ARL-TR-9217 ● JUNE 2021

DEVCOM ARMY RESEARCH LABORATORY

Aberdeen Architecture: High-Assurance Hardware State Machine Microprocessor Concept

by Patrick Jungwirth

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-9217 ● JUNE 2021

Aberdeen Architecture: High-Assurance Hardware State Machine Microprocessor Concept

Patrick Jungwirth
Computational and Information Sciences Directorate,
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
June 2021	Technical Report	November 2019 – February 2021

4. TITLE AND SUBTITLE
Aberdeen Architecture: High-Assurance Hardware State Machine Microprocessor Concept

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Patrick Jungwirth

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DEVCOM Army Research Laboratory
ATTN: FCDD-RLC-CA
Aberdeen Proving Ground, MD 21005

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-9217

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
In a traditional computer, an operating system manages computer system resources. Current microprocessors execute or run instructions without any verification or authentication. There is no difference between safe instructions, coding errors, and malicious instructions. Complete mediation is a computer security principle meaning to verify access rights and authority for every operation. The Aberdeen Architecture achieves complete mediation for instruction execution. The Aberdeen Architecture is also designed to block information leakage. It uses hardware-level state machine monitors for the trusted computing base. The state machine monitors provide security policies enforcing multiple information flow properties. The Aberdeen Architecture combines several protection methods to create a system security policy where the whole is greater than the individual security policies. The multiple security policies provide overlapping coverage, preventing brittleness and single-point security policy failures.

15. SUBJECT TERMS
computer architecture, high-assurance computing, state machine security policies, hardware kernel, RISC-V, Redstone Architecture

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Patrick Jungwirth
Unclassified	Unclassified	Unclassified	UU	123	19b. TELEPHONE NUMBER (Include area code) (410) 278-6174

Contents

List of Figures

List of Tables

1. Introduction

In a traditional computer, an operating system manages computer system resources. Current microprocessors execute or run instructions without any verification or authentication. There is no difference between safe instructions, coding errors, and malicious instructions. Complete mediation is a computer security principle from Saltzer and Schroeder (1975) meaning to verify access rights and authority for every operation. For the Aberdeen Architecture, we are proposing to move complete mediation down to the instruction level and check and verify permissions for each executing instruction—in other words, an "operating system" or hardware monitor to manage the instruction execution pipeline (executing microprocessor instructions).

Multiple state machine monitors implement hardware-level security policies. For example, the instruction execution state machine monitor provides complete mediation for instruction execution. The state machine monitors are at security level `0` and are completely isolated from the execution pipeline located at security level `1` (one level above the state machine monitors). More details on the state machine architectures will be provided in later sections.

The Aberdeen Architecture is specifically designed to prevent information leakage from shared computer resources. Shared hardware computer resources can be modulated (or manipulated) to leak information. In 1975, Saltzer found that covert channels in Multics operating systems can leak information (Lipner 1975). Bernstein (2005), Acıiçmez et al. (2007), Jiang and Fei (2017), Karimi et al. (2018), and Jungwirth and Hahs (2019) all illustrate how shared resources leak information. Complete time and space isolation is essential to prevent information leakage.

> Saltzer [11] has reported several attempts to build and measure covert channels on Multics [6]. These attempts involved processes "banging on the walls" of the confined environment via a combination of timing and paging rate. A channel of the order of a bit per second has been demonstrated, and channels of the order of tens of bits per second hypothesized. (Lipner 1975)

1.1 Some Current Architecture Challenges

In the early 1970s, Feustel considered the merits of new architectures not based on the von Neumann machine (1973). Today, tagged architectures have a renewed interest for computer security applications.

> In the von Neumann machine, program and data are equivalent in the sense that data which the program operates on may be the program itself. The loop which

> modifies its own addresses or changes its own instructions is an example of this. While this practice may be permissible in a minicomputer with a single user, **it constitutes gross negligence** in the case of multi-user machine where sharing of code and/or data is to be encouraged. (Feustel 1973)

Future architectures will be fundamentally different from the current von Neumann architecture (also called the Princeton architecture) (Nair 2015). Future architectures will need to protect the entire computer system (Jungwirth et al. 2019a). Current and future design engineers need to revisit security principles pioneered in the 1970s.

> This paper provides a historical perspective on the evolution of memory architecture, and suggests that the requirements of new problems and new applications are likely to fundamentally change processor and system architecture away from the currently established von Neumann model. (Nair 2015)

Sharing computer resources at the same time leads to information leakage (Lipner 1975; Jungwirth et al. 2019a). In addition, it correlates processes and hardware behavior. Correlated information flows leak information. There are several information flows in a microprocessor: instruction flow, memory access [flow], control flow, and data flow. With the exception of not-so-interesting programs, instruction flow, memory access, and control flow are all *data flow dependent*. A branch predictor is a hardware resource used to predict the destination address for branch instructions. A rogue process can modulate a branch predictor to leak information. To prevent information leakage, branch predictor coefficients should be *unique* for each process.

Speculative execution pipelines need to have a process ID tag field to enforce least privilege and privilege separation. Current cache banks have multiple cache lines that are used by multiple concurrently running processes. Current cache banks also allow for an attack process to flush a cache line used by another process. Current cache banks violate complete mediation, separation, and least privilege.

High-performance timers are shared across multiple processes. Many attacks utilize timing to steal sensitive information. Virtual timers have been proposed to decouple timing. Each process uses a virtual timer that only runs while the process is executing. Current microprocessor architectures are not taking advantage of control flow integrity. Control flow integrity ensures that an executing program is following an allowed path through its control flow diagram (Burow et al. 2017). Security and integrity ensure that data processing does not violate a security policy or data integrity policy (Denning 1976). The Aberdeen Architecture applies control flow integrity concepts to create a data flow integrity policy.

1.2 Aberdeen Architecture Hardware Security Policy Summary

Aberdeen Architecture uses state machines to enforce security policies. There are four main hardware-enforced security policies: (1) instruction execution, (2) page memory access, (3) control flow integrity, and (4) data flow integrity. State machines implement the security policies and are completely isolated from the execution pipeline. The Aberdeen Architecture is based on the research work found in Lipner (1975), Jungwirth et al. (2017, 2018a, 2018b, 2019b, 2020), Jungwirth and Ross (2019), Jungwirth (2020a), and Jungwirth and La Fratta (2015, 2016, 2017).

Current tagged computer architectures use a bit array to enforce security properties. This leads to a one-size-fits-all tagged architecture for computer security policies. The Aberdeen Architecture uses a two-level tagged architecture to simplify the security policy. The "global" policy establishes the sandbox (or fence) limit for each process. The "local" policy sets stricter limits for individual instructions. Exceeding the global bounds could potentially interfere with another process. Exceeding a local bound would normally only affect the process itself.

1.3 Microprocessor Architecture Information Flows

Information flow has focused on data and data processing steps. A wider view of information flows are needed to better secure microprocessor architectures. Table 1 lists four microprocessor information flows used to execute instructions. Future architectures need to take into account the data flow dependencies for information flows. The Aberdeen Architecture uses the information flows in Table 1 to create a high-assurance architecture. Space and time isolation is essential to prevent correlations (correlated behavior(s) cause information leakage) across multiple concurrently running processes.

Table 1 Aberdeen architecture information flows

Information Flow	Dependencies
Program instruction flow (integrity)	Data flow dependent
Control flow integrity	Data flow dependent
Memory access [flow] integrity	Data flow dependent
Data flow integrity	Data flow dependent

2. Historical Review

During a 2018 interview discussing Spectre and Meltdown attacks, distinguished cybersecurity researcher Bruce Schneier described the current state of the cybersecurity world (2018): "The security of pretty much every computer on the planet has just gotten a lot worse, and the only real solution – which, of course, is not a solution – is to throw them all away and buy new ones that may be available in a few years."

New software attacks by Spectre, Meltdown, and others (see Kovacs [2018] and Zurkus [2018]) exploit hardware design flaws and oversights. Now that the door has been opened, more software attacks are being launched against naïve hardware. There is hope for new, more secure architectures. The Defense Advanced Research Projects Agency's (DARPA's) System Security Integration Through Hardware and Firmware (SSITH) program goals are to develop new microprocessor architectures to counter software attacks against current inadequate security measures (DARPA 2017; Keller 2017; Hruska 2017; Blinde 2018; Rebello n.d.). In a 2017 DARPA article, SSITH program manager Linton Salmon stated, "This race against ever more clever cyber-intruders is never going to end if we keep designing our systems around gullible hardware that can be fooled in countless ways by software" (2017b).

2.1 Future Architectures

The only real long-term solution is to throw all of the current designs away and start over. We are living in a world where many, if not all, of the secure computing concepts were researched decades ago. Major operating system concepts were pioneered in Multics (Adleman et al. 1976; Karger and Schnell 2002) during the 1970s. Hardware-based operating systems were being considered in the 1970s (Sockut 1975; Brown et al. 1977; Higher Order Software, Inc. 1977). The i432 (Witten et al. 1983) microprocessor pioneered protected objects in the 1980s.

The foundational computer security philosophy is encapsulated in Saltzer and Schroeder's 1975 security principles (Saltzer and Schroeder 1975; Smith 2012). Security-tagged architectures have their roots in the late 1950's mainframe computers (Rice University 1962; Burroughs 1969; AEG Telefunken 1970; Feustel 1972, 1973). Up to the 1970s, tag bits were being used for reliability (Rice University 1962). In 1989, Bondi and Branstad researched security tag bits and flow control integrity (1989). There has been a renewed interest in applying tag bits for computer security (Alves-Foss et al. 2014). It is time to go back to the drawing board and develop new architectures based on sound security principles from the

ground up. Current and future engineers need an all-encompassing design philosophy: "Future Cyber Defenses Must Protect the Entire Architecture" (Jungwirth et al. 2019a).

There are three system parameters in computer architectures: cost, performance, and security. Cost is a function of performance and security. Performance is a function of cost and security. Current commodity computing has settled on high performance and low cost, while security is left out of the equation. With the end of Moore's law on the horizon, and processor speeds stuck in the low gigahertz range, parallelism is the new path forward for higher performance. Parallelism also offers a path forward for high-performance and high-assurance computing.

2.2 Principles and Challenges for Future Architectures

In current computer architecture terms, "bare metal" refers to the execution pipeline. Security policies need to have complete control over the execution pipeline and be implemented at a layer below the execution pipeline. The design goal is to make the execution pipeline security conscious. De Clercq and Verbauwhede (2017) emphasize the importance of placing the trusted computing base in hardware:

> The Trusted Computing Base (TCB) is the set of hardware and software components which are critical to the security of the system. … a TCB should be as small as possible in order to guarantee its correctness … To enforce a strong security policy, we recommend that the TCB consists of as little as possible software, while placing as much as possible security-critical functionality in hardware.

The proposed Aberdeen Architecture uses state machines to implement security policies in hardware below the execution pipeline. State machine security policies are isolated and not accessible from software. Aberdeen Architecture's state machines are the trusted computing base. Current execution pipelines run instructions without any authentication. Current execution pipelines violate Saltzer and Schroeder's principles of complete mediation, privilege separation, and least privilege principles. In other words, a confused deputy would provide better security (Hardy 1988; "Confused deputy problem" July 17, 2020).

Following a philosophy of the exokernel (Engler et al. 1995), the Aberdeen Architecture enforces basic security policies. Higher-level layers can implement specific algorithms, objects (library OS functions in [Engler et al. 1995]), hypervisors, unikernels, and so on.

> In separating protection from management, an exokernel performs three important tasks: (1) tracking ownership of resources, (2) ensuring protection by guarding all

> resource usage or binding points, and (3) revoking access to resources. (Engler et al. 1995)

Program shepherding (Kiriansky et al. 2002) is a software sandbox technique to enforce execution and control flow properties. Program shepherding implements a software-based control flow integrity technique. For the Aberdeen Architecture, a hardware state machine uses security tag bits to enforce the control flow integrity policy. The Aberdeen Architecture also adds a state machine to enforce the data flow integrity policy.

> Program shepherding prevents execution of data or modified code and ensures that libraries are entered only through exported entry points. Instead of focusing on preventing memory corruption, we prevent the final step of an attack, the transfer of control to malevolent code. This allows thwarting a broad range of security exploits with a simple central system that can itself be easily made secure. (Kiriansky et al. 2002)

The Aberdeen Architecture includes several state machines: instruction execution (integrity) monitor, memory page monitor, control flow monitor, data flow monitor, exception monitor, scheduler monitor, and interrupt monitor. The proposed state machines implement a foundational-level hardware security policy following Saltzer and Schroeder's security principles. Architecture objects are categorized by allowed operations. Aberdeen Architecture uses the Redstone Architecture's (OS Friendly Microprocessor Architecture's) security features (Jungwirth and LaFratta 2015) for the 0.1 and 0.2 security architecture layers.

One of Saltzer and Schroeder's guiding principles is "open design"; a secure system should only depend on a secret key, not on a confidential design (1975):

> Open design: The design should not be secret. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords.

In 1883, Kerchoffs wrote that the design of a cryptography system falling into an adversary's hands should not compromise messages (1883). Mann (2002) extends Kerckhoffs's principle to modern systems. Today, algorithms, like advanced encryption standard (AES), are widely published. The security of AES does not depend on a confidential algorithm—it depends on a secret key.

> Kerckhoffs's principle applies beyond codes and ciphers to security systems in general; every secret creates a potential failure point. Secrecy, in other words, is a prime cause of brittleness – and therefore something likely to make a system prone to catastrophic collapse. Conversely, openness provides ductility. (Mann 2002)

For example, two papers published in 1954 (Weaver and Newall 1954) and 1960 (Breen and Dahlbom 1960) contained the details for an "in-band" telephone network signaling system. In-band signaling combines data and control information on the same cable, but it provided *no authentication* for control information. In-band signaling was a "confidential" design published in two papers. In-band signaling violates several of Saltzer and Schroeder's security principles. For in-band signaling, the control signaling system was so simple, even an amateur could build one. A "blue box" ("Blue box" July 17, 2020) generated the control codes for the telephone network. Long distance phone calls were now free—until free became illegal. On display at the Computer History Museum is a blue box (Bell 1972) built by Steve Wozniak. In-band telephone network signaling suffered a catastrophic system collapse; it was replaced by out-of-band signaling. Similar to in-band signaling, the von Neumann architecture combines program instructions (control) with data. On a von Neumann machine, a simple buffer overflow can change "data" into program instructions.

The Sony BMG copy protection scandal was quickly identified as a rootkit (Russinovich 2005; Halderman and Felten 2006; "Sony BMG" [July 13, 2020]). Without an end-user agreement, the copy protection rootkit installed itself and left backdoors for hackers. Halderman and Felten of Princeton University labeled the copy protection as spyware (2006). The record label faced huge company image damages and class action lawsuits. The rootkit copy protection flagrantly violates Saltzer and Schroeder's "psychological acceptability principle": ease of use, simple to understand, and agree to use. No user would voluntarily install a copy protection scheme that phones usage information to a record label and installs additional vulnerabilities. It was a poor IP protection implementation that ruined a product.

Software exists at the execution pipeline level. Cyber protection software and malicious applications reside at the same security layer. Protection software does not have any advantage over a malicious application. The new class of attacks represented by Spectre (Kocher et al. 2018), Meltdown (Lipp et al. 2018), and related attacks (Kovacs 2018; Zurkus 2018) prove that malicious software and zero-day vulnerabilities have an advantage. The copy protection scandal is a case study for why software cannot protect software in a computer system. Future protection mechanisms (trusted computing base) must be rooted in hardware, completely isolated from software.

Another area of concern is information leakage and shared resources. As early as 1975, Saltzer and Lipner (Lipner 1975) pointed out that covert channels in Multics leak information. Many of these covert channels are from shared system resources. Bernstein (2005) illustrates how AES software's key bit-dependent execution times leak key bit information. A malicious program only needs to passively sense

execution timing to steal information. In 2007, Acıiçmez et al. demonstrated how a malicious program can modulate a hardware branch predictor (shared hardware resource) to leak information (2007). For a simple information leakage process, Jungwirth and Hahs (2019) illustrate how transfer entropy can be used to quantify information leakage.

The Spectre attack exploits information leakage during speculative execution. There are no security mechanisms to prevent information leakage. The Meltdown attack also takes advantage of the lack of protection to leak sensitive information. Timing attacks take advantage of high-precision timers to monitor execution times. Even if an architecture had no timers, the adversary could still use known execution times for the attack software to steal information leaking from another process. Timing is essential for GPS and process control. It is more than likely not practical to build a system without timers. Cache banks are another shared resource offering adversaries ample attack vectors (Jiang and Fei 2017; Karimi et al. 2018).

3. Motivation for Aberdeen Architecture

To truly implement Saltzer and Schroeder's security principles (Saltzer and Schroeder 1975; Smith 2012), *the security policy must be enforced from the hardware system architecture's lowest level*. Instruction execution must be at least one level above the security policy level (e.g., software cannot override any aspect of the security policy). In the TIARA architecture, Shrobe et al. (2009) points out the potential for security tag bits for high-assurance computer architectures: "Metadata-driven hardware interlocks make it practical to take the security principles of Saltzer and Schroeder seriously." Current computer architectures have not extended Saltzer and Schroeder's security principles down to instruction execution level, or below the instruction execution pipeline. The Aberdeen Architecture extends Saltzer and Schroeder's concepts to below the execution pipeline level.

The 1940 bridge over the Tacoma River opened on July 1, 1940, only to collapse on November 7, 1940. The bridge incorporated new design features that had unknown failure modes. The bridge collapsed during a helical oscillation caused by a 40 mi/h wind (64 km/h).

> The Tacoma Narrows bridge failure has given us invaluable information … It has shown every new structure [that] projects [or enters] into new fields of magnitude involves new problems for the solution of which neither theory nor practical experience furnish an adequate guide. (Ammann et al. 1941)

Speculative execution, cache bank operations, and branch predictors share hardware resources across multiple processes providing attack vectors. Spectre and Meltdown attacks were discovered in 2018. Both attacks were from unknown design oversights. Current and 20-plus-year-old microprocessors are vulnerable to Spectre and Meltdown attacks. For the Aberdeen Architecture, we want to use a simple design, avoid shared resources, and provide additional protection features. We do not want to find an enormous design flaw 20-plus years later.

3.1 Review of Saltzer and Schroeder's Security Principles

In this section, we review Saltzer and Schroeder's eight security principles and apply them to the Aberdeen Architecture. The Aberdeen Architecture's security policy provides for complete verification or complete mediation (verify authority and permissions) (Smith 2012) for instruction execution. The state machines' security policies provide for complete virtualization of the microprocessor execution pipeline. A virtual machine (Popek and Goldberg 1974) executes software using a "software based" virtual microprocessor. Virtual machines provide a property called introspection (Garfinkel and Rosenblum 2003). Introspection allows for verification and forensic analysis of software running on a virtual machine. The Aberdeen Architecture provides complete hardware support for full hardware-based virtualization. Current architectures mix hardware hypervisor features with the microprocessor architecture. An improved "hypervisor" that exists below the execution pipeline and is completely isolated from the execution pipeline is needed.

3.1.1 Economy of Mechanism

High-assurance operating systems can cost $10,000 per line of code (NICTA & UNSW 2009). The seL4 operating system used a computer-generated proof-of-correctness to significantly reduce the cost of verification and validation to around $1,000 per line of code. For the Aberdeen Architecture, we want a simple design that makes demonstrating high assurance significantly less expensive. Instead of a large monolithic operating system, we are proposing several small state machines to implement the security policy. The design goal is to reduce high test and evaluation costs for high-assurance certification. Eight small state machines are much less complex than a traditional operating system. Aberdeen Architecture provides another system architecture benefit: the state machine–based security policies will reduce the complexity of high-assurance microkernels. For example, we envision a seL4-like microkernel taking advantage of hardware security policies to reduce the number of lines of assembly and C codes. We would also like to be able to test the state machines in parallel, reducing the time required for verification.

The state machines are small compared to a microkernel. Small code sizes greatly simplify formal verification.

3.1.2 Fail-Safe Default

For fail safe default cases, Smith stated, “In computing systems, the safe default is generally ‘no access’; the system must specifically grant access to resources” (2012). Aberdeen Architecture uses “no access” by default. Each application is only allowed to access resources granted by the state machines’ security policies. Access for required library functions is registered during installation. No other library functions may be called. A von Neumann machine mixes instructions and data. A no-execute tag helps limit changing data into instructions. A better solution is found in a Harvard architecture, which completely isolates instructions and data. Complete isolation provides a much higher assurance for “fail-safe default” than a no-execute tag.

3.1.3 Complete Mediation

Complete mediation is defined as, “Access rights are completely validated every time an access occurs.” (Smith 2012). The Aberdeen Architecture provides complete mediation for instruction execution, memory page accesses, stack operations, control flow, and data flow. Aberdeen Architecture’s security policies are enforced by state machines and security tags. The architecture verifies each instruction execution, data processing operation, and memory page operation.

3.1.4 Open Design

Open design principle enforces high assurance by design: “The design should not be secret” (Saltzer and Schroeder 1975). From a test engineer’s point of view, how do you verify a system requirement if you do not know what the design is? Can a confidential design containing proprietary components ever be verified? “Security through obscurity” is the commonly used phrase for a confidential design. In-band telephone network signaling and rootkit copy protection illustrate flawed “confidential” designs with catastrophic endings. To truly evaluate a computer system, the test engineer needs *full system knowledge,* including all hardware and software. Otherwise, security is left to the hidden flaw. The algorithm for AES is a published open source document and it has been verified by the open source community. Spectre, Meltdown, and related attacks take advantage of hardware design oversights.

3.1.5 Separation of Privilege

A security system with several overlapping sensors is more effective than a system with a single point failure. Multiple security systems are preferred to a monolithic implementation. Dual keys on a safe deposit box and two-factor authentication are examples of good systems with separation of privilege. The Aberdeen Architecture uses multiple state machines instead of a single protection mechanism.

3.1.6 Least Privilege

The classic flawed example of privilege concentration (all rights in one place at the same time) is the monolithic kernel. All kernel routines have all rights to the system. Subverting one kernel routine gives the attacker all system access rights. This is often summarized as the "got root?" bumper sticker. Least privilege limits kernel routines, and application software to the minimum privileges required and no more privileges.

3.1.7 Least Common Mechanism

All shared resources result in covert channels leaking information. As early as 1975, Saltzer and Lipner (Lipner 1975) pointed out that covert channels in Multics leak information. In 2005, Bernstein (2005) showed how timing information leaks AES key bit information. In 2007, Acıiçmez et al. (2007) illustrated how a branch predictor (a shared hardware resource) can be manipulated or modulated to leak key bit information. Jungwirth and Hahs (2019) published a transfer entropy model to quantify information leakage.

3.1.8 Psychological Acceptability

Smith (2012) describes user acceptability of security mechanisms: "This principle essentially requires the policy interface to reflect the user's mental model of protection, and notes that users won't use protections correctly if the mechanics don't make sense to them." For the Aberdeen Architecture, we define psychological acceptability as a protection system that emphasizes security and balances trade-offs. We require simple protection mechanisms where the software developers can easily understand the design and performance trade-offs, and can easily write secure software for the architecture. We envision a high-assurance software compiler that intelligently manages secure information flow for the Aberdeen Architecture.

3.1.9 Work Factor

Smith (2012) describes the amount of effort to successfully attack a system. Work factor is the cost in time, resources, technology, and money to attack a system. The work factor for an in-band signaling telephone system was near zero. Systems with higher levels of assurance require more work and more money to attack. High-assurance systems avoid single point failures. A high-assurance system uses effective and simple isolation techniques. For example, a von Neumann architecture combines program instructions and data. A simple buffer overflow allows for data to become a malicious program. A Harvard architecture uses two separate busses and memories for instructions and data; therefore, a Harvard architecture provides simple and effective isolation between program instructions and data.

3.1.10 Compromise Recording

Smith (2012) describes the challenge with recording a system attack: "Saltzer and Schroeder were skeptical about the benefit of such [compromise] recordings. If the system couldn't prevent an attack that modified data, then the compromise recording itself might be modified or destroyed." In other words, would you trust a calculator that you knew was broken?

3.2 Trusted Computing Base Challenge

How do we completely isolate the trusted computer base from software applications? The simple answer is, to completely isolate the security primitives from software. A significant problem with current microprocessors is that hardware resources are shared across multiple processes. A better co-design approach is required for hardware security primitives to reduce the complexity of hypervisors and operating systems. Ideally, we would like a simple set of hardware security primitives that are easy to verify and significantly reduce the complexity of critical operating system security routines.

The von Neumann machine shares program instructions and data. Both instructions and data are integers. The execution pipeline in a von Neumann machine will simply execute the integers. It has no way of knowing the difference between the labels "data" and "program instruction". This is a fundamental flaw. A Harvard architecture completely separates program instructions from data using two separate busses and memories. Harvard architectures are commonly used in digital signal processing integrated circuits. The von Neumann machine does not isolate resources. It violates several of Saltzer and Schroeder's security principles.

Saltzer and Lipner (Lipner 1975), Bernstein (2005), Acıiçmez et al. (2007), Jiang and Fei (2017), and Karimi et al. (2018), Jungwirth and Hahs (2019) all illustrate how shared resources leak information. We need complete time and space isolation. From Acıiçmez et al.'s branch predictor attack, we cannot share branch predictor coefficients across multiple processes; each process must have its own set of branch predictor coefficients. For the cache bank attacks in Jiang and Fei (2017) and Karimi et al. (2018), we cannot allow a process (attack process) to manipulate or modulate cache bank lines of another process to leak information. The Spectre attack uses speculative execution to run malicious code before the hardware realizes the wrong execution path was taken. Hardware vulnerable to Spectre attacks violates complete mediation, least privilege, separation of privilege, and least common mechanism principles. A simple solution is to tag every micro-op pipeline operation with a process ID. If the operation accesses a resource with a different process ID, then raise a hardware exception.

In summary, the Aberdeen Architecture strictly enforces space and time partitions. No hardware unit, for example, a branch predictor, contains values from multiple processes. Each process has its own set of branch prediction coefficients. The cache bank memory pipeline from the OS Friendly Microprocessor Architecture (Redstone Architecture) provides the key technology to enforce complete time and space partitioning. It is time to consider new architectures.

We have computer system-of-systems composed of interconnected layers of hardware and software. What matters for the end user is the performance of the complete system. We need to improve the system performance, not just the operating system.

We want a simple and sound security policy that is easy for the software development engineer to understand. Complexity, and especially hidden complexity (proprietary, and/or confidential design), is the antithesis of good security. If you need a doctorate to understand the implementation, it is far too complex for the real world. Right now, commodity microprocessors cannot distinguish between good software, coding errors, and malicious software. A confused deputy provides better security. This is a fundamental flaw with current microprocessors. Future execution pipelines need a hardware monitor (e.g., a nano-kernel "operating system") to enforce a security policy for executing instructions.

Current microprocessors suffer from a "performance at all costs" mentality. Advance techniques, like speculative execution, branch prediction, cache line operations, high accuracy timing, and so on, are open doors for attackers. All are resources shared across multiple processes. We need much better space and time isolation.

The Aberdeen Architecture uses state machine monitors (nano-OS) for the trusted computing base. The state machine monitors provide complete mediation for instruction execution and memory page operations. The cache bank memory pipeline architecture from the OS Friendly Microprocessor Architecture provides the required space and time isolation in hardware.

3.3 Aberdeen Architecture Design Considerations

The Aberdeen Architecture is microprocessor instruction set architecture (ISA) agnostic and could support either a complex instruction set computer (CISC) or a reduced instruction set computer (RISC). For this report, we have chosen to focus on the RISC-V ISA. RISC-V is an open ISA with strong academic research support and support from computer industry captains: nVidia, Hitachi, Cadence, and Western Digital (RISC-V 2020). The architecture design goals are to create a high-assurance architecture using hardware security primitives to reduce OS complexity. We envision a more streamlined version of seL4 to take advantage of the hardware security primitives. We want the additional overhead for security to be negligible. Clock speeds for an equivalent reference architecture without security primitives will be similar to protected architecture. We picture a computer system composed of hardware and software, where the hardware security primitives reduce OS complexity and provide better system performance than a comparable architecture running a traditional OS. We believe the system performance for *state machine security policies + execution pipeline + guest OS* will be significantly better than a traditional computer system.

In the real world, physics and geometry bound solutions. For example, $x^2 = 9$ has two potential solutions, $x = +3$ and $x = -3$. Physics and geometry place limits on which solution is allowed. If something cannot be proved consistent according to theory or logic, physics or geometry can help rule out extreme cases that are unreasonable. We do not need to know if a program will run forever. We are not concerned with whether or not a halting program will ever stop. We need to show that a program cannot ever violate the system's security policy.

3.4 Aberdeen Architecture Security Policies

The primary function of the state machine hardware monitors is to provide complete mediation of instruction execution and only allow authorized information flow. Denning (1976) defines security as no unauthorized information flow is allowed: "'Secure information flow,' or simply 'security,' means here that no unauthorized flow of information is possible." The state machine monitors are completely isolated from the instruction execution pipeline. In a traditional

microprocessor, the execution pipeline exists at the bare metal security layer. For the Aberdeen Architecture, the state machine monitors reside at security layer `0` (one level below) and the execution pipeline at security layer `1`. The state machine monitors verify the following operations for each machine code instruction in real time:

- Is this operation allowed for the instruction class?
- Is this memory operation safe?
- Is this control flow valid?
- Is data flow valid?

Valid information flow is maintained through instruction execution, memory access, control flow integrity, and data flow integrity. Data flow integrity provides complete mediation for information flow as a program is running. Complete mediation is enforced by only allowing valid information flows during program execution. In 1981, Landwehr stated a set of rules for managing information flow (1981). The model tracks information flow from input information sources through data processing to output information sinks. Landwehr's five rules for managing information flow are as follows (Landwehr 1981):

(1) A set of objects, representing information receptacles (e.g., files, program variables, bits),
(2) A set of processes, representing the active agents responsible for information flow,
(3) A set of security classes, corresponding to disjoint classes of information,
(4) An associative and commutative class combining operator that specifies the class of the information generated by any binary operation on information from two classes, and
(5) A flow relation that, for any pair of security classes, determines whether information is allowed to flow from one to the other.

The state machine monitors use security tag fields (3), control flow integrity tags (3), and data flow integrity tags (3) to manage information flow from information source (1) to information sink (5). The state machine monitors are the active agents responsible (2) for complete mediation of information flow. The security tag bits contain the set of security information classes (3). The state machine monitors ensure information flow for associative and commutative data operations (4 and 5) and input information flows, data processing, and output information flows for each machine code instruction are valid (complete mediation of instruction execution).

Instruction execution (integrity) is a "streaming information flow". Control flow integrity security tags ensure program execution is along a valid control flow path.

Data flow integrity is similar to control flow integrity. Data flow integrity provides complete mediation from the data source through data processing and to a data sink.

The goal of data flow control is to prevent information leakage. The following example assumes security tags have been altered and other security settings changed to allow installing and running malware. An accounting program has a malicious subroutine. The malicious program is given access rights to sensitive data. Data flow integrity prevents the malicious subroutine from sending sensitive information to a non-allowed information sink. The state machine monitors enforce a data flow policy that prevents data leakage as described in the confinement problem (Lipner 1975).

3.5 Attack Model

For our attack model, we assume the hardware state machine monitors are completely isolated from the execution pipeline and all running software. We assume the state machine monitors have been semi-formally proven to formally proven correct. We assume the security tag bits are not accessible by the execution pipeline and running software. We assume the Aberdeen Architecture software installation tool correctly turns a binary execution file into a security-instrumented software application for the Aberdeen Architecture.

For our attack model, we assume a rogue software binary has been parsed by the Aberdeen Architecture installation tool, and the security policy properly configured. The Aberdeen Architecture will allow rogue software to run as long as it does not violate any state machine security policies. As an example, assume a malicious program has been given another process's pointer to an I/O port. A library function call to export a memory page would raise a hardware exception. The process ID for the attack process would not match the pointer's owner. I/O pointers cannot be read or modified by application software. If the rogue software attempted to read the address of a pointer, another hardware exception would occur.

3.6 Common Weakness Enumerations for Software-Facilitated Hardware Vulnerabilities

DARPA's System Security Integrated Through Hardware (SSITH) and firmware call for proposals (DARPA 2017) used seven Common Weakness Enumerations (CWEs) (DARPA 2017, Attachment 3 CWE Glossary) as security requirements. The Aberdeen Architecture provides protections against all seven CWEs by providing complete mediation for instruction execution, control flow integrity, and data flow integrity. Data flow integrity covers data security level, data integrity, and other data flow restrictions.

State machine monitors in the Aberdeen Architecture provide complete mediation for instruction execution, control flow verification, and data flow verification. Complete mediation provides a high-assurance sandbox environment to "interpret" instructions in hardware similar to a software virtual machine. By providing complete mediation at the lowest level, instruction execution, the architecture achieves high assurance. We review the seven CWEs (DARPA 2017, Attachment 3 CWE Glossary) and briefly describe the protection measures provided by the Aberdeen Architecture for each CWE.

1) Buffer Errors

This vulnerability allows inappropriate read and/or write access to locations within memory associated with variables, data structures or internal program data. Inappropriate access to memory is exploited to subvert the normal hardware operations creating security vulnerability in the hardware. (DARPA 2017, Attachment 3 CWE Glossary)

RISC-V Instruction	Operation	
`READ: LW R1,10(R2)`	`rd = mem_read(addr = (rs1 + Offset) )` `R1 = mem_read(R2 + 10)`	{1}

Common architectures, including RISC-V, use a von Neumann architecture. A fundamental problem with current microprocessors is the lack of isolation provided by a von Neumann machine. The first line of defense against buffer overflows needs to be at the hardware level. A true Harvard architecture is required to enforce hardware isolation between program instructions and data. Software isolation provided by current isolation kernels, hypervisors, and so on, are inadequate. Spectre, Meltdown, and related attacks illustrate how unknown hardware vulnerabilities can be exploited. Full memory protection requires complete mediation of instruction execution. All operations within an instruction **{1}** need to be validated before allowing the instruction to complete. In this report, **{1}** notation refers to short blocks of computer code similar to (1) notation used for equations. For complete instruction mediation, we would need to check the items listed in Table 2.

Table 2 Complete mediation for LW R1,10(R2)

Instruction Execution	Complete Mediation Tests
`0x00ABCD04 READ: LW R1,10(R2)`	Is the address 0x00ABCD04 valid for the running process? Is the memory page valid for the running process? Is the memory page executable code? Is the memory page Process ID valid?
`0x00ABCD04 READ: LW R1,10(R2)`	Are the control flow security tags valid?
`LW`	Is LW tag set?
`R2`	Is register R2 a valid pointer?
`R2 + offset`	Is the memory page pointed to by R2 + offset valid? Is the memory address pointed to by R2 + offset valid?
`mem_addr = R2 + offset`	Do the security tags for mem_addr allow for read access by process?
`R1`	Are the data flow integrity tags valid?
`R1`	Do register tags allow write access to R1?
`If all mediation tests pass, then →`	R1 = READ(mem_addr = R2 + offset)

2) Permission, Privileges, and Access Control

> This vulnerability allows execution of unauthorized operations in a system. A privilege vulnerability can allow inappropriate access to system privileges. A permission vulnerability can allow inappropriate permission to perform functions. An access vulnerability can allow inappropriate control of the authorizing policies for the hardware. (DARPA 2017, Attachment 3 CWE Glossary)

All resources are by default not allowed. All privileged operations are handled by the hardware state machines. For privilege escalation to occur, the state machine monitors would need to be directly subverted.

3) Resource Management

> This vulnerability allows improper use of the hardware resources that in turn allow external takeover of hardware resources. This includes improper access to hardware resources such as memory, CPU, and communication and/or preventing valid users from gaining access to these resources. (DARPA 2017, Attachment 3 CWE Glossary)

All requests for resources are managed by state machine monitors. The state machine monitors are completely isolated from the execution pipeline. Software has no access to state machine monitors. Side channels may leak information; however, the architecture isolates all resources. For example, branch predictors have a coefficient table for each process. A rogue process cannot modulate the

branch predictor to steal information from another process. Caches are isolated based on process IDs. A rogue process cannot modulate a cache bank or line to leak (steal) information from another process.

4) Code Injection

> This vulnerability allows inappropriate code to be injected into the hardware due to an inherent vulnerability in the hardware. This vulnerability allows introduction of malicious code to change the course of execution on the hardware. (DARPA 2017, Attachment 3 CWE Glossary)

There are two possible code injection attacks: an outside attacker or an inside operative with access to the software configuration and install tools. Our attack model assumes the configuration and install process is *trusted*. There are also two possible outsider system attacks: attacker has the system in hand or the attacker is attacking remotely. We will only consider noninvasive network cyber-attacks. We do not consider invasive system attacks in our attack model.

We assume the attacker has already succeeded in loading the attack "executable" payload and the executable is running. Complete mediation of instruction execution prevents malicious code from performing any illegal operations. We illustrated complete mediation for a load instruction in Section 3.4. The malicious code payload does not conform to the installed program's control flow graph and data flow graphs. A control flow violation would occur when the malicious program diverged from the control flow graph. A data flow integrity exception would occur if security and/or integrity policies are violated. A hardware exception would occur if an illegal instruction operation were attempted.

5) Information Leakage (also known as Information Exposure)

> This vulnerability allows inappropriate access to privileged information in the hardware through intentional or unintentional information sharing. This vulnerability includes inappropriate data transfers, caching mechanisms, and error handling but are not limited to these areas. (DARPA 2017, Attachment 3 CWE Glossary)

The combination of complete mediation for instruction execution and time-space isolation prevents processes from being correlated. Shared resources are completely space and time isolated. Aberdeen Architecture provides high-assurance space and time isolation to block information leakage.

6) Crypto Errors

> This vulnerability allows inappropriate use and execution of cryptography in hardware. This vulnerability includes inappropriate access to cryptographic keys

and inappropriate use of these keys to exfiltrate information or allow inappropriate access to hardware. (DARPA 2017, Attachment 3 CWE Glossary)

The current Aberdeen Architecture is a base architecture. A cryptographic engine can be connected through a DMA channel. Cryptographic keys are stored in protected memory blocks. Software does not have direct access to crypto keys. Figure 1 illustrates a protected I/O pointer for the OSFA architecture. Crypto keys can be protected using the same protected pointer type. A protected pointer points to a crypto key. As illustrated in the OSFA tech report, software knows the crypto-pointer is linked to a crypto key. Software does not have access to read/write to the crypto-pointer or the crypto key. Software can pass a protected crypto-pointer to a hardware cryptographic unit. The cryptographic core uses the crypto-pointer to access the cryptographic key. A hardware state machine can manage cryptographic operations and has read access to the crypto key.

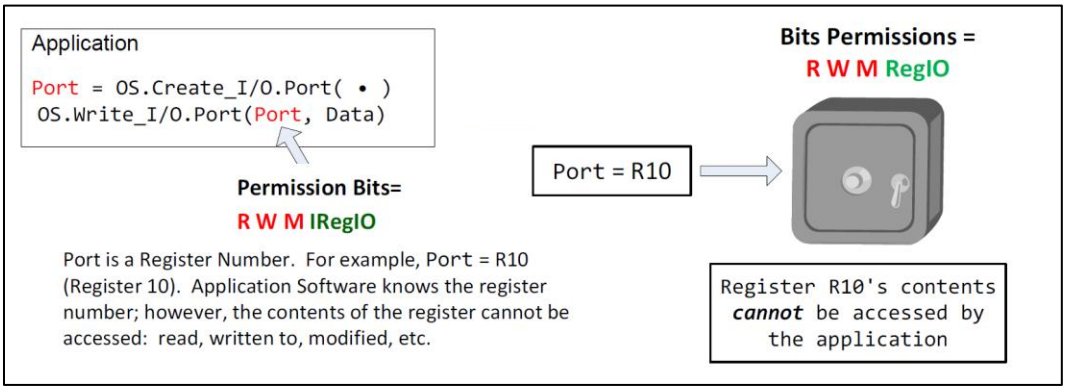


Fig. 1 OS friendly microprocessor architecture protected pointers (Jungwirth and LaFratta 2017)

7) Numeric Errors

This vulnerability allows exploitation of improper calculation or conversion of numbers and numeric types. Improper/incorrect calculations can allow subversion of security critical operational decisions and/or resource management. (DARPA 2017, Attachment 3 CWE Glossary)

Improper calculations or data conversion attacks are blocked by control flow integrity protections and data flow integrity protections. Data flow integrity in the Aberdeen Architecture provides security tags for secrecy levels, integrity, and data types. A gadget is a short block of code found in a common library function. A series of gadgets are called in an attacker predetermined order to craft an attack program. Control flow integrity provides specific entry and exit points for library functions. Code entry and exit points drastically limit opportunities for calling useful attack gadgets. The security tags also ensure that the secrecy level, integrity, and data types are compatible.

3.7 Data-Dependent Information Flows

Figure 2 illustrates simple data-dependent information flows. Program instruction flow is data dependent. The C code condition `if( arr[j] > arr[j+1])` results in data-dependent instruction execution. System state behavior must not be correlated with data-dependent program execution. The Aberdeen Architecture uses the four information flows in Table 1 to build a high-assurance architecture.

```
//  C code bubble sort routing

//  swap variables
void swap(int a, int b)  {
    int temp = a;
    a = b;
    b = a;
}

//  bubble sort function
int bubbleSort(int arr[], int n)  {
   int i, j;

   for (i = 0; i < n-1; i++)

      for (j = 0; j < n-i-1; j++)

         if (arr[j] > arr[j+1])
           swap(arr[j], arr[j+1]);

   return 0
}
```

Bubble Sort
Data Flow Dependent Program

Fig. 2 Bubble sort

3.8 Architecture Summary

The Aberdeen Architecture's goal is to provide complete mediation down to the instruction execution level (instruction execution integrity). All information flows are verified from information source to information sink. The state machine monitors are completely isolated from the execution pipeline and all running software. The state machines are the managers (controllers) for the execution pipeline. Multiple security policies are running simultaneously. In order to subvert the system, most if not all of the security monitors would need to be defeated.

As with the Tacoma Narrows Bridge, lessons learned need to be applied to new, more secure architectures. More complexity results in a higher probability of an unknown attack vector. Engineers can only make it more expensive to attempt to break in. Keep in mind, an *un-hackable* system cannot be built and it can *never* be fully tested.

4. Aberdeen Architecture

Podebrad et al. (2009) and Tiwari et al. (2011) point out that current computers are only designed for speed; security issues are completely ignored. It is time to develop new, more secure architectures. Resource sharing across concurrently running processes cannot be allowed. For example, the branch coefficients in a branch predictor are shared across multiple running processes. Allowing processes to manipulate cache lines or banks across multiple concurrent processes should not be allowed. The execution pipeline needs to support complete mediation for instruction execution.

> The analysis of currently available computer architectures has shown that such systems offer a lot of security gaps. This is due to the fact that in the past hardware has only been optimized for speed - never for security. (Podebrad et al. 2009)

> Almost every recent microarchitectural technique is built around the notion of optimizing the common case, an end achieved in large part through the addition of caches, status bits, exceptions, predictors, and other behaviors that modify the state of the machine. The problem is that, if one is protecting a secret or handling untrusted data, every operation performed on that secret will affect those internal states in one way or another. Non-interference requires that those affected internal states are then in no way visible to the other components, including either directly through the ISA, or indirectly through the resulting differences in behavior or timing. (Tiwari et al. 2011)

Figure 3 shows the classic 5-stage RISC execution pipeline. The classic RISC execution pipeline cannot distinguish between safe instructions, coding errors, and malicious instructions. Recent research demonstrates that security tag bits provide security awareness. The DARPA CRASH program, the DARPA SSITH program, and the Redstone Architecture (OS friendly microprocessor architecture) use security tag bits. The Redstone Architecture uses two levels of security tag bits. The Aberdeen Architecture uses the Redstone Architecture's execution pipeline for security level `1`. Level `0`, hardware-based state machine monitors, provides the foundational level security policy and complete mediation for instruction execution.

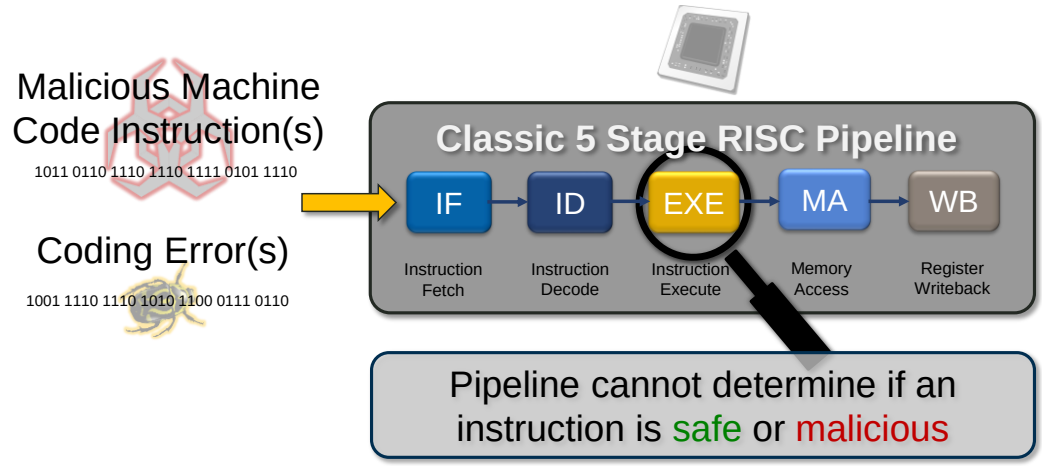


Fig. 3 Classic 5-stage RISC execution pipeline (Jungwirth 2020b)

The Aberdeen Architecture uses state machine monitors completely isolated from the execution pipeline to enforce hardware-based security policies. There are four main hardware-enforced security policies: (1) instruction execution [integrity], (2) page memory access control [memory integrity], (3) control flow integrity, and (4) data flow integrity. The Aberdeen Architecture extends the research work found in Lipner (1975), Jungwirth et al. (2017, 2018a, 2018b, 2019b, 2020), Jungwirth and Ross (2019), Jungwirth (2020a), and Jungwirth and La Fratta (2015, 2016, 2017).

The Aberdeen Architecture enforces Saltzer and Schroeder's security principles (1975) at the instruction execution level. *Shared hardware resources leak information*. Attack processes will maliciously modulate shared resources to maximize information leakage (Lipner 1975; Bernstein 2005; Acıiçmez et al. 2007; Jungwirth and Hahs 2019). The Aberdeen Architecture does not allow resource sharing across multiple, concurrent, running processes.

The Aberdeen Architecture's hardware-level security policy consists of several hardware monitors (nano-kernels). In Section 4.1, we present a historical review covering the development of hardware-based operating systems. In Sections 4.2–4.5, the Aberdeen Architecture is introduced and presented in detail.

4.1 Historical Review

In 1968, Dijkstra wrote the foundational paper for the modern operating system "The structure of the 'THE' - multiprogramming system" (1968). From the 1970s through the early 1990s, microprogramming was used to implement operating system primitives in hardware. Microprogramming was the first step toward a hardware-based operating system. In 1973, Goldberg was researching direct hardware execution for a virtual machine:

> An HVM [Hybrid Virtual Machine] is functional equivalent to a real machine. All instructions issued within the most privileged layer of the HVM are software interpreted while all non-privileged-layer instructions execute directly.

In 1975, Sockut (1975) published research on firmware/hardware support for operating systems. Brown et al. (1977) researched microprogramming to improve operating systems in 1977. Foster (1978) considered hardware enhancement for operating systems in 1978. In 1982, Kamibayashi et al. (1982) researched microcoded OS's and the "advantages of the efficiency which may be gained from microcoded operating system primitives." In the early 1990s, microprogramming was still being researched for operating systems (Papachristou and Gambhir 1991). Around that same time, Nakano et al. developed the first practical hardware-based operating system (1995, 1997, 1999). During the 2000–present time frame, hardware operating systems have reached the commercial world (Hardin 2001; Murtaza et al. 2006; Song et al. 2007; Vetromille et al. 2006; Yan et al. 2010; Oliveira et al. 2011; Ong et al. 2013; Moisuc et al. 2014; Stenquist 2014; Renesas 2021a, 2021b). In 2014, Renesas released the R-IN32M3 microcontroller with a hardware-based operating system (Renesas 2014; Renesas 2021a).

Tagged security computer architectures originated in the 1970s. In 1973, Feustel proposed using a tagged architecture to overcome limitations present in the von Neumann machine.

> Taken together, the arguments we have advanced provide a powerful incentive for further investigation and exploitation of tagged architecture. Such a machine may soon well be a replacement for today's widely accepted von Neumann architecture. (Feusel 1973)

In 1975, Saltzer and Schroeder proposed using tagged architectures for securing protected operations.

> Suppose, for example, that every location in memory were tagged with an extra bit. If the bit is **OFF**, the word in that location is an ordinary data or instruction word. If the bit is **ON**, the word is taken to contain a value suitable for loading into a protection descriptor register. … This kind of scheme is a particular example of what is called a tagged architecture. (Saltzer and Schroeder 1975)

In 1989, Bondi and Branstad researched a tagged architecture to simplify high-assurance certification:

> The architecture's hardware-enforced fine-grained mediation will …
> • permit sufficient simplification of security kernel and other associated trusted

> software to bring certification at TCSEC [Trusted Computer System Evaluation Criteria] level A1 (and beyond) within reach;
> • support highly secure data flow control. …

In the early 1980s, hardware optimizations like caches, execution pipelines, and speculative execution where not used in commercial microprocessors. In 1985, one paper (Gehringer and Keedy 1985) pointed out that tagged architectures were an 'unnecessary' complexity compared to software. Today, tagged architectures offer numerous computer security benefits for high-assurance architectures. DARPA's Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) 2011 program led to significant research into new tagged architectures for better computer security (Kenyon 2012; Smith n.d.). The website, www.crash-safe.org, hosts a number of CRASH architecture research papers (de Amorim et al. 2017; Chiricescu et al. 2013; Dhawan et al. 2017).

The DARPA System Security Integration Through Hardware and Firmware (SSITH) Program (Rebello n.d.; Chirgwin 2017; DARPA Microsystems Technology Office 2017; Hruska 2017; Keller 2017; Blinde 2018) promoted the development of new architectures to remove software initiated attacks against vulnerable hardware. The Register.com announced "DARPA seeks SSITH lords to keep hardware from the Dark Side" (Chirgwin 2017). SSITH architecture research papers, covering secure speculative execution (Jiang et al. 2018), information flow enforcement (Tarma et al. 2019), and cryptographic accelerators (Jiang et al. 2019), and defending against data oriented programming (DOP) attacks (Aga and Austin 2019) were recently published. Significant research efforts in the 2010s have led to renewed interest in tagged architectures for computer security (Aga and Austin 2019; Zeldovich et al. 2008; Shrobe et al. 2009; Shioya et al. 2009; Dhawan et al. 2012; Song and Alves-Foss 2013; and Song 2014).

Security techniques should be simple to understand and provide high assurance. Abadi et al. (2005) recommend for high-assurance systems:

> In order to be trustworthy, mitigation techniques should — given the ingenuity of would-be attackers and the wealth of current and undiscovered software vulnerabilities — be simple to comprehend and to enforce, yet provide strong guarantees against powerful adversaries. On the other hand, in order to be deployable in practice, mitigation techniques should be applicable to existing code (preferably even to legacy binaries) and incur low overhead.

The Redstone Architecture (OS Friendly Microprocessor Architecture [OSFA]) uses a tagged cache bank memory pipeline for high assurance and high performance (Jungwirth et al. 2017, 2018a, 2018b, 2019b, 2020; Jungwirth and Ross 2019; Jungwirth 2020a; Jungwirth and La Fratta, 2015, 2016, 2017). The Redstone

Architecture was patented, US9122610, in 2015. A security framework for the Redstone Architecture was patented, US10572687, in 2020. The Aberdeen Architecture uses the Redstone Architecture for security layer 1. Aberdeen Architecture is currently patent pending.

4.2 Aberdeen Architecture Philosophy and Goals

The Aberdeen Architecture manages objects (instructions, data and memory pages) by classes. The research Aberdeen Architecture uses the RISC-V instruction set. There are four instruction classes: (1) arithmetic/logic sequential, (2) load/store sequential, (3) conditional branch, and (4) jump. Each instruction class has a set of security properties enforced by state machine monitors. The instruction execution [integrity] state machine monitor provides complete mediation for instruction execution.

> High assurance systems of the future need a bottom-up, hardware and software co-design approach to security **[5]** and a hardware level root of trust **[6]-[7]**. We should also consider the high assurance levels achieved by state machines for safety critical applications where rigorous system verification is required. (Jungwirth et al. 2018b)

There are several memory page classes. Each class has a set of security properties enforced by state machine monitors. The memory page state machine monitor (MPSM) enforces security properties for several different memory page types. For example, the **`I/O_Page_Mem`** class provides for page-level input and output operations. No data operations may be performed on an **`I/O_Page_Mem`** class. This restriction enforces least privilege, privilege separation, and complete mediation for I/O. Data and math operations are restricted to "data" contained in a memory page type = **`Data_Page_Mem`** class. There are conversion instructions provided to convert memory pages between **`I/O_Page_Mem`** and **`Data_Page_Mem`** classes.

The memory state machine monitor provides complete mediation for load and store memory operations. The stack machine monitor provides complete mediation for all stack operations. The Aberdeen Architecture uses multiple stacks to provide least privilege and privilege separation for different stack types. For example, return addresses and call arguments are not placed on the same stack.

There may appear to be a large number of memory page types; however, we are balancing security policy, flexibility, OS complexity, and performance. Memory page classes provide least privilege, privilege separation, and complete mediation for each memory class. This simplifies OS complexity, provides higher performance, and provides a better security policy.

Aberdeen Architecture uses state machine monitors to enforce security polices based on Saltzer and Schroeder's security principles, and Landwehr's information flow control rules. The Aberdeen Architecture protects against all seven CWEs (DARPA 2017, Attachment 3 CWE Glossary) by providing complete mediation for instruction execution integrity, page memory access integrity, control flow integrity, and data flow integrity. Data flow integrity covers data security level, data integrity, and other data flow restrictions.

Aberdeen Architecture limits a rogue program to benign behavior. The goal of the Aberdeen Architecture is to limit a rogue program to only harm itself. As long as the architecture protects all programs, memory spaces, and so on, from a rogue program, damage is limited to only the rogue process. Complete mediation provides high assurance.

The Aberdeen Architecture provides complete mediation for instruction execution and memory operations. Control flow integrity verifies that program execution follows its control flow graph. Data flow integrity verifies information flow during program execution. The trusted computing base needs to be completely implemented in hardware and completely isolated from software. Aberdeen Architecture's security policy is enforced by several state machine monitors. The proposed state machines implement a foundational-level hardware security policy.

The Aberdeen Architecture includes several state machines: instruction execution monitor, memory page monitor, control flow monitor, data flow monitor, exception monitor, scheduler monitor, and interrupt monitor. Architecture objects are categorized by allowed operations. Aberdeen Architecture uses the Redstone Architecture's (OS Friendly Microprocessor Architecture) (Jungwirth and LaFratta 2015) pipeline and security features for the security layer 1.

> Software is mutable, generally buggier than hardware, might have coverage holes due to heterogeneity and layering, and might implement incorrect privacy notions. Hardware, however, is immutable and can sit between data sources (sensors) and data consumers (software accessing the data), guaranteeing coverage and a universal, minimum notion of privacy. (Sethumadhavan 2016)

De Clercq and Verbauwhede (2017) and Sethumadhavan (2016) recommended placing the trusted computing base in hardware. Complete mediation for instruction execution and memory operations has the potential of overcoming the limited success of previous memory protections focused on specific attack vectors (Suh et al. 2004).

> Unfortunately, it is very difficult to protect programs by stopping the first step of an attack, namely, exploiting program vulnerabilities to overwrite memory

> locations. There can be as many, if not more, types of exploits as there are program bugs. Moreover, malicious overwrites cannot be easily identified since vulnerable programs themselves perform the writes. Conventional access controls do not work in this case. As a result, protection schemes which target detection of malicious overwrites have only had limited success – they block only the specific types of exploits they are designed for. (Suh et al. 2004)

4.3 Aberdeen Architecture

We begin by describing the software design philosophy for the Aberdeen Architecture. This provides a foundation to describe protections provided by the Aberdeen Architecture. The software design philosophy describes how programs are structured to take advantage of the protection features provided by the Aberdeen Architecture. The next sections introduce the major features of the Aberdeen Architecture: ISA, memory architecture, tag protection bits, control flow integrity (CFI), and data flow integrity (DFI). Tag protection bits are the basis for instruction execution integrity, control flow integrity, and data flow integrity. For the Aberdeen Architecture, we consider the instruction execution flow as an information flow. Malicious manipulation of the instruction flow execution leaks information. We need to secure instruction execution just like securing data information flow using security and integrity parameters. For the Aberdeen Architecture, we group '**`data`**' information flow into a single class called data flow integrity. We now have four subclasses of information flow: instruction execution, page memory access, control flow integrity, and data flow integrity. High-assurance information security policies manage instruction execution, page memory access, control flow, and data flow.

In terms of a virtual machine, the security tag fields enable complete virtualization of the execution pipeline. The register security tags also virtualize the register file (registers `R0`, `R1`, `R2`, ··· `R31`). The security tag fields are "interpreted" in real time by the hardware state machine monitors. The monitors enforce the hardware-level security policies providing complete instruction execution mediation.

Aberdeen Architecture adds hardware-level nano-kernel state machine monitors to the Redstone Architecture. Figure 4 illustrates how the state machine enforces security policies for the execution pipeline. Redstone Architecture uses two-level security tag fields and cache bank memory pipeline architecture to provide high performance. The Aberdeen Architecture enforces complete mediation for instruction execution. The Aberdeen Architecture's state machines enforce high-assurance hardware-level security policies from architecture level `0` (most secure layer). Execution pipeline (security level `1`) runs software at security levels `2` and/or higher. In terms of a virtual machine, the state machine monitors provide full virtualization for the execution pipeline. The hardware security policy enforces

allowed instruction behavior for each instruction class. The Aberdeen Architecture uses security policies to enforce allowed operations for the four information flow classes. Instruction execution is considered an information flow class that is dependent on the data flow class. The Aberdeen Architecture uses multiple security policies to enforce allowed instruction execution. For example, a control flow monitor ensures that the executing program is following a valid control flow path. The next sections present a detailed description of the Aberdeen Architecture.

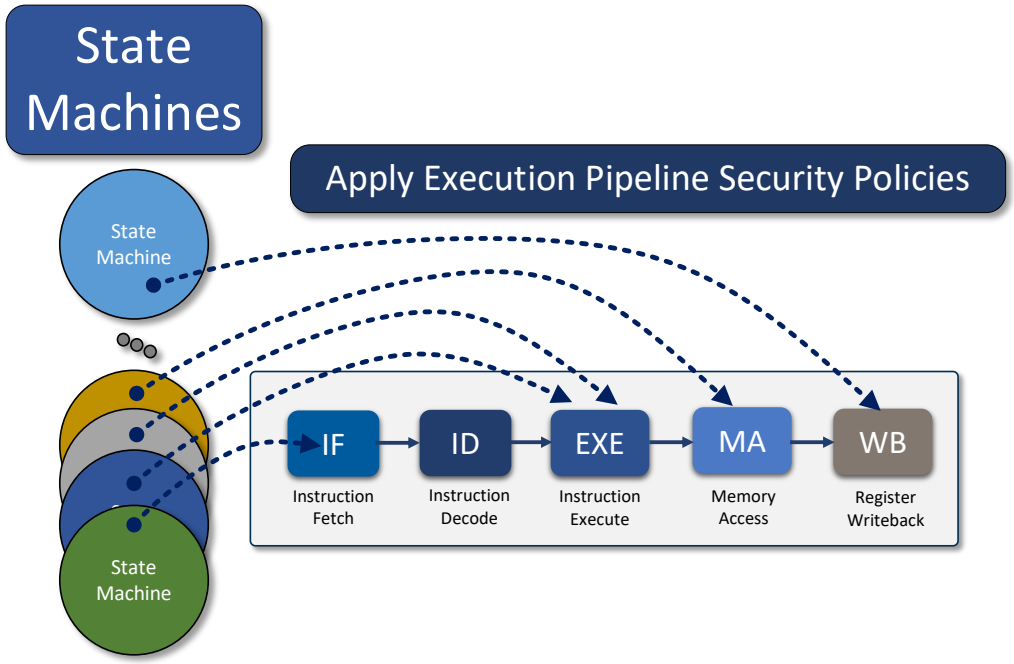


Fig. 4 State machine security policies

4.3.1 Aberdeen Architecture's Information Flow Classes

Aberdeen Architecture uses four information flow classes. Information flow has focused on data and data processing information flows. The current view of basing information flow only on data needs to be extended. A wider view of information flow is needed to better secure microprocessor architectures. Table 1 lists the four information flow classes. All information flows are data flow dependent. Data flow determines the path taken for program instruction flow (integrity), control flow graph (control flow integrity), and memory access flow (integrity). Data flow integrity covers integrity, secrecy, and other data characteristics (measurement units, and data type [record, float, integer, etc.]). Data flow dependencies need to be fully understood to create a secure architecture. As pointed out in Abadi et al. (2005), stack operations are not taken into account in a control flow graph. Stack operations can be data dependent or not. Stack memory access operations need to be secured. Control flow integrity needs to be supplemented with stack and memory protections. Data flow can drive stack and function call paths. We need to build up

a sound security policy starting from data flow, covering memory access, control flow, and program instruction flow.

4.3.2 Software Design Philosophy Introduction

A microprocessor is designed to run software. We introduce the software design philosophy for the Aberdeen Architecture to build up the architecture a step at a time. We will use the Sieve of Eratosthenes C program in Fig. 5 as a code example. Software for the Aberdeen Architecture is configured with a block structure. Single **`CALL`** entry and **`RETURN`** exit points restrict function entry and exit points (Kiriansky et al. 2002). We will build upon this code example to explain the operation of the Aberdeen Architecture.

Another process, or the function itself, is not allowed to **`CALL`** an instruction inside a function or code block. The block code structure is to prevent gadget attacks typically used against library functions (or dynamically linked library [DLL] functions). The block code structure is to enforce least privilege, privilege separation, and complete mediation principles from Saltzer and Schroeder. Tag bits are used to define the **`entry`** and **`exit`** code points. Tag bits can also be applied to exception handlers to ensure code execution follows an allowed path (control flow integrity). Figure 5 shows a C code version of the Sieve of Eratosthenes algorithm. Figure 6 illustrates the code block formed by the entry point (**`CALL`** to function) and exit point (**`RETURN`** from function). We will expand upon this code block structure for the Aberdeen Architecture.

The sieve code uses a packed bit array of consecutive 32-bit words to hold bits representing prime or not prime. In order to remove the square root function from the algorithm, we define two parameters, `R` and `LAST = R`^{2} `- 1`. This removes the square root function from the while loop: `while (base < R)`. Figure 7 shows an example list of prime numbers for `R = 37` and `LAST = 37`^{2} `– 1 = 1368`. Figure 8 shows the RISC-V assembly and machine codes for the Sieve of Eratosthenes routine. We will use the RISC-V Sieve of Eratosthenes program to describe the operation of the Aberdeen Architecture.

```
// Sieve of Eratosthenes
// uses packed bit array to store prime / not prime result
// pb[ ] = packed bit array, word aligned, e.g. addr = 0x100
// J. Ross and P. Jungwirth, Army Research Lab, October 2019
// C code is based on rosettacode.org/wiki/Sieve_of_Eratosthenes#Ada
// Algorithm see https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

#include <stdio.h>
#define R 37
#define LAST (R * R)
#define LWORD (LAST / 32)
int pb[LWORD + 1]; // = { [0 ... LWORD] = 0xffffffff } -- packed bit array

void setbits() {
   for (int i = 0; i < LWORD + 1; i++) pb[i] = 0xffffffff; }

void display() {
   int bit = 0; int word = 0; int shift = 0;
   printf("Prime Numbers are \n");
   for (int i = 1; i < LAST - 1; i++)
   {  word = i >> 5; shift = i & 0x1f;
      bit = (pb[word] >> shift) & 1;
      if (bit == 1) printf(" %d", i); }
}

void sieve() {
   int base = 2; int pbit = 0; int cnt = 0;
   int base_bit = 0; int base_word = 0; int base_shift = 0;
   int cnt_word = 0; int cnt_shift = 0; int cnt_mask = 0; int bit_mask = 0;
   while (base < R) {
      base_word = base >> 5; base_shift = base & 0x1f;
      base_bit = (pb[base_word] >> base_shift) & 1;
      if (base_bit)
      {  cnt = base << 1; // base + base;
         while (cnt < LAST)
         {  cnt_word = cnt >> 5; cnt_shift = cnt & 0x1f;
            cnt_mask = (1 << cnt_shift); bit_mask = ~cnt_mask;
            pbit = pb[cnt_word] & bit_mask;
            pb[cnt_word] = pbit;
            cnt = cnt + base; }
      }
      base = base + 1;
   }
}

int main() {
   setbits();  // set array of bits = 1's
   sieve();    // Call Sieve of Eratosthenes
   display();  // display prime number
   return 0;
}
```

Sieve of Eratosthenes Code

Fig. 5 Sieve of Eratosthenes in C code

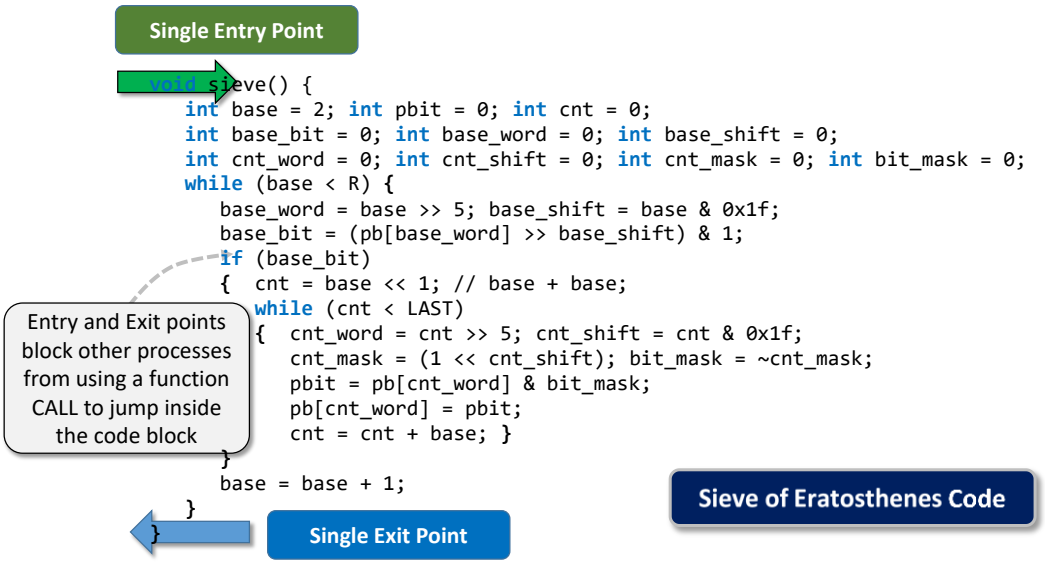


Fig. 6 Block code structure

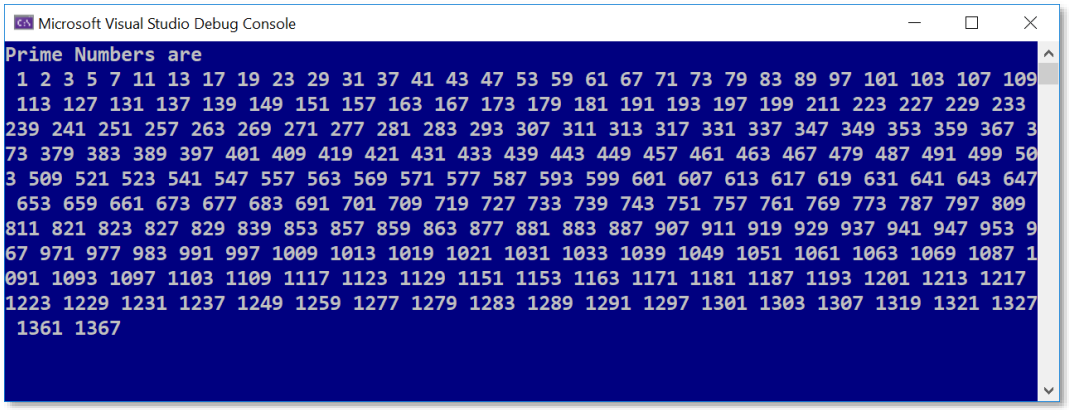


Fig. 7 Sieve of Eratosthenes results for R = 37

```
// Sieve of Eratosthenes – RISC-V Assembly Language
```

Memory Address	Machine Code	Assembly Code	Code Description
0x0000 0008: <sieve>	0x00200613	li a2, 2	# base = a2 = 2
0x0000 000c:	0x10000513	li a0, 0x100	# pb[0] = 0x100
0x0000 0010:	0x00100893	li a7, 1	# a7 = 1
0x0000 0014:	0x06300813	li a6, 0x63	# LAST = R^2 -1 = 100 - 1
0x0000 0018:	0x00a00313	li t1, 0x10	# t1 = R = 16
0x0000 001c:	0x00c0006f	j 28 <L1>	# jump to <L1>
0x0000 0020: <L3>	0x00160613	addi a2, a2, 1	# a2 = base = base +1
0x0000 0024:	0x04660a63	beq a2, t1, 78 <L2>	# if base = R then <L2> Done
0x0000 0028: <L1>	0x40565793	srai a5, a2 ,0x5	# a5 = word offset
0x0000 002c:	0x00279793	slli a5, a5, 0x2	# a5 = byte offset [note 1]
0x0000 0030:	0x00f507b3	add a5, a0, a5	# a5 = pb[0] + byte offset
0x0000 0034:	0x0007a783	lw a5, 0(a5)	# a5 = LW(addr = a5)
0x0000 0038:	0x40c7d7b3	sra a5, a5, a2	# a5 = a5 >> a2 [note 2]
0x0000 003c:	0x0017f793	andi a5, a5, 1	# a5 = pb[word, bit number]
0x0000 0040:	0xfe0780e3	beqz a5, 20 <L3>	# if a5 = bit = 0 the <L3>
0x0000 0044:	0x00161693	slli a3, a2 ,0x1	# a3 = cnt = base + base
0x0000 0048: <L4>	0x4056d793	srai a5, a3, 0x5	# a5 = word offset from a3
0x0000 004c:	0x00279793	slli a5, a5, 0x2	# a5 = byte offset
0x0000 0050:	0x00f507b3	add a5, a0, a5	# a5 = pb[0] + byte offset
0x0000 0054:	0x0007a583	lw a1, 0(a5)	# a1 = LW(addr = a5 + 0)
0x0000 0058:	0x00d89733	sll a4, a7, a3	# a4 = 1 << cnt = 0••010••00
0x0000 005c:	0xfff74713	not a4, a4	# a4 = 1•••0•••11
0x0000 0060:	0x00b77733	and a4, a4, a1	# clear bit
0x0000 0064:	0x00e7a023	sw a4, 0(a5)	# update word
0x0000 0068:	0x00c686b3	add a3, a3, a2	# cnt = cnt + base
0x0000 006c:	0xfcd85ee3	ble a3, a6, 48 <L4>	# if less then <L4>
0x0000 0070:	0x00160613	addi a2, a2, 1	# base = base + 1
0x0000 0074:	0xfa661ae3	bne a2, t1, 28 <L1>	# if base != R then <L1>
0x0000 0078: <L2>	0x00000513	li a0, 0	# clear a0
0x0000 007c: <return>		ret	

Fig. 8 RISC-V Sieve of Eratosthenes

The Aberdeen Architecture is instruction-set agnostic; however, to describe the architecture's operations, we need a specific instruction set. For this technical report, we will use the RISC-V 32-bit integer instruction set. Code fragment **{2}** shows the first line of the RISC-V code for the Sieve of Eratosthenes. We will add tag bits to the machine code instructions for control flow integrity, data flow integrity, memory access integrity, and instruction execution integrity. Memory integrity is enforced by instruction execution integrity, control flow integrity, and data flow integrity. For example, Aberdeen Architecture uses separate stacks for "data" and return addresses. Mixing stack classes violates Saltzer and Schroeder's principles of least privilege, privilege separation, and complete mediation. More details about memory architecture will be presented in Section 4.3.3.

Memory Address	Instruction Machine Code	RISC-V Assembly Language
0x0000 0008	0x0020 0613	<sieve> li a2, 2 # base = a2

{2}

4.3.3 Instruction Set Architecture (ISA)

The Aberdeen Architecture is ISA agnostic. Both complex instruction set computer and reduce instruction set computer ISAs can be used for the Aberdeen Architecture. For this technical report, we have chosen to use the RISC-V reduce instruction set computer ISA. The open source and extendable RISC-V ISA has an established user community and software development ecosystem. Current proprietary microprocessor ISAs were never designed to be extended. RISC-V promotes its open design philosophy by creating an architecture that is designed to be extended by the user community. We take advantage of this design philosophy to create the Aberdeen Architecture. In addition, the small number of instruction classes simplifies the design and simplifies establishing high assurance.

Table 3 lists the four classes of RISC-V instructions: (1) arithmetic/logic sequential, (2) load and store sequential, (3) conditional branch, and (4) jump. A sequential instruction increments the program counter register (PCR) by one instruction (4 bytes for RISC-V). RISC-V arithmetic and logic instructions are sequential. An example sequential class instruction is shown in Table 3 and Fig. 9. Load/store instructions are sequential and read or write to memory. Figure 10 illustrates load and store class instructions. The conditional branch instruction class is shown in Fig. 11. A conditional branch instruction has two possible next instructions. If the branch condition is `false`, the program counter is incremented by one instruction (4 bytes). If the branch condition is `true`, the program counter is loaded with the address for the destination instruction. Figure 12 describes the jump instruction class. The jump instruction class has only one destination address. A jump

instruction loads the program counter with the destination address contained in the jump instruction. The four instruction class definitions provide for least privilege execution for each type of instruction. The Aberdeen Architecture uses the four classes of instructions for instruction execution integrity.

Table 3 Aberdeen Architecture instruction classes

Instruction Class	RISC-V Instruction	Instruction Operation	Next Instruction
(1) Sequential	`add R1,R2,R3`	R1 = R2 + R3	PCR = PCR + 4
(2a) Load Sequential	`lw R1, 0(R2)`	R1 = read_mem(0 + R2)	PCR = PCR + 4
(2b) Store Sequential	`sw R4, 0(R5)`	write_mem(0 + R5) = R4	PCR = PCR + 4
(3) Conditional Branch	`ble R3, R6, 48`	**Branch if less than** **if**(R3 < R6) then 48 **else** next instruction	PCR = 48 (**true**) PCR = PCR + 4 (**false**)
(4) Jump	`j 28`	Jump to address = 28	PCR = 28

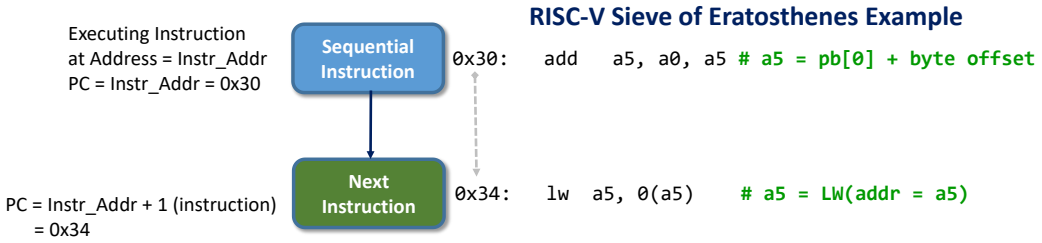


Fig. 9 Sequential instruction class

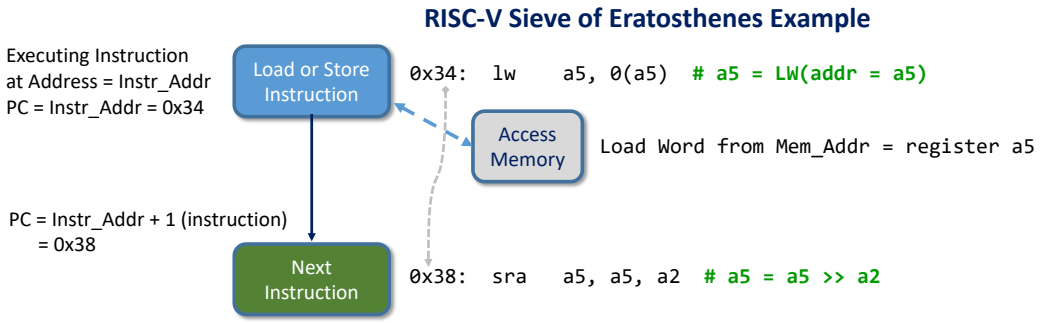


Fig. 10 Load and store instruction class (same as sequential with memory access)

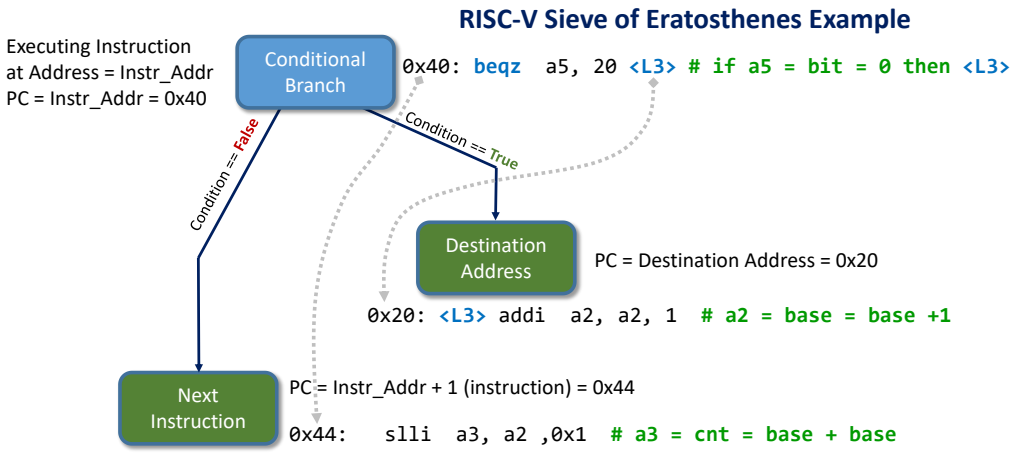


Fig. 11 Conditional branch instruction class

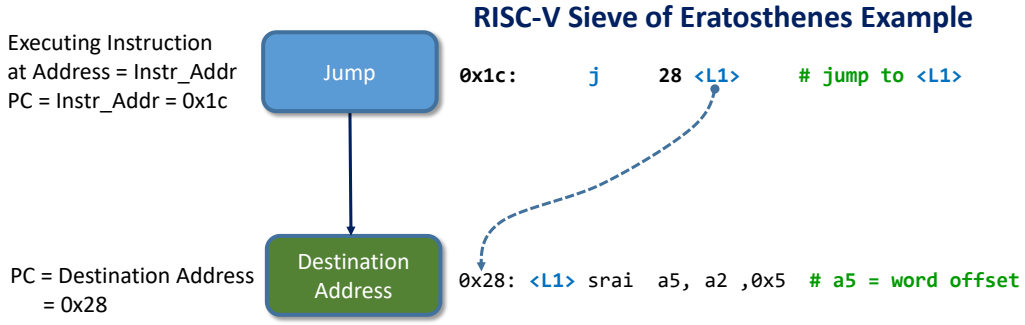


Fig. 12 Jump instruction class

The instruction execution monitor provides a hardware virtual machine for instruction execution. The execution of the instruction must pass all of the state machine monitors' verifications in order for the instruction to complete execution. If a state machine monitor's verification fails, a hardware-level exception is raised.

4.3.4 Memory Page Background

The Aberdeen Architecture uses the memory classes defined in Fig. 13. The classes extend the memory classes described in the research paper "Hardware Security Kernel for Cyber Defense" (Jungwirth et al. 2019b). A von Neumann machine mixes program instructions and data. There is no difference between program instructions and data; both are integers. In a von Neumann machine, a simple buffer overflow "converts `data`" into program instructions. The von Neumann architecture violates several of Saltzer and Schroeder's security principles. The Aberdeen Architecture's memory classes restrict permitted operations to provide least privilege and privilege separation for each memory class.

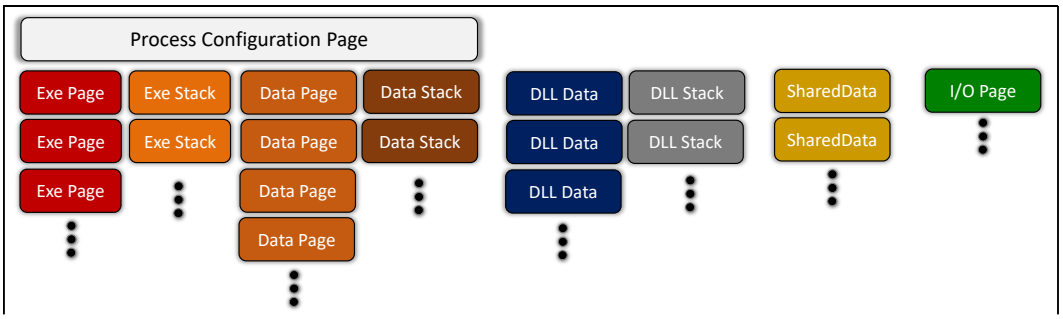


Fig. 13 Memory classes. Each class supports least privilege, privilege separation, and complete mediation.

`Process_Config_Page` contains the start-up or boot parameters for the process. Only the state machine monitors may read a **`Process_Config_Page`** memory page. The **`Process_Config_Page`** contains the global security settings for a process. For a more secure environment, encryption and a digital signature can be used to protect the integrity of the **`Process_Config_Page`**.

The classic stack mixes data, function call parameters, return addresses, and so on. Data, function call parameters, and return addresses are different object classes. Each class has a set of authorized and unauthorized operations. Mixing stack classes violates Saltzer and Schroeder's security principles. On the surface, more stack classes may seem like an increase in complexity; however, we are restricting each stack class to least privilege. The multiple stack classes provided for the same functionality as the conventional mixed class stack with significantly better computer security.

The **`Exe_Page`** memory page class is execute only (unless page is being loaded). **`Exe_Stack_Page`** provides function call and return stack operations for a running process. We seek to completely isolate **`Exe_Stack`** from data. **`DLL_Data`** and **`DLL_Stack`** memory pages support function calls to library functions. **`DLL_Data`** and **`DLL_Stack`** memory pages restrict operations to those limited to support function calls.

The **`Shared_Data`** memory page supports shared memory between two processes. Shared memory operations are restricted to operations permitted for sharing memory between two or more running processes.

The **`I/O_Page`** is the only memory page class that supports input and output. The **`I/O_Page`** class limits input and output to a single memory class. **`I/O_Page`** supports least privilege and privilege separation for input and output operations. No other memory page class may be used for I/O. This is not as flexibles as C-pointer arithmetic operations; however, it is much more secure. The **`I/O_Page`** class supports complete mediation for all input and output operations.

4.3.5 Tag Protection Bits

Saltzer and Schroeder promoted tagged memory for computer security back in 1975:

> Suppose, for example, that every location in memory were tagged with an extra bit. If the bit is **OFF**, the word in that location is an ordinary data or instruction word. If the bit is **ON**, the word is taken to contain a value suitable for loading into a protection descriptor register. … This kind of scheme is a particular example of what is called a tagged architecture.

With the DARPA CRASH (DARPA 2010) and DARPA SSITH (Salmon 2017a) programs, there has been a renewed interest in tag bits for high-assurance computer systems. There is an underlying assumption for tag bits. Software cannot access tag security bits. Software cannot force the architecture to leak information about the tag security bits. Without complete isolation, tag security bits are not able to protect a system.

The Aberdeen Architecture uses the security tag field architecture from the OS Friendly Microprocessor Architecture (Redstone Architecture). A brief introduction to tag fields for the Aberdeen Architecture is presented next. More details will be provided in later sections.

Table 4 illustrates the instruction word tag protection bit fields for the Aberdeen Architecture. The tag bits are "attached" to the instruction; however, only the hardware state machines can access the tag bits. Tag bits are assumed to be created by a trusted authority and/or trusted process. The tag bits allow the state machine security policies to verify instruction execution (complete mediation). The tag bits are completely isolated from all software and the execution pipeline.

Table 4 Aberdeen Architecture instruction format

Memory Address	RISC-V Machine Code		Local Tag Fields		
0x0000 0034	0x0007a783		Exe Tags	CFI Tags	DFI Tags

The **Process_Mem_Page** is shown in Table 5. The memory page contains process start, resource information, and process shutdown information. For better security, the **Process_Mem_Page** should be encrypted and digitally signed. Table 6 shows the local security tag bits for the register file. There are four categories of tag bits: (1) register read / write / modify / stack operations / protected register, (2) register load/store, (3) control flow integrity, and (4) data flow integrity.

Table 5 Aberdeen Architecture process configuration memory page

Configuration Information
•••
Configuration Information

Table 6 Aberdeen Architecture register file format

Aberdeen Architecture		Local Tag Fields		
Register Number		RWM LD/ST	CFI Tags	DFI Tags
R0				
R1				
•••				
R31				

In Table 7, the **Mem_Page** contains global and local security tag fields. The global tags set the security sandbox (fence) limit for the memory page. The local security tag fields can further restrict the bounds on allowed memory accesses.

Table 7 Aberdeen Architecture memory page format

Global Tags				
		Local Tag Fields		
Memory Address (4096 bytes)		Load/Store	CFI Tags	DFI Tags
0x0000				
0x0004				
•••				
0x0ffc				

4.3.6 Harvard Machine Architecture

A von Neumann architecture is compared to a Harvard architecture in Fig. 14. In a von Neumann machine, there is no difference between data and instructions. This is a fundamental security flaw. A simple buffer overflow can convert "data" into program instructions. A Harvard architecture has complete hardware isolation between data and program instructions. A return-oriented programming attack requires malicious control of a stack and access to gadgets to implement the steps required for the attack (Abadi et al. 2005). On x86 architectures, instructions have variable byte lengths that offer more possibilities to find useful instructions for malicious operations. Göktaş et al. (2014) show how von Neumann machine stack and gadget attacks are becoming more sophisticated:

> ROP [return-oriented programming] exploitation is based on an attacker controlling the stack of a program. After corrupting the stack and controlling the return address of an executing function, when the function returns, control is diverted to a gadget specified by the attacker's payload. Since gadgets are small sequences of code that end with a **ret** [return instruction], similar to the return-

oriented gadget shown in Fig. 3a, the attacker can carefully position data on the stack to make the program jump from gadget to gadget, chaining together the final code.

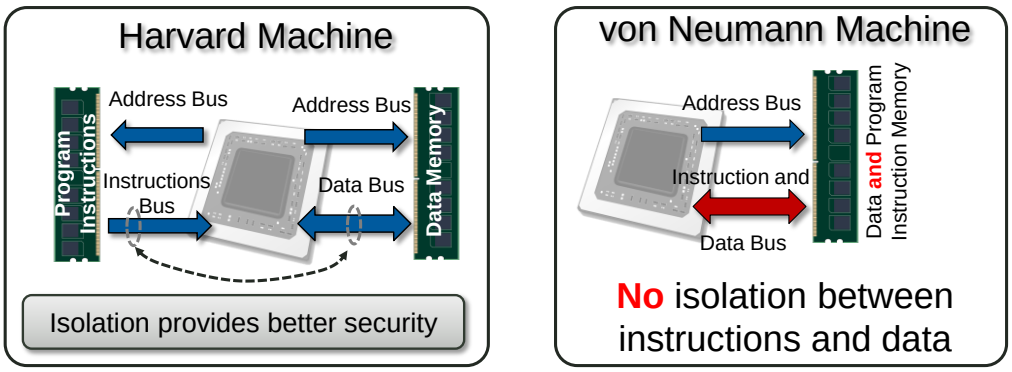


Fig. 14 Harvard machine compared to von Neumann machine (Jungwirth 2020b)

New architectures need better stack isolation and security tags to block gadget attacks. Function single entry and exit points could potentially eliminate most gadget attacks. Single entry and exit points would reduce the number of gadgets available from greater than 100,000 (Göktaş et al. 2014) to a relatively small number. Instruction words should all have the same length (one instruction per memory word). Note, the RISC-V architecture allows for byte, half-word, and word memory accesses. For security, a simpler architecture only offering word memory accesses would be a better architecture.

4.3.7 Aberdeen Machine Architecture

Aberdeen Architecture adds hardware-level state machine security policies to the Redstone Architecture. The core features for the Redstone Architecture's software security framework is the basis for the state machine security policies. The Redstone Architecture uses an extended Harvard architecture. The cache bank memory pipeline is illustrated in Fig. 15. A technical report covering the Redstone Architecture is found in Appendix A. The next section presents an introduction to the state machine security policies.

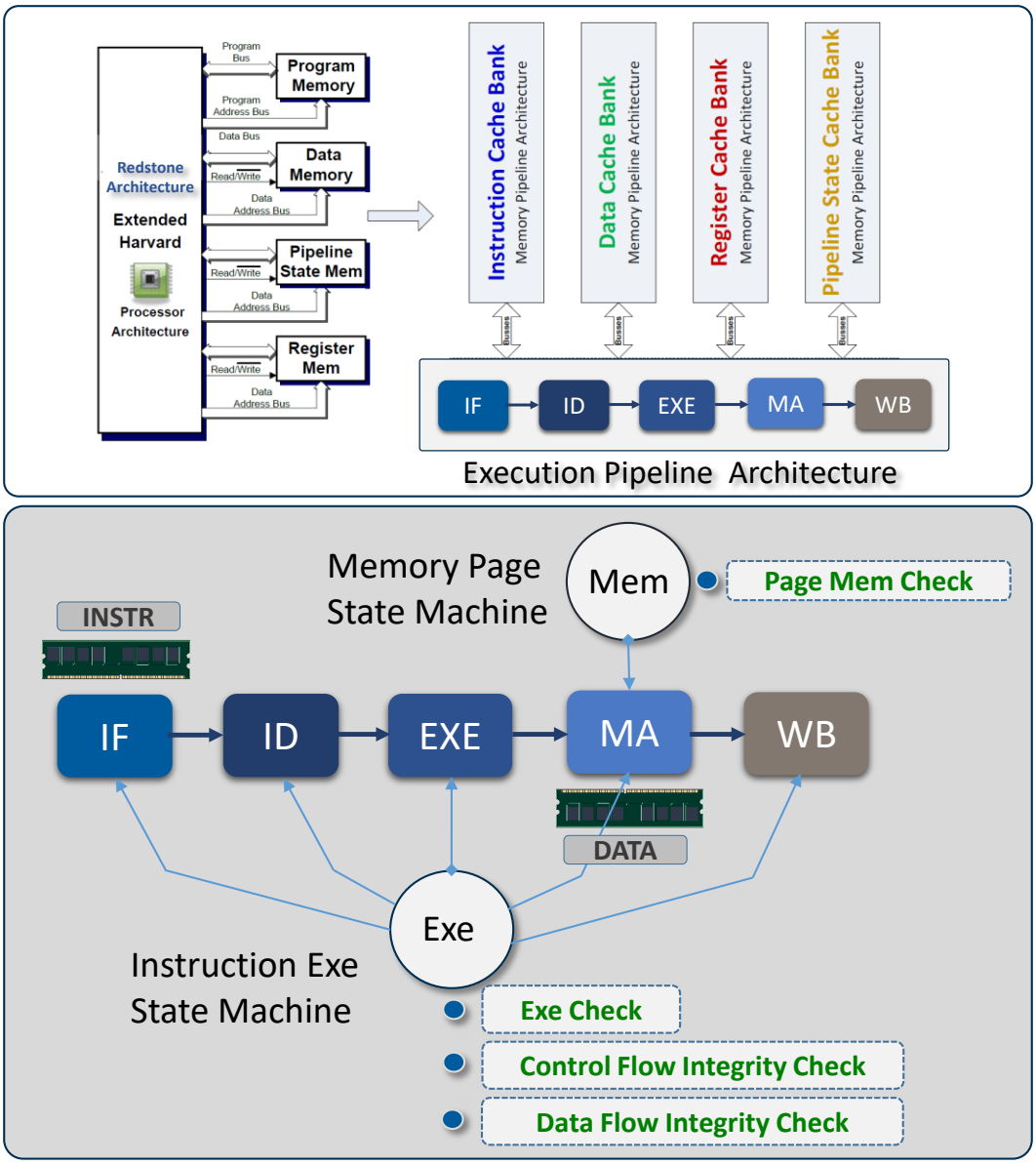


Fig. 15 Instruction execution pipeline: state machine controller

4.3.8 State Machine Security Policy Introduction

An introduction to the Instruction Execution Pipeline State Machine Controller is presented in Fig. 15. State machines verify security policies for instruction execution. Aberdeen Architecture considers instruction execution as an information flow. Security policies verify information flow properties during instruction execution. In terms of a virtual machine, the state machines' security policies create a virtual machine–based execution pipeline. Instruction execution policies are verified during instruction execution. If one of the security policies fails, a hardware-level exception is issued.

During instruction execution, control flow integrity is verified; data flow integrity is verified; memory and memory page operations are verified; and instruction execution is verified. Figure 15 summarizes the main state machine security policies. The paper "Security Tag Computation and Propagation in OSFA" describes data flow integrity for the Redstone Architecture (Jungwirth et al. 2018b).

A high-precision control flow implementation uses unique labels for each branch in the control flow graph. Each node (code block) has entry and exit links (graph edges). The destination address for jump/branch instruction is the entry point for a code block. A jump or branch instruction forms the exit point for a code block. For the high-precision case (Abadi et al. 2005), each edge (branch/jump control flow change) in the control flow graph can be exactly identified by its unique label. Less precise implementations reuse labels (labels are not unique).

High-precision implementation requires considerably more resources (memory) than the low-precision implementation. For the Aberdeen Architecture, the labels are scalable. Higher precision requires more bits for the edge labels. For the basic implementation, we consider a small number of labels combined with function single entry and exit points. Single entry and exit points significantly reduce gadget-based attacks. A malicious program uses function `CALL` instructions to gadgets (short sequence of useful hacking instructions followed by a return statement). Several gadgets are called in sequence to launch an attack. Low-precision control flow graphs provide less protection against gadget attacks. The Aberdeen Architecture uses single point function call entry and exit points to significantly reduce (best case would be to completely prohibit gadget attacks). The single point entry and exit code block tags strengthen the protections provided by lower-precision control flow protections.

Figure 16 presents an introduction to the instruction execution state machine controller. The four classes of instruction execution control flows are highlighted in blue. Sections 4.3.2 and 4.3.3 describe the control flow properties of the four instruction classes. Figure 16 also introduces memory operations: stack operations, memory page allocate, and memory page deallocate. The state machine for stack operations is introduced in Fig. 17. Stack and memory page security policies will be considered later in Section 4.4. We begin be looking at control flow integrity for instruction execution next.

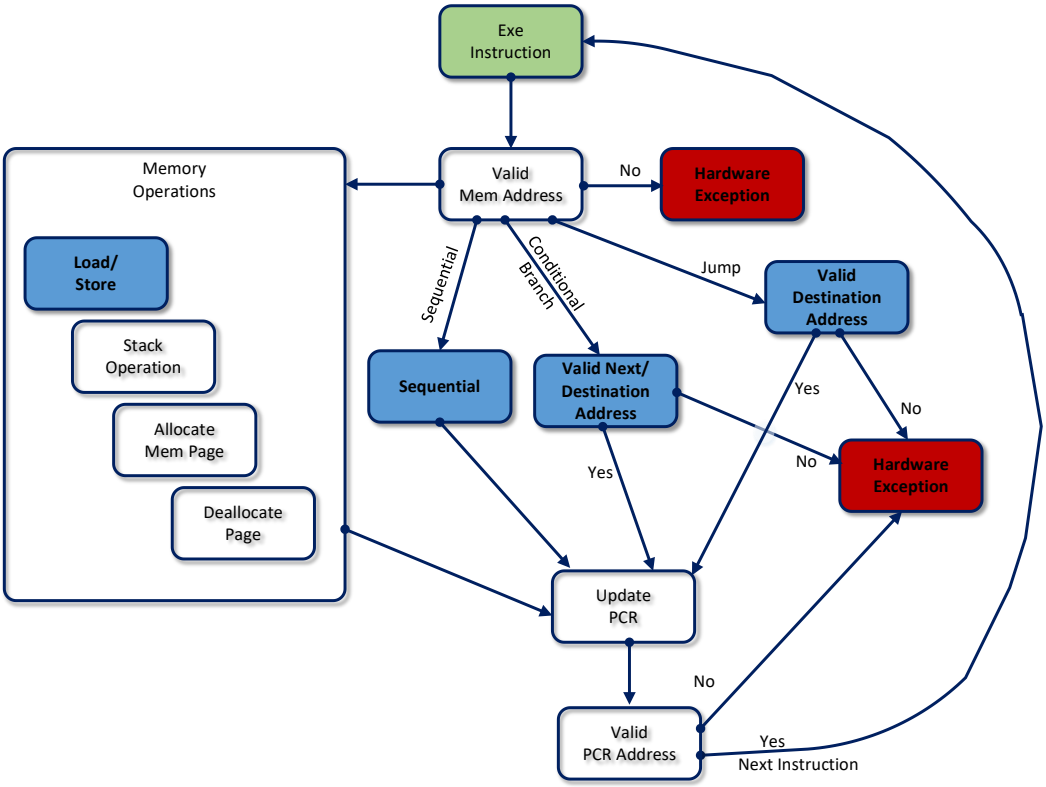


Fig. 16 Instruction execution state machine

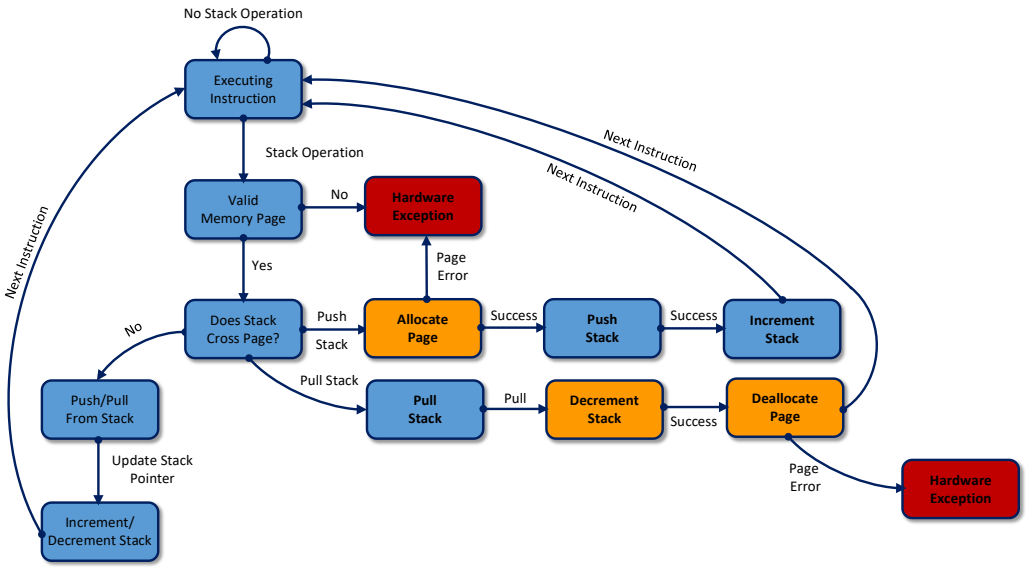


Fig. 17 Stack machine state machine

4.3.9 Control Flow Integrity

Abadi et al.'s paper (2005) reviewed control flow integrity methods and renewed interest in control flow integrity for software security. Alves-Foss et al. (2014) researched security tagging bits for high assurance: "Metadata-driven hardware interlocks make it practical to take the security principles of Saltzer and Schroeder [59] seriously." For static programs, a control flow graph shows all possible execution paths for the program. In general, a control flow graph is undecidable when considering self-modifying programs, dynamic dispatch, and just-in-time compiling (dynamic re-compilation). For simplicity, we will consider static control flow graphs. The Aberdeen Architecture's control flow integrity methods extend the research concepts described in "Security Tag Fields and Control Flow Management," (Jungwirth and Ross 2019), "Hardware Security Kernel for Cyber Defense," (Jungwirth et al. 2019b), and "Hardware Security Kernel for Managing Memory and Instruction Execution" (Jungwirth et al. 2020).

> CFI [control flow integrity] enforcement is effective against a wide range of common attacks, since abnormal control-flow modification is an essential step in many exploits — independently of whether buffer overflows and other vulnerabilities are being exploited. … We have examined many concrete attacks and found that CFI enforcement prevents most of them. … Of course, CFI enforcement is not a panacea: exploits within the bounds of the allowed CFG [control flow graph] (e.g., Chen et al. [2005]) are not prevented. (Abade et al. 2009)

Figure 18 presents the control flow graph for the Sieve of Eratosthenes code in Fig. 8. Aberdeen Architecture adds single entry and exit points for function calls, exception handlers, and interrupt requests. The single entry and exit points provide for Saltzer and Schroeder's security principles: least privilege, privilege separation, and complete mediation. The single entry and exit points are to help block gadget attacks and improve precision for control flow integrity.

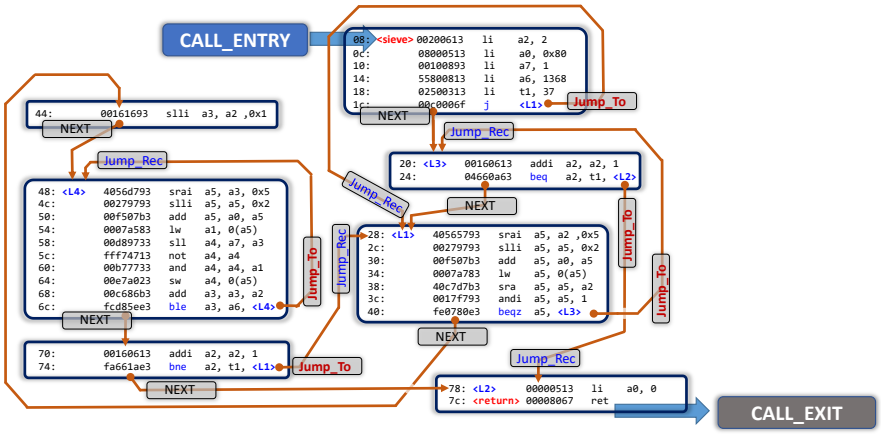


Fig. 18 Control flow graph for Sieve of Eratosthenes

Control flow integrity techniques have trade-offs between protection level, overhead, and implementation difficulty. Software control flow integrity techniques typically have a high overhead; however, some methods offer very high levels of precision (fine grain protection control). The Aberdeen Architecture considers a hardware implementation of a simple set of rules to enforce CFI. The proposed methods do have limitations; however, we believe the protection offered, complexity, and ease of implementation are well balanced.

> Undoubtedly, even loose forms of CFI harden binaries against attacks. Normally, control-hijacking exploits are able to redirect execution to any instruction in the binary. On x86 architectures, which use variable-length instructions and have no alignment requirements, an attacker can redirect control to virtually any executable byte of the program. If we consider every executable byte as a potential control-flow target, then CFI blocks more than 98% of these targets [17]. But, is the remainder 2% enough for attackers exploiting a program? (Göktaş et al. 2014)

Figure 19 compares high-precision control flow tags to low-precision tags. The high-precision tag uses a large (for example, 32 bit) random integer to link the conditional branch to its destination address. With 2^{32} = 4,294,967,296 ≈ $4.3 \cdot 10^9$ possible tag combinations, each branch can have a unique tag. Low-precision tags use only a few tag values and have to reuse tag values for other branches.

Address	Label	Machine Code	Assembly Language		Tag Fields
0020:	<L3>	0x00160613	addi a2, a2, 1		CFI Tag = 0xFC66BC28 (32 bit random integer)
		▪▪▪			▪▪▪
		▪▪▪			▪▪▪
0040:		0xfe0780e3	beqz a5, 20 <L3>		CFI Tag = 0xFC66BC28 (32 bit random integer)

Address	Label	Machine Code	Assembly Language		Tag Fields
0020:	<L3>	0x00160613	addi a2, a2, 1		CFI Tag = 0x28 (8 bit random integer)
		▪▪▪			▪▪▪
		▪▪▪			▪▪▪
0040:		0xfe0780e3	beqz a5, 20 <L3>		CFI Tag = 0x28 (8 bit random integer)

Fig. 19 High- and low-precision CFI tags

To address stack flow integrity issues, the Aberdeen Architecture uses several stacks to separate program execution (**CALL** and **RETURN**) from program data flow. More on stack and data control flow will be presented in later sections. Kiriansky et al. (2002) describes the problems with mixing data, **CALL**/**RETURN**, etc. on a stack:

> Many entities participate in transferring control in a program execution. Compilers, linkers, loaders, runtime systems, and hand-crafted assembly code all have legitimate reasons to transfer control. Program addresses are credibly manipulated by most of these entities, e.g., dynamic loaders patch shared object functions; dynamic linkers update relocation tables; and language runtime systems modify dynamic dispatch tables. Generally, these program addresses are intermingled with and indistinguishable from data. In such an environment, preventing a control transfer to malicious code by stopping illegitimate memory writes is next to impossible. It requires the cooperation of numerous trusted and untrusted entities that need to check many different conditions and understand high-level semantics in a complex environment.

Abadi et al. (2005) describe how stack operations reduce the precision of a control flow graph: "In particular, a finite CFG [control flow graph] does not capture the dynamic execution call stack ..." Göktaş et al. [2014] point out the importance of control flow to counter modern attacks: "As existing defenses like ASLR, DEP, and stack cookies are not sufficient to stop determined attackers … In its ideal form, CFI prevents flows of control that were not intended by the original program …"

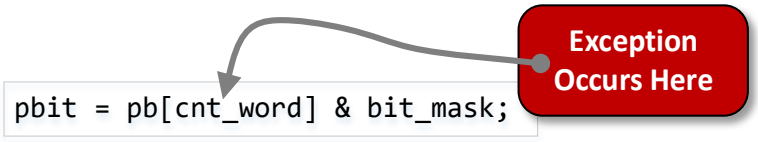
A mixed stack violates several of Saltzer and Schroeder's security principles. The Aberdeen Architecture incorporates a data flow integrity policy to provide additional protections for better control flow graph precision. In other words, control flow integrity is a function of data flow integrity.

Buffers can be used to keep track of the last N_I instructions, last N_B branches, and last N_{PC} program counter values. Unfortunately, finite buffers lack control flow graph precision. Arbitrarily large buffers are required to track large software programs and data-driven function calls like recursion. CFI buffer lengths are briefly discussed in (Jungwirth and Ross 2019). Aberdeen Architecture uses block entry tags to ensure the current instruction has a valid control flow path back to the block entry point. When the corresponding block exit point is reached, the path from start to end can be deleted since control flow from block start to block end has completed.

Figures 20–23 illustrate single point entry and exit points for exception handlers and interrupt request and return handlers. This is to ensure that exception handlers and return from interrupt requests cannot jump to a maliciously selected return address to enable a gadget attack.

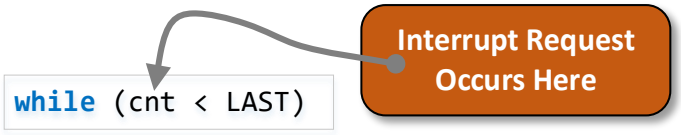
The single point entry and exit points are to provide specific fixed entry and exit points. The fixed entry and exit points limit rogue behavior by reducing to eliminating gadget code start points. There is some lost flexibility and additional code required; however, the additional protection is well worth the single entry and

exit point cost. Figure 22 illustrates an exception occurring in the array `pb[ ▪ ]` in **{**3**}**. The exception is handled by a single exit point, then either a local routine or a function call occurs to handle the exception. Return from exception in **{**3**}** is handled by a single routine. The exception handlers provide single entry and exit points and prohibit "spaghetti code" exception handlers enabling gadget attacks.



{3**}**

Figure 23 covers interrupt requests following the same idea for exception handlers. An interrupt occurs during the `while` loop in **{**4**}**. Single point entry and exit points cover a local interrupt handler and a function call to handle the interrupt request. Again, more code is required; however, the security benefits of single entry and exit points far out way the cost.



{4**}**

Instruction Class	CFI Link Type	CFI Tag Type
Arithmetic/Logic Instruction	Sequential	Sequential Tag
Load/Store	Load/Store Sequential	Load/Store Sequential Tag
Conditional Branch	Conditional Branch	Conditional Branch Tag
Jump	Jump	Jump Tag
CALL	Function Call	CALL Tag
RETURN	Function Return	RETURN Tag
EXCEPTION	Exception Link	EXCEPTION Tag
EXCEPTION Return	Exception Return Link	Exception Return Tag
EXCEPTION Process Terminate	Exception Terminate Link	Exception Terminate Tag
INTERRUPT	INTERRUPT Handler Link	Interrupt Handler Tag
INTERRUPT RETURN	Interrupt Return Link	Interrupt Return Tag
INTERRUPT EXCEPTION Process Terminate	INTERRUPT EXCEPTION Process Terminate	INTERRUPT EXCEPTION Process Terminate Tag

Fig. 20 Aberdeen Architecture CFI link types

```
// Sieve of Eratosthenes
// C code is based on https://rosettacode.org/wiki/Sieve_of_Eratosthenes#Ada
// Algorithm see https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
```

```
void sieve() {
   int base = 2; int pbit = 0; int cnt = 0;
   int base_bit = 0; int base_word = 0; int base_shift = 0;
   int cnt_word = 0; int cnt_shift = 0; int cnt_mask = 0; int
bit_mask = 0;
   while (base < R) {
      base_word = base >> 5; base_shift = base & 0x1f;
      base_bit = (pb[base_word] >> base_shift) & 1;
      if (base_bit)
      {  cnt = base << 1; // base + base;
         while (cnt < LAST)
         {  cnt_word = cnt >> 5; cnt_shift = cnt & 0x1f;
            cnt_mask = (1 << cnt_shift); bit_mask = ~cnt_mask;
            pbit = pb[cnt_word] & bit_mask;
            pb[cnt_word] = pbit;
            cnt = cnt + base; }
      }
      base = base + 1;
   }
}
```

Sieve of Eratosthenes Code

```
// Sieve Exception Handlers
EXCEPTION HANDLER // C code placeholder
// HANDLE EXCEPTION HERE (LOCAL)
   int local_exception_handler(  ◙  )
// CALL EXCEPTION HANDLER (USING FUNCTION CALL)
   int call_exception_handler(  ◙  )
// EXCEPTION HANDLERS MUST ALL RETURN HERE
   int handle_return_to_local_code(  ◙  )
   int single_point_exception_handler_return(  ◙  )
```

Single Point Exit and Return Points for Exceptions

```
// Sieve Exception IRQ Handlers
IRQ HANDLER // C code placeholder
// HANDLE EXCEPTION HERE (LOCAL)
   int local_IRQ_handler(  ◙  )
// CALL IRQ HANDLER (USING FUNCTION CALL)
   int call_IRQ_handler(  ◙  )
// IRQ RETURNS MUST ALL RETURN HERE
   int single_point_IRQ_handler_return(  ◙  )
```

IRQ Handler Single Exit and Return Points

Fig. 21 Exception and IRQ handlers exit and return points

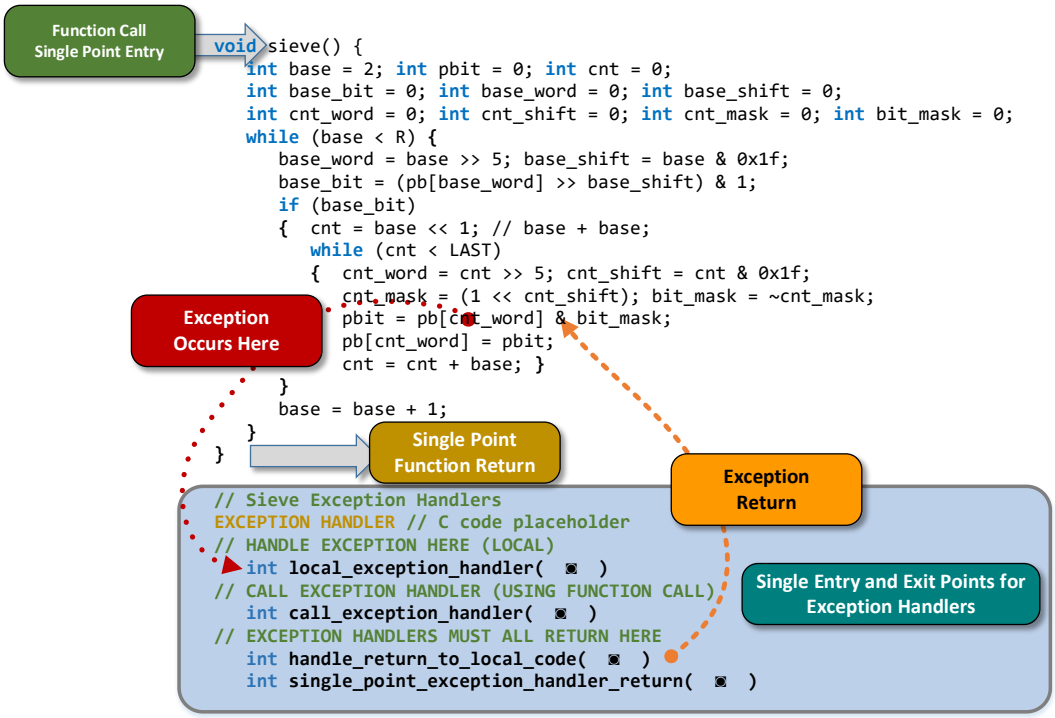


Fig. 22 Single point entry and exit exception handlers

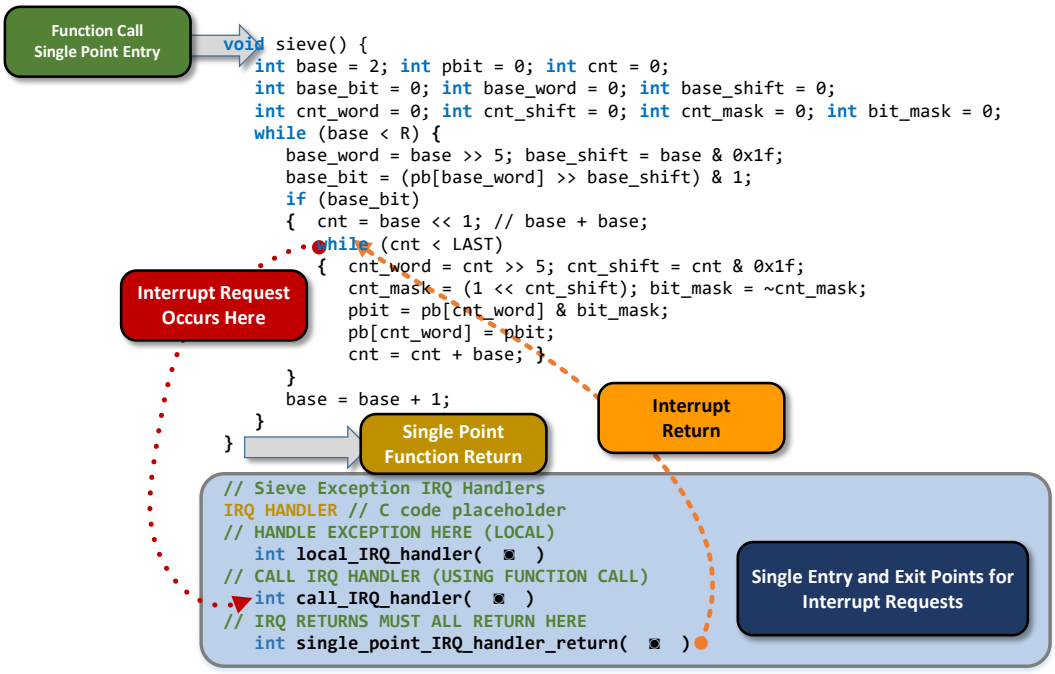


Fig. 23 Single point interrupt request entry and exit points

4.3.9.1 Sequential Control Flow Integrity Class Instructions

Sieve of Eratosthenes control flow graph has single point function entry and exit points shown in Fig. 24. Figure 24 shows the code blocks consisting of sequential instructions with branch and jump instructions connecting the blocks together. The tags **NEXT**, **JUMP_To**, and **JUMP_Rec** illustrate how the code blocks form a linked list. Function calls are only allowed to the **CALL_ENTRY** point: **<sieve>**. The single point function return is **<return>**. Function calls, jump instructions, branch instructions, and so on, from another code block to internal code in Fig. 24 are not allowed. The single point entry and exit points are to prevent gadget code attacks. The single point code entry and exit points provide better control flow graph precision for a limited number of tags (or labels).

Figure 25 shows how arithmetic and logic sequential instructions and `LOAD/STORE` sequential instructions form a linked list A sequential instruction simply advances the program counter by one instruction in (1). For RISC-V, a single instruction is 4 bytes in length. `LOAD` and `STORE` instructions are sequential instructions that access memory. `LOAD/STORE` control flow tags are required to access memory. Figure 26 shows a load instruction with control flow integrity tag = `LOAD`.

Sequential Instruction Control Flow Integrity Class Property		
$PCR(n + 1) = PCR(n) + 4$ bytes	1 instruction is 4 bytes long	(1)

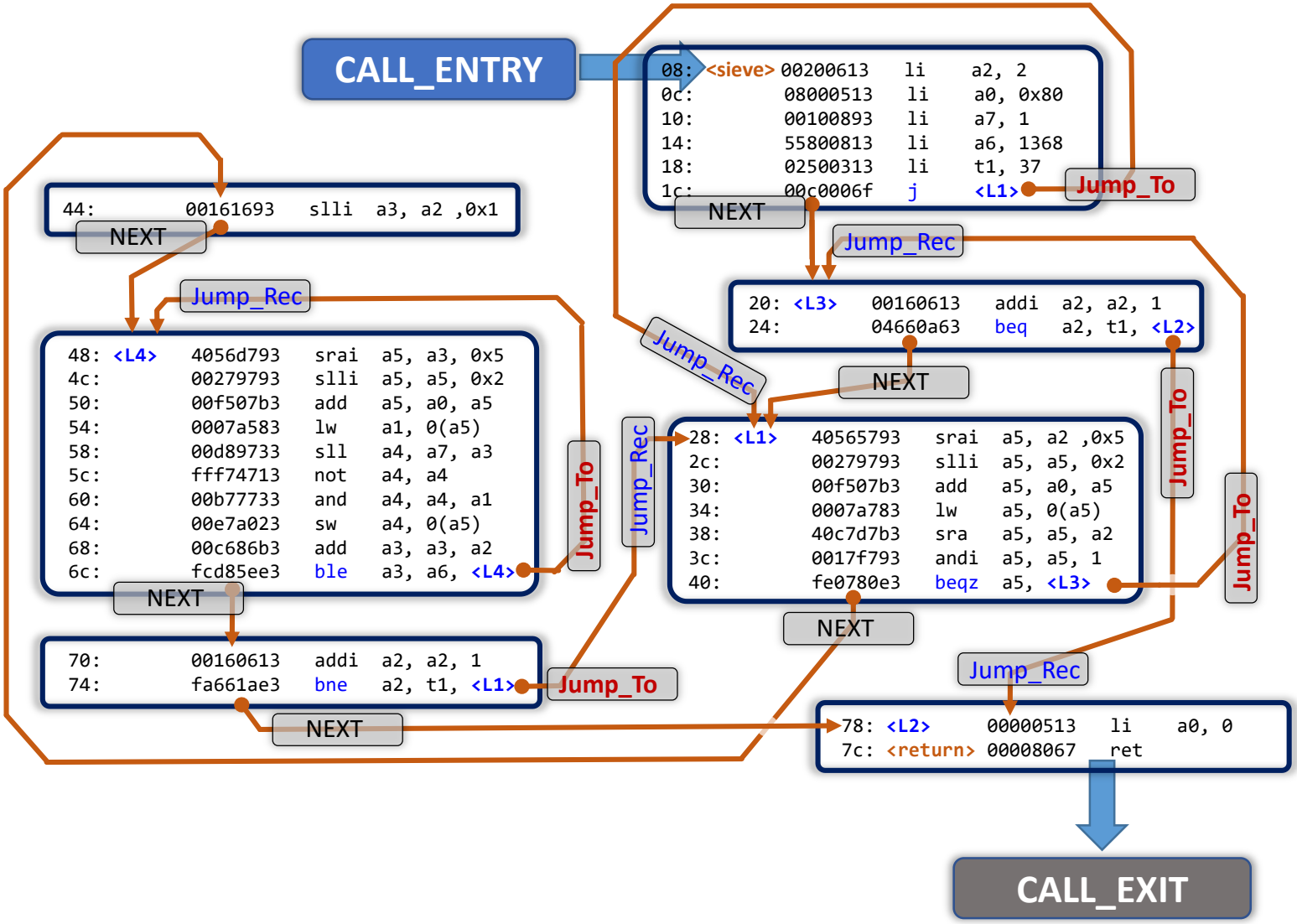


Fig. 24 Control flow graph for Sieve of Eratosthenes

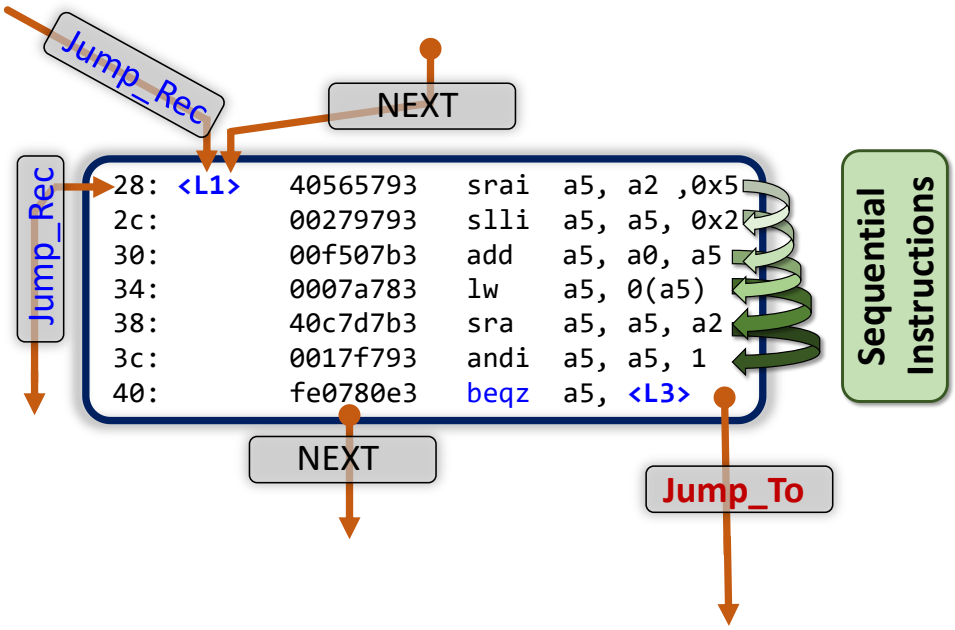


Fig. 25 Control flow graph for arithmetic and logic sequential instructions

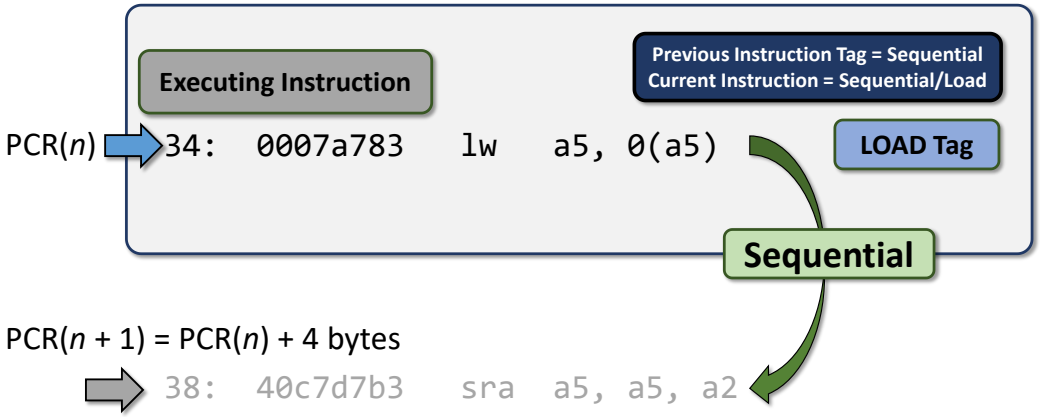


Fig. 26 Control flow graph for LOAD sequential instruction

4.3.9.2 Jump Instruction Control Flow Graph

Figure 27 illustrates the control flow properties for a JUMP instruction. There is only one next address: the destination address. Equation (2) shows the next PCR value, $PCR(n+1)$, is simply the destination address.

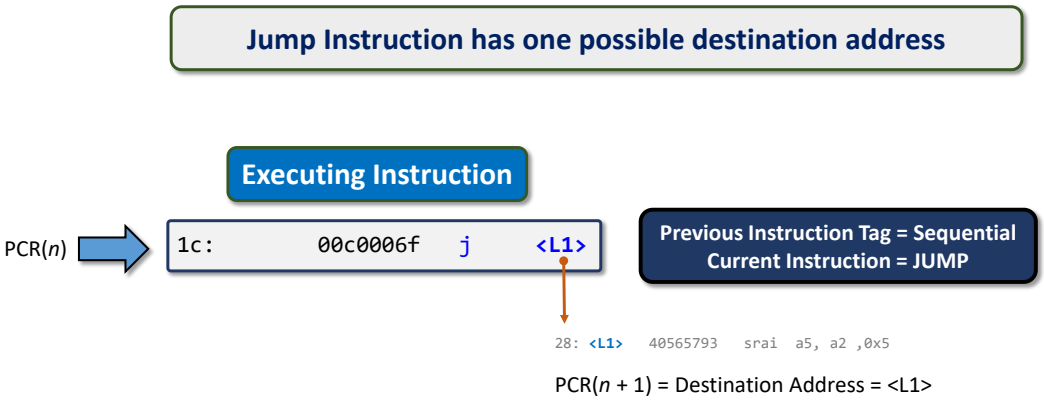


Fig. 27 Control flow graph for JUMP instruction

JUMP Instruction Control Flow Integrity Class Property	
$PCR(n+1) = \text{Destination Address}$	(2)

4.3.9.3 Branch Instruction Control Flow Graph

The control flow graph properties for a branch instruction are illustrated in Fig. 28. There are two possible next instruction addresses. If the branch condition is **`true`**, the next address is the destination address. Else (condition is **`false`**), the next address is the same as a sequential instruction; next address is the next sequential instruction. Equation (3) describes the PCR values for a branch instruction.

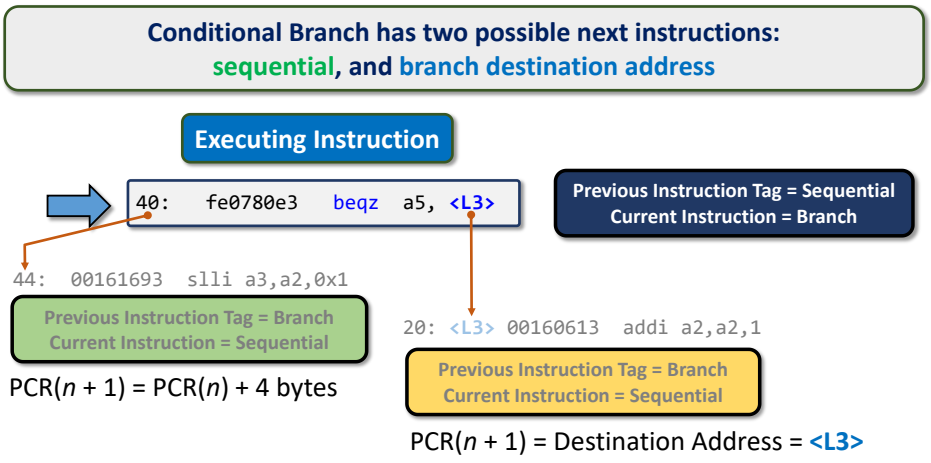


Fig. 28 Control flow graph for branch instruction

Branch Instruction Control Flow Integrity Class Property

```
if( condition == TRUE )
   PCR(n + 1) = Destination Address

else // condition == FALSE
   PCR(n + 1) = PCR(n) + 4 bytes // (same as sequential instruction)
```

(3)

4.3.9.4 Sieve of Eratosthenes RISC-V Code with Control Flow Integrity Tags

Figure 29 illustrates control flow integrity tags for the Sieve of Eratosthenes RISC-V code. The instruction **START** field tag is the **END** tag from the previous instruction. The **EXE** tag is the instruction execution tag. There are four execution classes of instructions: arithmetic and logical sequential, LOAD/STORE sequential, jump, and conditional branch. Figure 30 shows the single entry and exit points for the Sieve of Eratosthenes code. The **EXE** tags place limits on the RISC-V instructions behavior: **SEQ** = sequential instruction, **LOAD** = load sequential instruction, **STORE** = store sequential instruction, **BR** = conditional branch, and **JMP** = jump. For example, a conditional branch instruction (as illustrated in Fig 28) has two possible next instructions, either sequential or destination address.

```
// Sieve of Eratosthenes – RISC-V Assembly Language [21]-[22]

Memory Address       Machine Code       Assembly Code
0x0000 0008: <sieve>  0x00200613   li    a2, 2
0x0000 000c: 0x10000513      li    a0, 0x100
0x0000 0010: 0x00100893      li    a7, 1
0x0000 0014: 0x06300813      li    a6, 0x63
0x0000 0018: 0x00a00313      li    t1, 0x8
0x0000 001c: 0x00c0006f      j     28 <L1>
0x0000 0020: 0x00160613 <L3> addi  a2, a2, 1
0x0000 0024: 0x04660a63      beq   a2, t1, 78 <L2>
0x0000 0028: 0x40565793 <L1> srai  a5, a2 ,0x5
0x0000 002c: 0x00279793      slli  a5, a5, 0x2
0x0000 0030: 0x00f507b3      add   a5, a0, a5
0x0000 0034: 0x0007a783      lw    a5, 0(a5)
0x0000 0038: 0x40c7d7b3      sra   a5, a5, a2
0x0000 003c: 0x0017f793      andi  a5, a5, 1
0x0000 0040: 0xfe0780e3      beqz  a5, 20 <L3>
0x0000 0044: 0x00161693      slli  a3, a2 ,0x1
0x0000 0048: 0x4056d793 <L4> srai  a5, a3, 0x5
0x0000 004c: 0x00279793      slli  a5, a5, 0x2
0x0000 0050: 0x00f507b3      add   a5, a0, a5
0x0000 0054: 0x0007a583      lw    a1, 0(a5)
0x0000 0058: 0x00d89733      sll   a4, a7, a3
0x0000 005c: 0xfff74713      not   a4, a4
0x0000 0060: 0x00b77733      and   a4, a4, a1
0x0000 0064: 0x00e7a023      sw    a4, 0(a5)
0x0000 0068: 0x00c686b3      add   a3, a3, a2
0x0000 006c: 0xfcd85ee3      ble   a3, a6, 48 <L4>
0x0000 0070: 0x00160613      addi  a2, a2, 1
0x0000 0074: 0xfa661ae3      bne   a2, t1, 28 <L1>
0x0000 0078: 0x00000513 <L2> li    a0, 0
0x0000 007c: <return>                ret
```

Tag Fields		
Instruction Start	Instruction Exe	Instruction End
START = CALL;	EXE = LOAD;	END = SEQ
START = SEQ;	EXE = LOAD;	END = SEQ
START = SEQ;	EXE = LOAD;	END = SEQ
START = SEQ;	EXE = LOAD;	END = SEQ
START = SEQ;	EXE = LOAD;	END = SEQ
START = SEQ;	EXE = JMP;	END = JMP
START = BR;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = BR;	END = BR|SEQ
START = SEQ|BR|JMP;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = LOAD;	END = SEQ
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = BR;	END = SEQ|BR
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ|BR;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = LOAD;	END = SEQ
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = STORE;	END = SEQ
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = BR;	END = SEQ|BR
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = BR;	END = SEQ|BR
START = SEQ;	EXE = SEQ;	END = SEQ
START = SEQ;	EXE = RTN;	END = RTN

Fig. 29 Sieve of Eratosthenes RISC-V code and control flow integrity tags

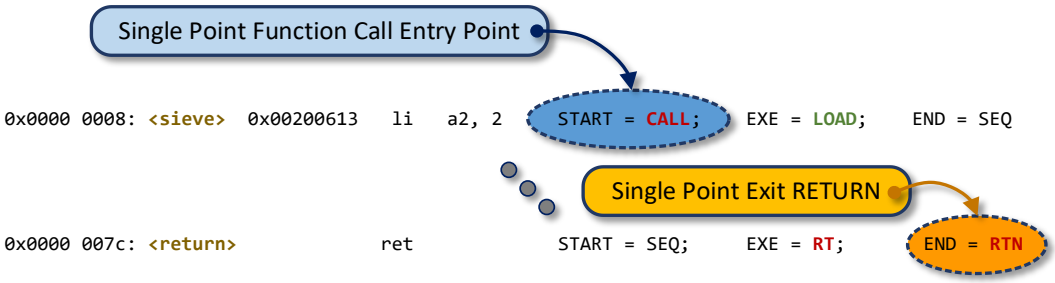


Fig. 30 Single point function entry and exit points

In Fig. 31, multiple control flow paths can lead to instruction **<L1>**. The **START** tags show that the previous instructions can be either sequential, conditional branch, or jump. As shown in Abadi et al. (2005), large random integers can provide higher levels of control flow graph precision. The low precision provided by a handful of control flow tags limits the control flow behavior; however, it does not limit the control flow behavior as much as unique tags used for high-precision control flow. For a practical application (Göktaş et al. 2014), control flow precision requires a balancing act: How much precision is enough given a limited amount of resources? We believe control flow tags combined with instruction execution tags, memory integrity, and data flow integrity tags provides 'the whole is greater than the sum of the parts' level of protection. Abadi et al. (2009) considers how dynamic stack behavior is not captured in the control flow graph. The Aberdeen Architecture uses stack state machines, stack isolation, and the separation provided by a Harvard architecture to enforce least privilege for stacks (see Section 3.1 in Jungwirth et al. [2019b] and Section 4.3.10 covering data flow integrity state machine monitors).

> Preferably, control-flow enforcement should be as precise as possible. However, even the reliance of CFI on a finite CFG implies a lack of precision. In particular, a finite CFG does not capture the dynamic execution call stack; we address this limitation in Section 5.4. Furthermore, without some care, schemes based on IDs and ID-checks may be more permissive than necessary. (Abadi et al. 2009)

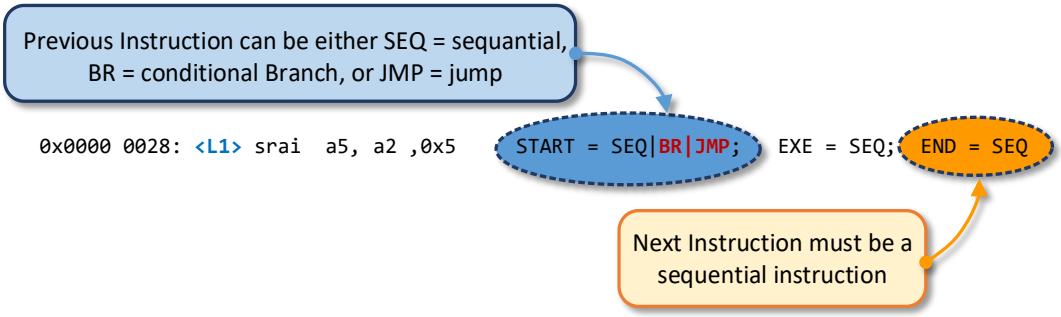


Fig. 31 START and END instruction execution control flow tags

4.3.10 Data Flow Integrity

Information integrity is defined as dependability and trustworthiness of information (Mandke and Nayar 2000). A sound security policy only allows authorized information flows. Denning 1976 formalized secure information flow. Unfortunately, current commodity microprocessors still do not provide protections against unauthorized information flow.

> The security mechanisms of most computer systems make no attempt to guarantee secure information flow. "Secure information flow," or simply "security," means here that no unauthorized flow of information is possible. (Denning 1976)

Taint analysis (Venkataramani et al. 2008; Schwartz et al. 2010; Chen et al. 2011; Kim et al. 2014; Prakash et al. 2015) typically implements a small number of security tags to isolate two classes of data. Taint status is used to monitor data flow integrity. Safe or trusted data is *untainted*. When trusted data is mixed with untrusted data, the taint status is changed to untrusted or tainted (Schwartz et al. 2010): "Data from trusted sources starts out as untainted … Taints are then propagated as values are copied or used in computation. To detect potential attacks, a tainting scheme looks for unsafe uses of tainted values." Venkataramani et al. (2008) extends taint propagation to control flow integrity applications using a tainted jump policy to catch control flow hijacking attacks:

> A prototypical application of dynamic taint analysis is attack detection. Table III shows a typical attack detection policy which we call the **tainted jump policy**. … The goal of the tainted jump policy is to protect a potentially vulnerable program from control flow hijacking attacks. …. A control flow exploit, however, will overwrite jump targets (e.g., return addresses) with input-derived values. The tainted jump policy ensures safety against such attacks by making sure tainted jump targets are never used.

Castro et al. (2006) illustrate the connections between control-flow integrity and data flow integrity. A control flow graph shows all possible comparison (control flow if-then-else, et al.) software execution paths. During program execution, a data flow graph shows how data types interact as data flows from a data source to a data sink. Castro et al. (2006) describe the three process steps to monitor data flow integrity:

> Data-flow integrity enforcement has three phases. The first phase uses static analysis to compute a data-flow graph for the vulnerable program. The second instruments the program to ensure that the data-flow at runtime is allowed by this graph. The last one runs the instrumented program and raises an exception if data-flow integrity is violated.

Song et al. (2016) researched hardware-assisted data flow isolation. Data flow integrity ensures data flow follows an allowed path in a data flow integrity graph. Data flow integrity provides integrity and confidentiality guarantees. Song et al. explain:

> For example, to protect the **integrity** of sensitive data, we can enforce the Biba Integrity Model [6]. In particular, we can use the tag to indicate integrity level (`IL`) of the corresponding data: sensitive data has **`IL1`** and normal data has **`IL0`**. Next, we assign `IL` to write operations based on the data-flow. That is, we use static analysis to identify write operations that can manipulate sensitive data, and allow them to set the memory tag to **`IL1`**; all other write operations will assign to the tag to **`IL0`**. Finally, when loading sensitive data from memory, we check if the tag is **`IL1`** … HDFI [Hardware-Assisted Data-Flow Isolation] can also be used to enforce **confidentiality**, i.e., the Bell–LaPadula Model [5]. For instance, to protect sensitive data like encryption keys, we can set their tag to **`SL1`** (secret level 1), and enforce that all untrusted read operations (e.g., when copy data to an output buffer) can only read data with tag **`SL0`**.

Control flow [monitoring] precision describes how well a control flow protection algorithm detects malicious control flow behavior. Current control flow techniques do not take into account stack behavior (Abadi et al. 2005). Göktaş et al. (2014) described return-oriented program attacks against a program's stack. Stack and memory protections are required to strengthen control flow and data flow integrity protections.

> ROP [return-oriented program] exploitation is based on an attacker controlling the stack of a program. After corrupting the stack and controlling the return address of an executing function, when the function returns, control is diverted to a gadget specified by the attacker's payload. (Göktaş et al. 2014)

Aberdeen Architecture uses state machines to enforce security policies. There are four main hardware-enforced security policies: (1) instruction execution [integrity], (2) page memory access [integrity], (3) control flow integrity, and (4) data flow integrity. Aberdeen Architecture focuses on the whole is greater than the sum of the parts security policy. By using simple rules for information flows (1)–(4), Aberdeen Architecture implements a high-assurance security policy in hardware. Aberdeen Architecture provides complete mediation for instruction execution. State machines implement the security policies and are completely isolated from the execution pipeline.

Control flow integrity ensures a program follows a valid execution path on a control flow graph. During program execution, data flow integrity verifies that (1) the data-driven control flow path is valid; and (2) data flow properties of security, integrity,

and accuracy are valid. For example, for eight levels for security and integrity, security level `0` = most secure level and integrity level `0` = highest integrity. Pressure transducer data source, P_t, outputs a current proportional to pressure, $P_t = k \cdot P$. P_t has security = `3` and integrity = `2`. The conversion constant, C = 1 Torr/mA has security = `7` and integrity = `0`. Pressure result, R, is found in (4) and C code in **{**5**}**. For (4), the security and integrity values (5) are found using the upper bound in (6). Security can be up-converted; however, security cannot be down-converted. We cannot take sensitive information and change it to open source information. A constant value, like π, has a high integrity because it is an exact value. If we approximate π as 3.14, it has a lower integrity than the exact value. Integrity is calculated as a lower bound. An hourglass has integrity = `7` (lowest integrity), whereas an atomic clock has integrity = `0` (highest integrity). Fake news has integrity = `7`.

$$R[\text{Torr}] = P_t[\text{mA}] * C\left[\frac{\text{Torr}}{\text{mA}}\right] \qquad \text{Sensor conversion equation} \qquad (4)$$

```
RTorr = P_Transducer*C_Torr_per_mA;
```
C Code **{**5**}**

$P_t.security = 3,$
$P_t.integrity = 2$

$C.security = 7,$
$C.integrity = 0$

Security and Integrity Properties:
(`0` = most secure; `7` = least secure)
(`0` = highest integrity; `7` = lowest integrity) (5)

$R.security =$
$upper\_bound\ (3,7) = 3$

$R.integrity =$
$lower\_bound\ (2,0) = 2$

Security and Integrity Properties:
(`0` = most secure; `7` = least secure)
(`0` = highest integrity; `7` = lowest integrity) (6)

We can also place restrictions on the security and integrities values R is allowed to have. Equation (7) illustrates.

$R.max\_security = 4$,
$R.min\_security = 5$

$R.max\_integrity = 0$,
$R.min\_integrity = 4$

If the integrity and security values are outside the ranges, then an exception occurs. (7)

Data flow drives the execution of software. Conditional branches determine the execution path taken in a control flow graph. As pointed out by Abadi et al. (2005), stack operations are not captured in a control flow graph. For example, given two classes of information—open source and company proprietary—the lattice security model illustrates allowed information flows. Open source information can be upgraded to company proprietary; however, company proprietary cannot be downgraded to open source.

The simple data flow example in {5} does not block all potential information leaks. Denning (1976) illustrates how a simple **if** statement can leak information. In {5}, RTorr leaks information to Low_Pressure. Practically speaking, the information leak shown in **{**6**}** is only a problem if the information flows to a data sink that is accessible by a malicious program. We must block this potential attack vector.

> The primary difficulty with guaranteeing security lies in detecting (and monitoring) all flow causing operations. This is because all such operations in a program are not explicitly specified – or indeed even executed! As an example, consider the statement **if** a = **0** **then** b := 0; if b ≠ 0 initially, testing b = **0** on termination of this statement is tantamount to knowing whether a = **0** or not. In other words, information flows from a to b regardless of whether or not the then clause is executed. (Denning 1976)

```
RTorr = P_Transducer * C_Torr_per_mA;

Low_Pressure = FALSE;
if(RTorr < 100) {
  // Information leaks from RTorr to Low_Pressure
    Low_Pressure = TRUE; }
```

Example: C Code Information Leakage **{**6**}**

To block the information leakage in **{**6**}**, we need to reconsider single point code block entry and single point code block exit points. Figure 32 illustrates the control flow graph for the code in **{**6**}**. The true and false condition paths leak information from **RTorr** to **Low_Pressure**. For the code block defined by the **if** statement, we need to set the security tags for **Low_Pressure** equal to the security tags for **RTorr**. This ensures that the data flow from information source, **RTorr**, to information sink, **Low_Pressure**, follows a valid data flow integrity "path."

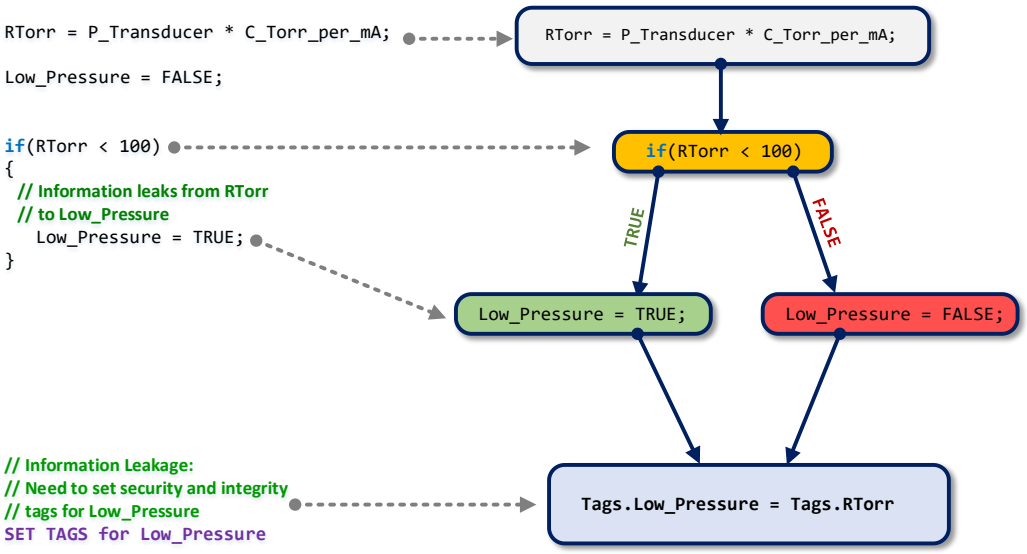


Fig. 32 Control flow graph and data flow integrity (security and integrity tags)

Figure 33 illustrates a data flow leakage path in the RISC-V Sieve of Eratosthenes code. Figure 34 shows a partial data flow integrity graph for the first half of the Sieve of Eratosthenes RISC-V program. Figure 34 also shows how integrity and security tags for data flow from information source (point where data originates) to data sink (point where data is no longer used). Data flow integrity verifies the security and integrity tags during instruction execution. Aberdeen Architecture verifies instruction execution by checking (1) instruction execution tags, (2) control flow graph integrity tags, (3) data flow integrity tags, and (4) memory access tags. Hardware state machines monitor the (1)–(4) tags. By using four simple state machines to provide overlapping security policy coverage, the Aberdeen Architecture is able to use low-moderate precision tags to provide high assurance and complete mediation of instruction execution. The paper "Security Tag Computation and Propagation in OSFA" (Jungwirth et al. 2018b) describes security tag propagation for the Redstone or OS Friendly Microprocessor Architecture.

Aberdeen Architecture can also support integrity tags for accuracy and measurement units. We could specify accuracy tags of `0` = 64 bit, `1` = 32 bit, `2` = 16 bit, `3` = 8 bit. For measurement units, we could define tags for `0` = volt, `1` = ampere, `2` = ohm, and `3` = power.

```
<sieve>  li    a2, 2           # base = a2
                               a2.security = constant.security = 7
                               a2.integrity = constant.integrity = 0

// create buffer a0 with start address = a0, end address = a1
         li    a0, 0x100       # buffer pb[0] = address = 0x100
                               a0.Security = 7, a0.integrity = 0
         li    a1, 0x200: a1.Security = 7, a1.integrity = 0
// create buffer a0            # Aberdeen Architecture Instruction
      AA.cb    a0,a0,a1        a1.security = 7, a1.integrity = 0
                               a0.security = 7, a0.Integrity = 0
                               a0.security  = Upper_Bound( 7 , 7 ) = 7
                               a0.integrity = Lower_Bound( 0 , 0 ) = 0

         li    a7, 1           # a7 = 1
                               a7.Security = 7, a7.Integrity = 0

         li    a6, 0x63        # LAST = R^2 -1 = 100 - 1
                               Tags Security = 7, Integrity = 0

         li    t1, 0x8         # t1 = R = 10
                               Tags Security = 7, Integrity = 0

          j      <L1>          # jump to <L1>
                               Tags Security = 7, Integrity = 0

<L3>     addi  a2, a2, 1       # a2 = base = base +1
Data Flow Integrity            constant.security = 7, constant.integrity = 0
Information Leak               a5.security = 7, a5.integrity = 0 (if a5 = bit = 0)
                               a2.security = 7, a2.integrity = 0
                               a2.security  = Upper_Bound( 7 , 7, a5) = 7
                               a2.integrity = Lower_Bound( 0 , 0, a5 ) = 0

<L1>     srai  a5, a2 ,0x5     # a5 = word offset
                               a5.security  = Upper_Bound( a2 , 7 ) = 7
                               a5.integrity = Lower_Bound( a2 , 0 ) = 0

         slli  a5, a5, 0x2     # a5 = byte offset [note 1]
                               a5.security  = Upper_Bound( a5 , 7 ) = 7
                               a5.integrity = Lower_Bound( a5 , 0 ) = 0

         add   a5, a0, a5      # a5 = pb[0] + byte offset
                               a5.security  = Upper_Bound( a0 , 7 ) = 7
                               a5.integrity = Lower_Bound( a0 , 0 ) = 0

         lw    a5, 0(a5)       # a5 = LW(addr = a5)
                               constant.security = 7, constant.integrity = 0
                               Read_Mem(0+a5).security = tag.security = 7
                               Read_Mem(0+a5).integrity = tag.integrity = 0
                               a5.security  = Read_Mem(0+a5).security) = 7
                               a5.integrity = Read_Mem(0+a5).integrity) = 0

         sra   a5, a5, a2      # a5 = a5 >> a2 [note 2]
                               a5.security  = Upper_Bound( a5 , a2 ) = 7
                               a5.integrity = Lower_Bound( a5 , a2 ) = 0

         andi  a5, a5, 1       # a5 = pb[word, bit number]
                               a5.security  = Upper_Bound( a5 , 7 ) = 7
                               a5.integrity = Lower_Bound( a5 , 0 ) = 0

         beqz  a5, <L3>        # if a5 = bit = 0 the <L3>
                               Tags Security = 7, Integrity = 0
```

Information Leak

Fig. 33 Sieve of Eratosthenes data flow integrity

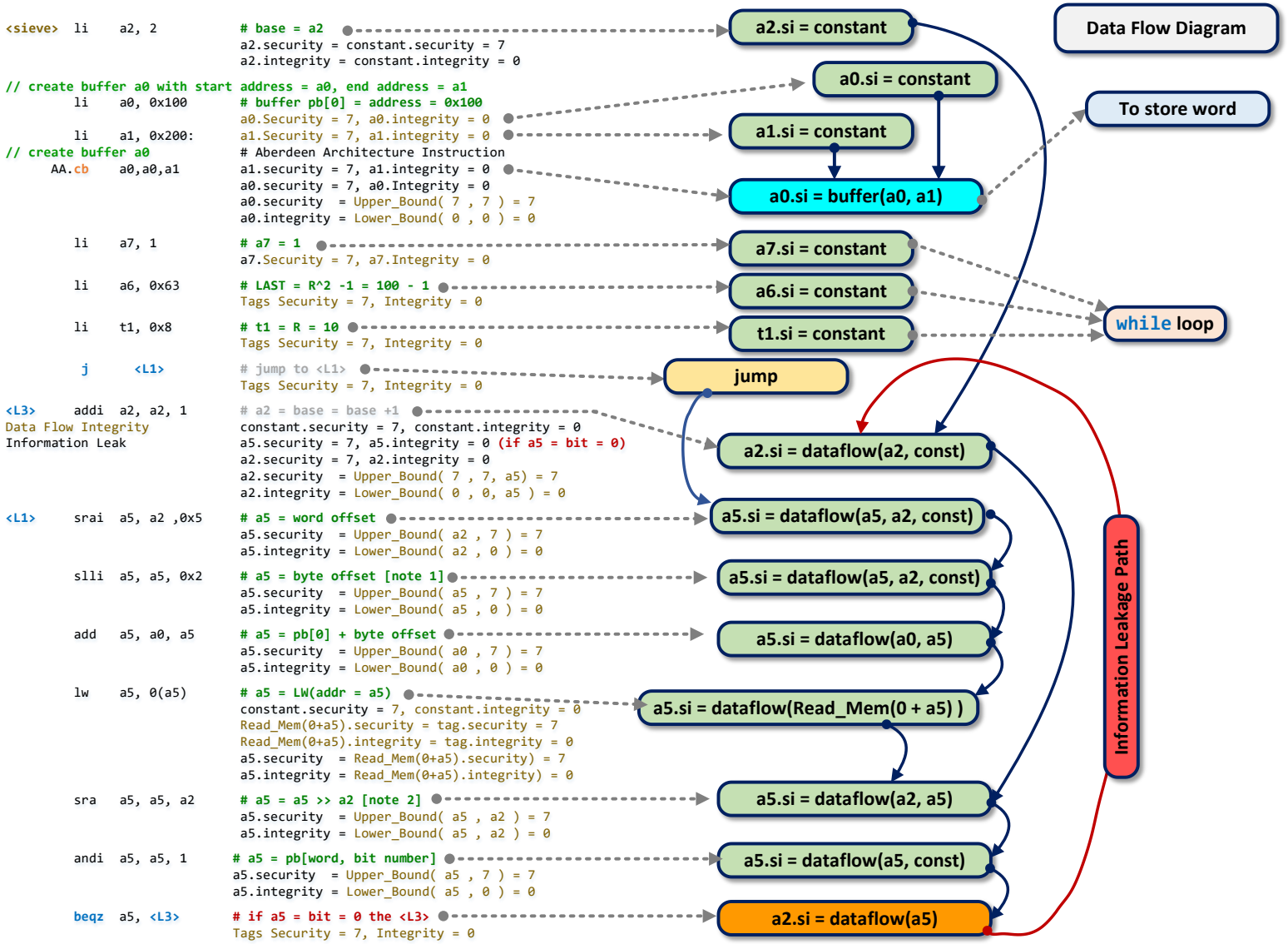


Fig. 34 Partial data flow diagram for Sieve of Eratosthenes

4.3.11 System Architecture

The Aberdeen Architecture provides complete mediation for instruction execution. Hardware-level security policies are enforced by state machine monitors in Fig. 35. Data flow integrity, control flow integrity, and memory access policies are verified during instruction execution. Memory access policy verifies load/store and stack memory operations. Aberdeen Architecture uses multiple stacks to isolate control information (`CALL`, `RETURN`, etc.) from "data". Data and control information stack mixing is not allowed. The state machine security policies located at security level `0` (most secure) are the trusted computing base. Memory integrity is more secure than executing code. A memory access violation will raise a hardware-level exception.

Instruction execution monitor relies on the data flow integrity monitor, control flow integrity monitor, and memory access monitor. Memory access monitor manages load/store operations, stack operations, and memory page operations. More OS relevant functions, process context switch, interrupt handler, and exception handler are included for instruction execution.

Hardware-level security policies are defined for levels `0`–`0.9`. Instruction execution occurs at security level `1`. Guest operating systems reside at security level `2`. Application software is placed at security levels `3` and above.

Level `0` is the hardware state machine monitors. We could use a configuration file at policy level `0.1` to configure some of the state machine settings. Policy level `0.2` could define scratchpad memory for the state machines. Level `0.5` defines the memory access policies. Memory access integrity is more important than instruction execution. Numerous cyber researchers point to memory manipulation as a common attack vector. A hardware hypervisor could be placed at security level `0.7`.

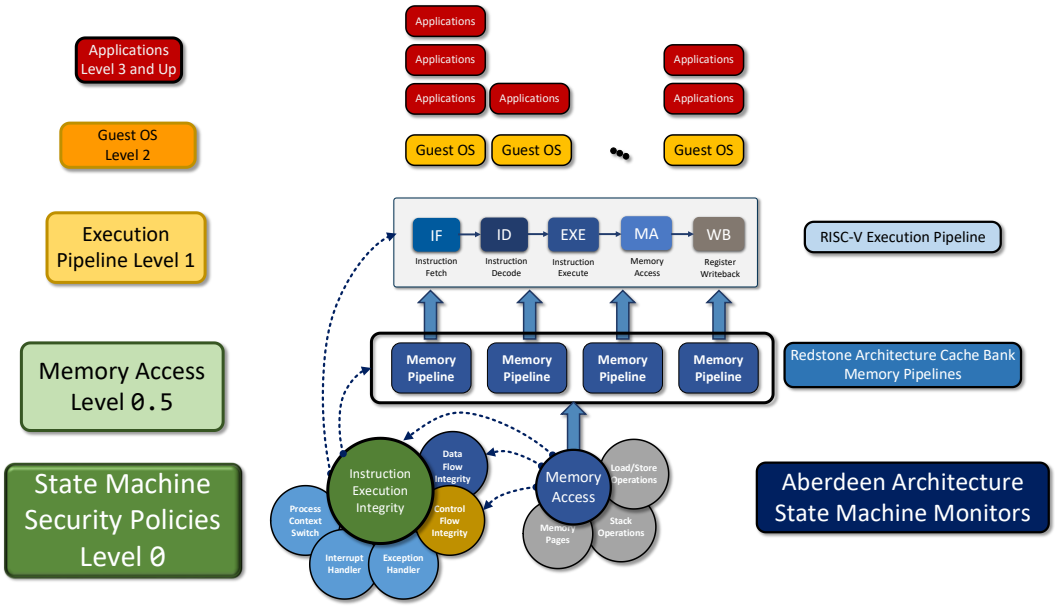


Fig. 35 Aberdeen architecture security levels. Hardware state machine monitors enforce security policies. Memory access policy is below the execution pipeline. Execution Pipeline cannot change memory policy. Execution Pipeline sits at security level 1. Guest OS and Applications software are at less secure levels.

4.4 Aberdeen Architecture State Machine Monitors

In this section, instruction execution is explained in several steps. The control flow graph forms the global structure for a running process. The data flow graph describes navigating the control flow graph and local level of information flow. The data flow subgraphs describe local information flows covering information sources, data processing, and information sinks. Instruction execution follows allowed control flow graph paths and data flow graph paths. As described in Section 4.3.3, there are four classes of instructions. The four instruction classes each have a set of allowed operations. The instruction classes allow Saltzer and Schroeder's security principles to be applied to instruction execution. We begin by describing basic instruction execution and build our way up to the complete Aberdeen Architecture.

4.4.1 Instruction Execution

In a conventional microprocessor, the number of combinations of instructions makes formal statements difficult (exponential growth). For the Aberdeen Architecture, a framework for formal proofs limits the number of combinations to a practical number. For the Aberdeen Architecture, there are four possible instruction classes for the previous instruction, four classes for the current instruction, and four classes for the next instruction. The Aberdeen Architecture

has a total of $4^3 = 64$ cases to consider. As illustrated in Table 1, data flow determines the instruction execution path on a control flow graph.

4.4.1.1 Sequential Instruction Execution Classes

Figure 36 illustrates sequential instruction class execution. If the previous instruction was a sequential or load/store sequential instruction, the program counter is advanced by one instruction to the currently executing instruction. If the previous instruction was a conditional branch, there are two possible paths to reach the currently executing sequential instruction. If the branch condition evaluates to true, the next instruction is found at the branch destination address (program counter is loaded with the branch destination address). If the branch condition evaluates to false, the next instruction is found by advancing the program counter by one instruction (`PC = PC + 1` instruction, just like the sequential instruction class). If the previous instruction was a jump instruction, the next instruction is the jump destination address (PC = jump destination address).

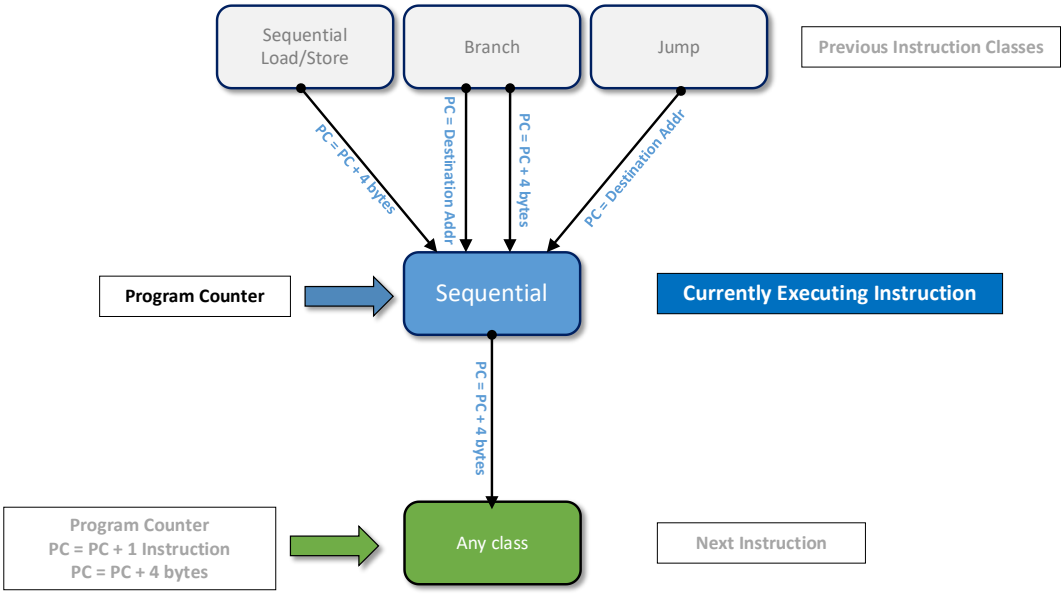


Fig. 36 Sequential and load/store sequential instruction execution

The instruction flow in Fig. 36 can be used to develop a proof-by-induction. For a proof-by-induction, we show the instruction execution properties hold for $n = 2$ instructions, and then show the instruction execution properties hold for $n = k + 1$ instructions. For a more complete proof of high assurance, several protection properties need to be demonstrated: data flow integrity, stack and memory behavior, and register behavior.

Figure 37 shows part of the Sieve of Eratosthenes **<L1>** code block. There are several sequential class instructions in a row. The instruction execution tag maps to the **EXE** field found in Fig. 29. The bottom part of the figure illustrates some of the properties verified during execution of **lw a5, 0(a5)** instruction. Memory access flow tag is set to **LOAD**. This tag allows memory read operation. Data flow integrity checks are completed during instruction execution (see Instruction Execution block). Each memory address also has data flow integrity tags that are completely isolated from data.

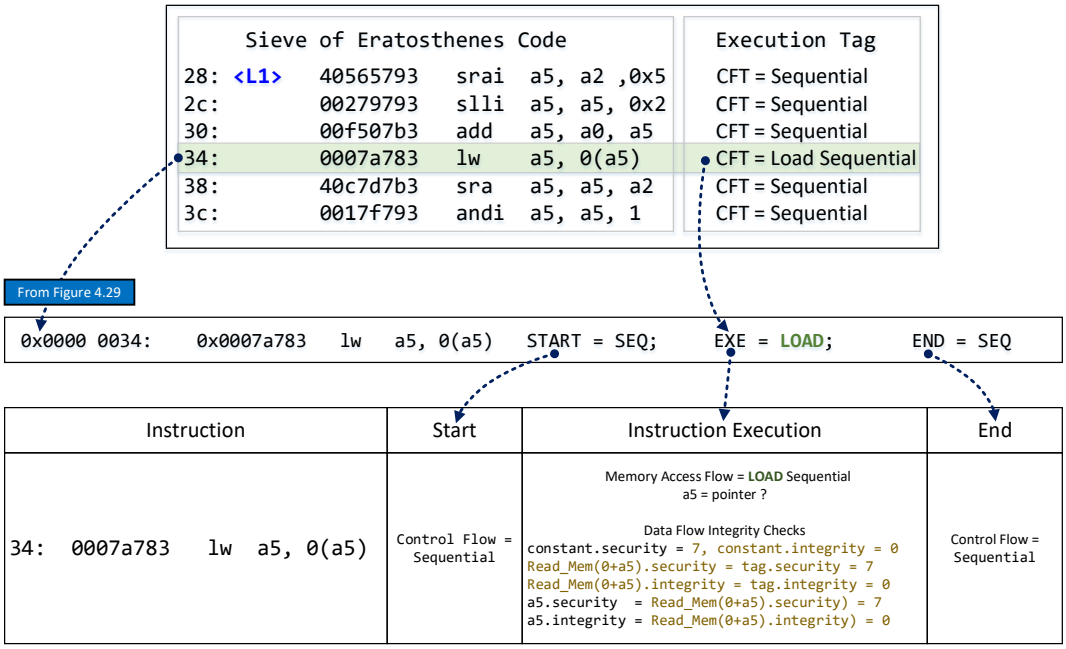


Fig. 37 LOAD sequential instruction execution

4.4.1.2 Branch Instruction Execution Class

The branch instruction execution class is illustrated in Fig. 38. For the previous instruction, there are four possible instruction classes. The branch instruction has two possible next instructions. For condition = **TRUE**, the next instruction is located at the branch destination address (PC = destination address). For condition = **FALSE**, the next instruction is the same as the sequential execution class (PC = PC + 1 instruction). Figure 39 shows a code block for `beqz a5, 20 <L3>` from Fig. 29 and control flow links for `begz` branch instruction class. The START control flow tags shows the previous instruction class was sequential. The execution tag shows the instruction class is branch. The END tag control flow links show that the two possible next instruction classes are sequential or branch.

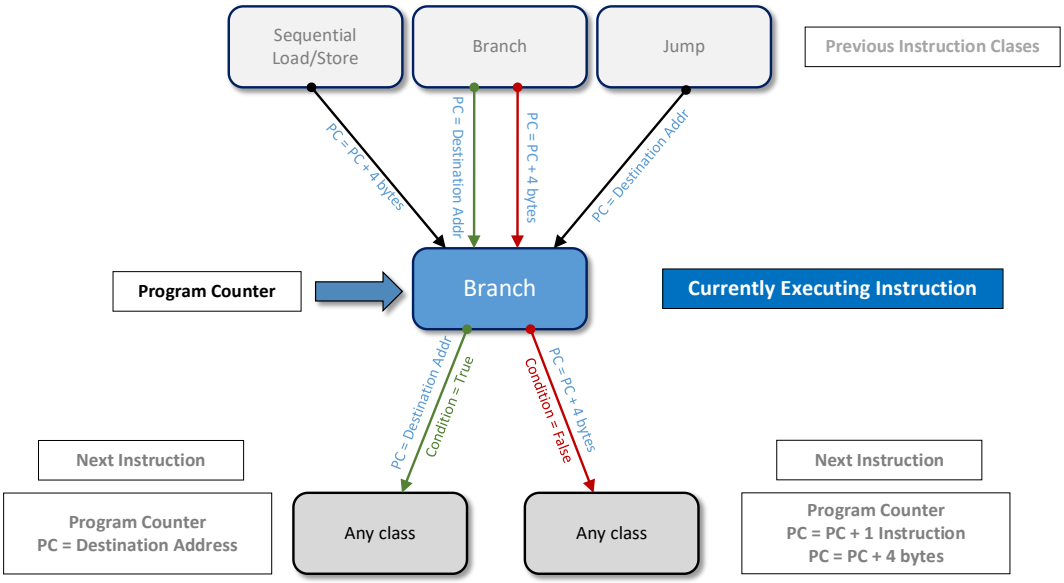


Fig. 38 Branch instruction execution

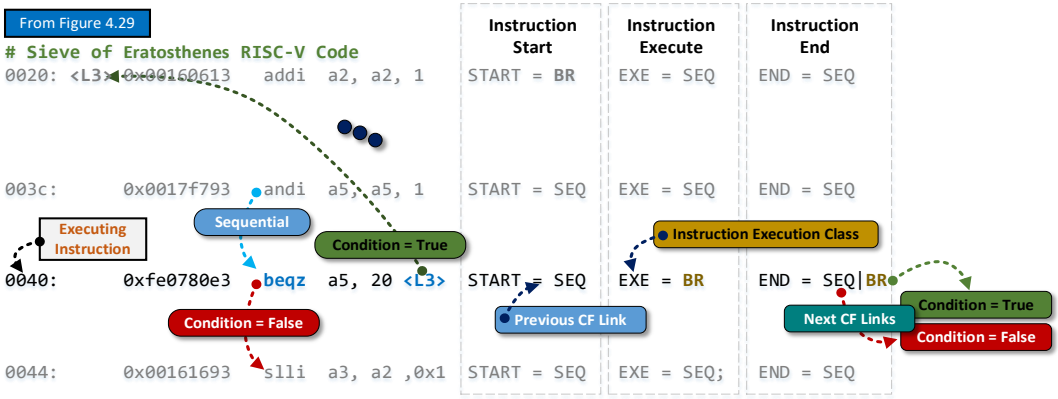


Fig. 39 Branch instruction execution example

4.4.1.3 Jump Instruction Execution Class

The jump instruction execution class is similar to the branch instruction. As shown in Fig. 40, the jump instruction class only has one next instruction: the destination address. Figure 41 shows the control flow link tags and instruction execution tag for a jump instruction.

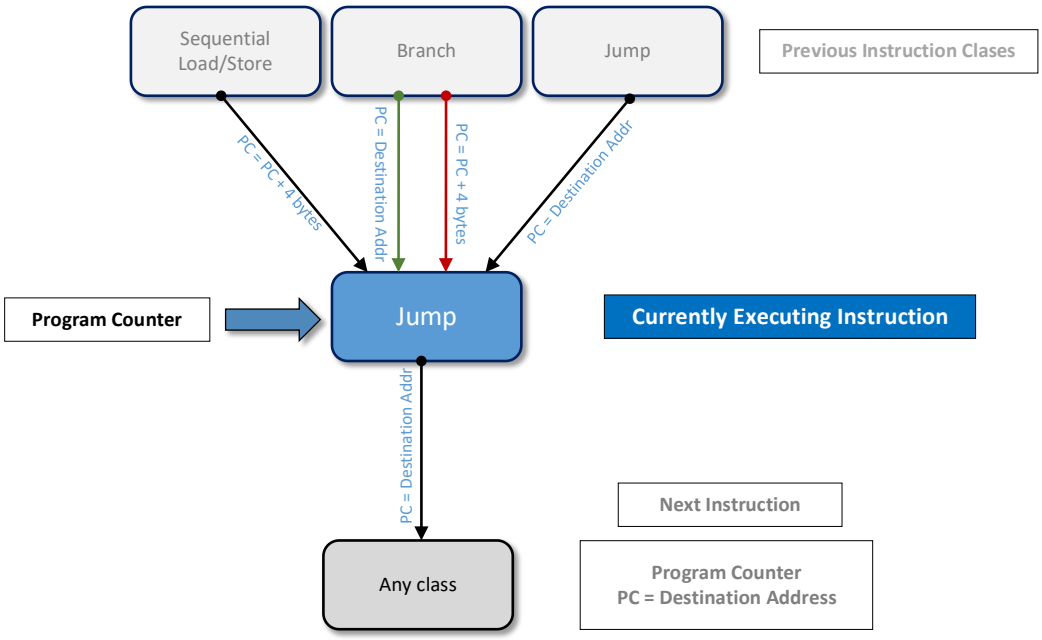


Fig. 40 Jump instruction execution

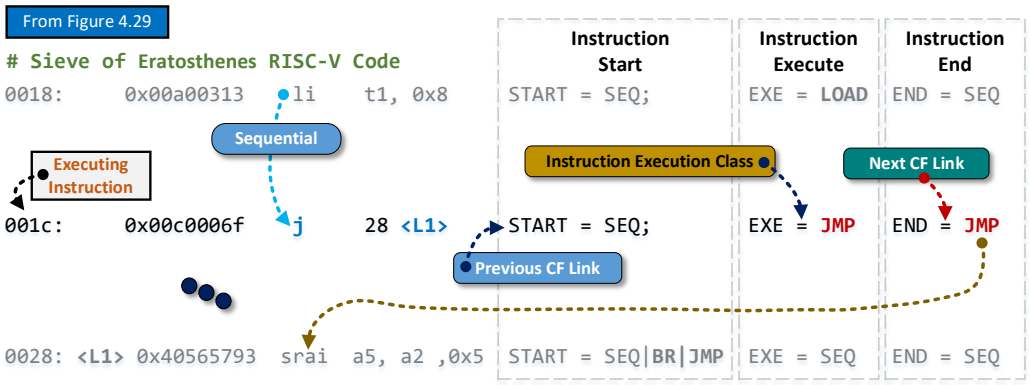


Fig. 41 Jump instruction execution example

4.4.1.5 Stack Operations

Stack and memory protections are required to strengthen control flow and data flow integrity protections. A control flow graph does not consider stack behavior (Abadi et al. 2005). Control flow and stack both must be protected. Abadi et al. note, “Of course, CFI enforcement is not a panacea: exploits within the bounds of the allowed CFG (e.g., Chen et al. [2005]) are not prevented.” Return-oriented programming (ROP) is a common stack attack, and CFI fails to block it. An ROP attack corrupts the stack and maliciously modifies the return address for an executing function (Göktaş, et al. 2014). Current software architectures combine data and control information on the *same* stack. This design philosophy violates Saltzer and

Schroeder's security principles of least privilege, privilege separation, and complete mediation. Current stack implementations suffer the same isolation issues found in a von Neumann machine. All control information *must* be completely isolated from data. To isolate control and data, separate stacks are required similar to instruction and data isolation provided by a Harvard architecture.

Figure 42 illustrates stack state machine operations (Jungwirth 2020b). Control flow operations are saved on the **EXE_STACK**. Data is placed on **DATA_STACK**. The two stacks completely isolate control information from data. A malicious data stack operation cannot overwrite or modify control flow information on the control flow stack. A process ID (PID) provides a second level of isolation. Each process has its own **EXE_STACK** and **DATA_STACK**. This provides a second level of isolation for stack information.

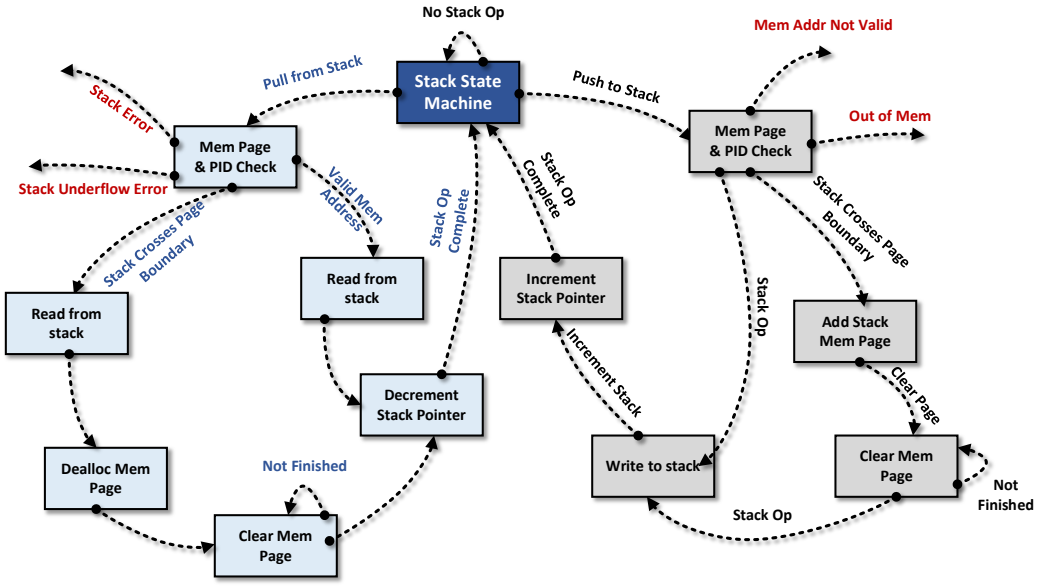


Fig. 42 Stack push and pull operations

4.4.1.6 Memory Page Operations

Memory access and page operations are presented in Sections 4.3.4, 4.3.5, 4.3.8, and 4.3.9; and Jungwirth et al. (2019b) and Jungwirth and Ross (2019). I/O and data page operations are presented in Jungwirth et al. (2019b).

4.4.2 State Machine Monitors Introduction

The Aberdeen Architecture's state machine monitors are the trusted computing base. State machine monitors interpret the security tag bits during instruction execution. Each instruction class has a set of allowed and prohibited operations. The security tag bits define the boundaries between allowed and prohibited

operations. As each instruction is executed, the security tags define the limits for control flow behavior, memory access operations, data flow behavior, and instruction execution. Instructions that violate one or more of the security properties will raise a hardware-level exception. Section 4.5 describes the hardware-level security policies enforced by the state machine monitors in detail.

Additional information covering control flow integrity is found in the paper "Security Tag Fields and Control Flow Management" (Jungwirth and Ross 2019). Data flow integrity for the Redstone Architecture is covered in the paper "Security Tag Computation and Propagation in OSFA" (Jungwirth et al. 2018b). Redstone Architecture's instruction execution and page memory management is covered in "Hardware Security Kernel for Cyber-Defense" (Jungwirth et al. 2019b) and "Cyber Defense through Hardware Security" (Jungwirth et al. 2018a). Aberdeen Architecture builds on the control flow and data flow integrity ideas for the Redstone Architecture.

4.5 State Machine Monitors

Aberdeen Architecture's state machine monitors are the trusted computing base. Aberdeen Architecture's data flow integrity was presented in section 4.3.10, Figs. 33 and 34. Control flow integrity is presented in Section 4.3.9. Both data flow integrity and control flow integrity are scalable. More field labels for control flow integrity and data flow integrity provide higher levels of precision. Memory page integrity monitor is covered in Section 4.3.4. Instruction execution integrity is found in Sections 4.3 and 4.4. Following the software description philosophy in Section 4.3.2, we present the RISC-V Aberdeen Architecture version of the Sieve of Eratosthenes in Section 4.5.1.

Monitors are presented in reverse order (Data Flow Integrity, Control Flow Integrity, Memory Page Monitor, and Instruction Execution Monitor) to build up to instruction execution class. Instruction execution is a function of all information flow classes. Section 4.5 finishes up with an introduction to "OS" support function monitors: scheduler, interrupt handler, and exception handler.

4.5.1 RISC-V Aberdeen Architecture version of the Sieve of Eratosthenes

Figure 43 introduces instruction execution for the Aberdeen Architecture. The control flow diagram for the Sieve of Eratosthenes code is found in Fig. 24. Control flow integrity is validated by verifying the execution path that follows the control flow security tags. Figure 29 shows the instruction execution tags. The previous instruction's **END** tag is the same as the current instruction's **START** tag. The **EXE** tags ensure that the proper control flow path is followed.

In Fig. 43, the load immediate instruction, `li a7, 1`, is a sequential class instruction (see Fig. 36). The instruction loads the constant `1` into register `a7`. The data memory access type is `load immediate`. The data integrity flow tags for a constant are security = `7` (lowest security) and integrity = `0` (highest integrity). The Aberdeen Architecture uses the register tags from the Redstone Architecture to track data flow integrity through register calculations, and memory read and write operations. When a value is read into a register, the memory word tag fields are assigned to the register. When a value is stored to memory, the register tag values are saved with the memory word.

Single point entry and exit points are shown in Fig. 44. The single entry and exit points significantly reduce the chances of a gadget attack against code inside the single entry and exit points. The control flow tags for code inside the single entry and single exit blocks also block function calls. The two simple control flow mechanisms, single entry/exit points, and control flow tags provide a high degree of protection without a high cost (memory and chip area). Control flow protections are known to lack coverage for stack operations. Aberdeen Architecture also includes memory protections for stacks, data, executable code, and I/O. The overlapping state machines' protections provide the whole is greater than the sum of the parts level of protection.

Instruction execution in Fig. 43 shows the interaction of all the state machines to implement Saltzer and Schroeder's complete mediation principle (verification of operations and authority) for instruction execution. Complete mediation for instruction execution requires high precision for protections. The Aberdeen Architecture takes advantage of the overlapping protections provided by lower precision protection mechanisms. Partial complete mediation is a practical level of mediation suitable for an actual implementation. The precision level is scalable; more security tags provide greater precision. Here we are interested in a balance between protection cost (memory and circuits) and protection level. Figure 45 presents the ranges for mediation. Complete mediation verifies operations and authorities without any ambiguities (highest level of precision). Complete mediation is not practical for all applications. In addition, when using multiple protection mechanisms with moderate levels of precision, an approximation to complete mediation is possible. Near complete mediation reduces the ambiguities to a small level where the available attack vectors are difficult to nearly impossible to exploit. For computer code running on the Aberdeen Architecture, the multiple overlapping protection mechanism for near-complete mediation can be greater than the sum of the parts and provide a practical implementation for complete instruction mediation.

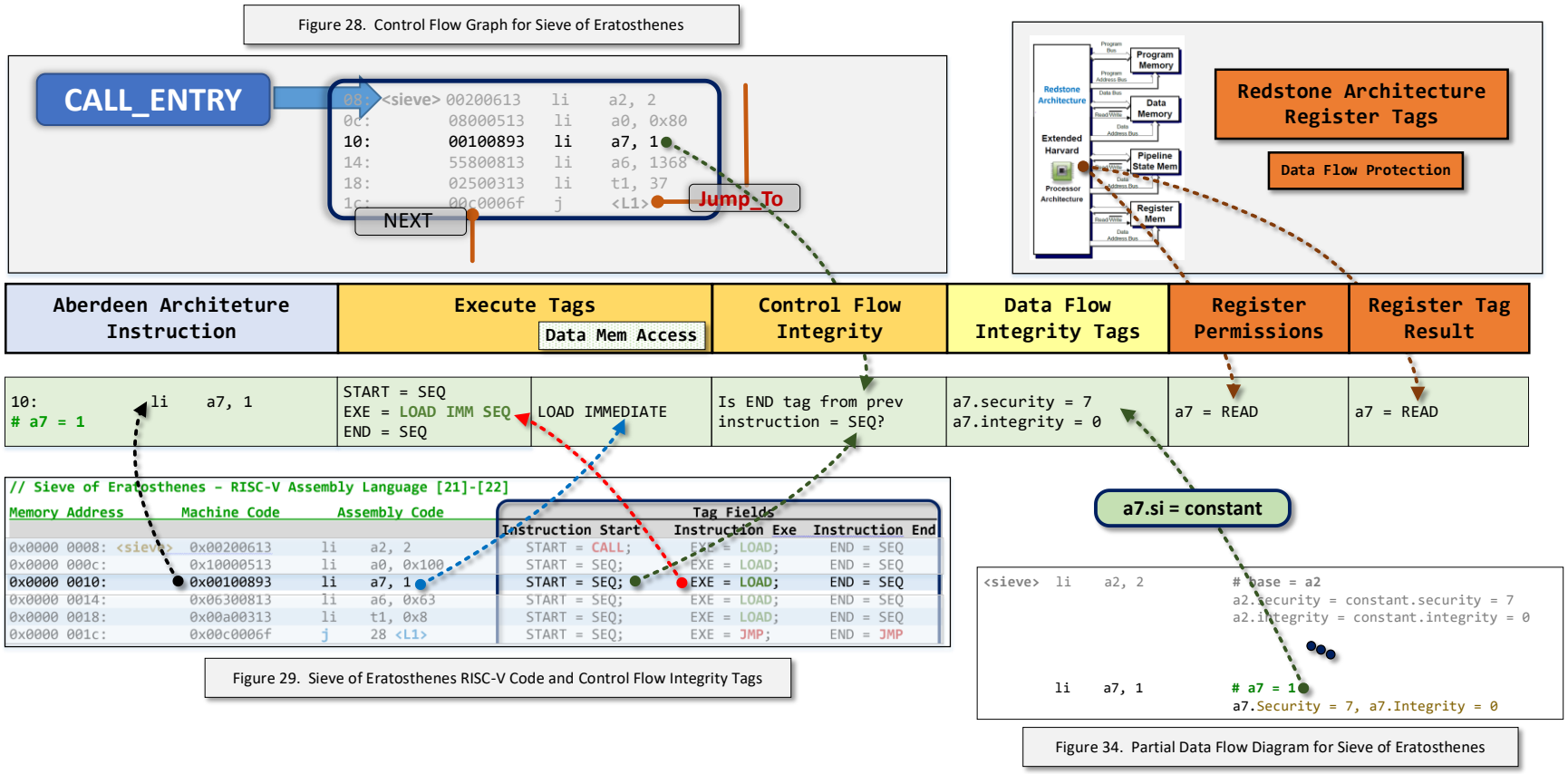


Fig. 43 Aberdeen Architecture RISC-V Sieve of Eratosthenes code instruction execution example

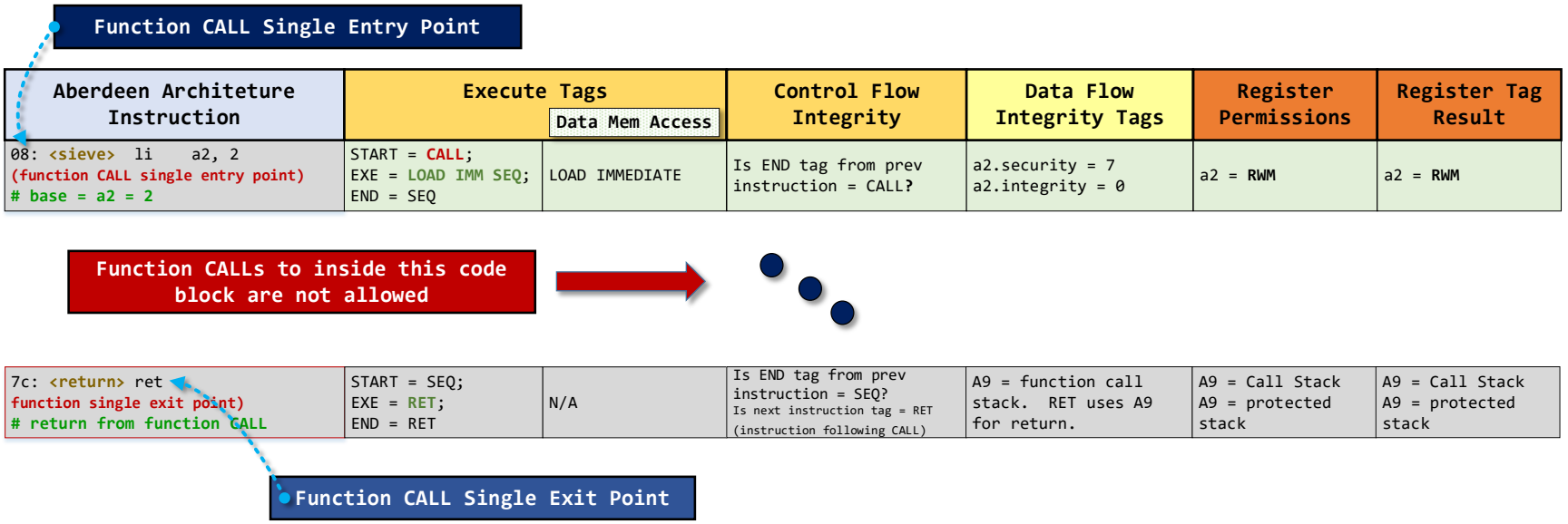


Fig. 44 Sieve of Eratosthenes single entry and exit code points

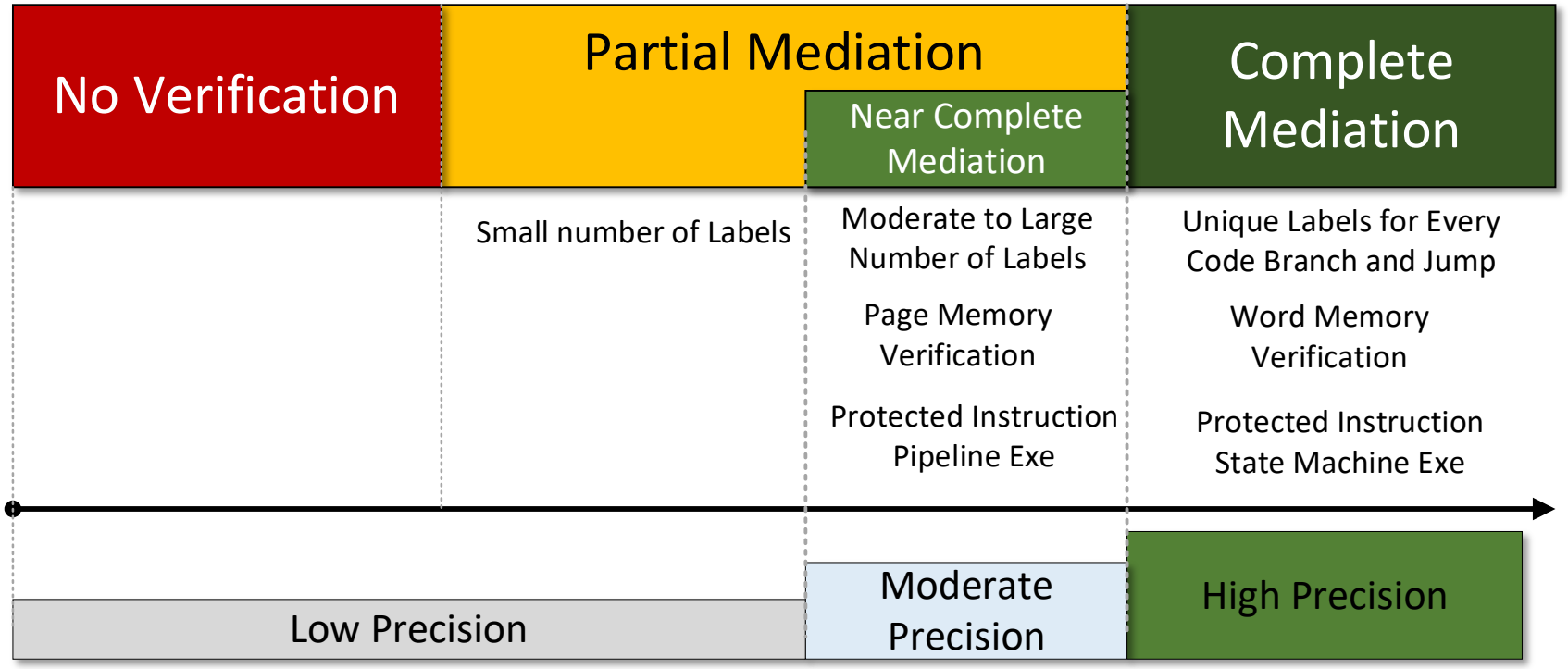


Fig. 45 Partial, near-complete, and complete mediation ranges

The Aberdeen Architecture uses the register and memory tags from the Redstone Architecture to implement protected buffers. In Fig. 46, Aberdeen Architecture protected state machine "monitor call" instruction **AA.Create_Buffer** creates a protected pointer to a buffer. An example of a protected pointer was introduced in Fig. 44. A memory state machine controller creates the protected pointer. The register tags are set to an array pointer. An array pointer cannot be "read" by the running program. A pointer may be copied to other registers; however, the new registers are upgraded to an array pointer. A register without the register tag "array pointer" cannot be used to read or write to memory. Section 4.3.4 presented an introduction to the Aberdeen Architecture memory map.

An Aberdeen Architecture **JUMP** instruction is illustrated in Fig. 47. The control flow path from the previous instruction is shown. The **JUMP** instruction has one possible next instruction: the jump destination address. The control flow graph for the **<L1>** instruction can accept control flow tags **SEQUENTIAL**, **BRANCH**, and **JUMP**. A larger number of control flow labels can improve control flow precision. The Aberdeen Architecture conditional branch instruction is shown in Fig. 48.

LOAD and STORE memory instructions for the Aberdeen Architecture are presented in Fig. 49. The LOAD instruction reads and STORE instruction writes a memory word pointed to by a register tag = pointer register. For arrays or memory words, a protected register prevents misuse of read and write operations. To read or write to memory, the running process ID must match the process ID tag for the memory page. Shared memory pages support multiple processes sharing memory. Each memory address has memory tags. A read operation sets the register tags to the stored memory tags. A write operation saves the memory word and register tags. The memory tags provide for data flow integrity.

The complete Sieve of Eratosthenes RISC-V program for the Aberdeen Architecture is presented in Fig. 50. Control flow, data flow, memory access flow, and instruction execution are presented. Security tag fields are completely isolated from the executing program. Security tag fields are created by parsing a binary or high-level language to generate control flow, data flow, and memory access patterns.

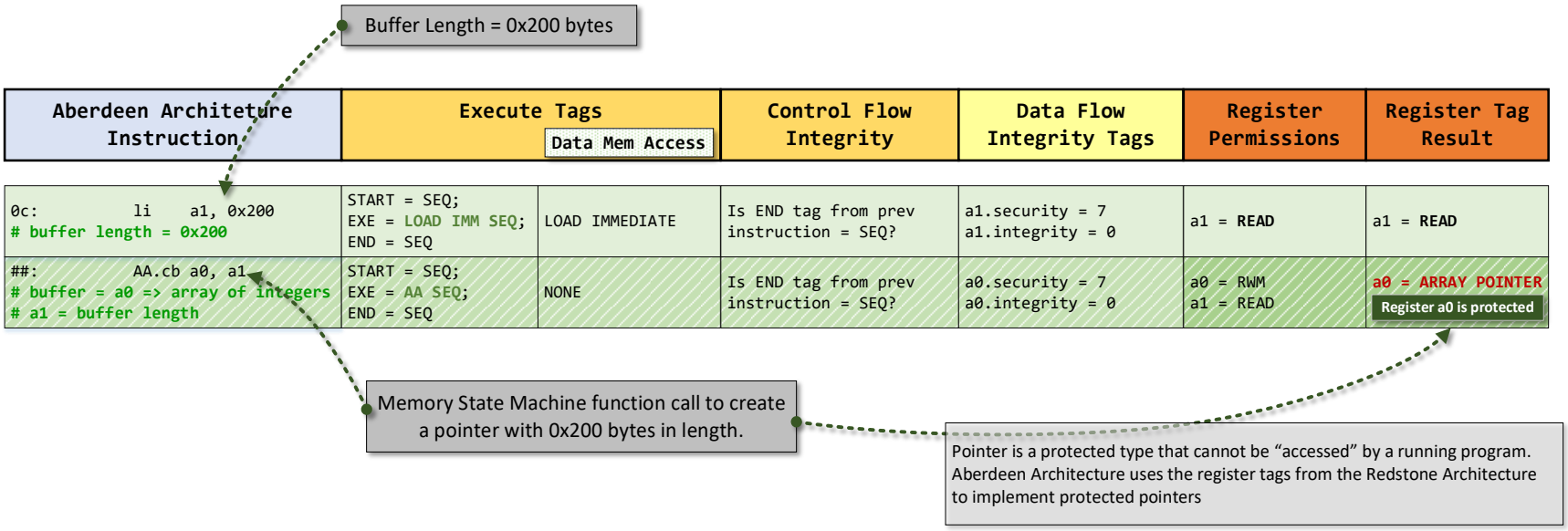


Fig. 46 Aberdeen Architecture creates protected buffer

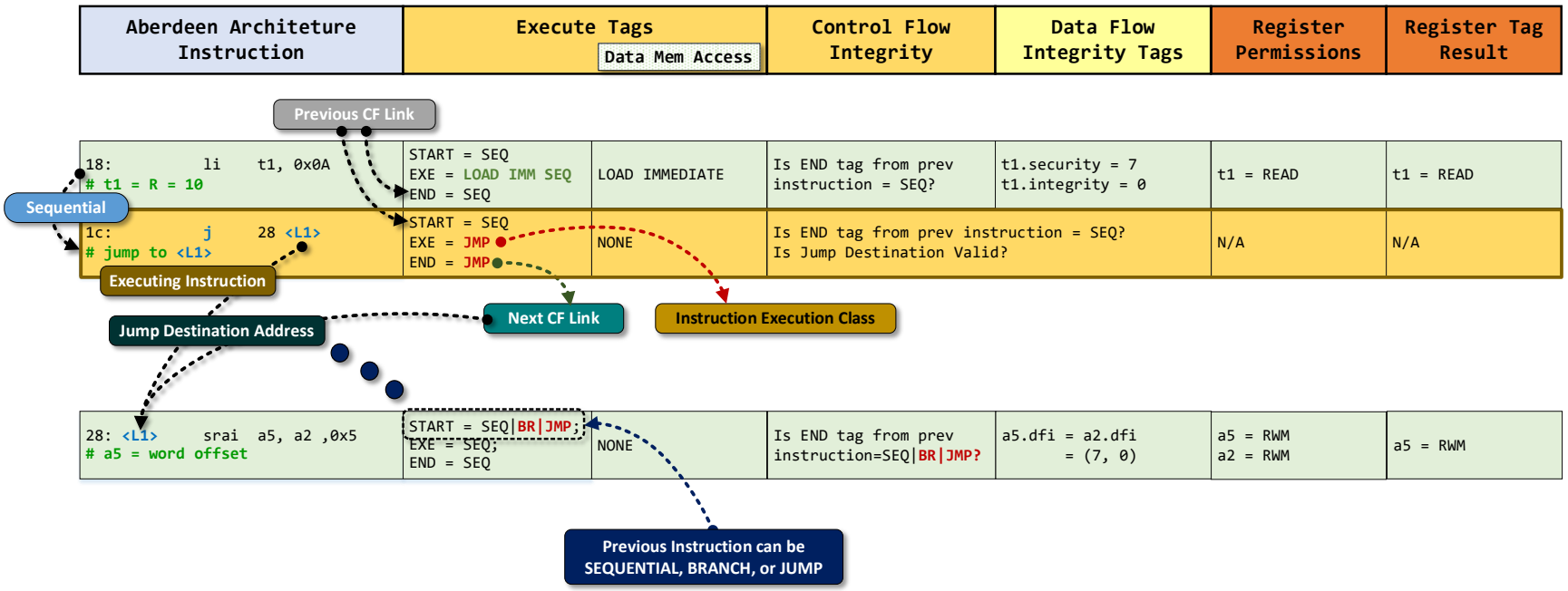


Fig. 47 Aberdeen Architecture JUMP instruction execution

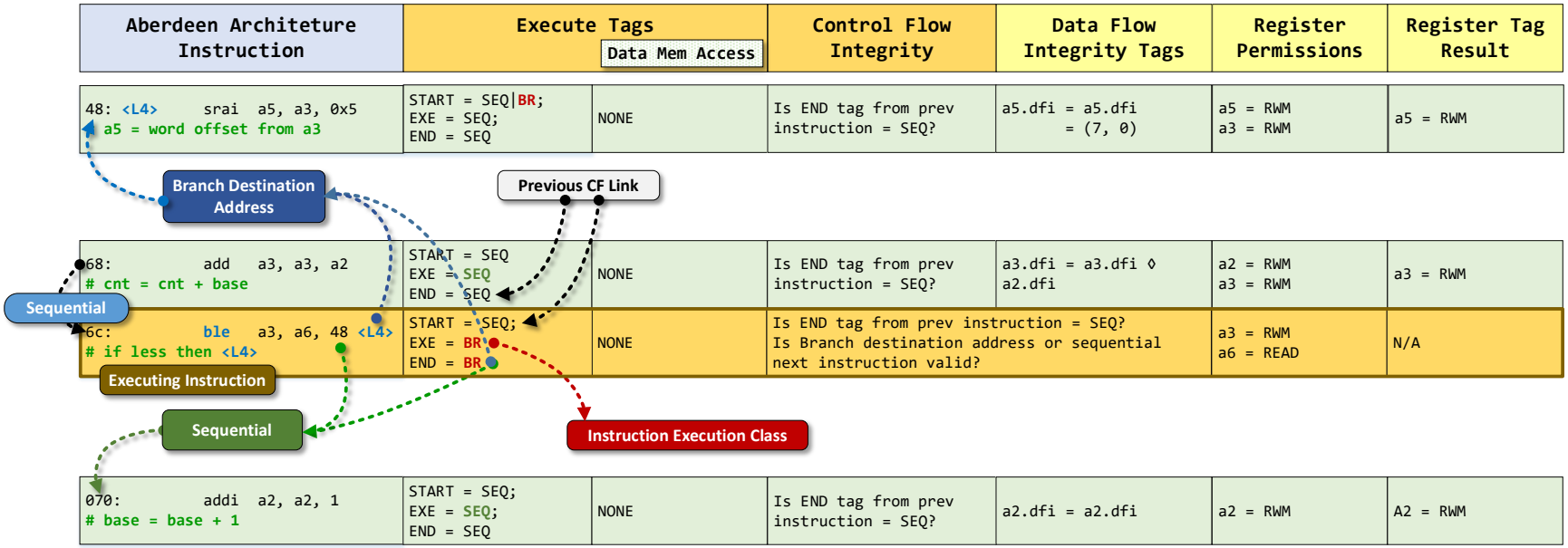


Fig. 48 Aberdeen Architecture conditional branch

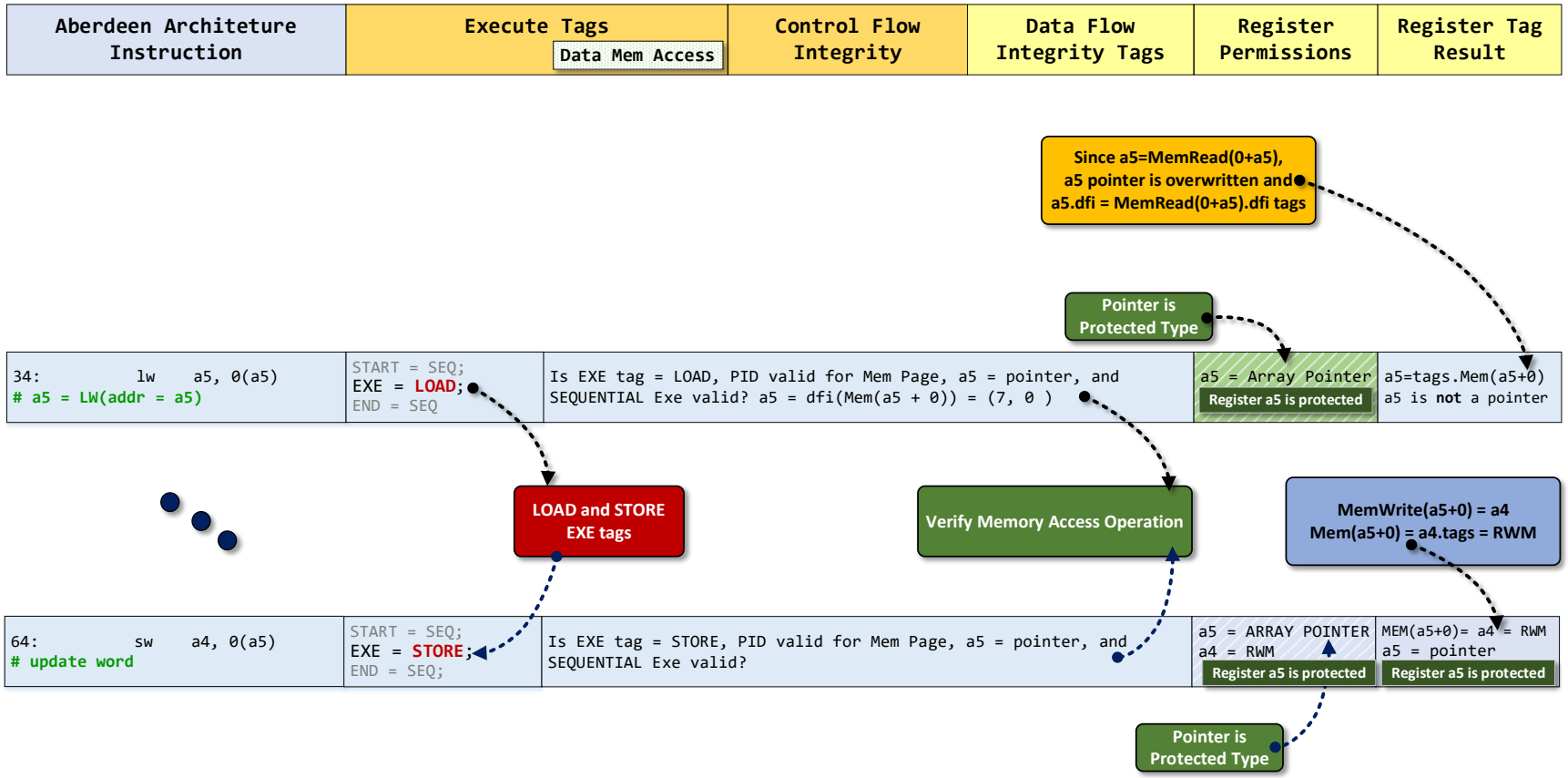
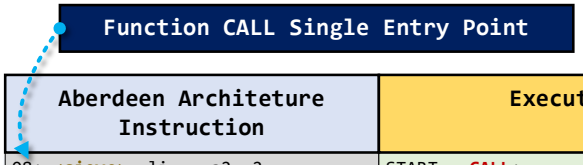


Fig. 49 Aberdeen Architecture LOAD and STORE memory instructions



Aberdeen Architeture Instruction	Execute Tags	Data Mem Access	Control Flow Integrity	Data Flow Integrity Tags	Register Permissions	Register Tag Result
08: <sieve> li a2, 2 (function CALL single entry point) # base = a2 = 2	START = CALL; EXE = LOAD IMM SEQ; END = SEQ	LOAD IMMEDIATE	Is END tag from prev instruction = CALL?	a2.security = 7 a2.integrity = 0	a2 = RWM	a2 = RWM
0c: li a1, 0x200 # buffer length = 0x200	START = SEQ; EXE = LOAD IMM SEQ; END = SEQ	LOAD IMMEDIATE	Is END tag from prev instruction = SEQ?	a1.security = 7 a1.integrity = 0	a1 = READ	a1 = READ
##: AA.cb a0, a1 # buffer = a0 => array of integers # a1 = buffer length	START = SEQ; EXE = AA SEQ; END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a0.security = 7 a0.integrity = 0	a0 = RWM a1 = READ	a0 = ARRAY POINTER Register a0 is protected
10: li a7, 1 # a7 = 1	START = SEQ EXE = LOAD IMM SEQ END = SEQ	LOAD IMMEDIATE	Is END tag from prev instruction = SEQ?	a7.security = 7 a7.integrity = 0	a7 = READ	a7 = READ
14: li a6, 0x63 # LAST = R^2 - 1 = 100 - 1	START = SEQ EXE = LOAD IMM SEQ END = SEQ	LOAD IMMEDIATE	Is END tag from prev instruction = SEQ?	a6.security = 7 a6.integrity = 0	a6 = READ	a6 = READ
18: li t1, 0x0A # t1 = R = 10	START = SEQ EXE = LOAD IMM SEQ END = SEQ	LOAD IMMEDIATE	Is END tag from prev instruction = SEQ?	t1.security = 7 t1.integrity = 0	t1 = READ	t1 = READ
1c: j 28 <L1> # jump to <L1>	START = SEQ EXE = JMP END = JMP	NONE	Is END tag from prev instruction = SEQ? Is Jump Destination Valid?		N/A	N/A
20: <L3> addi a2, a2, 1 # a2 = base = base +1	START = BR; EXE = SEQ; END = SEQ	NONE	Is END tag from prev instruction = BR?	a2.dfi = a2.dfi = (7, 0)	a2 = RWM	a2 = RWM
24: beq a2, t1, 78 <L2> # if base = R then <L2> Done	START= SEQ; EXE = BR; END = SEQ|BR	NONE	Is END tag from prev instruction = SEQ? Is Branch destination address or sequential next instruction valid?		a2 = READ t1 = READ	N/A
28: <L1> srai a5, a2 ,0x5 # a5 = word offset	START = SEQ|BR|JMP; EXE = SEQ; END = SEQ	NONE	Is END tag from prev instruction=SEQ|BR|JMP?	a5.dfi = a2.dfi = (7, 0)	a5 = RWM a2 = RWM	a5 = RWM
2c: slli a5, a5, 0x2 # a5 = byte offset [note 1]	START = SEQ; EXE = SEQ END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a5.dfi = a5.dfi = (7, 0)	a5 = RWM	a5 = RWM

Fig. 50 Aberdeen Architecture RISC-V Sieve of Eratosthenes code

Aberdeen Architeture Instruction	Execute Tags	Data Mem Access	Control Flow Integrity	Data Flow Integrity Tags	Register Permissions	Register Tag Result
30: add a5, a0, a5 # a5 = pb[0] + byte offset	START = SEQ; EXE = SEQ; END = SEQ	none	Is SEQUENTIAL Execution Valid?	a5.dfi = a0.dfi = (7, 0)	a5 = RWM a0 = array pointer Register a0 is protected	a5 = Array Pointer Register a5 is protected
34: lw a5, 0(a5) # a5 = LW(addr = a5)	START = SEQ; EXE = LOAD; END = SEQ	Is EXE tag = LOAD, PID valid for Mem Page, a5 = pointer, and SEQUENTIAL Exe valid? a5 = dfi(Mem(a5 + 0)) = (7, 0)			a5 = Array Pointer Register a5 is protected	a5=tags.Mem(a5+0) a5 is **not** a pointer
38: sra a5, a5, a2 # a5 = a5 >> a2 [note 2]	START = SEQ; EXE = SEQ END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a5.dfi = a5.dfi ◊ a2.dfi = (7, 0)	a5 = RWM a2 = RWM	a5 = RWM
3c: andi a5, a5, 1 # a5 = pb[word, bit number]	START = SEQ; EXE = SEQ END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a5.dfi = a5.dfi = (7, 0)	a5 = RWM	a5 = RWM
40: beqz a5, 20 <L3> # if a5 = bit = 0 the <L3>	START= SEQ; EXE = BR; END = SEQ|BR	NONE	Is END tag from prev instruction = SEQ? Is Branch destination address or sequential next instruction valid?		a5 = **RWM**	N/A
44: slli a3, a2 ,0x1 # a3 = cnt = base + base	START = SEQ; EXE = SEQ END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a3.dfi = a2.dfi = (7, 0)	a3 = RWM a2 = RWM	a3 = RWM
48: <L4> srai a5, a3, 0x5 # a5 = word offset from a3	START = SEQ|BR; EXE = SEQ; END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a5.dfi = a5.dfi = (7, 0)	a5 = RWM a3 = RWM	a5 = RWM
4c: slli a5, a5, 0x2 # a5 = byte offset	START = SEQ; EXE = SEQ; END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a5.dfi = a5.dfi = (7, 0)	a5 = RWM	a5 = RWM
50: add a5, a0, a5 # a5 = pb[0] + byte offset	START = SEQ; EXE = SEQ; END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a5.dfi = a5.dfi ◊ a0.dfi = (7, 0)	a5 = RWM a0 = Array Pointer Register a0 is protected	a5 = Array Pointer Register a5 is protected
54: lw a1, 0(a5) # a1 = LW(addr = a5 + 0)	START = SEQ; EXE = LOAD MEM; END = SEQ	Is EXE tag = LOAD, PID valid for Mem Page, a5 = pointer, and SEQUENTIAL Exe valid?			a1 = RWM a5 = ARRAY POINTER Register a5 is protected	a1 = ReadMem(0+a5) a5 = Array Pointer Register a5 is protected

Fig. 50 Aberdeen Architecture RISC-V Sieve of Eratosthenes code (continued)

Aberdeen Architeture Instruction	Execute Tags	Data Mem Access	Control Flow Integrity	Data Flow Integrity Tags	Register Permissions	Register Tag Result
58: sll a4, a7, a3 # a4 = 1 << cnt = 00•••1•••000	START = SEQ; EXE = SEQ END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a4.dfi = a7.dfi ◊ a3.dfi	a4 = RWM a7 = RWM a3 = RWM	a4 = RWM
5c: not a4, a4 # a4 = 11•••0•••111	START = SEQ; EXE = SEQ END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a4.dfi = a4.dfi	a4 = RWM	a4 = RWM
60: and a4, a4, a1 # clear bit	START = SEQ; EXE = SEQ END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a4.dfi = a4.dfi ◊ a1.dfi	a4 = RWM a1 = READ	a4 = RWM
64: sw a4, 0(a5) # update word	START = SEQ; EXE = STORE; END = SEQ	Is EXE tag = STORE, PID valid for Mem Page, a5 = pointer, and SEQUENTIAL Exe valid?			a5 = ARRAY POINTER a4 = RWM	MEM(a5+0)= a4 = RWM a5 = pointer Register a5 is protected
68: add a3, a3, a2 # cnt = cnt + base	START = SEQ; EXE = SEQ; END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a3.dfi = a3.dfi ◊ a2.dfi	a2 = RWM a3 = RWM	a3 = RWM
6c: ble a3, a6, 48 <L4> # if less then <L4>	START = SEQ; EXE = BR; END = BR	NONE	Is END tag from prev instruction = SEQ? Is Branch destination address or sequential next instruction valid?		a3 = RWM a6 = READ	N/A
070: addi a2, a2, 1 # base = base + 1	START = SEQ; EXE = SEQ; END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a2.dfi = a2.dfi	a2 = RWM	A2 = RWM
74: bne a2, t1, 28 <L1> # if base != R then <L1>	START = SEQ; EXE = BR; END = BR	NONE	Is END tag from prev instruction = SEQ? Is Branch destination address or sequential next instruction valid?		a2 = RWM t1 = READ	N/A
78: <L2> AA.dp a0, 0 # deallocated memory	START = SEQ; EXE = SEQ; END = SEQ	Deallocate memory pointed to by a0	Is END tag from prev instruction = SEQ?	a0.dfi = (7, 0)	a0 = POINTER Register a0 is protected	a0 = READ
7c: <return> ret function single exit point) # return from function CALL	START = SEQ; EXE = RET; END = RET	N/A	Is END tag from prev instruction = SEQ? Is next instruction tag = RET (instruction following CALL)	A9 = function call stack. RET uses A9 for return.	Register A9 is protected CALL stack	Register A9 is protected CALL stack

Function CALL Single Exit Point

Fig. 50 Aberdeen Architecture RISC-V Sieve of Eratosthenes code (continued)

4.5.2 Data Flow Integrity Monitor

Table 8 summarizes Aberdeen Architecture's data flow integrity policies. The Data Flow Integrity Monitor performs data integrity and data security tag checks listed in Fig. 44. Instruction execution integrity monitor, described in Section 4.4.5, demonstrates the behavior of all of the state machines. Data integrity and data security use lattice operators to compute the resultant integrity level and security level. For arithmetic and logic calculations in (8), registers `rd` = destination register, `rs1` = source register 1, and `rs2` = source register 2. For arithmetic and logic calculations, the resultant *security tag* in (9) is the *highest-level tag* in the calculation. For the integrity tags in (10), **`rd.tag.integrity`** is equal to the *least integrity level* for `rs1` and `rs2`. Global register tags in (11) can be used to define minimum and maximum values for integrity and security.

Table 8 Data flow integrity policies summary

$rd = rs1 \odot rs2$	where $\odot$ is the arithmetic or logic operation for RISC-V arithmetic or logic instruction. For example, `rd = rs1 + rs2`	(8)
$rd.tag.security = \max(rs1.tag.security, rs2.tag.security)$	arithmetic and logic **security** tag result	(9)
$rd.tag.integrity = \min(rs1.tag.security, rs2.tag.security)$	arithmetic and logic **integrity** tag result	(10)
$rd.tag.security.range = (2, 7)$ $rd.tag.integrity.range = (0, 5)$	Global register tags can be used to define security and integrity ranges	(11)

Data integrity verification checks data integrity tags in (10) and data security tags in (9) for the information flow in (8). The memory access tag for **`LOAD`** and **`STORE`** instructions provides data flow verification for memory accesses.

4.5.3 Control Flow Integrity Monitor

In Table 9, control flow integrity supports (1) code block labels, block start, block end, (2) **`CALL`**, **`RETURN`** single point entry and exit points, (3) exception single point entry/exit points, (4) interrupt request single point entry/exit points, and (5) control flow instruction labels and links (instruction linked list). Control flow integrity operations are illustrated in Fig. 44. Control flow integrity was discussed in Section 4.3.9. Instruction execution integrity routine, described in Section 4.4.5, demonstrates the behavior of all of the state machines.

Table 9 Control flow label summary

Control Flow Label	Control Flow Description	Figures
CALL	Function Call Single Entry Point	24, 29, 30, 44
RETURN	Function Return Single Exit Point	24, 29, 30, 44
EXCEPTION_CALL	Exception Single Entry Point	21
EXCEPTION_RET	Exception Single Entry Return	21, 22
IRQ_CALL	Interrupt Single Entry Point	21, 23
IRQ_RETURN	Interrupt Single Entry Return	21, 23
AL_SEQUENTIAL	Arithmetic/Logic Sequential Instruction	9, 25, 36
LOAD_IMM_SEQ	Load immediate sequential Instruction	43, 46
LOAD_SEQUENTIAL	Load from memory sequential instruction	10, 26, 37, 49
STORE_SEQUENTIAL	Store to memory sequential instruction	10, 49
AA.SEQUENTIAL	**Aberdeen Architecture Sequential Class Privileged Instructions**	46, 50
BRANCH	Branch instruction	11, 38, 39, 48, 50
JUMP	Jump instruction	12, 27, 40, 41, 47, 50

4.5.4 Memory Page Monitor

Section 4.3.4 and Fig. 13 introduce page memory classes. Figure 17 describes the operation of the stack state machine. The memory classes extend the memory classes described in the paper "Hardware Security Kernel for Cyber Defense" (Jungwirth et al. 2019b). Memory Page Monitor Supports: memory page classes, Exe Mem Page, Exe Stack Mem Page, Data Stack Mem Page, Create Stack Pointer and Deallocate Stack Pointer; Virtual Page Table Mem Page, OS Table Mem Page; LOAD/STORE memory access; I/O Page Class; Data Page Class; Data Stack Page; conversions between allowed classes; and Process Configuration Page (protected page class).

Each memory page class provides least privilege, complete mediation, and privilege separation. Memory pointers, stack, **IO_Page**, buffers, and so on, also have a set of allowed memory operations to support least privilege, complete mediation, and privilege separation. For a process's data stack to be used by a DLL, the data stack memory page must be converted to **DLL_Stack** using an Aberdeen Architecture protected instruction. Once memory page class = **DLL_Stack**, the running process cannot access the stack space. When the DLL completes, it sets the page class back to Data Stack for the running process.

4.5.5 Instruction Execution Monitor

The Aberdeen Architecture's instruction execution monitor verifies four instruction classes shown in Table 10. `AA_Protected` instructions are sequential class instructions that call state machine monitor operations. For an example state

machine operation, see AA.Create_Buffer. Instruction classes are based on the behavior of the PCR. Sequential instruction classes **AL_SEQUENTIAL**, **IMM_SEQUENTIAL**, **LS_SEQUENTIAL**, and **AA_Protected** advance the PCR to the next instruction. **CALL** and **RETURN** instructions are a protected "jump"-like instruction. BRANCH instruction class has two next instruction addresses: PCR = next instruction (same as sequential class) and PCR = branch destination address. The JUMP instruction simply jumps to the jump destination address (PCR = destination address).

Table 10 Instruction class summary

Instruction Class	Control Flow Description	Figures
(1.1) AL_SEQUENTIAL	Arithmetic and Logic Sequential Instruction Class	9, 25, 36
(1.2) IM_SEQUENTIAL	Load immediate sequential Instruction	43, 46
(1.3) LS_SEQUENTIAL	LOAD/STORE Sequential Instruction	10, 26, 37, 49
(2) AA_Protected	Aberdeen Architecture **Protected Instructions**	46, 50
(3) BRANCH	Branch instruction	11, 38, 39, 48, 50
(4) JUMP	Jump instruction	12, 27, 40, 41, 47, 50

As listed in Table 10, sequential instructions cover (1.1) register-to-register arithmetic and logic instructions; (1.2) load immediate, and (1.3) LOAD and STORE instructions. The sequential class instructions advance the program counter to the next instruction. Aberdeen Architecture instructions are protected instructions directly executed by the Aberdeen Architecture state machine controllers. For example, an operating system or user program can call an Aberdeen Architecture instruction to create a buffer. The buffer is managed and protected by state machines. Aberdeen Architecture instructions also include stack operations, I/O operations, and memory page operations. Section 4.5.6 presents a pseudo-code implementation for the instruction execution monitor.

4.5.6 Instruction Execution State Machine Monitor

A simplified state machine instruction execution monitor is presented in Fig. 51. Section 4.5.6 and Fig. 52 present a pseudo-code implementation for the instruction execution monitor. The instruction execution state machine uses (1) data flow control state machine monitor, (2) control flow state machine monitor, and (3) memory access state machine. Together the four state machines provide high-assurance instruction execution for the Aberdeen Architecture. The instruction execution monitor operation is described for the Sieve of Eratosthenes RISC-V code in Fig. 44. Each instruction class uses a line of code from Fig. 44 to explain the operation of the instruction execution state machine. For example, code {7} handles the arithmetic and logic sequential instruction class. Code {8} shows an

example of a protected instruction, `AA.create_buffer`. Protected instructions are executed by state machine controllers.

```
case Arithmetic_Logic_SEQUENTIAL:                                    {7}
```

20: <L3> addi a2, a2, 1 # a2 = base = base +1	START = BR; EXE = SEQ; END = SEQ	NONE	Is END tag from prev instruction = BR?	a2.dfi = a2.dfi = (7, 0)	a2 = RWM	a2 = RWM

```
// Aberdeen Architecture Protected Instructions
// Executed by State Machine Controllers                             {8}
case AA.Create_Buffer:
```

0c: li a1, 0x200 # buffer length = 0x200	START = SEQ; EXE = LOAD IMM SEQ; END = SEQ	LOAD IMMEDIATE	Is END tag from prev instruction = SEQ?	a1.security = 7 a1.integrity = 0	a1 = READ	a1 = READ
##: AA.cb a0, a1 # buffer = a0 => array of integers # a1 = buffer length	START = SEQ; EXE = AA SEQ; END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a0.security = 7 a0.integrity = 0	a0 = RWM a1 = READ	a0 = ARRAY POINTER Register a0 is protected

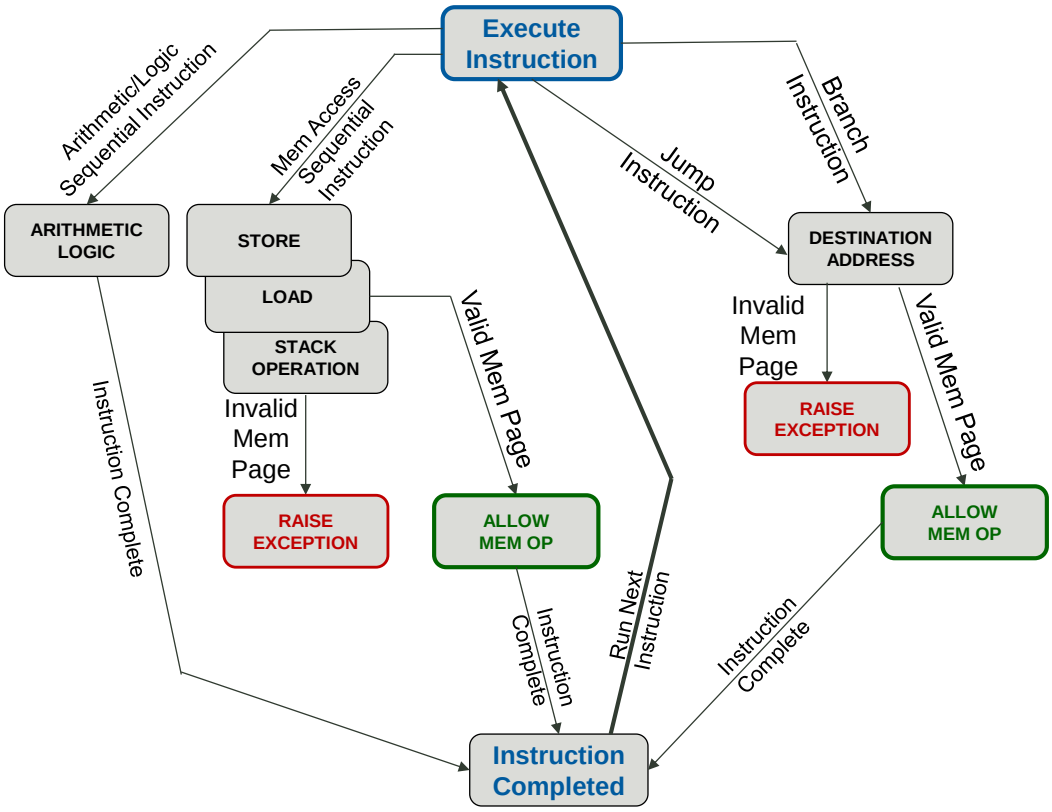


Fig. 51 Simplified execution monitor state machine

```
int Instruction_Exe_Monitor(int PID, int instruction_class, reg_type registers,
data_mem_type &data)
{
   switch(instruction_class){
```

20: <L3> addi a2, a2, 1 # a2 = base = base +1	START = BR; EXE = SEQ; END = SEQ	NONE	Is END tag from prev instruction = BR?	a2.dfi = a2.dfi = (7, 0)	a2 = RWM	a2 = RWM

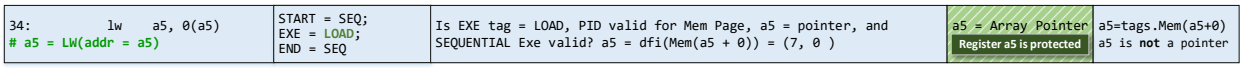
```
      case Arithmetic_Logic_SEQUENTIAL:
         if( ( Control_Flow (PCR-1, PCR) == ALLOWED)   &&
             ( Mem_Access_Tag == NONE    )              &&
             ( Exe_Page.Tag == EXE )                    &&
             ( Exe_Page.PID == Running_Process_ID )  )  &&
             ( Register_Tags[rd] != PROTECTED)     )    &&
             ( Register_Tags[rd] == at least WRITE)    )
         {
            if (Register[rd].dfi.bounds ALLOWED for Register[rs1].dfi ◊
                Register[rs2].df)
            {
         // ◊ = data flow integrity lattice operator
               Register[rd].dfi  = Register[rs1].dfi ◊ Register[rs2].dfi;
               Register[rd].tags = Register[rs1].tags ◊ Register[rs2].tags;

         // RISC-V Core executes
         // rd = rs1 ⊙ rs2; where ⊙ = arithmetic/logic
            }
            else
               throw Register_Bounds_EXCEPTION;
         }
         else
            throw AL_SEQ_EXCEPTION;
      break;
```

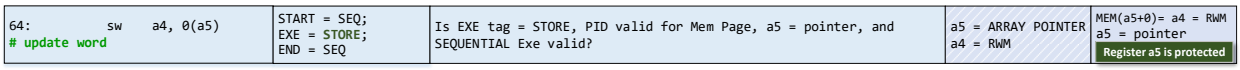
10: li a7, 1 # a7 = 1	START = SEQ EXE = LOAD IMM SEQ END = SEQ	LOAD IMMEDIATE	Is END tag from prev instruction = SEQ?	a7.security = 7 a7.integrity = 0	a7 = READ	a7 = READ

```
      case LOAD_IMM_SEQUENTIAL: // LOAD CONSTANT
         if( ( Control_Flow (PCR-1, PCR) == ALLOWED)  &&
             ( Mem_Access_Tag == NONE    )            &&
             ( Exe_Page.Tag == EXE )                  &&
             ( Exe_Page.PID == Running_Process_ID )   &&
             ( Register_Tag[rd] != PROTECTED)     )   &&
             ( Register_Tag[rd] == at least WRITE)    )
           {
              Register[rd].security  = 7;   // lowest security
              Register[rd].integrity = 0;  // highest integrity
              Register[rd].tags = Register[rd].tags;
           // RISC-V Core executes
           // Register[rd] = immediate value
           }
         else
            throw LOAD_IMM_EXCEPTION;
      break;
```

Fig. 52 Instruction execution state machine monitor




```
case LOAD_WORD_SEQUENTIAL:
   addr = register[rs] + offset;
   if( ( Control_Flow (PCR-1, PCR) == ALLOWED)  &&
       ( Exe_Page.Tag == EXE )                  &&
       ( Exe_Page.PID == Running_Process_ID )   &&
       ( Mem_Access_Tag == LOAD   )             &&
       ( Mem_Page.PID == Process.PID            &&
       ( Mem_Addr == VALID                      &&
       ( Mem_Page.tag == at least READ)      )  &&
       ( Register[rd] == at least WRITE)      )  &&
   {
      if (Register[rd].dfi.bounds ALLOWED for Register[rs1].dfi ◊
            Register[rs2].df)
      {
         Register[rd].dfi  = Mem(addr = rs + offset).dfi;
         Register[rd].tags = Mem(addr = rs + offset).register_tags;
         // RISC-V core executes
        // Register[rd] = Mem(addr = rs + offset);
      }
      else
        throw Register_Bounds_EXCEPTION;
   }
   else
      throw LOAD_IMM_EXCEPTION();
break;
```




```
case STORE_WORD_SEQ
   addr = register[rs] + offset;
   if( ( Control_Flow (PCR-1, PCR) == ALLOWED)  &&
       ( Exe_Page.Tag == EXE )                  &&
       ( Exe_Page.PID == Running_Process_ID )   &&
       ( Mem_Access_Tag == STORE   )            &&
       ( Mem_Page.PID == Process.PID            &&
       ( Mem_Addr == VALID                      &&
       ( Mem_Page.tag == at least WRITE)     )  &&
       ( Register[rd].tag == at least READ)  )  &&
       ( Register[rs1].tag == POINTER )
      {
         Mem(addr = rs + offset).dfi = Register[rd].dfi;
         Mem(addr = rs + offset).register_tags = Register[rd].tags;
      // Mem(addr = rs + offset = Register[rd];
      }
   else
      throw STORE_EXCEPTION;
   break;
```

Fig. 52 Instruction execution state machine monitor (continued)

0c: li a1, 0x200 # buffer length = 0x200	START = SEQ; EXE = LOAD IMM SEQ; END = SEQ	LOAD IMMEDIATE	Is END tag from prev instruction = SEQ?	a1.security = 7 a1.integrity = 0	a1 = **READ**	a1 = **READ**
##: AA.cb a0, a1 # buffer = a0 => array of integers # a1 = buffer length	START = SEQ; EXE = AA SEQ; END = SEQ	NONE	Is END tag from prev instruction = SEQ?	a0.security = 7 a0.integrity = 0	a0 = RWM a1 = READ	a0 = **ARRAY POINTER** Register a0 is protected

```
// Aberdeen Architecture Protected Instructions
// Executed by State Machine Controllers
   case AA.Create_Buffer:
       if(rd.tag != BUFFER|STACK|PROTECTED)
          rd.tag == BUFFER;
          ALLOCATE_MEM_PAGE();
       else
          throw Create_Buffer_ERROR;
   break;

   case AA.CALL: // similar to jump with stack operation
      if( CALL Destination Address  == VALID)   &&
        ( Exe_Page.Tag == EXECUTE )                                    &&
        ( Exe_Page.PID == Running_Process_ID )                    &&
        ( Exe_Mem_Page.PID == Process.PID                         &&
        ( Destination_Addr == VALID                               )
      { // State Machine Executes Protected Instruction
         PCR = DESTINATION ADDRESS;
         PUSH_Exe_Stack();
      }
      else
         throw CALL_EXCEPTION;
   break;

   case AA.RETURN:
      if( (PCR(n) == RET instruction) &&
          (PCR(n+1) == ACCEPT_RETURN)
         {
            PCR = PCR + 1;
            Update_Exe_Stack();
         }
       else
          throw RETURN_EXCEPTION;
   break;


   case AA.Data_Stack:
      Update_Data_Stack();
   break;

   case AA.Data_Stack_to_DLL_Stack:
      if(rd.tag == DATA_STACK)
          rd.tag == DLL_STACK;
      else
         throw STACK_TYPE_ERROR;
   break;
```

Fig. 52 Instruction execution state machine monitor (continued)

```
    case AA.DLL_Stack_to_Data_Stack:
        if(rd.tag == DLL_STACK)
           rd.tag == Data_STACK;
        else
           throw STACK_TYPE_ERROR;
    break;

    case AA.IO_Page_to_Data:
        if(rd.tag == IO_PAGE)
           rd.tag == Data_PAGE;
        else
           throw IO_PAGE_TYPE_ERROR;
    break;

    case AA.Data_to_IO_Page:
        if(rd.tag == DATA_PAGE)
           rd.tag == IO_PAGE;
        else
           throw DATA_PAGE_TYPE_ERROR;
    break;

    case AA.Open_IO_Port:
         rd.tag = IO_Port;
         rd = Port_Controller_Address(create, rd);
         if Port_Controller_Address == NULL then
         {
            rd.tag = NULL;
            throw Open_Port_Exception;
        }
        Clear_IO_Page(rd);
    break;

    case AA.Close_IO_Port:
         Clear_IO_Page(rd);
         Port_Controller_Address(free, rd);
         if Port_Controller_Address != 0 then Close_Port_Exception;
    break;

    default:
      throw SEQUENTIAL_INSTRUCTION_EXCEPTION;
    break;
```

24: beq a2, t1, 78 <L2> # if base = R then <L2> Done	START= SEQ; EXE = BR; END = SEQ|BR	NONE	Is END tag from prev instruction = SEQ? Is Branch destination address or sequential next instruction valid?	a2 = **READ** t1 = READ	N/A

```
case BRANCH:
    if( ( Control_Flow (PCR(n-1), Destination_Addr, SEQ ) == ALLOWED)   &&
        ( Exe_Page.Tag == EXECUTE )                                     &&
        ( Exe_Page.PID == Running_Process_ID )                          &&
        ( Destination_Addr == VALID                                     &&
        ( Sequential_Addr  == VALID                                     && )
    {
        Register.PCR.tags = ???
       // PCR = Destination Address or Next Sequential Address
    }
    else
       throw BRANCH_EXCEPTION;
break;
```

Fig. 52 Instruction execution state machine monitor (continued)

1c: j 28 <L1> # jump to <L1>	START = SEQ EXE = JMP END = JMP	NONE	Is END tag from prev instruction = SEQ? Is Jump Destination Valid?	N/A	N/A

```
   case JUMP:
     if( ( Control_Flow (PCR(n-1), Destination_Addr ) == ALLOWED)     &&
           ( Exe_Page.Tag == EXECUTE )                                 &&
           ( Exe_Page.PID == Running_Process_ID )                      &&
           ( Exe_Mem_Page.PID == Process.PID                           &&
           ( Destination_Addr == VALID                                 )
       {
           Register.PCR.tags = ???
          // Jump to Destination Address, PCR = Destination Addres
       }
       else
          throw JUMP_EXCEPTION;
   break;

// invalid instruction type
    default:
       throw Instruction_Type_Exception }
    break;
```

Fig. 52 Instruction execution state machine monitor (continued)

4.6 Aberdeen Architecture Two-State Machine Simulation

C code for a limited prototype RISC-V-based Aberdeen Architecture simulation is found in Appendix B. The prototype only simulates (1) simple control flow integrity, and (2) simple page memory verification. A fully functional prototype would require several more protection features to be implemented and additional functionality to fuse the outputs from the multiple state machine monitors.

Figure 53 illustrates a simple control flow violation (see Appendix C). Executing instruction is a **JUMP** instruction. Control flow tags are set for BRANCH instruction. Since the executing instruction violates the control flow tags, a control flow violation occurs. The **case** **BRANCH** in Fig. 52 raises a **BRANCH_EXCEPTION** as illustrated in Fig. 53.

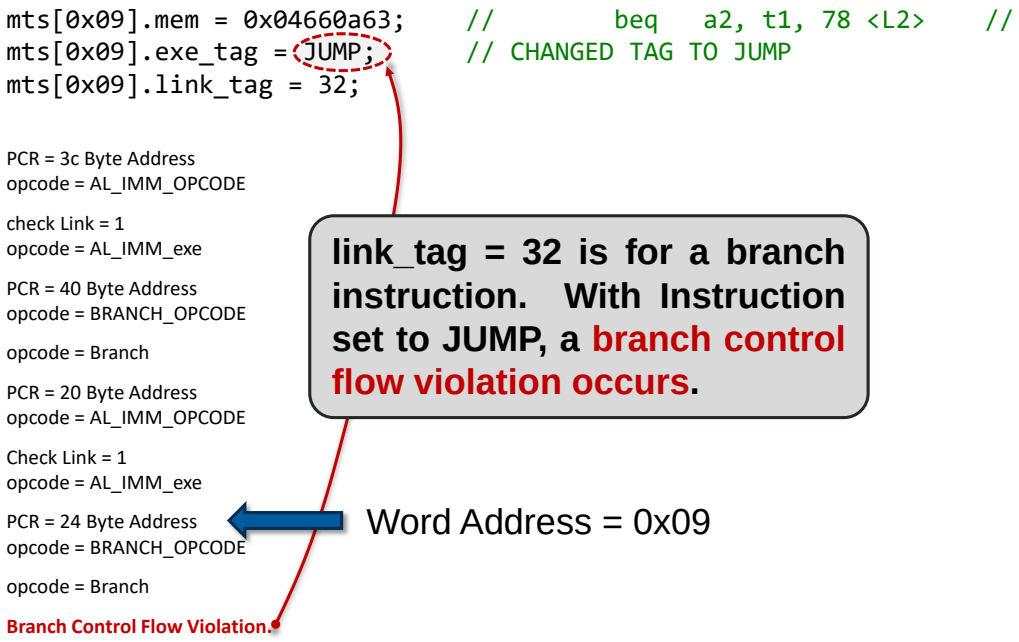


Fig. 53 Control flow state machine simple control flow graph exception

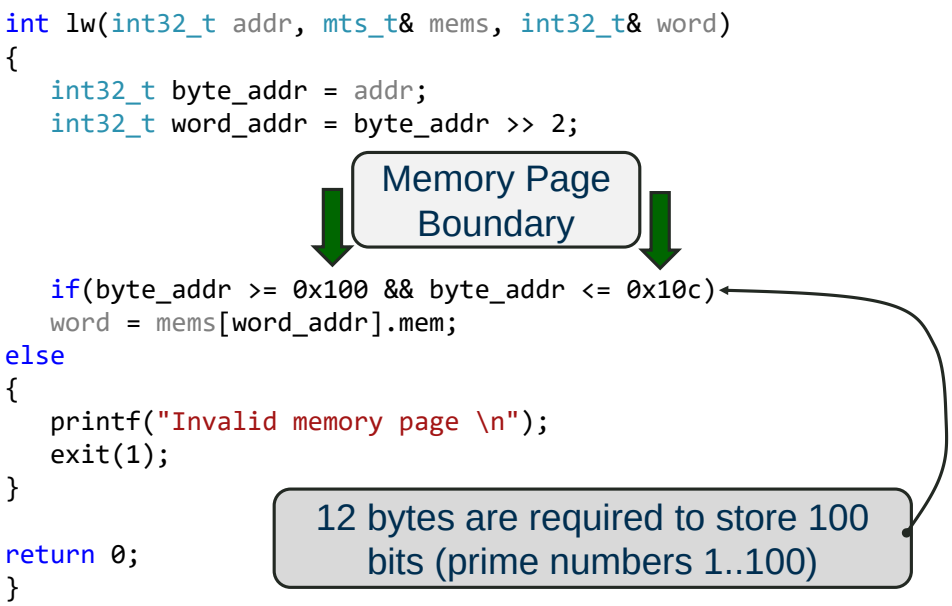


Fig. 54 Memory page state machine simple memory page exception

Figure 54 illustrates simulating a simple page memory boundary for the sieve array of bits. For each bit, `0` = not prime, and `1` = prime. Accessing memory outside the memory page (`0x100` through `0x10c`) will raise a simulated hardware exception.

4.7 Summary of Aberdeen Architecture State Machine Monitors

A brief summary of the Aberdeen Architecture is presented in Fig. 55. The Aberdeen Architecture adds state machine monitor protection mechanisms to the Redstone Architecture presented in Appendix A.

1. Aberdeen Architecture -- State Machine Monitors
    - 1.1. Aberdeen Architecture Protection Mechanisms
        - 1.1.1. Data Flow Integrity Monitor
        - 1.1.2. Control Flow Integrity Monitor
        - 1.1.3. Memory Access Page Monitor
        - 1.1.4. Instruction Execution Monitor
    - 1.2. Data Flow Integrity Monitor (Table 8)
    - 1.3. Control Flow Integrity Monitor (Table 9)
        - 1.3.1. Control Flow Tags (SEQUENTIAL, BRANCH, JUMP)
        - 1.3.2. Control Flow Code Blocks
        - 1.3.3. CALL/RETURN Instructions
        - 1.3.4. IRQ/RETURN Code Blocks
        - 1.3.5. EXCEPTION/RETURN Code Blocks
    - 1.4. Memory Page Monitor (§ 4.3.4)
        - 1.4.1. Memory Page Classes
            - 1.4.1.1. Exe Mem Page
            - 1.4.1.2. Stack Mem Page
            - 1.4.1.3. Exe Stack Mem Page
            - 1.4.1.4. Create Stack Pointer // Deallocate Stack Pointer
            - 1.4.1.5. Virtual Page Table Mem Page
            - 1.4.1.6. OS Table Mem Page
    - 1.5. Instruction Monitor (§ 4.3.1)
        - 1.5.1. RISC-V instruction set architecture (§ 4.3.3)
        - 1.5.2. Classes of RISC instructions (§ 4.4.1)
            - 1.5.2.1. Sequential Execution
                - 1.5.2.1.1. Register-to-Register
                - 1.5.2.1.2. Load Immediate
                - 1.5.2.1.3. Load/Store
            - 1.5.2.2. Direct Jump
            - 1.5.2.3. Branch
            - 1.5.2.4. Aberdeen Architecture Protected Instructions. (§ 4.4, and § 4.5)
                - 1.5.2.4.1. Buffer, IO, Memory Page, et al. Pointers
                - 1.5.2.4.2. Stack Pointers
                - 1.5.2.4.3. Allocate/Deallocate Memory Pages
                - 1.5.2.4.4. CALL/RETURN Instructions
2. Redstone Architecture
    - 2.1. Aberdeen Architecture uses high assurance features in Redstone Architecture
        - 2.1.1. Tag Files
        - 2.1.2. Register Tag Fields
        - 2.1.3. Local and Global Tag Fields
    - 2.2. Aberdeen Architecture uses cache bank memory pipeline from Redstone Architecture
        - 2.2.1. Round Robin scheduler uses cache bank memory pipeline for context switches

Fig. 55 Aberdeen Architecture summary

5. Conclusions

This technical report describes a high-assurance computer architecture that achieves complete mediation (Saltzer and Schroeder 1975, Smith 2012) for instruction execution. The Aberdeen Architecture uses hardware-level state machine monitors for the trusted computing base. The state machine monitors provide security policies enforcing multiple information flow properties. The state machines provide complete mediation for instruction execution based on four information flow classes. The Aberdeen Architecture combines several protection methods to create a system security policy where the whole is greater than the individual security policies. The multiple security policies provide overlapping coverage preventing brittleness and single-point security policy failures. The Aberdeen Architecture fully virtualizes the execution pipeline and register file, providing complete time and space separation between software and the security policies.

The Aberdeen Architecture is currently patent pending.

6. Future Research Areas

The Aberdeen Architecture requires a high-assurance compiler to take advantage of the security tag features and state machine controllers. A high-level language is required for software developers. The compiler for the high-level language needs to determine and implement the security details for the programmers.

The high-assurance security features from the Aberdeen Architecture can simplify the implementation of high-assurance microkernels. We envision using the security features from the Aberdeen Architecture to develop a streamlined version of seL4. The AA-seL4 would run as a guest OS using the nano-kernel OS features provided by the hardware-level state machines.

An out-of-order instruction execution architecture is possible. Each execution thread requires a PID to isolate threads, processes, and hardware states.

7. References

Abadi M, et al. Control-flow integrity principles, implementations, and applications. ACM CCS; 2005 Nov. pp 340–353

Abadi M, et al. Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security. 2009 October;13(1):Article 4.

Acıiçmez O, et al. Predicting secret keys via branch prediction. Proceedings of the cryptographers' track at the RSA conference on topics in cryptology. 2007 Feb 5–9. pp 225–242. [accessed 2021 June 9]. https://eprint.iacr.org/2006/288.pdf

Adleman N, et al. Multics security integration requirements. 1976 Mar. Report No.: ESD-TR-76-354 [accessed 2021 June 9]. https://apps.dtic.mil/dtic/tr/fulltext/u2/a041514.pdf

AEG Telefunken. TR441: Characteristics of the RD441 [German]. AEG Telefunken Manual, DBS 180 0470, Konstanz, Germany; 1970.

Aga M, Austin T. Smokestack: thwarting DOP attacks with runtime stack layout randomization. IEEE/ACM International Symposium on Code Generation and Optimization (CGO); 2019. pp 26–36. doi: 10.1109/CGO.2019.8661202.

Alves-Foss J, et al. A new operating system for security tagged architecture hardware in support of multiple independent levels of security (MILS) compliant systems. 2014 Apr. AFRL Technical Report: AFRL-RI-RS-TR-2014-088. www.dtic.mil/dtic/tr/fulltext/u2/a602198.pdf

Ammann O, et al. The failure of the tacoma narrows bridge, a report to the administrator. Report to the Federal Works Agency, Washington, 1941. https://authors.library.caltech.edu/45680/1/The%20Failure%20of%20the%20Tacoma%20Narrows%20Bridge.pdf

Bell G. Wozniak's blue box. Computer History Museum; 1972. Catalog No.: 102713487. http://www.computerhistory.org/collections/catalog/102713487

Bernstein D. Cache-timing attacks on AES; 2005. https://www.semanticscholar.org › Papers › Cache-timing attacks on AES

Blinde L. Galois awarded $4.5M DARPA contract to strengthen hardware security. Intelligence Community News; 2018 Jan 26 [accessed 2021 Apr 26]. https://intelligencecommunitynews.com/galois-awarded-4-5m-darpa-contract-to-strengthen-hardware-security/

Blue box. Wikipedia; n.d. [accessed 2021 June 9]. http://en.wikipedia.org/wiki/Blue_box

Bondi J, Branstad M. Architectural support of fine-grained secure computing; 1989 Dec 4–8. pp 121–130.

Breen C, Dahlbom C. Signaling systems for control of telephone switching. Bell System Technical Journal. 1960 November;39(6):1381–1444. archive.org/details/bstj39-6-1381

Brown G, et al. Operating system enhancement through firmware. Proceedings of the 10th annual workshop on Microprogramming, ACM SIGMICRO. 1977 Sept.;8(3):110–133. https://dl.acm.org/citation.cfm?id=800102.803324

Burow N, et al. Control-flow integrity: precision, security, and performance. ACM Computing Surveys. April 2017;50(1):1–33. https://doi.org/10.1145/3054924.

Burroughs Corp. Burroughs B6500 information processing systems reference manual. Burroughs Corp.; 1969.

Castro M, et al. Securing software by enforcing data-flow integrity. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06). USENIX Association; 2006. pp 147–160.

Chen Z, et al. Dynamic taint analysis with control flow graph for vulnerability analysis. First International Conference on Instrumentation, Measurement, Computer, Communication and Control; 2011. pp 228–231. doi: 10.1109/IMCCC.2011.66.

Chirgwin R. DARPA seeks SSITH lords to keep hardware from the dark side. The Register; 2017 Apr 21. [accessed 2021 June 9]. https://www.theregister.com/2017/04/12/darpa_ssith_program/

Chiricescu S, et al. SAFE: a clean-slate architecture for secure systems. IEEE International Conference on Technologies for Homeland Security (HST); 2013 Nov 12–14. pp 570–576.

Confused deputy problem. Wikipedia; n.d. [accessed 2021 June 9]. https://en.wikipedia.org/wiki/Confused_deputy_problem

DARPA Microsystems Technology Office. Broad agency announcement, system security integrated through hardware and firmware (SSITH); 2017 Apr 19.

DARPA. Clean-slate design of resilient, adaptive, secure hosts (CRASH); 2010 June. DARPA-BAA-10-70.

de Amorim A, et al. A verified information-flow architecture (long version). crash-safe.org; n.d. [accessed 2017 May 10]. http://www.crash-safe.org/assets/verified-ifc-long-draft-2013-11-10.pdf

De Clercq R, Verbauwhede I. A survey of hardware-based control flow integrity (CFI). pp 4-5, 2017 31 Jul. arxiv.org/ftp/arxiv/papers/1706/1706.07257.pdf

Denning D. A lattice model of secure information flow. ACM. 1976 May;19(5):236–243.

Dhawan U, et al. Architectural support for software-defined metadata processing. crash-safe.org. [accessed 2017 May 10]. www.crash-safe.org/assets/PUMP-ASPLOS-2015.pdf

Dhawan U, et al. Hardware support for safety interlocks and introspection. IEEE Adaptive Host and Network Security Workshop; 2012 Sep 14.

Dijkstra E. The structure of the 'THE'-multiprogramming system. Communications of the ACM. 1968 May;11(5):341–346.

Engler D, et al. Exokernel: an operating system architecture for application-level resource management. ACM SIGOPS Operating Systems Review. 1995 Dec;29(5):251–266.

Feustel E. The rice research computer - a tagged architecture*. ACM AFIPS Proceedings of the Spring Joint Computer Conference. 1972 May 16–18. pp 369–377.

Feustel E. On the advantages of tagged architecture. IEEE Transactions on Computers. 1973 July;C-22(7).

Foster C. Hardware enhancement of operating systems. University of Massachusetts, Amherst; 1978 Nov 23. www.dtic.mil/docs/citations/ADA062462

Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. NDSS. 2003 Feb;3:191–206. https://suif.stanford.edu/papers/vmi-ndss03.pdf

Gehringer E, Keedy J. Tagged architecture: how compelling are its advantages? Proceedings of the 12th annual international symposium on computer architecture. 1985 June 17–19. pp 162–170,

Göktaş E, et al. Out of control: overcoming control-flow integrity. IEEE Symposium on Security and Privacy; 2014. pp 575–589.

Goldberg R. Architecture principles for virtual computer systems [master's thesis]. Harvard University; 1973 Feb. www.dtic.mil/dtic/tr/fulltext/u2/772809.pdf

Halderman J, Felten E. Lessons from the Sony CD DRM episode. Center for Information Technology Policy, Department of Computer Science, Princeton University; 2006 Feb. [accessed 2021 June 9]. https://www.copyright.gov/1201/2006/hearings/sonydrm-ext.pdf

Hardin D. Real-time objects on the bare metal: an efficient hardware realization of the Java/sup TM/ Virtual Machine. Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. 2001 May 2–4. pp 53–59.

Hardy N. The confused deputy: (Or why capabilities might have been invented). Association for Computing Machinery. 1988 Oct.;22(4):36–38. https://dl.acm.org/doi/10.1145/54289.871709

Higher Order Software, Inc. Techniques for operating system machines; 1977 July. (Report No.: 7). https://apps.dtic.mil/sti/citations/ADA095989

Hruska J. DARPA, University of Michigan Team Up to Build 'Unhackable' chip. ExtremeTech; 2017 Dec 22 [accessed 2021 Apr 26]. https://www.extremetech.com/extreme/261052-darpa-university-michigan-team-build-unhackable-chip

Jiang Z, Fei Y. A novel cache bank timing attack. IEEE/ACM International Conference on Computer-Aided Design; 2017. pp 139–146.

Jiang Z, et al. High-level synthesis with timing-sensitive information flow enforcement. ACM Proceedings of the International Conference on Computer-Aided Design; 2018. Article 88, pp 1–8. doi.org/10.1145/3240765.3243415

Jiang Z, et al. Designing secure cryptographic accelerators with information flow enforcement: a case study on AES, ACM/IEEE design automation conference (DAC); 2019. pp 1–6.

Jungwirth P, inventor. Computer security framework and hardware level computer security in an operating system friendly microprocessor architecture. US Patent 10,572,687. Granted 2020a 25 Feb.

Jungwirth P. Hardware security kernel for managing memory and instruction execution [UMBC cyber presentation]; 2020b Feb.

Jungwirth P, Hahs D. Transfer entropy quantifies information leakage. IEEE SouthEastCon; 2019 Apr.

Jungwirth P, La Fratta P. OS friendly microprocessor architecture. Army Research Laboratory (US); 2017 Apr. Report No.: ARL-SR-0370. https://apps.dtic.mil/sti/pdfs/AD1032088.pdf

Jungwirth P, LaFratta P, inventors. OS friendly microprocessor architecture. US Patent 9,122,610. Granted 2015 Sep 1.

Jungwirth P, La Fratta P. OS friendly microprocessor architecture. SPIE Defense + Security Cyber Sensing Conference; Baltimore, MD; 2016 Apr 17–21.

Jungwirth P, Ross J. Security tag fields and control flow management. IEEE SouthEastCon 2019; 2019 Apr.

Jungwirth P, et al. Secure computing architecture: a direction for the future -- the OS friendly microprocessor architecture. IEEE HPEC; 2017. [accessed 2021 June 9]. http://ieee-hpec.org/2017/techprog2017/index_htm_files/67.pdf

Jungwirth P, et al. Cyber defense through hardware security. 2018a Apr. Paper 10652-22. (Disruptive Technologies in Information Sciences, Vol. 10652). https://doi.org/10.1117/12.2302805

Jungwirth P, et al. Security tag computation and propagation in OSFA. SPIE Defense + Security; 2018b Apr.

Jungwirth P, et al. The future of cybersecurity workshop. IEEE SouthEastCon; 2019a Apr.

Jungwirth P, et al. Hardware security kernel for cyber-defense. Disruptive Technologies in Information Sciences II; Proc. SPIE 11013; 2019b 10 May. doi.org/10.1117/12.2513224

Jungwirth P, et al. Hardware security kernel for managing memory and instruction execution [Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, presentation]. 2020 Feb 28.

Kamibayashi N, et al. HEART: an operating system nucleus machine implemented by firmware. ACM Proceedings of the first international symposium on architectural support for programming languages and operating systems; 1982 Mar 1–3. pp 195–204. https://dl.acm.org/citation.cfm?id=801843

Karger P, Schnell R. Thirty years later: lessons from the Multics security evaluation. IEEE Annual Computer Security Applications Conference; Las Vegas, NV; 2002 Dec 9–13. pp 119–126.

Karimi E, et al. A timing side-channel attack on a mobile GPU. IEEE International Conference on Computer Design; 2018. pp 67–74.

Keller J. Five organizations working with DARPA to develop design tools for cyber security and trusted computing. Military & Aerospace Electronics; 2017 Dec 13 [accessed 2021 Apr 26]. https://www.militaryaerospace.com/articles/2017/12/design-tools-cyber-security-trusted-computing.html

Kenyon H. DARPA's CRASH program reinvents the computer for better security. Breaking Defense; 2012 Dec 21. https://breakingdefense.com/2012/12/darpa-crash-program-seeks-to-reinvent-computers-for-better-secur/

Kerchoffs A. La cryptographie militaire [French]; 1883. (Also see https://en.wikipedia.org/wiki/Auguste_Kerckhoffs).

Kim J, et al. Survey of dynamic taint analysis. 4th IEEE International Conference on Network Infrastructure and Digital Content; 2014. pp 269–272. doi: 10.1109/ICNIDC.2014.7000307.

Kiriansky V, et al. Secure execution via program shepherding. Proceedings of the 11th USENIX Security Symposium; 2002 Aug 5–9; San Francisco, CA; http://groups.csail.mit.edu/commit/papers/02/RIO-security-TM-625.pdf

Kocher P, et al. Spectre attacks: exploiting speculative execution; 2018 Jan 3 [accessed 2021 June 9]. arxiv.org/pdf/1801.01203.pdf

Kovacs E. Foreshadow/L1TF. SecurityWeek.Com; 2018 Aug 15. https://www.securityweek.com/foreshadowl1tf-what-you-need-know

Landwehr C. Formal models for computer security. ACM Computing Surveys. 1981 September;13(3):247–277.

Lipner S. A comment on the confinement problem. Association for Computing Machinery. 1975 November;9(5):192–196.

Lipp M, et al. Meltdown. 2018 Jan 3. [accessed 2021 June 9]. arxiv.org/pdf/1801.01207.pdf

Mann C. Homeland insecurity. The Atlantic Monthly. 2002 September;290(2):81–102.

Moisuc E, et al. Hardware event handling in the hardware real-time operating systems. Proceedings of the 18th International Conference on System Theory; 2014 Oct 17–19. pp 54–58.

Murtaza Z, et al. Silicon real time operating system for embedded DSPs. IEEE 2006 International Conference on Emerging Technologies; 2006 Nov 13–14. pp 188–191.

Nair R. Evolution of memory architecture. IEEE Proceedings. 2015 August;103(8):1331–1345.

Nakano T, et al. Hardware implementation of a real-time operating system. IEEE Proceedings of the 12th TRON Project International Symposium; 1995 Nov 28 Nov – 1995 Dec 2. pp 34–42.

Nakano T, et al. Performance evaluation of STRON: a hardware implementation of a real-time OS. IEICE Transactions Fundamentals. 1999 Nov;E82-A(11):2375–2382.

Nakano T, Komatsudaira Y, Shiomi A, Imai M. VLSI implementation of a real-time operating system. Proceedings of ASP-DAC '97: Asia and South Pacific Design Automation Con; 1997 Jan 28–31. pp 679680.

Mandke VV, Nayar MK. Implementing information integrity technology - a feedback control system approach. In: van Biene-Hershey ME, Strous L, Editors. Integrity and Internal Control in Information Systems. IICIS 1999. Springer; 2000. (IFIP - The International Federation for Information Processing, Vol 37). https://doi.org/10.1007/978-0-387-35501-6_3

[NICTA] National ICT Australia Ltd, [UNSW] University of New South Wales. Trustworthy embedded systems: ERTOS-2 project plan 2009–2013; 2009 July. https://ts.data61.csiro.au/publications/papers/ERTOS_09.pdf

Oliveira A, et al. The ARPA-MT embedded SMT processor and its RTOS hardware accelerator. IEEE Transactions on Industrial Electronics. 2011 March;58(3):890–904.

Ong S, et al. SEOS: Hardware implementation of real-time operating system for adaptability. 2013 First International Symposium on Computing and Networking; 2013 Dec 4–6. pp 612–616.

Papachristou C, Gambhir S. Microcontrol architectures with sequencing firmware and modular microcode development tools. Microprocessing and Microprogramming. 1991 March;29(5):303–328.

Podebrad I, et al. List of criteria for a secure computer architecture. IEEE Third International Conference on Emerging Security Information, Systems and Technologies, Secureware '09; 2009 June. pp 76–80.

Popek J, Goldberg R. Formal requirements for virtualizable third generation architectures. Communications of the ACM. 1974;17(7):412–421. doi:10.1145/361011.361073.

Prakash A, et al. On the trustworthiness of memory analysis—an empirical study from the perspective of binary execution. IEEE Transactions on Dependable and Secure Computing. 2015 1 Sep–Oct;12(5):557–570. doi: 10.1109/TDSC.2014.2366464.

Rebello K. System security integration through hardware and firmware (SSITH). Defense Advanced Research Projects Agency; n.d. [accessed 2021 Apr 26]. https://www.darpa.mil/program/ssith

Renesas. Renesas expands ecosystem for its R-IN32M3 industrial network devices to deliver up to 5X improvement in overall network performance; 2014 Jun 18 [accessed 2021 Jun 9]. https://www.renesas.com/us/en/about/press-room/renesas-expands-ecosystem-its-r-in32m3-industrial-network-devices-deliver-5x-improvement-overall

Renesas. Renesas data sheet R-IN32M3 series. 2021a Jan 12. https://www.renesas.com/sg/en/document/dst/r-in32m3-series-datasheet?r=1215991

Renesas. R-IN32M3 ASSP for multi-protocol support; 2021b. https://www.renesas.com/sg/en/products/factory-automation/multi-protocol-communication.html

Rice University. Rice University computer-basic machine operation. Rice University; rev. 1962.

RISC-V Members. 2020 July. https://riscv.org/members-at-a-glance/

Russinovich M. Sony, rootkits and digital rights management gone too far. 2005 Oct. https://blogs.technet.microsoft.com/markrussinovich/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far/

Salmon L. System security integrated through hardware and firmware (SSITH). Proposers Day Presentation, Defense Advanced Research Projects Agency; 2017a Apr 21. https://www.darpa.mil/attachments/SSITHProposersDay20170422.pdf

Salmon L. Baking hack resistance directly into hardware. Defense Advanced Research Projects Agency; 2017b 10 Apr. https://www.darpa.mil/news-events/2017-04-10

Saltzer J, Schroeder M. The protection of information in computer systems. Proceedings of the IEEE. 1975 Sept;63(19):1278–1308.

Schneier B. The security of pretty much every computer on the planet has just gotten a lot worse. CNN.com, 5 Jan 18.

http://www.cnn.com/2018/01/04/opinions/security-of-nearly-every-computer-has-just-gotten-a-lot-worse-opinion-schneier/index.html

Schwartz E, et al. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). IEEE Symposium on Security and Privacy; 2010. pp 317–331. doi: 10.1109/SP.2010.26.

Sethumadhavan S. Hardware-enforced privacy. IEEE Computer. 2016 October;49:10.

Shioya R, et al. Low-overhead architecture for security tag. IEEE Pacific Rim International Symposium on Dependable Computing. 2009 Nov 16–18. pp 135–142.

Shrobe H, et al. Trust-management, intrusion tolerance, accountability, and reconstruction architecture (TIARA). Massachusetts Institute of Technology: 2009 June. Report No.: AFRL-RI-RS-TR-2009-271. www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA511350

Smith JM. Clean-slate design of resilient, adaptive, secure hosts (CRASH). Defense Advanced Research Projects Agency; n.d. https://www.darpa.mil/program/clean-slate-design-of-resilient-adaptive-secure-hosts

Smith RE. A contemporary look at Saltzer and Schroeder's 1975 design principles. IEEE Security Privacy. 2012 Nov;10(6):20–25. doi:10.1109/MSP.2012.85. ISSN 1540-7993

Sockut G. Firmware/hardware support for operating systems: principles and selected history. ACM SIGMICRO Newsletter. 1975 Dec;6(4):17–26. https://dl.acm.org/citation.cfm?id=1217198

Song J. Security tagging for a real-time zero-kernel operating system [dissertation]. University of Idaho; 2014 Oct.

Song J, Alves-Foss J. Security tagging for a zero-kernel operating system. IEEE 46th Hawaii International Conference on System Sciences (HICSS); 2013.

Song C, et al. HDFI: Hardware-assisted data-flow isolation. IEEE Symposium on Security and Privacy (SP); 2016. pp 1–17. doi: 10.1109/SP.2016.9.

Song M, et al. Reducing the overhead of real-time operating system through reconfigurable hardware. 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007); 2007 Aug 29–31. pp 1–4.

Sony BMG copy protection rootkit scandal. Wikipedia; 2020 July 13. https://en.wikipedia.org/wiki/Sony_BMG_copy_protection_rootkit_scandal

Stenquist C. HW-RTOS improved RTOS performance by implementation in silicon [white paper]. Renesas R-IN32M3 Industrial Network ASSP; 2014 May. https://www.renesas.com/en-eu/media/support/partners/r-in-consortium/technology/R-IN32_HWRTOS_Whitepaper_5_20_14.pdf

Suh G, et al. Secure program execution via dynamic information flow tracking. ACM ASPLOS XI Proceedings of the 11th international conference on Architectural support for programming languages and operating systems; 2004 Oct 7–13. pp 85–96.

Taram M, et al. Context-sensitive fencing: securing speculative execution via microcode customization. ACM Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems; 2019. pp 395–410. doi.org/10.1145/3297858.3304060

Tiwari M, et al. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. ACM Proceedings of the 38th Annual International Symposium on Computer Architecture; 2011 June 4–8. pp 189–200,

Venkataramani G, et al. FlexiTaint: a programmable accelerator for dynamic taint propagation. IEEE 14TH International Symposium on High Performance Computer Architecture; 2008. pp 173–184. doi: 10.1109/HPCA.2008.4658637.

Vetromille M, et al. RTOS scheduler implementation in hardware and software for real time applications. Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06); 2006 June 14–16. pp 1–6.

Weaver A, Newall N. In-band single frequency signaling. Bell System Technical Journal. 1954 Nov;33(6):1309–1330. https://archive.org/details/bstj33-6-1309

Witten I, et al. An introduction to the architecture of the Intel iAPX 432. IEEE Software & Microsystems. 1983 April;2(2):29–34.

Yan L, et al. Hardware implementation of muC/OS-II based on FPGA. 2010 Second International Workshop on Education Technology and Computer Science; 2010 Mar 6–7. pp 825–828.

Zeldovich N, et al. Hardware enforcement of application security policies using tagged memory. Proceedings of the 8th USENIX conference on Operating systems design and implementation; 2008 Dec. pp 225–240.

Zurkus K. Side-channel vulnerability portsmash steals keys. Infosecurity Magazine; 2018 Nov 6. www.infosecurity-magazine.com/news/side-channel-vulnerability/

Appendix A. OS Friendly Microprocessor Architecture Tech Report (Redstone Architecture)

This appendix appears as a pdf attachment.

Appendix B. In-Progress Prototype Aberdeen Architecture Simulation Code

This appendix appears as a pdf attachment.

Appendix C. Limited Simulation Presentation

This appendix appears as a pdf attachment.

List of Symbols, Abbreviations, and Acronyms

AA	Aberdeen Architecture
AES	advanced encryption standard
CFI	control flow integrity
CWE	Common Weakness Enumeration
DFI	data flow integrity
DLL	dynamically linked library
ISA	instruction set architecture
PCR	program counter register
RISC	reduced instruction set computer
SSITH	System Security Integration Through Hardware and Firmware
TCB	Trusted Computing Base

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

1 (PDF) DEVCOM ARL FCDD RLD DCI TECH LIB

1 (PDF) DEVCOM ARL FCDD RLC CA P JUNGWIRTH