DASICS-Security Processor Design White Paper

Authors: Jin Yue, Yang Chengyuan, Yibin Xu, Lu Tianyue, Chen Mingyu, Chen Mingyu

Date: 2023-04-18T10:29:07+00:00

Abstract

The open-source, shared, and collaborative software development model has spurred the vigorous growth of fields such as the Internet and artificial intelligence. However, this model has also led to increasing complexity in software development, evidenced by reliance on numerous developers to collaboratively develop a single software system, frequent invocation of third-party code libraries, and the management and maintenance of a massive overall codebase. This complex software development paradigm results in a high probability of introducing security vulnerabilities at the development level. For instance, software developers inevitably need to invoke third-party code libraries yet lack assurance of their security, leading to vulnerabilities exploitable by attackers due to the invocation of unreliable third-party libraries, thereby posing risks of information leakage and tampering. Once vulnerabilities are discovered in a frequently used third-party library, the impact often extends to numerous software systems developed using that library. Memory access vulnerabilities constitute the predominant category of software security vulnerabilities. To address the software security challenges posed by these memory access vulnerabilities, academia and industry have proposed a series of software and hardware memory protection mechanisms. These protection mechanisms, on one hand, employ Data Flow Integrity (DFI) technology to perform strict checking and restriction on the data flow of untrusted software code, preventing illegal memory operations through out-of-bounds checks on data boundaries or compliance verification of data sources. Representative works in this domain include Intel's MPX and MPK technologies, ARM's MTE technology, and the CHERI security architecture led by the University of Cambridge, among others. On the other hand, Control Flow Integrity (CFI) technology is utilized to prevent malicious control flow hijacking, such as Intel's CET technology, ARM's BTI technology, and PA technologies. However, these memory protection methods suffer from various issues to varying degrees, including overly coarse granularity of isolation objects, vulnerability of security metadata to attacks, excessive hardware implementation and performance overhead, and the requirement for signif-

icant modifications and recompilation of existing third-party code. We propose the DASICS secure processor design to address the problems of overly coarse isolation object granularity, low metadata security, and excessive performance overhead in existing security protection technologies, while also focusing on dynamic privilege partitioning, memory protection within the same-level address space, and cross-layer call checking, which have received limited attention in prior work. This realizes a secure processor design based on dynamic privilege partitioning by code segments, providing hardware-assisted efficient software memory protection, ensuring the secure invocation and execution of third-party code, and offering solid security guarantees and support for open-source-based software development.

Full Text

DASICS Secure Processor Design White Paper v1.0.0

Institute of Computing Technology, Chinese Academy of Sciences
Advanced Computer Systems Laboratory, Architecture Group

2 Untrusted Third-Party Code Libraries

Many low-level software components (such as operating systems and network protocol stacks) and high-performance third-party libraries (e.g., SQLite, ASL, Crypto++, Qt) are developed in memory-unsafe C/C++ languages. Numerous third-party libraries contain potential memory access vulnerabilities. For instance, the XML parsing library libxml2 has suffered from heap buffer overflow vulnerabilities, the HTML filtering library htmlawed has experienced remote code execution vulnerabilities, and the open-source anti-malware toolkit ClamAV has contained buffer overflow flaws. When integrating these libraries, developers often overlook their inherent unsafe characteristics (such as the absence of input boundary checks), resulting in exploitable security vulnerabilities in their own software that can lead to information leakage and data tampering.

Memory access vulnerabilities represent the primary threat to contemporary software security. These vulnerabilities occur when programs access or modify unauthorized memory regions due to programming errors or malicious attacks, potentially disrupting normal execution, causing system crashes, or enabling exploitation. Common memory access vulnerabilities include buffer overflows, use of uninitialized memory, null pointer dereferences, and heap overflows. Such flaws can cause program crashes, data loss, system failures, and information leakage, posing significant risks to system security. Figure 1 [Figure 1: see original paper] presents Google's statistics on difficult-to-fix security vulnerabilities in the Chrome codebase, revealing that 36.1% of threats stem from Use-after-Free attacks arising from improper memory management, while 32.9% originate from overflow attacks and other issues. Overall, approximately 96% of reported vulnerabilities are memory access vulnerabilities.

The root cause of typical memory access vulnerabilities in third-party libraries lies in the lack of boundary checking for untrusted code and insufficient control flow validation. A prominent example is the Heartbleed vulnerability (CVE-2014-0160) in OpenSSL versions prior to 1.0.1, which occurred when implementing the TLS heartbeat mechanism without validating input lengths. This missing boundary check enabled attackers to conduct overread buffer attacks against clients and servers, causing widespread critical information leakage. More severe memory vulnerabilities allow attackers to rewrite instruction execution addresses, diverting program control flow to attacker-controlled code and enabling more flexible, threatening attacks with Turing-complete capabilities.

Since third-party libraries typically execute within the same process address space as developer-written code, vulnerabilities in these libraries can easily compromise the entire application. For programs written in languages like C/C++ that lack built-in memory safety checks, any line of code can theoretically access any location within the process address space without restriction. Once a third-party library vulnerability is exploited, attackers can readily access arbitrary data and code within the process space, steal or modify information, hijack program execution, and even attempt operating system privilege escalation. Furthermore, vulnerabilities in third-party drivers within the OS kernel can provide access to entire system memory.

Existing research on memory vulnerability protection generally follows two directions: data flow integrity (DFI) techniques that efficiently partition software regions and enforce boundary checks, and control flow integrity (CFI) techniques that restrict jumps to prevent control flow hijacking. Different protection methods exhibit varying characteristics in terms of completeness, usability, and performance overhead, requiring careful trade-offs in practical system design.

3 Related Work Comparison

Defense mechanisms for memory safety can be categorized into pure software approaches and hardware-software collaborative methods. Pure software approaches operate at the programming language, compiler, and loader levels, while hardware-software collaborative methods include constructing trusted execution environments and implementing intra-process address space isolation and control flow protection.

3.1 Pure Software Approaches

Pure software defenses can be implemented at multiple layers. At the programming language level, rewriting untrusted code in memory-safe languages like Rust can resolve most memory access vulnerabilities. However, this approach faces significant practical challenges, as porting the vast and diverse ecosystem of third-party libraries from C/C++ to safe languages represents an enormous engineering effort.

At the compiler level, runtime boundary checking can be enforced by inserting instrumentation code into source code. Examples include GCC and Clang's Stack Protector technology, which prevents stack attacks by setting and verifying a stack guard variable during function calls and returns, and Microsoft's AddressSanitizer (ASan) compiler plugin, which injects detection code at compile-time to identify memory errors during execution.

At program loading time, Address Space Layout Randomization (ASLR) provides defense by randomizing the layout of heap, stack, and shared library mappings, making it difficult for attackers to predict target code locations.

For control flow protection, software relies primarily on CFI technology, which statically analyzes source or binary code to construct a control-flow graph (CFG) and monitors at runtime whether execution remains within the CFG. A CFG consists of basic blocks (continuous code segments without control transfer instructions) and directed edges representing control flow transfers.

While pure software methods provide certain security guarantees, code instrumentation (as in ASan) introduces significant performance degradation by adding checks before and after memory accesses. Additionally, mechanisms that protect security metadata (such as CFG data) must ensure this metadata cannot be tampered with or forged by attackers.

3.2 Trusted Execution Environment

Trusted Execution Environment (TEE) is a hardware-software collaborative security architecture that constructs isolated secure computing environments through CPU time-multiplexing or dedicated memory address spaces. Using hardware isolation, encryption, or integrity verification, TEEs prevent malicious programs from reading or tampering with internal data and control flows from outside the environment. TEEs have been widely adopted in mobile devices and cloud computing, with representative technologies including Intel SGX, AMD SEV-SNP, ARM CCA and TrustZone, and RISC-V Penglai and Keystone.

However, specific TEE technologies often depend on particular architectures or platforms, increasing costs for device migration and software portability. Moreover, TEEs require substantial hardware resources (such as Merkle trees for memory integrity protection) to maintain security. The overhead for data interaction between TEE-internal and external programs is considerable, and the performance cost of secure environment creation and context switching is non-negligible. Consequently, TEEs are suitable for relatively independent, small-scale security-critical code but less appropriate for scenarios requiring frequent interaction. While theoretically feasible to run large applications entirely within a TEE, isolating multiple modules within such programs—particularly third-party libraries—remains an unaddressed challenge.

3.3 Address Space Security Isolation

Beyond external isolation provided by TEEs, protection against internal software threats (such as untrusted third-party libraries) is essential. Without internal code isolation where all code shares identical privileges and can access any data or jump to any code within the address space, untrusted code could steal sensitive data (e.g., user passwords) or jump to malicious code for more threatening attacks. Therefore, software internals require privilege partitioning and isolation of access and jump regions based on different code modules.

Internal software protection technologies must partition privileges among code fragments and restrict memory access and jump targets for untrusted third-party libraries to prevent out-of-bounds data tampering and control flow hijacking. These technologies can be classified into data access isolation and control flow protection approaches. Figure 2 [Figure 2: see original paper] illustrates threats from untrusted libraries within software.

3.3.1 DFI Methods Data Flow Integrity (DFI) techniques partition and isolate data accessible to different code fragments within software, checking memory accesses against these isolation rules to prevent out-of-bounds access and illegal operations, thereby blocking theft and tampering of sensitive information. DFI methods can be further divided into pointer-based permission attachment and memory data-based permission attachment.

Pointer-based permission techniques, represented by MPX and CHERI, define each pointer's accessible range and permitted operations on pointed-to data, enforcing checks during pointer-based memory accesses. MPX (Memory Protection Extensions) is Intel's hardware extension for preventing memory access vulnerabilities, adding four 128-bit bound registers, specialized instructions, and configuration/status registers. MPX checks pointer references against preset ranges during execution. Due to limited bound registers, additional pointer metadata must be stored in memory tables. This approach requires extensive recompilation of third-party libraries and suffers severe performance degradation from instrumentation of every memory access instruction, leading Intel to deprecate MPX in 2019.

CHERI (Capability Hardware Enhanced RISC Instructions) is another hardware-software collaborative memory protection technology that extends traditional pointers to 128-bit capabilities containing valid range, permissions, and validity information, with dedicated instructions for modifying capability data. While CHERI provides pointer-level memory safety and theoretically defends against most overflow attacks, its extended pointers incur substantial memory access overhead and require significant compiler modifications and third-party library recompilation.

Memory data-based permission techniques, represented by MTE and PKU, isolate data regions through memory tagging, where code holds tags to specify access permissions for corresponding memory regions. MTE (Memory Tagging

Extension) in ARMv8.5-A partitions memory into 64-byte chunks with unique tags used alongside pointers to verify correct memory locations. MTE performs memory protection using hardware tags without code modifications and provides software tagging for custom schemes. PKU (Protection Key Unit), exemplified by Intel's MPK, uses 4 spare bits in page table entries as "color" keys to partition memory at page granularity. A special PKRU register tracks the current program's key, detecting illegal accesses when a page's key mismatches. PKU achieves memory isolation with minimal hardware overhead but lacks systematic protection for PKRU integrity, requiring binary scanning to ensure libraries cannot modify PKRU instructions. Additionally, PKU supports only 16 keys, insufficient for complex programs with numerous isolation requirements.

3.3.2 CFI Methods Control Flow Integrity (CFI) technology originally aimed to construct complete program control flow representations (e.g., CFGs), but practical implementations focus on efficiently protecting critical control flow data (e.g., function pointers, return addresses on stack) and constraining indirect jump targets (e.g., to function entry points). Major technologies include CET, BTI, and PA.

CET (Control-flow Enforcement Technology) is Intel's hardware-level security technology that prevents control flow hijacking through two mechanisms: Indirect Branch Tracking (IBT) defends against JOP and ROP attacks by setting landing pad instructions for indirect jumps to legitimate targets (e.g., function entries), and Shadow Stack (SS) maintains an additional stack to track function return addresses, pushing return addresses on calls and verifying them on returns to detect tampering.

BTI (Branch Target Identification) in ARMv8.5 similarly counters JOP attacks by setting BTI instructions at specific locations during compilation to define indirect jump targets, increasing the difficulty of chaining gadgets through indirect jumps.

PA (Pointer Authentication) is ARM's pointer integrity verification technology that uses cryptographic integrity protection for return addresses. To conserve hardware, PA utilizes unused high bits in virtual address pointers. The callee computes an authentication code for the return address, embeds it in the pointer's high bits (called PAC, Pointer Authentication Code), and pushes this authenticated address onto the stack. Verification instructions inserted before function returns detect tampering and trigger exceptions on mismatches.

3.3.3 Security Metadata Protection Existing address space isolation methods share a common problem: while using security metadata (e.g., MTE tags, PKU keys) for protection, they lack safeguards for the metadata itself, allowing malicious programs to escalate privileges or disable isolation by modifying security metadata. For example, programs can freely manipulate PKRU registers to disable coloring protection.

Research addressing security metadata protection falls into two categories: binary scanning methods that inspect and rewrite application or library binaries to ensure untrusted code cannot contain metadata-modifying instructions. For instance, the academic ERIM approach prevents third-party libraries from including WRPKRU instructions through binary scanning, though this cannot support dynamically generated code. User-mode privileged function methods construct a privileged function in user space for dedicated security metadata management and operations, with any metadata operations outside this function triggering exceptions. The academic Donky approach exemplifies this by using user-mode interrupts to establish a Domain Monitor for PKRU management, restricting modifications to this function and controlling jumps into it to prevent hijacking. This resembles further privilege set partitioning within user space but may incur performance overhead from frequent privilege switching.

Security metadata protection often requires combining DFI and CFI: DFI methods isolate the metadata as sensitive data to prevent untrusted access and tampering, while CFI methods ensure that code authorized to modify this metadata cannot be hijacked for malicious modifications.

3.4 Common Issues in Security Mechanism Design

We summarize that security mechanisms for memory access vulnerabilities face several practical challenges when deployed:

3.4.1 Trade-off Between Protection and Overhead Widely-applicable processor security solutions inevitably face trade-offs between performance/hardware overhead and protection capability. Mechanisms with strong protection may incur substantial performance overhead from frequent checks (e.g., MPX's per-pointer permission table lookups). Protection based on privileged states may require numerous privilege transitions, also introducing significant overhead. Additionally, hardware-software collaborative mechanisms relying on extra security metadata can cause substantial hardware overhead; for instance, CHERI's pointer extension increases storage overhead several-fold, creating considerable storage and access costs for pointer-intensive applications. Therefore, security mechanism design must consider not only security but also the cost of achieving it.

3.4.2 Portability Issues Some security mechanisms face rewriting and recompilation difficulties when protecting existing third-party code bases, creating portability problems. Many academic approaches add new instructions or extend pointer semantics, breaking compatibility with legacy binary libraries. Deploying these mechanisms requires recompiling all unsafe third-party libraries, which is labor-intensive and often impossible for closed-source libraries, limiting widespread adoption. Thus, portability must be considered in security mechanism design.

3.4.3 Privilege Transition Security Checks DFI and CFI typically add fine-grained protection within the same privilege level. However, during privilege transitions, higher-privilege code can modify lower-privilege code and data. Without restrictions on privilege transitions, DFI and CFI protections could be bypassed through privileged calls, including system calls and virtual machine traps. System calls, while essential interfaces for applications to use OS functionality, can be exploited by untrusted code to circumvent security mechanisms. For example, under MPK protection, code can use the madvise system call (intended to provide kernel memory access hints for performance) to clear contents of specific memory pages despite MPK protection, or use brk and sbrk system calls (intended for heap management) to reclaim and reallocate sensitive heap data.

Therefore, beyond data isolation, boundary checking, and control flow restrictions, security mechanisms should also intercept and filter system calls and other privilege transitions to prevent attacks that exploit these mechanisms.

4 DASICS Design Philosophy

Based on the challenges identified above, we propose an efficient, low-overhead, and portable processor security enhancement technology for applications within a single address space, integrating data flow integrity, control flow integrity, and system call security.

Attack Model: Our secure processor design assumes software can be divided into code fragments from different sources, including developer-written code that can be controlled and recompiled as needed, and third-party library code that may originate from closed-source binaries or dynamically generated code that developers cannot analyze or modify. We treat this latter category as untrusted code likely containing exploitable memory access vulnerabilities and potentially dangerous operations like privilege escalation through system calls. Attackers can exploit these vulnerabilities for control flow hijacking, data tampering, and theft of critical information.

Code as Subject: To achieve security isolation within an address space, we must first identify appropriate security subjects. Traditional processes or threads are unsuitable subjects within a single address space; function libraries are too coarse-grained, while pointers or instructions are too fine-grained. We observe that code fragments (continuous address ranges of executable code) within a program serve as appropriate subjects for privilege partitioning. For example, a library function contains many continuous code fragments that typically implement related functionality. A given code fragment accesses a deterministic data range and performs deterministic operations at a specific time, enabling spatial partitioning of data accessed by different code (e.g., limiting pointer ranges) and operational restrictions (e.g., limiting read/write access to specific addresses) to achieve temporal security isolation. We must also dynamically adjust these partitions and restrictions over time to ensure security isolation for

the same code fragment in different contexts (e.g., when a function is called multiple times to process different data locations). Additionally, we must restrict entry and exit points of these code fragments (e.g., functions can only be entered through calls and exited through returns, not jumped into mid-execution or exited to non-caller code) to constrain control flow.

Therefore, we select code fragments as security subjects—the principle of "code as subject."

Dynamic Permissions: Based on the code-as-subject concept, our security mechanism dynamically sets permissible data boundaries and jump targets before invoking a code fragment according to its presumed data access range. The code fragment can freely access and jump within allowed ranges, but any out-of-bounds access or illegal jump triggers an exception to block the operation. Permissions are automatically revoked upon the code fragment's return.

Dynamic permission partitioning addresses several requirements: dynamic setting and allocation of permissions due to limited security resources while needing to isolate numerous code fragments in complex programs; support for different permission settings for the same code fragment at different times (e.g., function A may access private data when called by trusted code but must be prohibited from such access when called by untrusted code); and handling scenarios that cannot be statically partitioned, such as function pointers where the caller cannot predetermine the function's behavior and data access range.

Built-in Trusted Computing Base: As previously discussed, security resource management requires a trusted core. We need a trusted code base (TCB) to uniformly manage permissions and maintain metadata, which must satisfy fundamental properties: the TCB itself must be difficult to attack, and transitions between the TCB and untrusted code must be efficient to minimize performance overhead. The TCB must: allocate and recycle security resources to support dynamic partitioning for different code fragments or the same fragment at different times; set permissible data regions and operations before transferring control to untrusted code to ensure data flow integrity; restrict its own entry points to prevent untrusted code from hijacking control flow into the TCB to modify security metadata; trigger exceptions when untrusted code violates permissions, enabling error handling or dynamic correction; and provide trusted call functionality for operations beyond untrusted code's accessible range.

Regarding TCB implementation, we do not adopt privilege level partitioning within the same address space. Instead, we recognize that the TCB itself can be managed as a special code fragment. Transitions between the TCB and untrusted code do not require complex privilege switching but can directly jump under security rule control. Thus, the TCB is essentially a code fragment built into the application.

We call this security processor design based on dynamic permission partitioning by code fragments **Dynamic in-Address-Space Isolation by Code Segments (DASICS)**. Specifically, DASICS comprises several components: region

partitioning, code permission configuration, control flow restriction, permission checking, permission exception handling, main function calls, and system call interception.

4.1 Region Partitioning

DASICS performs permission partitioning at the code fragment level, where a code fragment is defined as a continuous range of executable code in address space (e.g., code within a non-inline function). Fragment ranges are specified by start and end addresses. DASICS follows the principle that higher-privilege code fragments can restrict lower-privilege fragments. As shown in Figure 3 [Figure 3: see original paper], these restrictions operate in two dimensions: vertical restrictions where higher privilege levels can set permissions for lower privilege levels (e.g., in RISC-V, M-mode trusted code can restrict S-mode, which can restrict U-mode), and horizontal restrictions where code within the same privilege level is divided into trusted and untrusted zones, with trusted zones restricting untrusted zones' access permissions. For example, in user mode, the main function serves as the trusted zone while third-party library functions constitute the untrusted zone, with the main function setting permissions before library calls. In kernel mode, the kernel scheduler is the trusted zone while untrusted third-party driver code resides in the untrusted zone, with the kernel setting permissions before using these drivers.

Currently, trusted/untrusted zone partitioning is determined through manual address space annotation, with DASICS placing trusted code fragments in statically designated address spaces during linking. All other address space code is considered untrusted. Trusted zones can access all data at the same privilege level, while untrusted zones cannot access trusted zone data, thereby achieving access region isolation between different privilege codes.

4.2 Code Permission Partitioning

In DASICS, code fragment permissions consist of access permissions and jump permissions. Access permissions are defined by data access boundaries, with DASICS adding several sets of boundary registers in hardware, each describing an accessible region's range and permitted operations (read/write). Jump permissions are similarly defined by jump range registers specifying allowable jump target addresses for library functions.

Before invoking untrusted code, the trusted zone allocates boundary registers and configures the untrusted code's permissions, including memory access and jump permissions, to constrain its data flow and control flow. Upon the untrusted code's return, boundary register resources are recycled. For control flow restriction, beyond limiting jump targets through range registers, DASICS also constrains entry points from untrusted to trusted zones, allowing transitions only through function returns, main function call entries, or same-privilege interrupts, preventing trusted code from being intercepted and exploited.

4.3 Permission Checking

DASICS access permission checking is implemented in the processor's instruction fetch and memory access units. When untrusted code performs memory accesses (including instruction fetch, reads, or writes), the access address is checked against boundary registers to verify whether it falls within allowed ranges and whether the operation type is permitted. Any out-of-bounds access or unauthorized operation triggers a same-privilege interrupt for further handling (e.g., error exit).

Control flow restriction checking occurs during jump instruction execution, primarily verifying trusted-to-untrusted function calls. When control transfers from trusted to untrusted code via function calls, the return address in trusted code is recorded. Upon return from untrusted code, the actual return address is compared against the recorded address; any mismatch indicates potential tampering, triggering an interrupt and error exit.

These permission checks enable DASICS to prevent attacks such as buffer overflows and control flow hijacking through return address modification.

4.4 Same-Privilege Exception Handling

DASICS supports registration of exception handlers in the trusted zone to handle access or control flow exceptions. When an exception occurs, a same-privilege interrupt is triggered (e.g., user-mode interrupt in user mode), jumping to the exception handler which reads hardware privilege registers to determine the exception cause and perform appropriate actions such as error exit or dynamic permission correction before resuming execution. Exception handlers also support dynamic registration to enable different handling for unified exceptions across various scenarios.

4.5 Trusted Calls

Similar to OS system calls, DASICS provides trusted calls for untrusted zones to perform operations beyond their accessible range. This is necessary when pre-configured data ranges are insufficient for untrusted code functionality. For example, when a library function needs expanded data access permissions or larger data regions, it can return to the main function to update boundary registers before resuming execution.

To support trusted calls, the trusted zone must register trusted call entry addresses as additional legal trusted zone entry points. Untrusted code can set trusted call parameters and directly jump to these entries, which is faster than system calls as it avoids costly privilege transitions.

4.6 System Call Interception

To prevent attacks exploiting cross-privilege calls, DASICS incorporates system call interception. Hardware monitoring logic captures system call instructions

(e.g., RISC-V ecall), triggering user-mode interrupts that jump to the trusted zone exception handler. Based on the exception cause (user-mode system call), the handler redirects to a system call exception processing function that performs security checks, such as verifying whether the code fragment can invoke the system call, checking whether parameters exceed the fragment's permission range, and proxying unsafe system calls by copying parameters and executing the call from the main function. In-application system call checking enables fine-grained management of various file and device resources.

4.7 Advantages of DASICS

Compared to previous memory vulnerability protection mechanisms, DASICS offers several advantages:

- **Comprehensive Protection**: DASICS ensures data flow integrity through memory isolation between different privilege subjects and control flow integrity through paired call-return checks. It provides bidirectional protection with mutual support, minimizing risks of security metadata tampering while protecting application data.
- **Efficient Protection**: Transitions between trusted and untrusted zones in DASICS involve direct jumps, avoiding the interrupt-based switching of mechanisms like Donky and eliminating frequent kernel/monitor transitions required by other protection schemes.
- **Fine-Grained Protection**: DASICS achieves pointer access restrictions through dynamic boundary register-based partitioning, providing finer granularity than page-based mechanisms like MPK. Code fragments enable more granular privilege partitioning without relying on hardware IDs or tags.
- **Dynamic Protection**: DASICS implements dynamic region partitioning through trusted zone configuration and dynamic permission partitioning through allocation and recycling of security resources (access boundary registers and jump range registers), enabling flexible, scenario-specific protection with temporal security policies.
- **Multi-Privilege Protection**: DASICS supports multi-level privilege protection (kernel and user modes), applicable to both user-mode code protection (see Use Case 1) and kernel code protection (see Use Case 2).
- **Strong Portability**: Under DASICS, users only need to partition third-party libraries into untrusted zones without recompiling them, reducing recompilation difficulties. When source code modification is possible, simple API additions at function entry/exit points enable further security enhancements.

5 Prototype 1.0 Implementation

We have implemented DASICS Prototype 1.0 based on the above design, including hardware and software components. The hardware prototype is built on the

open-source RISC-V NutShell processor, while the software prototype modifies BBL (Berkeley Boot Loader) and Linux 4.18.2.

5.1 Hardware Implementation

The DASICS hardware prototype extends the open-source RISC-V NutShell processor. In NutShell's six-stage in-order pipeline, we added support for DASICS CSRs, DASICS exceptions, and new DASICS instructions, primarily modifying the decode module (IDU), CSR execution module, and load-store unit (LSU), with total changes under 500 lines of Chisel code.

For DASICS CSRs, we added 43 logical 64-bit CSR registers to support DASICS security functions:

DASICS CSR Name	CSR Number	Description
dsmcfg & dumcfg	0xBC0 & 0x5C0	S-mode/U-mode master function configuration registers
dsmbound0 & dsmbound1	0xBC1 & 0xBC2	S-mode master function boundary registers
dumbound0 & dumbound1	0x5C1 & 0x5C2	U-mode master function boundary registers
dlcfg0 & dlcfg1	0x881 & 0x882	Library function configuration registers
dlbound0 … dlbound31	0x883 … 0x8A2	Library function boundary registers
dmaincall	0x8A3	Master function call entry address
dretpc	0x8A4	Library function return address
dretpcfz	0x8A5	Active zone return address

- dsmcfg and dumcfg share a physical CSR to enable DASICS protection for S-mode/U-mode and include clear bits to reduce context switch overhead.
- dsmbound0/1 and dumbound0/1 define trusted zone boundaries for S-mode and U-mode; code outside these boundaries is untrusted.
- Each dlcfg contains eight 4-bit tiny configs (odd-indexed configs are unused zeros), with each tiny config corresponding to a dlbound pair, specifying permissions (read/write/execute) for the code fragment within those bounds. dlcfg and dlbound together form a DASICS library function lookup table for exception detection.
- dmaincall stores the master function call entry address, which library functions are permitted to jump to.

- dretpc records the allowed return target address after library function calls. When the master function jumps to a library function, the hardware automatically stores the next instruction address in dretpc.
- dretpcfz is the active zone return address, where an active zone is untrusted code that external code can jump to arbitrarily, but which can only return to external code through function returns.

To prevent untrusted code from modifying these CSRs to disable DASICS protection, DASICS restricts CSR access to trusted zone code with appropriate privilege levels. For example, dsmbound0/1 can only be accessed by M-mode and S-mode master functions, while dlcfg can only be accessed by M-mode, S-mode, and U-mode master functions.

To enable exceptions on DASICS violations and system call interception, we added eight DASICS exceptions:

DASICS Exception Name	Exception Code	Description
DASICS_U/S_{{{INST}}}_{{{FAULT}}}	0x19	U-mode/S-mode jump exception
DASICS_U/S_{{{LOAD}}}_{{{FAULT}}}	0x1b	U-mode/S-mode read permission violation
DASICS_U/S_{{{STORE}}}_{{{FAULT}}}	0x1d	U-mode/S-mode write permission violation
DASICS_U/S_{{{ECALL}}}_{{{FAULT}}}	0x1f	U-mode/S-mode system call exception

- DASICS S-mode/U-mode execution exceptions occur when untrusted functions jump to trusted zone targets not equal to dretpc or dmaincall, when active zone functions attempt jumps outside return addresses, or when untrusted code jumps outside allowed ranges.
- DASICS S-mode/U-mode memory access exceptions trigger when untrusted code attempts to access addresses outside permitted ranges (i.e., when access address and operation type find no match in the DASICS library function lookup table), generating read/write exceptions based on the instruction type. These exceptions cancel pending memory requests in the LSU module.
- DASICS S-mode/U-mode system call exceptions trigger upon detecting RISC-V system call instructions (ecall), enabling interception checks in the exception handler.

In NutShell's instruction decode module, we added decoding logic for the new dasicsret instruction, which resembles JALR but only permits master function calls. To accelerate DASICS exception handling, we implemented the RISC-V

N extension draft (version 1.1) on NutShell, enabling delegation of DASICS exceptions to user mode through medeleg/sedeleg configuration. Figure 4 [Figure 4: see original paper] illustrates the exception handling flow when DASICS exceptions are delegated to user mode.

5.2 Software Implementation

In BBL, we added DASICS mechanism initialization, including setting DASICS exception delegation and configuring dsmbound to cover the entire memory space. This approach is necessary because BBL cannot determine the S-mode trusted zone size during initialization and will later jump to S-mode trusted code that modifies dsmbound via new SBI calls. We also added an SBI call for dsmbound modification that can only be invoked by S-mode trusted code, otherwise returning an error.

In Linux, we added DASICS exception handling logic and integrated DASICS into user program ELF loading. For exception handling, we extended Linux's existing exception flow to save/restore DASICS context and delegate DASICS U-mode exceptions to user mode. We also provide default master function call entry functions and default DASICS user-mode exception handlers. For ELF loading, we modified fself.c to identify dasics partitions in user programs and configure dumbound registers to establish user program master function regions.

6 DASICS Application Scenarios

This chapter presents several examples of how DASICS can efficiently enhance system security.

6.1 Use Case 1: Trusted Rust + Untrusted C Libraries

As previously mentioned, using safe languages like Rust can resolve most memory vulnerabilities, but such languages involve complex development and difficult porting. Rewriting all existing code in safe languages would require enormous effort. With DASICS, developers can build trusted zone code in safe languages while constraining untrusted library code through DASICS mechanisms. This ensures trusted zone code remains free of memory vulnerabilities while significantly reducing rewriting workload. DASICS-constrained untrusted zone code has limited permissions, substantially reducing the impact of potential security vulnerabilities. Since DASICS configures security permissions through hardware register operations independent of programming language, it supports various languages and provides flexible security configuration.

6.2 Use Case 2: Kernel Protection + Third-Party Drivers

Many security mechanisms rely on a secure kernel that sets security permission data to check or restrict user-mode programs. However, kernel mode itself runs substantial third-party code including file systems, device drivers, and network

stacks that OS developers do not trust, creating security risks. Both third-party modules and kernel code run at privileged levels; once kernel security is compromised, most security mechanisms become ineffective. DASICS enables trusted/untrusted zone isolation within the operating system itself, allowing system developers to place third-party modules in kernel-space untrusted zones and restrict their execution permissions, mitigating OS security risks from third-party kernel modules. DASICS's hardware-based protection supports kernel address isolation without breaking existing kernel-user isolation while adding new isolation layers within kernel mode.

6.3 Use Case 3: Single-Level OS (LibraryOS)

In cloud computing scenarios, applications often do not require complex OS functionality, while complex operating systems reduce overall system efficiency. The LibraryOS or Unikernel concept addresses this by transforming required OS functions into user-space libraries, reducing overhead and improving platform adaptability. However, this approach sacrifices traditional OS security, namely kernel-user isolation. Achieving sufficient security requires additional lightweight isolation mechanisms. Integrating DASICS into LibraryOS can restrict user program memory access permissions at function granularity, isolating application code from kernel functionality code within LibraryOS to avoid security risks. Compared to protection mechanisms based on multiple protection state transitions, DASICS incurs lower performance overhead, preserving LibraryOS's original performance advantages.

Note: Figure translations are in progress. See original paper for figures.