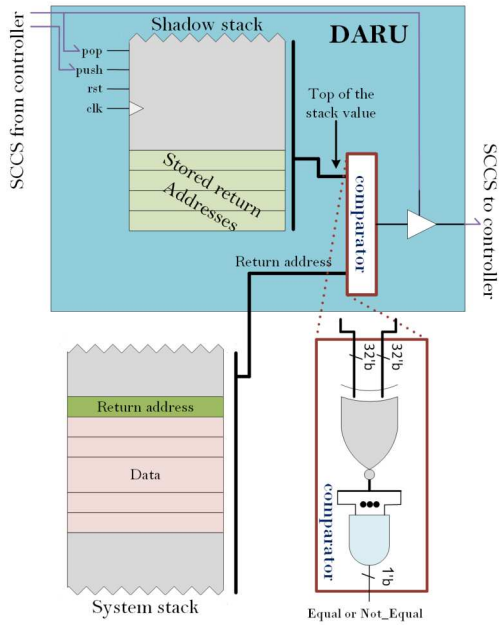

Fig. 5. DARU shadow stack architecture

code pointers [27-29] to their encryption in memory [30], [31] or tagging [32], [33]. The ideal solution can combine several defense principles, and must present some fundamental requirements, including:

- complete transparency with respect to the user;
- a low impact on additional consumption in terms of area and power;
- a negligible increase in application execution times;
- the inviolability of its control structures and of any sensitive data stored to apply the protection;

V. Proposed Security Remedy

As discussed in the previous sections, DARU and DAWU control the AFTAB's incoming and outgoing links. One of the primary purposes of separating these two parts is to have configurable gateways, which helps us easily add our configurable hardware security modules and protect the processor against various attacks such as memory corruption.

For many instructions, some handshaking occurs between the controller and DURU or DAWU to ensure the instructions are not malicious. Many attacks can be detected in the execution stage by using the lightweight hardware security modules that are in DARU and DAWU. AFTAB's controller is directly connected to these two modules in a 2-way handshaking by security check control signals (SCCSs). These signals are issued based on the type of instructions in two steps, first DARU and DAWU record some parameters according to the instruction being executed. Next, they send some security feedback to the processor controller to ensure the executing instruction is not malicious. In the following, we will show a case to detect the buffer-overflow attack in Return-Oriented Programming (ROP) or any memory corruption for an interrupt return address that is a major security concern.

Fig. 5 shows the security module addition of DARU to ensure the correct return address of ROP and interrupts. This

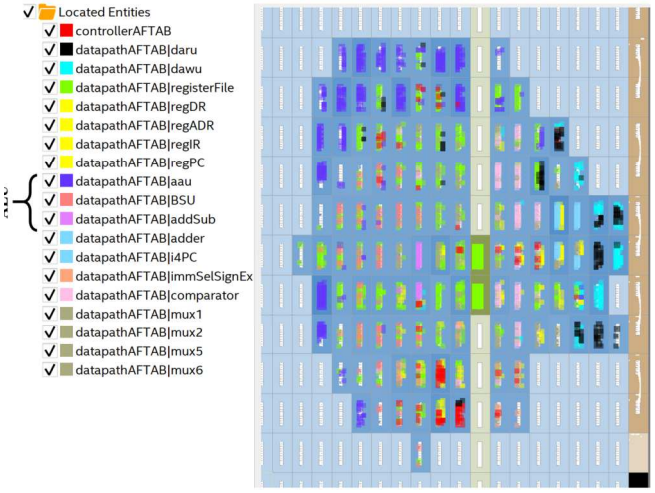

Fig. 6. Chip Planner View and Design Mapping

is achieved by a lightweight SRAM stack memory. In this scenario, the first step is to store the correct return addresses. In this architecture, when a function call or an interrupt occurs in a program, the controller issues a push to the shadow stack to place the address of the next instruction, i.e., PC+4, at the top of the stack. More return addresses are pushed on the stack for recursive function calls in the program. The stack behavior is the same for all function calls in the program, and the last PC address of the last function call is stored at the top of the stack.

In the next step, once a return instruction is executed, the controller issues a stack pop operation and contents will be compared with the address in the return instruction. The comparison is simply made with an array of XNOR followed by the AND gates shown in Fig. 5. The return instruction will be executed normally if the result of stack compare is true, otherwise, otherwise the program will exit abnormally before the execution stage gets completed.

VI. AFTAB Implementation

To assess the functionality of the AFTAB processor, several programs have been simulated on it successfully. Afterward, we have described the AFTAB architecture in VHDL and have implemented it on both Cyclone 10 GX and Cyclone IV FPGAs. The Intel Cyclone 10 GX devices with the M20K memory blocks enable utilizing 20 Kbits of embedded memory for AFTAB's register-file, instead of using logic elements [34]. Also, the memory capacity corresponding to Cyclone IV with M9K memory blocks is 9 Kbits [35]. Because of employing these memory blocks, the number of logic elements is noticeably reduced. Table I depicts hardware utilization in terms of logic elements, registers and memory bits.

TABLE I. AFTAB Resource Utilization

AFTAB Architecture	Hardware Resources		
	Total Logic Utilization	*Total Registers*	*Total Memory Bits*
Cyclone 10 GX	978	639	2048 (2 RAM Blocks)
Cyclone IV	2089	590	2048 (2 RAM Blocks)

To investigate the FPGA mapping of AFTAB, the FPGA layout has been extracted, as shown in Fig. 6. The FPGA's logic elements and the M20K memory blocks are denoted respectively with the blue and the yellow column in the background. For more clarification on the layout, each component is represented with a color corresponding to color mapping items shown on the left-hand side of the layout.

VII. Conclusions

This paper presented a secure RISC-V architecture. The design of the processor separates its main computation engine from its memory accessing parts that we refer to as gateways. We showed the general architecture of read and write gateways, i.e., DARU and DAWU. The architectures of the gateways were discussed and we showed provisions that we have put in these units for configurability purposes.

After this discussion, we showed that by only making changes to the processor's gateways hardware security features could be incorporated into the processor. The AFTAB processor designed as such was synthesized, and synthesis results showed the gateways and the real-estate that is dedicated to these units. We have shown that independent design of secure configurable memory gateways has a low hardware overhead, and at the same time adds flexibility to a processor. Independent design of our gateways means that they can be used for any processor, be it RISC-V or any other embedded processor. The gateways designed as such eliminate heavy burdens of system's required bussing.

References

[1] C. Cowan, et al. "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." In *USENIX security symposium*, vol. 98, pp. 63-78, 1998.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. "Control-flow integrity principles, implementations, and applications." *ACM Transactions on Information and System Security (TISSEC)* 13, no. 1, pp. 1-40, 2009.

[3] P. Team. "PaX address space layout randomization." *http://pax. grsecurity. net/docs/aslr. txt,* 2003.

[4] Y. Yanqiu, W. Zhenyu, and Z. Lijun. "Research on Control Flow Integrity Verification in ARM Architecture." *Computer Engineering*, pp. 151-155, 2015.

[5] R. Ramakesavan, D. Zimmerman, and P. Singaravelu. "Intel memory protection extensions (intel mpx) enabling guide.", 2015.

[6] "Intel: Control-Flow Enforcement Technology Review", 2016.

[7] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan. "Shakti-t: A risc-v processor with light weight security extensions." In *Proceedings of the Hardware and Architectural Support for Security and Privacy*, pp. 1-8, 2017.

[8] A. De, A. Basu, S. Ghosh, and T. Jaeger. "FIXER: Flow integrity extensions for embedded RISC-V." In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 348-353, 2019.

[9] M. Werner, R. Schilling, T. Unterluggauer, and S. Mangard. "Protecting risc-v processors against physical attacks." In 2019 Design, Automation & Test in Europe *Conference & Exhibition (DATE)*, pp. 1136-1141, 2019.

[10] A.S. Waterman, "Design of the RISC-V instruction set architecture.", University of California, Berkeley, 2016.

[11] E. Flamand, et al. "GAP-8: A RISC-V SoC for AI at the Edge of the IoT." *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018.

[12] S. Bailey, et al. "A mixed-signal risc-v signal analysis soc generator with a 16-nm finfet instance." *IEEE Journal of Solid-State Circuits* 54, no. 10, pp. 2786-2801, 2019.

[13] E. Torres-Sánchez, J. Alastruey-Benedé, and E. Torres-Moreno. "Developing an AI IoT application with open software on a RISC-V SoC." In *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, pp. 1-6. IEEE, 2020.

[14] I. Zagan, C.A. Tanase, and V.G. Gaitan. "Hardware Real-time Event Management with Support of RISC-V Architecture for FPGA-Based Reconfigurable Embedded Systems." *Advances in Electrical and Computer Engineering* 20, no. 1, pp. 63-70, 2020.

[15] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. "Return-oriented programming: Systems, languages, and applications." *ACM Transactions on Information and System Security (TISSEC)* 15, no. 1, pp. 1-34, 2012.

[16] D. Kanter, "RISC-V offers simple, modular ISA." *Microprocessor Report*, 2016.

[17] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanovi´c, RISC-V Foundation, December 2019.

[18] P. Prinetto and G. Roascio. "Hardware Security, Vulnerabilities, and Attacks: A Comprehensive Taxonomy." In *ITASEC*, pp. 177-189. 2020.

[19] R. Baldoni, R. De Nicola, and P. Prinetto. "The Future of Cybersecurity in Italy: Strategic Focus Areas." *Consorzio Interuniversitario Nazionale per l'Informatica - CINI*, 2018. ISBN: 9788894137330.

[20] A. One. "Smashing the stack for fun and profit." *Phrack magazine* 7, no. 49, pp. 14-16, 1996.

[21] S. Designer. "Getting around non-executable stack (and fix)." Available: https://seclists.org/bugtraq/1997/Aug/63, 1997, [Online; accessed 07-June-2021].

[22] H. Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." In *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 552-561, 2007.

[23] C.W. Enumeration. "CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer.", Available: https://cwe.mitre.org/data/definitions/119.html, 2021. [Online; accessed 07-June-2021].

[24] S. Bhatkar, D.C. DuVarney, and R. Sekar. "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits." In *USENIX Security symposium*, vol. 12, no. 2, pp. 291-301, 2003.

[25] G.F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. "Surgically returning to randomized lib (c)." In *2009 Annual Computer Security Applications Conference*, pp. 60-69, 2009.

[26] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl. "From hack to elaborate technique—a survey on binary rewriting." *ACM Computing Surveys (CSUR)* 52, no. 3, pp. 1-37, 2019.

[27] H. Ozdoganoglu, T.N. Vijaykumar, C.E. Brodley, B.A. Kuperman, and A. Jalote. "SmashGuard: A hardware solution to prevent security attacks on the function return address." *IEEE Transactions on Computers* 55, no. 10, pp. 1271-1285, 2006.

[28] A. Francillon, D. Perito, and C. Castelluccia. "Defending embedded systems against control flow attacks." In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pp. 19-26, 2009.

[29] C. Bresch, et al. "Stack redundancy to thwart return oriented programming in embedded systems." *IEEE Embedded Systems Letters* 10, no. 3, pp. 87-90, 2018.

[30] Q. Pengfei, Y. Lyu, J. Zhang, D. Wang, and G.Qu. "Control flow integrity based on lightweight encryption architecture." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, no. 7, pp. 1358-1369, 2017.

[31] Y. Li, Z. Dai, and J. Li. "A control flow integrity checking technique based on hardware support." In *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pp. 2617-2621, 2018.

[32] V. Kuznetzov, et al. "Code-pointer integrity." In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pp. 81-116, 2018.

[33] N. Roessler and A. DeHon. "Protecting the stack with metadata policies and tagged hardware." In *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 478-495, 2018.

[34] Intel® Cyclone® 10 GX Core Fabric and General Purpose I/Os Handbook. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-10/c10gx-51003.pdf, 2021.

[35] Cyclone IV Device Handbook, Volume 1, Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf