# Verifying Hardware Resource Isolation Using Mandatory Access Control

Christopher Nokes*, Kostas Amberiadis†, D. Richard Kuhn†, Edwards E. Reed‡, Michael Zuzak*
*Rochester Institute of Technology, Email: {crn2267, mjzeec}@rit.edu
†National Institute of Standards and Technology, Email: {kostas.amberiadis, d.kuhn}@nist.gov
‡AESEC Global Services, Inc., Email: ed.reed@aesec.com

***Abstract*—Resource isolation plays a crucial role in ensuring the security of hardware systems. Existing hardware security verification measures can identify improper isolation of hardware resources, but coverage is incomplete. As the search space increases with hardware size, the chances of an integrity flaw slipping through conventional verification methodologies also increase. In this work, we propose the application of Mandatory Access Control (MAC) models to the hardware resource isolation problem as an improvement to existing System-on-Chip (SoC) verification environments. To do so, we extend the definitions of subject and object, conventionally defined for the software domain, to a hardware context, allowing existing time-tested MAC policy models and verification methodologies to be directly applied to test the isolation of shared hardware resources. Using these definitions, we frame hardware isolation as a MAC problem and adopt the Biba interpretation of the Bell-LaPadula (BLP) model for hardware resource isolation. We use this model to propose a security verification methodology, in which subject-to-object interactions are defined as transactions, and are verified with respect to a security policy. Transactions that violate this security policy constitute a hardware isolation violation. We then use this transaction-based methodology to implement SystemVerilog-based verification environments for three open-source RISC-V cores. These environments are used to discover six instances of improper resource isolation (one novel, two previously documented, and three artificial test cases) with a simulation runtime overhead of less than 10%. These violations constitute security vulnerabilities in hardware, which a malicious user could exploit to access or modify memory regions or hardware peripherals that should be isolated, thereby violating trust.**

***Index Terms*—Hardware Verification, Mandatory Access Control, Hardware Vulnerability Detection**

## I. Introduction

The assurance of proper resource isolation in hardware systems is a critical goal of hardware security verification. Doing so reduces the likelihood of design flaws that may compromise both the integrity and confidentiality of a system. The MITRE Common Weakness Enumeration (CWE) database highlights a number of common weaknesses related to the improper sharing of hardware resources (e.g., [1], [2]) in the published list of top hardware CWEs [3].

Prior work has explored a variety of approaches for hardware verification, including those focused on ensuring proper resource isolation [4], [5]. Common approaches include fuzz testing [6]–[8], formal verification [9], [10], and automatic security assertion generation with checking [11], [12]. Such methods have shown promise, identifying multiple security vulnerabilities in prominent open-source hardware systems [6]–[8]. However, these methodologies can fail to detect integrity flaws when the underlying security policy is incomplete or implicit. In this work, we present a complementary verification approach that enhances existing methods for detecting hardware design flaws related to the improper isolation of hardware resources. Such vulnerabilities are common and can severely compromise trust in a hardware system if exploited.

In software systems, the proper isolation and controlled sharing of data among software processes has been extensively studied [13]. This led to the introduction of mandatory access control (MAC), which is a formally proven access control policy for environments requiring strict, provable enforcement [14]. MAC enforces access rules regarding how active entities, known as *subjects*, can interact with passive entities, known as *objects*, based on a formal security policy. The Bell-LaPadula model (BLP) was formulated to provide a model through which the MAC policy could be implemented and verified [15]. K. J. Biba later proposed an integrity-focused interpretation of BLP to ensure the integrity of shared data objects is maintained [16]. The Biba interpretation of BLP operates on the principle that higher-integrity subjects should not be corrupted by lower-integrity objects. This ensures that data in a system is not modified by insufficiently trustworthy subjects.

While the Biba interpretation of BLP has been widely studied in the context of software security, its underlying formalism makes it applicable to verify isolation in non-software contexts, as long as the MAC definitions are properly upheld. This motivates our work. We develop and propose mathematical definitions that facilitate the application of MAC in a hardware context. MAC policy models, such as BLP, have been used by major standardization and government agencies to ensure that information confidentiality and integrity of shared data in computing systems is maintained (i.e., data objects are accessed only by authorized parties) [14], [17]. Hence, by extending the MAC policy from a software to a hardware context, well-established methods for verifying MAC policies in information systems can be used to provide a formal, standardized, and automated approach for verifying hardware resource isolation.

### A. Contributions

In this work, we explore isolation of and access to shared hardware resources (e.g., communication interfaces) in embedded systems using MAC policy. To do so, we formalize

hardware-centered definitions of subjects, objects, transactions, and access modes. We then propose and implement a security verification methodology for system integrity that employs random and directed-random testing to verify the integrity of shared hardware resources using the Biba interpretation of BLP. We evaluate this approach using three open-source RISC-V processor cores[1] with built-in memory protection (i.e., a physical memory protection (PMP) module). The contributions of the work are as follows:

1) We propose using MAC policies from the information security domain to verify the integrity and proper isolation of shared hardware resources. We formulate the interactions between hardware resources (e.g., a processor and a DMA peripheral) as a MAC problem, enabling established policy models (e.g., Biba interpretation of BLP) to be used in hardware verification.
2) We formalize definitions of subject, object, transaction, and access-mode for a MAC policy that defines interactions between hardware resources. These definitions allow existing MAC security policy models from information security to be adapted for verifying hardware.
3) We develop a set of rules for secure devices, wherein these rules are representative of MAC policies and, therefore, a violation of these rules constitutes a security violation. Furthermore, we define these rules in such a way that they may be seamlessly integrated with existing verification environments. As such, these rules serve to detect integrity violations in the hardware when triggered by other means (e.g., fuzzing, directed testing, etc.).
4) We implement a SystemVerilog-based hardware verification methodology based on this MAC policy abstraction and demonstrate that the proposed approach can seamlessly integrate into conventional test environments.

To evaluate the proposed approach, we apply it to verify three open-source RISC-V processors. Our approach identifies integrity violations in each core, including one previously undocumented integrity flaw in the CV32E40S [18]. This flaw has been reported and acknowledged by the maintainers of the core, demonstrating the utility of our approach. We release the verification code implemented for the CV32E40S and Ibex as open-source at https://github.com/cnokes14/Hardware-MAC

## II. Preliminaries

### A. Biba Interpretation of Bell-LaPadula Policy Model

The Biba interpretation of the BLP model is an access control policy model to ensure the integrity of data in secure computer systems [16]. Specifically, Biba integrity aims to ensure that a computing system, if properly initialized, is protected from "unprivileged, intentionally malicious modification" [16]. Traditionally, Biba integrity is used in an information security context, focused primarily on how software is designed after a hardware system has been fully specified [19].

[1]Although RISC-V processors serve as our analysis platform, the definitions are generic and can be applied to other hardware systems.

The Biba interpretation of BLP governs the interactions between a set of *subjects (S)* and *objects (O)*. *Subjects* are active, information-processing elements of a system. *Objects* are passive entities that store information. Both subjects and objects have an assigned integer integrity level. Higher integrity levels indicate more trustworthy subjects/objects. To maintain data integrity in a properly initialized system, the Biba interpretation of BLP requires four rules to be enforced:

1) **Simple Integrity:** a subject must not read an object at a lower integrity level (no read down).
2) **Star (*) Integrity:** a subject must not write to an object at a higher integrity level (no write up).
3) **Invocation Property:** a subject must not make service requests of a higher integrity subject or object.
4) **Tranquility:** a subject or object must not change integrity level while being referenced.

### B. Related Work on Hardware Verification

Verification of security in hardware has been widely studied [4], [5], [20]. Common methodologies include functional, formal, assertion-based, and information flow tracking verification. We describe each strategy in turn.

Functional testing involves the direct testing of a device under test (DUT) using a mix of randomized and directed stimuli [4], [21]. Functional testing is commonly performed using a variety of tools in a Universal Verification Methodology (UVM) framework [22]–[24]. UVM's compartmentalized and modular framework allows for quick substitution of elements to perform different tests, or to test multiple similar DUTs with slightly differing components [25]. This also allows high-level verification tools to be built during early stages of RTL development, as well as integration with low level software simulation and verification [26].

Formal verification proves that certain mathematical properties are met by hardware components, which is particularly appealing in the security domain, as it allows for a formal guarantee that certain security policies are upheld [27], [28]. Formal verification, while ideal, is often prohibitively computationally complex in larger devices, making formal verification of security policies at the system level difficult.

Assertion-based testing inserts policies as verification statements that are checked during simulation [5], [29]. These assertion statements must be simultaneously broad and precise. They must cover the growing search space of a device and must be properly defined such that they ensure security policies are maintained. Furthermore, these assertions are often challenging to reach during simulation. As such, a large section of assertion research involves automation of assertion statement generation and tests to trigger them [11], [30], [31]. Assertion-based verification has excelled in the validation of functional requirements, but the verification of non-functional requirements remains an open research question. While assertions have successfully identified various security vulnerabilities [32], [33], the requirements for human expertise and task automation allow some errors to slip through. Ultimately, assertions become a

stronger tool with stronger underlying security policies to define them.

Information flow tracking (IFT) checks to ensure that certain properties of data flow are maintained (e.g., data from different integrity domains are not cross-contaminated) [20], [34], [35]. IFT has been applied to the hardware resource isolation problem and has successfully located certain integrity flaws, however, these applications have either denoted hardware faults as out of scope [36], or have not caught flaws in the shared domain [37]. Prior work has also explored automatically integrating security policy enforcement during synthesis [38], however, such strategies incur hardware overhead and are focused on runtime policy enforcement, rather than verification.

As noted, prior verification methodologies have effectively identified isolation-related bugs in open-source hardware designs. However, we identify two remaining gaps that motivate our work: 1) many existing efforts explicitly out-scope certain hardware faults, limiting the detected vulnerabilities, and 2) integrity violations that arise from bugs in shared domains between devices have been missed. To address these challenges, we propose a verification strategy intended to operate alongside and enhance approaches used in prior work. Specifically, we explore the explicit mapping of a MAC policy onto hardware to check transactions between physical components, thereby narrowing the gap between high-level security policies and low-level hardware interactions.

## III. Security Verification of Hardware with Mandatory Access Control Policy

In order to verify hardware resource isolation using existing MAC policy models, we must construct a mapping of the subject-object paradigms, developed in a software/information security context, onto generic hardware systems. We must provide these definitions in a way that they can be used to draw the same formal conclusions as their software equivalents.

In this section, we first provide motivation for the application of MAC models to hardware. We then define generic hardware subjects, objects, and transactions as applicable to any MAC policy model. We additionally provide the framework for the implementation of the Biba interpretation of BLP using these definitions. To illustrate these definitions, we provide examples of valid and invalid transactions for a single-core processor.

### A. Overview and Motivations

To motivate the proposed approach and formulate the hardware resource isolation problem addressed in this work, we begin with an example. In Fig. 1, we show a system with multiple hardware IPs interacting through a common bus. While this example is generic, it can be expanded to represent any set of shared hardware resources and IPs connected via a bus or network-on-chip. Specifically, Fig. 1 depicts four subjects: a DMA, a DSP, and two subjects originating from a processor (e.g., separate cores); four objects are also defined: a USB interface, two memory segments, and a sensor peripheral. We define each component as a subject or object depending on whether it actively distributes access requests, or passively responds to access requests, respectively. Each subject and object has access to a common bus, by which they may interact with each other through fixed transactions.

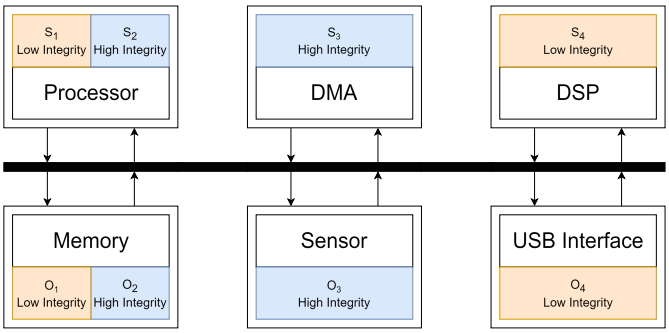


Fig. 1: Hardware subject-object topology

High integrity subject $S_2$ is running biometric authentication, and low-integrity subject $S_1$ is running user code. A hardware vulnerability exists in $S_2$, where, if the shared data bus does not match the Open Bus Interface (OBI) protocol, memory accesses have not been properly specified, leading to undefined behavior. This behavior may result in random memory addresses being accessed via read or write operations.

An attacker controlling $S_1$ wishes to skip to a subroutine that should only occur once authentication is completed. In all low-integrity memory, the attacker stores a jump instruction containing a target subroutine address. They then perform fault injection to violate the OBI protocol on the shared bus and therefore generate undefined memory accesses from $S_2$. One of these undefined instruction fetches accesses a low-integrity section of memory. This sends $S_2$ to the attacker's target subroutine, skipping the authentication stage.

We identify this security violation as a hardware isolation problem, since low-integrity resource $O_1$, which should be readable and writeable by low-integrity core $S_1$, but only writeable by high-integrity core $S_2$, can be read by $S_2$. By allowing a high-integrity subject to read from a low-integrity object, we allow an insufficiently trustworthy component to influence a high-integrity component. This could allow the attacker to reconfigure the device into an insecure state, such as disabling memory protection or bypassing authentication.

This vulnerability may be hard to detect using conventional verification strategies. Automatic test pattern generation and policy-based testing do not necessarily define all possible points of interaction between shared hardware components, meaning invalid requests may not be recognized, even if triggered. While flow-based verification may define all possible interactions, the invalid interaction in this example occurs in a shared domain that is likely to have downgraded integrity [36]. Hence, errors that occur on non-obvious paths with effects that are likely to be considered permissible in the shared domain are a challenge for these methods.

Policy models developed in a software context have been shown to provably identify improper flow of information between processes. In order to detect isolation violations similar

to those in our example, we spend the remainder of this section discussing applications of these policy models in a hardware context. We outline a set of definitions that enable similar theoretical rigor, and with this basis, we define a set of requirements to verify hardware resource isolation.

### B. Definition of Subject

A subject $S$ is an active entity (e.g., a process or thread) that can perform an action on an object resource [16]. In the context of the Biba interpretation of BLP, each subject $S$ has an integrity level $L(S)$, which reflects its trustworthiness, and defines what actions it is allowed to perform on objects of different integrity levels. A subject $S_H$ is considered to be higher integrity than subject $S_L$ if $L(S_H) > L(S_L)$. A subject with higher integrity is assumed to be more secure than one of lower integrity. Thus, we define a hardware subject as follows:

**Definition 1** (Hardware Subject)**.** *Any active hardware component with an integrity level that can initiate transactions with one or more objects.*

We further specify that a hardware subject is the largest possible active component such that, for any section of that component, all other sections of the component must share the same integrity level. Subject integrity is treated as immutable during operation, and is defined by the verifier of the device or security policy. Examples of hardware subjects include CPUs, GPUs, DSPs, and DMA controllers.

### C. Definition of Object

An object $O$ is a passive entity that may be accessed by subjects. Each object has an associated integrity level $L(O)$, which reflects both criticality (for writes) and trustworthiness (for reads), and determines which subjects (by integrity level) may access it and in what ways. An object $O_H$ is considered to be higher integrity than object $O_L$ if $L(O_H) > L(O_L)$. An object with higher integrity is assumed to be more secure than one of lower integrity. We define a hardware object as follows:

**Definition 2** (Hardware Object)**.** *Any passive hardware component with an integrity level that contains device state that can be shared or modified by one or more subjects.*

We further specify that a hardware object is the largest possible passive component such that all sections of the component must share the same integrity level.

For example, we consider sections of memory, cache, hardware peripherals (e.g., a timer or communication interface), and certain internal status and configuration registers to be objects, since they must be accessed by an active subject (i.e., they are passive) and should only be accessed by active components with a sufficient integrity level. If a passive shared component does not have a defined integrity level (i.e., no access permissions or requirements are given by the specification), it is assumed to have the lowest integrity.

In accordance with the Tranquility Property, an object may not have its integrity level changed while it is being accessed.

### D. Definition of Transaction

In the information security domain, subject-to-object interactions (accesses) are classified into four categories: read, write, execute, and append [15]. In the hardware domain, we define subject-to-object interactions as transactions as follows:

**Definition 3** (Transaction)**.** *Any access of an object by a subject in which the state of the object is altered or viewed.*

The only way hardware subjects may access hardware objects is by a read or a write. While execute transactions often exist in hardware state (e.g., the RISC-V PMP has read, write, and execute permission bits [39]), these are considered to be read transactions by BLP [15]. We adopt this same approach. Any requirement of an integrity model that is not specifically a read or a write can be defined as a subset of one or both of these.

### E. Active Component Interactions

Two subjects do not directly communicate. Suppose there is a processor with two cores, each with unique integrity levels, $S_A$ and $S_B$. To properly model interactions between the two, we define mediator objects $O_{A \to B}$ and $O_{B \to A}$, where $O_{A \to B}$ is written to by $S_A$ and read from by $S_B$, and $O_{B \to A}$ is written to by $S_B$ and read from by $S_A$. The integrity of each object is defined by the subject that reads from them (i.e., $L(O_{A \to B}) = L(S_B)$, $L(O_{B \to A}) = L(S_A)$). We show this example configuration in Fig. 2, where $L(S_A) < L(S_B)$.

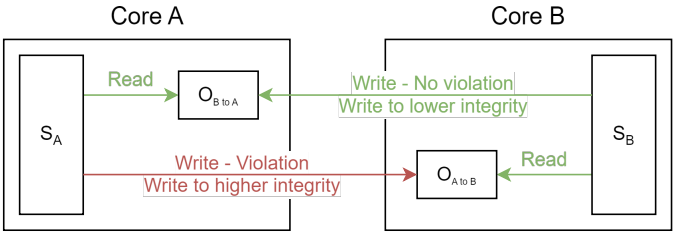


Fig. 2: Interaction between two hardware subjects

For example, for a CPU subject $S_{CPU}$ to send a message to DMA subject $S_{DMA}$, it must send a message to object $O_{DMA}$, which is read by $S_{DMA}$. If $S_{CPU}$ is of lower integrity than $O_{DMA}$, this transaction should be flagged, as a lower level subject should not be able to write to a higher level object.

### F. Rules for Biba Integrity Enforcement in Hardware

Using our definitions of subject and object, we re-frame the Biba interpretation of the BLP in the context of hardware. This is equivalent to the conventional Biba interpretation of BLP.

1) **Simple Integrity Property ("no read down"):** A hardware subject $S$, as previously defined, may read a hardware object $O$ if and only if $L(S) \leq L(O)$. This ensures a trusted subject $S$ cannot read data from an unreliable source object $O$ (e.g., a high-integrity processor core cannot execute instructions retrieved from a compromised low-integrity region of memory).
2) **Star (*) Integrity Property ("no write up"):** A hardware subject $S$, as previously defined, may write to hardware

object $O$ if and only if $L(S) \geq L(O)$. This ensures an untrusted subject $S$ may not alter the state of a trusted object $O$ (e.g., a low-integrity processor core should not be able to disable memory protection).

3) **Invocation Property:** Given that we define all possible hardware transactions as either reads or writes, invocations are considered to be a subset of writes. They are therefore covered inherently by the *-Integrity Property.
4) **Tranquility Property:** The tranquility property asserts that a subject or object must not be allowed to transition to a higher integrity state during normal operation. Normal operation must be predetermined by the policy verifier, and transitions between integrity states that occur during normal operation must be flagged as invalid. A subject or object may freely transition to a lower integrity state, thereby satisfying the *Weak Tranquility Property* (e.g., a user must use system calls to invoke the higher integrity operating system). Alternatively, the *Strong Tranquility Property* holds that a subject or object must never transition integrity level, even to a lower integrity level. To observe the principle of least privilege, we adopt the *Weak Tranquility Property* and, for brevity, refer to it as the *Tranquility Property* for the remainder of this work.

### G. Example: Processor with Three Peripherals

A single-core processor ($S_{\text{Processor}}$) has access to three peripherals ($O_{\text{Memory}}$, $O_{\text{USB Interface}}$, and $O_{\text{Sensor}}$). The processor has a medium integrity level, such that $L(S_{\text{Processor}}) < L(O_{\text{Memory}})$, $L(S_{\text{Processor}}) = L(O_{\text{USB Interface}})$, and $L(S_{\text{Processor}}) > L(O_{\text{Sensor}})$ [2]. $O_{\text{Target}}$ will be used as a general term to reference an object targeted by a transaction attempt by $S_{\text{Processor}}$.

According to the Biba interpretation of BLP, the following properties must be fulfilled in hardware:

**Simple Integrity - No Read Down:** The processor may read from the peripheral if and only if $L(S_{\text{Processor}}) \leq L(O_{\text{Target}})$. Otherwise, any read access from $S_{\text{Processor}}$ to $O_{\text{Target}}$ is invalid.

- **Valid Interaction:** The hardware subject $S_{\text{Processor}}$ attempts to read from object $O_{\text{Target}} = O_{\text{Memory}}$. $S_{\text{Processor}}$ therefore has the same or lower integrity level than $O_{\text{Target}}$ ($L(S_{\text{Processor}}) \leq L(O_{\text{Target}})$). The processor reads from memory. This is a valid transaction, since the lower integrity (untrustworthy) subject may depend on a higher integrity (trustworthy) object. This ensures that a higher level subject may not retrieve data from lower integrity objects that may introduce vulnerabilities.
- **Violation:** The hardware subject $S_{\text{Processor}}$ attempts to read from $O_{\text{Target}} = O_{\text{Sensor}}$. $S_{\text{Processor}}$ therefore has a greater integrity level than target peripheral $O_{\text{Target}}$ ($L(S_{\text{Processor}}) > L(O_{\text{Target}})$). This is an invalid transaction, since it allows the higher integrity subject to use data from the lower integrity object. This allows lower integrity components to introduce vulnerabilities to higher integrity components (e.g., executing untrusted code).

[2]This example takes place while the processor is in a high integrity state. The processor may switch to a lower integrity state according to the Tranquility Property, so that it may read from the sensor.

**Star (*) Integrity - No Write Up:** The processor may write to the peripheral if and only if $L(S_{\text{Processor}}) \geq L(O_{\text{Target}})$. Otherwise, any write access from $S_{\text{Processor}}$ to $O_{\text{Target}}$ is invalid.

- **Valid Interaction:** The hardware subject, $S_{\text{Processor}}$, writes to object $O_{\text{Target}} = O_{\text{USB Interface}}$. $S_{\text{Processor}}$ has an integrity level greater than or equal to a peripheral $O_{\text{Target}}$ ($L(S_{\text{Processor}}) \geq L(O_{\text{Target}})$). This is valid, as it is permissible for an insecure object to depend on a secure subject.
- **Violation:** The hardware subject, $S_{\text{Processor}}$, writes to object $O_{\text{Target}} = O_{\text{Memory}}$. $S_{\text{Processor}}$ therefore has a lower integrity level than a peripheral $O_{\text{Target}}$ ($L(S_{\text{Processor}}) < L(O_{\text{Target}})$). This is an invalid interaction, since it allows lower integrity components to distribute untrustworthy information to higher integrity components (e.g., writing malicious code to secure instruction memory).

### H. MAC Policy Models for Hardware Security Verification

Using these definitions of hardware subjects and objects, we frame isolation violations as invalid transactions. We propose a verification method in which each path on which transactions occur is monitored; we call this monitor a *transaction checker*. These transaction checkers are deliberately designed to be applied to most widely used forms of hardware verification (i.e., as instantiated blocks during simulation, as assertions, as formal checking policies, etc.). The sole underlying requirement is that all transactions that occur are passed through a transaction checker, which compares them against the MAC model.

We define transaction checkers to have three stages: translation, where signals are converted to transactions; mapping, where a transaction is mapped from a subject to an object; and verification, where the transaction is checked against the set of valid accesses according to the underlying security policy.

To demonstrate the application of this checker, we consider a system with two processor cores ($S_H$ and $S_L$) and two memory segments ($O_H$ and $O_L$), connected by a shared bus. By our definition of hardware subjects, we must also define objects to mediate subject-to-subject interactions ($O_{S_H \to S_L}$ and $O_{S_L \to S_H}$), as described in Section III. Using this, we may define all valid transactions that may occur between subjects and objects according to the Biba interpretation of BLP.

- $S_H$ and $S_L$ may write to $O_L$. $S_L$ may read from $O_L$.
- $S_H$ and $S_L$ may read from $O_H$. $S_H$ may write to $O_H$.
- $S_H$ and $S_L$ may write to $O_{S_H \to S_L}$. $S_L$ may read from $O_{S_H \to S_L}$.
- $S_H$ and $S_L$ may read from $O_{S_L \to S_H}$. $S_H$ may write to $O_{S_L \to S_H}$. Because this object is to handle messages sent from $S_L$ to $S_H$, it should be entirely inactive.

To verify this system, we develop a set of *transaction checkers* which compare actual transactions between shared components against the set of valid interactions defined above. These transaction checkers should be placed on all possible paths through which $S_H$ and $S_L$ may write to the shared data bus. We do not need to define transaction checkers on paths where objects write to the shared data bus because objects are passive by definition; if an object could perform actions without

a request from an active object, then it must be defined as a subject. We note that these checkers are passive verification elements *only*, and are not included in the synthesized design.

A transaction checker must know the subjects, objects, and access control matrix. This enables the checker to verify all information transmitted into the shared domain.

The subjects and objects in a system, as well as their integrity levels, are inferred from the system specification using the definitions provided in Sec. III-B/III-C. We consider any subject or object with an undefined integrity level to be at the lowest possible integrity level. In the following sections, we evaluate this approach by implementing these transaction checkers for the integrity verification of three open-source RISC-V cores.

## IV. Evaluation of MAC-Based Security Verification

To evaluate the proposed MAC-based formulation for the security verification of hardware, we applied our subject and object definitions to three open-source cores: OpenHWGroup's CV32E40S [18], lowRISC's Ibex [40], and OpenRISC's MOR1KX [41]. These specific cores were selected because they are broadly adopted and have been used as benchmarks by prior work, facilitating comparison. These cores are solely used to demonstrate the applicability of the proposed verification methodology. We draw no comparisons of these cores with respect to alternative RISC-V cores. To begin the evaluation, we outline the experimental setup used for verification.

### A. Verification Environment for RISC-V Cores

To validate the implementation of hardware isolation features, we first defined a set of subjects and objects in each core based on the definitions presented in Section III. We then determined all paths by which transactions between subjects and objects may occur. Each transaction checker took into account only the necessary transaction details: integrity status of the core, read/write enable, address, and security policy. Each transaction checker was defined as a SystemVerilog (CV32E40S and Ibex) or Verilog (MOR1KX) module. The generic layout of the test environment is shown in Fig. 3.

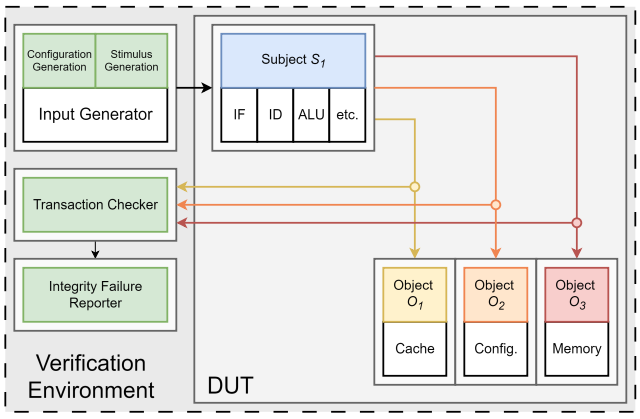


Fig. 3: Transaction checker test environment example layout

When checked, a transaction goes through the following steps: 1) The transaction is converted into a generic read or write transaction[3] based on previous definitions in Sec. III. 2) The transaction is mapped from a specific subject to a specific object. 3) The transaction is checked for in the set of all valid transactions as defined by the Biba interpretation of BLP, and by the system-specific security policy. If the transaction does not exist in that set, then it is flagged as a violation.

Given this methodology, it is possible to apply design checkers to various stimulus-based and formal testing environments. For the evaluation presented in this work, we opted to primarily use randomized stimulus-based testing. However, a formal-methods-based approach will be considered in future work.

### B. Evaluation Test Cases

To evaluate the proposed MAC-based verification of hardware resource isolation, we implemented the verification approach outlined in Sec. III for three prevalent open-source RISC-V processors. Each core was run at the most recent version of the public repository, with the exception of the Ibex, which was tested using an older version of the repository to detect a bug that had been present in live code.

We additionally verified that the test environment could detect multiple errors artificially inserted by the authors, which are based on bugs that have been identified as hard to locate by other verification means [42]–[44]. These examples were run on the most recent version of the Ibex due to its stability. A set of all test cases run is shown in Table I.

TABLE I: Test cases for the transaction checker environment. Bugs marked as artificial were inserted by the authors to recreate historically hard to find bugs; bugs marked as novel were not documented prior to this work.

| Bug # | Core | Artificial? | Novel? | Property Violated? |
|---|---|---|---|---|
| Bug 1 | CV32E40S | No | Yes | Simple, *-Integrity |
| Bug 2 | Ibex | No | No | Simple Integrity |
| Bug 3 | MOR1KX | No | No | *-Integrity |
| Bug 4 | Ibex | Yes | No | Simple, *-Integrity |
| Bug 5 | Ibex | Yes | No | *-Integrity |
| Bug 6 | Ibex | Yes | No | Simple, *-Integrity |

In each core, we ran randomized simulations with transaction checkers enabled. We report the discovered violations below; violations discovered in live code are given a more verbose explanation, whereas violations deliberately inserted by the authors are described only briefly to compare to existing work.

#### 1) *Bug 1: CV32E40S - Undefined Memory Access*

The bug identified in the CV32E40S, which was previously unknown, originates from the load-store unit and the bus interface. Both of these components have FIFOs with a depth of three, but with an unprotected access of two bits (i.e., an

[3]For the RISC-V cores evaluated, defining a transaction as read or write usually only involved tapping the write enable signal.

index of $11_2$ would cause an index out-of-bounds error). The core uses the OBI standard for bus communication; when this protocol is violated, it is possible for this buffer overflow error to occur. This results in various signals in the core having undefined values, which allows the instruction read enable bit to be undefined while the address points to an invalid peripheral or segment of memory. If an attacker were to violate the protocol of the bus, they may force the CV32E40S to perform an invalid read while at a high integrity level, allowing the attacker to perform code injection. This is a violation of the Simple Integrity Property. This was discovered by the transaction checker placed between the active component $S_{Core}$ and its interface with the bus. The transaction checker would map the transaction to a memory segment object $O_{\text{Memory Segment N}}$, and if an fetch occurred such that $L(S_{Core}) > L(O_{\text{Memory Segment N}})$, an invalid transaction was raised.

*2) **Bug 2: Ibex - Insufficient Cache Checking***

The bug present in the Ibex originates from the instruction fetch. In a prior version of the repository (commit *5e7c2cf*), instruction request address checking by the PMP was only performed for requests leaving the core. Requests sent to the instruction cache were not checked, and the cache was not reset when switching integrity modes. If an attacker were to fill the instruction cache with insecure data, they may allow the fetch to access these insecure instructions, thereby performing code injection. This is a violation of the Simple Integrity Property. We detected this violation using the transaction checker placed between the instruction fetch unit $S_{Fetch}$ and the cache. Depending on the target address of the cache access, the transaction checker would map the request to a memory segment object $O_{\text{Cache Segment}}$. The invalid transaction alert was raised whenever a fetch occurred such that $L(S_{Fetch}) > L(O_{\text{Cache Segment}})$, where $L(O_{\text{Cache Segment}})$ is equivalent to the integrity level of the associated memory block.

*3) **Bug 3: MOR1KX - Insufficient Configuration Control***

The bug present in the MOR1KX originates from the configuration registers. Checks of the integrity level of the core against the required permissions of the configuration register are insufficient for many registers. This allows an attacker to write to privileged registers in an unprivileged state, presenting a clear violation of the Star Integrity Property. We detected this violation using the transaction checker placed between the core $S_{Core}$ and its access path to the configuration registers. The transaction checker would map the request to a configuration register object $O_{\text{Configuration Register N}}$. The invalid transaction alert was raised whenever a write request to an internal register occurred, such that $L(S_{Core}) < L(O_{\text{Configuration Register}})$.

*4) **Bug 4: Ibex - Erroneous Data Access Address***

This bug, inserted manually by the authors, randomly alters the output data request address after it was checked by the PMP. This bug represents arbitrary data changes that do not violate flow policies and occur after the policy checking process [36], [37]. This bug was detected by the transaction checker when the erroneous request accessed an invalid section of memory.

*5) **Bug 5: Ibex - Failed Memory Protection Lock***

This bug, inserted manually by the authors, disabled locking on the Machine-Mode Whitelist Policy (MMWP) and Machine-Mode Lockdown Policy (MML) registers [44]. This allows the lock to be disabled outside of reset, which is forbidden by the RISC-V standard [39]. Removing these protections would allow a user to incorrectly alter the PMP during runtime, allowing invalid memory accesses to occur. This bug was detected during a directed test that incorrectly reconfigured the PMP, then ran an invalid memory instruction, which was flagged.

*6) **Bug 6: Ibex - Incorrect Reset State***

This bug, inserted manually by the authors, sets a small percentage of randomly selected control and status registers to have an incorrect value after reset [42], [43]. This bug was detected because certain control and status registers may allow bypassing of protection mechanisms, thereby allowing erroneous access to protected memory.

### *C. Discussion and Analysis*

These results demonstrate the application of MAC, traditionally limited to the information security and software domains, to hardware isolation verification. Using only the provided subject, object, and transaction definitions, we successfully identified security vulnerabilities in each of the target RISC-V cores. These vulnerabilities originate from hardware flaws, and are therefore not identifiable using conventional MAC enforcement, which considers only transactions that have software-observable side-effects. Moreover, in the case of the CV32E40S, a security-focused core in ongoing development, a security vulnerability that was previously unknown was discovered and disclosed to the maintainers using the proposed methodology.

Table II shows the simulation runtime of each processor across ten million cycles with and without the transaction checker enabled. This provides an estimate of the overhead added to existing verification environments.

TABLE II: Simulation time for 10 million cycles with and without the transaction checker

| **Configuration** | **Min (s)** | **Max (s)** | **Avg (s)** | **Sim. Time Overhead** |
|---|---|---|---|---|
| CV32E40S No Checker | 464 | 492 | 481 | - |
| CV32E40S w/ Checker | 490 | 546 | 524 | 9% |
| Ibex No Checker | 104 | 113 | 107 | - |
| Ibex w/ Checker | 104 | 121 | 111 | 3% |
| MOR1KX No Checker | 199 | 244 | 239 | - |
| MOR1KX w/ Checker | 236 | 253 | 246 | 3% |

As shown, simulation run time increased by less than 10% in each processor with the transaction checkers included. Additionally, further runtime overhead reduction could be achieved through continued optimization or the use of assertion statements. Based on the evaluation, the proposed hardware resource isolation verification methodology effectively detected bugs that have proven hard to identify in prior work. Moreover, the proposed approach can be cleanly integrated alongside

existing verification strategies, with minimal overhead, to enhance the effectiveness of security verification.

## V. Conclusion

We mapped MAC policy models, conventionally used in the information security domain, to the hardware domain to facilitate the verification of hardware resource isolation. We performed this mapping in such a way that existing MAC policy models, specifically the Biba interpretation of BLP in this work, can be used to determine whether an arbitrary transaction in hardware violates device integrity. This enables security verification of hardware against MAC policy models. Our definitions were used to develop transaction checker components in three open-source RISC-V devices. These components identified integrity violations (i.e., security vulnerabilities) in each device.

## Acknowledgment

The opinions, recommendations, findings, and conclusions in this article do not necessarily reflect the views or policies of NIST or the United States Government.

## References

[1] MITRE Corporation, "CWE-1189: Improper Isolation of Shared Resources on System-on-a-Chip (SoC)," 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/1189.html

[2] ——, "CWE-1191: On-Chip Debug and Test Interface with Improper Access Control," 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/1191.html

[3] ——, "2021 CWE Most Important Hardware Weaknesses," 2021. [Online]. Available: https://cwe.mitre.org/scoring/lists/2021_CWE_MIHW.html

[4] R. Saravanan *et al.*, "The fuzz odyssey: A survey on hardware fuzzing frameworks for hardware design verification," in *Proceedings of the Great Lakes Symposium on VLSI 2024*, 2024, pp. 192–197.

[5] H. Witharana *et al.*, "A survey on assertion-based hardware verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–33, 2022.

[6] J. Hur *et al.*, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[7] V. Gohil *et al.*, "Mabfuzz: Multi-armed bandit algorithms for fuzzing processors," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[8] R. Kande *et al.*, "{TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities," in *31st USENIX Security Symposium*, 2022.

[9] X. Guo *et al.*, "Automatic code converter enhanced pch framework for soc trust verification," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 12, pp. 3390–3400, 2017.

[10] C. Kern *et al.*, "Formal verification in hardware design: a survey," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 2, pp. 123–193, 1999.

[11] H. Witharana *et al.*, "Automated generation of security assertions for rtl models," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 19, no. 1, pp. 1–27, 2023.

[12] A. Ayalasomayajula *et al.*, "Prioritizing information flow violations: Generation of ranked security assertions for hardware designs," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2024, pp. 128–138.

[13] R. Shu *et al.*, "A study of security isolation techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, pp. 1–37, 2016.

[14] D. C. Latham, "Department of defense trusted computer system evaluation criteria," *Department of Defense*, vol. 198, 1986.

[15] D. E. Bell *et al.*, "Secure computer system: Unified exposition and multics interpretation," *Defense Technical Information Center*, 1976.

[16] K. J. Biba *et al.*, "Integrity considerations for secure computer systems," 1977.

[17] V. C. Hu *et al.*, "Verification and test methods for access control policies/models," *NIST Special Publication*, vol. 800, p. 192, 2017.

[18] OpenHW Group, "CV32E40S," 2021. [Online]. Available: https://github.com/openhwgroup/cv32e40s

[19] K. Harley *et al.*, "Information integrity: Are we there yet?" *ACM Computing Surveys (CSUR)*, vol. 54, no. 2, pp. 1–35, 2021.

[20] W. Hu *et al.*, "Hardware information flow tracking," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–39, 2021.

[21] A. Jayasena *et al.*, "Directed test generation for hardware validation: A survey," *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–36, 2024.

[22] S. Qamar *et al.*, "A comprehensive investigation of universal verification methodology (uvm) standard for design verification," in *Proceedings of the 2020 9th International Conference on Software and Computer Applications*, 2020, pp. 339–343.

[23] N. Harshitha *et al.*, "An introduction to universal verification methodology for the digital design of integrated circuits (ic's): A review," in *International Conference on AI and Smart Systems*, 2021.

[24] D. Lohmann *et al.*, "Extending universal verification methodology with fault injection capabilities," in *2018 IEEE 9th Latin American Symposium on Circuits & Systems (LASCAS)*. IEEE, 2018, pp. 1–4.

[25] R. Wang *et al.*, "The art of portable and reusable uvm shared system memory model verification methodology across multiple verification platforms," in *Proc. Design Verification Conf.(DVCon US 2017)*, 2017.

[26] S. Morgansgate *et al.*, "Generic core-monitor for hardware/software co-debugging targeting emulation platform," *Proceedings of the Design and Verification Conference and Exhibition*, 2022.

[27] A. L. Duque Antón *et al.*, "Vericheri: Exhaustive formal security verification of cheri at the rtl," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[28] K. Cheang *et al.*, "Verifying risc-v physical memory protection," *arXiv preprint arXiv:2211.02179*, 2022.

[29] U. Alsaiari *et al.*, "Hardware trojan detection using reconfigurable assertion checkers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 7, pp. 1575–1586, 2019.

[30] M. W. Anwar *et al.*, "A unified model-based framework for the simplified execution of static and dynamic assertion-based verification," *IEEE Access*, vol. 8, pp. 104 407–104 431, 2020.

[31] Z. Yan *et al.*, "Assertllm: Generating hardware verification assertions from design specifications via multi-llms," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, 2025, pp. 614–621.

[32] H. Witharana *et al.*, "Directed test generation for activation of security assertions in rtl models," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 4, pp. 1–28, 2021.

[33] A. Ayalasomayajula *et al.*, "Automatic asset identification for assertion-based soc security verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 10, 2024.

[34] W. Hu *et al.*, "Property specific information flow analysis for hardware security verification," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[35] A. Meza *et al.*, "Security verification of the opentitan hardware root of trust," *IEEE Security & Privacy*, vol. 21, no. 3, pp. 27–36, 2023.

[36] A. Ferraiuolo *et al.*, "Verification of a practical hardware security architecture through static information flow analysis," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 555–568.

[37] C. Song *et al.*, "Hdfi: Hardware-assisted data-flow isolation," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 1–17.

[38] S. Paria *et al.*, "Dispel: A framework for soc security policy synthesis and distributed enforcement," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2024.

[39] RISC-V International, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture ," 2025. [Online]. Available: https://riscv.org/specifications/ratified/

[40] lowRISC, "Ibex," 2018. [Online]. Available: https://github.com/lowRISC/ibex

[41] OpenRISC, "mor1kx," 2012. [Online]. Available: https://github.com/openrisc/mor1kx

[42] MITRE Corporation, "CWE-276: Incorrect Default Permissions," 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/276.html

[43] ——, "CWE-1221: Incorrect Register Defaults or Module Parameters," 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/1221.html

[44] ——, "CWE-1231: Improper Prevention of Lock Bit Modification," 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/1231.html