计算机研究与发展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(3): 768−781, 2026

RISC-V SBI 固件安全启动过程的形式化验证

杨叶倩　戴鸿君
（山东大学软件学院　济南　250101）
（山东大学智能创新研究院　济南　250101）
（202320842@mail.sdu.edu.cn）

Formal Verification for Safe Startup Process of the RISC-V SBI Firmware

Yang Yeqian and Dai Hongjun
（*School of Software*, *Shandong University*, *Jinan* 250101）
（*Academy of Intelligent Innovation*, *Shandong University*, *Jinan* 250101）

Abstract　The RISC-V SBI (supervisor binary interface) specification establishes a standardized interface between RISC-V hardware architectures and operating systems. In comparison to traditional firmware implementations for x86 and ARM platforms, the RISC-V SBI firmware demonstrates superior compliance with architectural standards and offers enhanced reliability. However, the current SBI implementations still exhibit significant security concerns, particularly regarding unverified boot process correctness and vulnerabilities in PMP (physical memory protection) mechanisms for physical address access control. To this end, a closed-loop formal verification framework driven by theorem proving is proposed, developing the first formally verified SBI firmware—SeSBI. By integrating Dafny and Isabelle/HOL, a layered strategy is employed to formally verify specification properties across startup process. And to improve verification efficiency, a RISC-V hardware simulation strategy is introduced, unifying the modeling of firmware-hardware interactions, thereby significantly reducing verification overhead. Finally, the functional correctness of SeSBI in the boot process is successfully verified. The results demonstrate that SeSBI implements the SBI standard boot interface with more concise code and high reliability, providing the first formally verified, trusted SBI firmware implementation for the RISC-V ecosystem. Future work will expand verification coverage to more advanced properties and optimize tooling for greater efficiency.

Key words　RISC-V; SBI firmware; formal verification; theorem proving; functional correctness

摘　要　RISC-V SBI（supervisor binary interface）标准定义了 RISC-V 架构硬件与操作系统的接口规范。相比 x86 和 ARM 固件，SBI 固件更合规、可靠；但现有版本仍存在启动正确性未验证、PMP（physical memory protection）物理地址访问等安全漏洞。为此，提出了定理证明驱动的闭环形式化验证框架，开发经形式验证的 SBI 固件——SeSBI，并结合 Dafny 与 Isabelle/HOL，以分层策略实现对启动过程的形式验证。为提高验证效率，提出 RISC-V 硬件模拟策略，统一建模固件与硬件的交互，从而显著降低验证开销。最终成功验证了 SeSBI 固件启动过程的功能正确性。结果表明，SeSBI 以更简洁的代码实现 SBI 标准启动过程接口，并具备高度可靠性，为 RISC-V 生态系统提供了经形式验证的可信 SBI 固件实现。未来工作将扩展验证范围至更多高级属性，并优化验证工具以提升效率。

收稿日期：2024-11-25；**修回日期**：2025-09-11
基金项目：国家重点研发计划项目（2023YFB4503901）；山东省重大科技创新工程项目（2022CXGC020103）；济南市新高校 20 条项目（202333041）
This work was supported by the National Key Research and Development Program of China (2023YFB4503901), the Key Scientific and Technological Innovation Project of Shandong Province (2022CXGC020103), and the New 20 Measures for Universities in Jinan (202333041).
通信作者：戴鸿君（dahogn@sdu.edu.cn）

关键词 RISC-V；SBI 固件；形式化验证；定理证明；功能正确性

中图法分类号 TP316；TP302；TP309

DOI: 10.7544/issn1000-1239.202440915 **CSTR:** 32373.14.issn1000-1239.202440915

可靠且安全的基础软件(如操作系统)已成为现代软件技术开发和应用中的重要研究趋势[1]。与传统基础软件不同，固件通过一组外围设备与外界交互，其代码更接近底层，重要部分通常由不安全的低级语言构建，如 C 语言。偏底层的特性为固件埋下安全隐患[2]。因此，固件的安全性亟需保障。

随着开源指令集 RISC-V 的发展，具有接口标准约束的监管二进制接口 SBI(supervisor binary interface)固件相比其他没有标准规定的固件更加规范和标准化[3]。图 1 展示了 RISC-V SBI 规范。

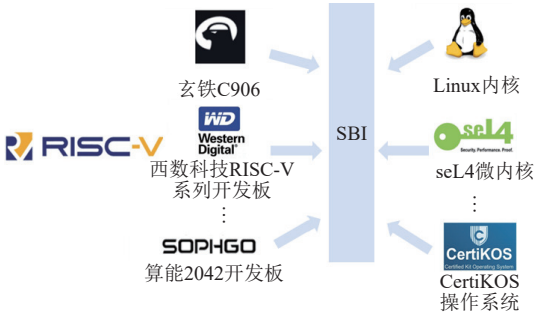


Fig. 1 Specification for RISC-V SBI

图 1 RISC-V SBI 规范

由图 1 可知，SBI 规范确保了固件的跨平台特性，即不论 RISC-V 生态系统内的底层硬件有何差异，固件与操作系统之间的基本交互都能保持一致。对于在不同的 RISC-V 平台上开发可移植和可靠的操作系统，该一致性促进 SBI 固件在不同环境中的广泛适用。

截至目前，市面上已有许多 SBI 固件实现，例如 OpenSBI[4] 和 RustSBI[5]，但这些固件仍存在安全问题：代码越多，漏洞存在的可能性就越大[6]；固件可信执行环境(trusted execution environment, TEE)验证依旧存在漏洞等。由此来看，若 SBI 固件的代码实现更加精简，并且其安全特性能够得到验证，可在一定程度上缓解在安全方面的风险。

近年来，随着形式化验证(formal verification, FV)技术的发展，越来越多的软件程序通过该技术进行验证，其可靠性大大提高。通过形式化验证技术，采用基于严谨的逻辑推理，能够对目标系统进行形式规约及推理验证，确保实现与其规范一致，不存在代码漏洞[7]。据统计，截至目前针对 SBI 固件的形式化验证研究十分匮乏，原因主要在于：

1)缺乏 RISC-V SBI 固件的定制形式化验证框架。SBI 固件与底层硬件交互，执行包含不同的权限模式，其数据变量的定义存在冗余性，形式化验证较为复杂。

2)缺乏 SBI 抽象标准需求的形式化验证。SBI 标准正不断被更新，目前已更新至 3.0 版本，说明 SBI 标准仍存在一定的问题。

而相比于其他架构固件，RISC-V SBI 固件的形式化验证存在先天性优势：具备一定的需求标准。若能利用该条件形式化验证 SBI 固件，开发出可信 SBI 固件，将是 RISC-V 架构生态系统的一块里程碑。

针对上述问题，本文提出了一种 RISC-V SBI 固件形式化验证框架，并基于 SBI 规范与形式化验证结果，迭代开发了一种已通过形式化验证的 SBI 固件——SeSBI(SecuritySBI)。本验证框架引入分层验证策略(layered verification strategy)，通过分别验证可执行规范层与抽象规范层，能够在 SBI 固件开发早期发现漏洞，减少后期测试和维护的成本。目前，SeSBI 源代码已在 GitHub[8] 发布，后续将随时进行更新。

本文主要展示了 SeSBI 的形式化验证框架组成及固件实现，其包含分层形式化验证过程及固件源码。此外，本文成功解决了 5 个关键技术挑战：

1)为提高验证效率，采用基于统一特权级的硬件语义对 SeSBI 系统状态进行建模，并实现其理论接口。

2)采用硬件模拟策略形式化描述固件系统状态，解决验证器无法直接模拟程序的硬件环境的问题。

3)填补当前 RISC-V SBI 固件形式验证框架缺口。

简而言之，本项工作的主要贡献包括：

1)提出基于 Dafny 与 Isabelle 的 RISC-V SBI 固件分层形式化验证框架，定制化 SBI 固件形式化验证。

2)基于该框架与 SBI 规范，设计并实现新的 SBI 固件 SeSBI。该固件足够简洁、紧凑，适合实际使用。

3)通过分层策略验证 SBI 固件在抽象层与可执行规范层的安全属性，确保全方面验证 SeSBI。

4)提出 RISC-V 固件硬件模拟策略，并采用层间一致性精化关系推理，提升可执行层的形式化验证的验证速度。

5)提出统一的固件系统状态规约语义，完善

Isabelle 对验证目标语言切换的统一理论接口。

1 相关工作

Dijkstra[7] 首创形式化验证的概念，即通过严格的逻辑推理开发正确可靠的程序。程序测试能证明错误的存在，但不能证明错误的不存在。形式化验证技术采用严格语义的逻辑推理，可证明程序软件不存在漏洞。这是因为，逻辑语义能够完整描述系统的状态和特征，令研究人员在程序开发早期挖掘代码中的需求和漏洞[9]。

目前，形式化验证技术被广泛应用于多个领域，如航空航天、可信基础软件等。2015 年，美国军方对“小鸟”无人机的核心控制程序进行了形式化验证，成功弥补潜在的安全漏洞，并抵御了黑客程序针对控制系统长达 6 个月的攻击。基础软件的形式化验证也吸引着越来越多研究人员的注意[10-15]，如：著名的嵌入式操作系统微内核 SeL4(secure embedded L4)的安全性和功能正确性已经自 2009 年起逐步经过了完整的形式化验证[10]。操作系统内核开发项目 CertiKOS 创建了安全、可验证的操作系统内核，确保在多核处理器上运行的应用程序具备正确性和安全性[15]。

根据国内外现有研究，针对基础软件的形式化验证技术主要分为 2 种：模型测试(model checking)和定理证明(theorem proving)[16]。二者的主要区别在于使用的逻辑规则不同。模型测试是基于有限状态机(finite-state machine, FSM)模型对所要研究的软件系统目标进行建模，通过自动搜索有限状态空间及其搜索路径来建立模型与规范之间的一致性。而定理证明是将软件系统和系统规范或需求基于某类形式化语言表达为数学逻辑公式，从公理的角度描述软件模型的需求，其所验证的软件模型的范式是不受限制的，因此，定理证明广泛应用于基础软件设计正确性的验证。

而操作系统底层软件作为计算机系统的核心，确保其安全性和可靠性至关重要。可信微内核 SeL4 是由 Gernot Heiser 团队开发的经过完整形式化验证的 L4 微内核[10]。自 2004—2014 年内，该团队逐步对 SeL4 模块的各个部分进行形式化验证，从而逐步改进提高各模块的可靠性。在形式化验证过程中，该团队采用分层策略，将 SeL4 的形式化验证分为 3 层，即抽象规范(abstract specification)层、可执行规范(executable specification)层和手动 C 代码实现层。其验证从中间层开始，逐渐向不同的方向扩展验证。此外，在层间一致性验证过程中，SeL4 微内核的系统状态基于硬件的状态机建模，包括寄存器、内存等。通过验证不同层的系统状态转换的一致性，成功证明了 SeL4 功能的实现完全基于所开发的规范层。国内研究人员也展开了对微内核形式化验证的相关研究，例如：钱振江团队[17] 针对操作系统复杂庞大的设计、其正确性难以描述并验证的问题，提出了一种汇编级的系统状态模型，使用定理证明工具 Isabelle/HOL 对其进行描述并成功验证了系统模块的功能语义的正确性，基于此，最终实现了安全可信操作系统——VSOS。

综上所述，目前在嵌入式操作系统领域，形式化验证主要用于 Linux 内核[9-17]，很少涉及对固件的相关研究，特别是 RISC-V 体系结构固件。并且，内核形式化验证的技术无法适用于 SBI 固件。这是因为，SBI 固件的系统调用既需要与底层硬件通信，也需要与系统软件交互，其主要功能所在固件代码由 C 语言与汇编语言组合而成[18]。而内核的关键功能则主要由 C 语言组成[19]。因此，不管从编写语言还是与底层硬件的交互，内核在形式定义与规约时均比固件更简单。

而在 RISC-V 架构方面，针对系统安全已有一些验证工作正在推进[20-22]，如芮志清等人[20] 通过国密算法强化了 RISC-V 安全启动链；RISC-V 内存模型的同地址顺序一致性经过了定理证明[21]。但这些工作依赖硬件改造、架构适配，或仅验证其他结构的属性，均未解决 SBI 固件的形式化验证问题。并且，SBI 固件的形式验证技术需要对 SBI 固件进行完整的 RISC-V 架构硬件与软件行为上的模拟。然而，现有 Linux 的形式验证技术多从软件行为出发，无法彻底解决硬件行为模拟问题。与其他形式化的验证研究对象相比，SBI 固件位于操作系统体系结构的底部，并且在初始化底层硬件设备和引导操作系统方面起着更关键的作用[23-24]。除此之外，SBI 固件标准开源并符合一定的标准规范，这种特性导致 SBI 固件进行形式化验证具有“天然”的优势。因此，开发一个经过形式化验证的 SBI 固件对于 RISC-V 生态系统来说，具有里程碑式的意义。

2 预备知识

2.1 RISC-V SBI

RISC-V SBI 全称为监管二进制接口标准，制定了底层硬件设备与操作系统的接口规范，存在于操作系统内核与机器固件之间的标准化交互层中[25]。

SBI 标准设计框架包含 3 种不同的执行模式：用户模式（U 模式）、监管模式（S 模式）、机器模式（M 模式）。这 3 种执行模式的执行权限级别依次升高。其中，U 模式允许安全执行应用程序，S 模式有效管理系统资源和中间服务，M 模式提供强大的系统控制和直接的硬件交互。SBI 通过为 S 模式和 M 模式之间的服务请求定义标准接口来促进这些交互，从而简化软件开发，增强系统的安全性和可移植性。

其中，M 模式直接管理硬件资源和系统级中断，是权限最高的模式，功能主要为：发送处理器间中断（IPI）的中断管理；为调度程序设置定时器；管理性能计数器等，对于系统性能监控至关重要。

目前，SBI 固件已被许多研究人员基于 SBI 规范以不同方式开源实现，如前文提到的 OpenSBI 和 RustSBI。其中，OpenSBI 是基于 C 语言和汇编语言的比较完整的 SBI 固件，实现了多个 SBI 接口，包括时钟扩展接口、PMU 扩展接口、IPI 扩展接口等。而 RustSBI 基于嵌入式开发语言 RUST 编写，与 OpenSBI 相比具有更快的开发速度，并将 SBI 固件系统与 RUST 状态相结合。然而，由于未经过严格的形式化验证，它们仍存在一定的问题。而传统的程序测试无法证明其所具备的特性，从而无法保证其运行的正确性。

因此，基于 RISC-V SBI 固件形式化验证，目前还未出现相应的研究，但其对 RISC-V 生态来说，具有十分重要的意义。

2.2 形式化验证工具及系统软件验证

本文的形式验证框架基于 Isabelle/HOL 与 Dafny 组成，二者均为现有常用的交互式验证工具。

首先，Isabelle/HOL 是国际上应用最广泛的交互式定理证明器之一[26]，采用高阶逻辑作为核心建模语言。基于该工具，研究人员能够以可读性较高的 Isar 脚本描述公理以及推理规则与待证定理，通过内置和外接的 Sledgehammer 工具、Nitpick、SMT 等自动化组件快速搜索证明子目标，再由用户利用结构化文本完成人机协作式证明。Isabelle/HOL 提供模块化理论库和强大的 code generation 框架，可将经过机器检验的函数导出为 SML 类型、OCaml 类型、Haskell 类型 或 Scala 类型代码，便于将形式化结果嵌入工程系统。

然后，Dafny 是微软研究院推出的面向验证的命令式/函数式混合语言及其自动证明环境[27]。其内建了不可变/可变数据结构、代数数据类型与后置条件、前置条件、循环不变式等规范机制，借助后台 Z3 SMT 求解器实现“单键式”自动验证，即研究人员能够在 Dafny 代码中写下形式化规约，与此同时，编译器验证相应的逻辑规则，在绝大多数情形下无需交互式干预即可完成证明。并且，Dafny 支持逐步精化和模块化验证，能够从抽象接口逐级落地至可执行细节，并保持机器可检的一致性保证。因此，在系统软件场景中，Dafny 常用来验证内核模块、分布式协议及安全关键算法，为上层应用提供形式化可信根。

一般来说，系统软件形式验证[28]是基于上述验证工具，利用数学逻辑与自动化推理技术，对操作系统、固件及关键运行时库的功能正确性、安全属性和并发一致性进行机器可检验的证明。验证流程主要包括：

1）抽象语义建模。将系统调用、异常处理等行为刻画为状态机或操作语义。

2）属性规格化。前/后置条件、不变式或时序逻辑形式化安全、活性与隔离目标。

3）自动-交互证明。借助模型检查、SMT 求解或交互式定理证明器（如 Isabelle/HOL，Coq，Dafny，TLA+）生成并验证证明义责。

4）代码精化映射。经过验证的模型逐层落地到 C 语言和汇编语言实现乃至二进制镜像，确保实现与规格的可追溯一致性。

3 SBI 固件形式化验证设计

为实现 SBI 固件（SeSBI）的形式化验证与开发，根据 SBI 固件的架构特点与代码特性，本文创新地提出了一种基于简化分层验证策略的 RISC-V SBI 固件形式化验证框架。其中，“简化”体现在 SBI 固件的形式验证是初步的可完整闭环的形式验证，并且是可基于系统特性进行扩展的形式验证。未来将进一步地实现对 SBI 固件的深度扩展。

形式验证前，本文首先初步实现了 SBI 规范中规定的基础接口，包含：串口初始化、PMP 配置初始化、MPP 寄存器设置初始化和时钟初始化。其中，对启动过程影响最关键的接口是：串口初始化、PMP 配置初始化、时钟初始化。这些关键接口确保启动链安全可控。这些关键接口在主流 RISC-V Linux 启动过程中覆盖大于 85% 的 SBI 调用场景；其余常用扩展已完成实现并集成到 Github[8] 仓库，证明脚本沿用统一模板即可完成验证。鉴于篇幅与评审周期限制，本文报告关键扩展的全栈验证结果；完整覆盖将作为后续工作开源持续发布。验证根据不同验证层级的特性，分别采用 Dafny 和 Isabelle/HOL 定理证明器

对各层进行功能正确性形式化验证。概括来讲, SeSEI 形式化验证可分为 2 部分: SBI 固件抽象规范层的形式验证和 SBI 固件可执行规范层的形式验证, 如图 2 所示。各层形式化验证的具体过程包括: 基于 Isabelle/HOL 的抽象规范验证和基于 Dafny 的可执行规范验证。

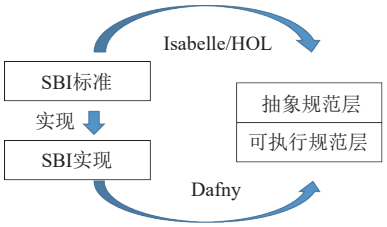


Fig. 2 The workflow of SeSBI formal verification

图 2 SeSBI 形式化验证工作流程

1)基于 Isabelle/HOL 的抽象规范验证。首先根据 RISC-V SBI 规范, 对 SeSBI 系统模型形式规约。主要采用霍尔逻辑, 在 Isabelle 中统一固件系统语义形成理论接口, 进而, 形式描述 RISC-V 架构下 SBI 固件系统。针对 SBI 规范中已实现的关键功能, 在抽象规范层面, 本文对这些功能进行 100% 覆盖率的形式验证。其中, 为统一验证语义, 将多特权级执行环境映射到汇编语言级别, 并形式说明 SBI 固件在所有功能接口上的功能正确性。然后, 基于固件语义接口及 Isabelle/HOL 交互式定理证明器, 形式化验证抽象层所规约的属性。

2)基于 Dafny 的可执行规范验证。基于 C 语言和汇编语言实现 SeSBI 已实现的所有关键功能, 在可执行代码层面进行覆盖率为 100% 的功能正确性验证, 具体实现过程见第 6 节。本节是基于 SeSBI 固件实现的功能代码抽象至可执行规范层, 主要包括物理内存保护配置、时钟初始化。

4 基于 Isabelle/HOL 的抽象规范验证

Isabelle/HOL 交互式定理证明器属高层次逻辑框架, 能够清晰地描述固件的抽象行为和规范。因此, 本文采用 Isabelle/HOL 定理证明器对 SBI 固件的抽象规范层进行规约及验证。该过程主要包括: SBI 固件理论接口定制、启动文件和功能初始化文件的形式验证。研究期间, 为更好地贴合 RISC-V SBI 固件验证迭代研究, 本文将 RISC-V SBI 硬件语义封装成 Isabelle/HOL 抽象理论接口, 从而加速验证 SeSBI 的抽象规范。SeSBI 验证主要包含:

1)SBI 固件理论接口定制。基于 Isabelle/HOL 形式说明多模式下的系统状态并定制理论接口。

2)启动文件的形式验证。基于理论接口形式说明与验证 SBI 固件汇编启动过程。

3)功能初始化文件的形式验证。基于理论接口形式说明与验证 SBI 固件底层设备初始化过程。

4.1 SBI 固件理论接口定制

根据 SBI 标准规定[18], SBI 固件运行在不同模式下, 这一点使得 SBI 固件的验证过程需要抽象在不同的特权模式中。因此, 相同的数据结构在不同的验证过程的定义则显得十分冗余。同时, SBI 固件的形式化验证也属于迭代开发过程, 不断更新、重新定义数据结构也是造成冗余性的主要问题。因此, 基于 Isabelle/HOL, 本文提出将 RISC-V 架构下 SBI 固件系统常用的数据结构及其行为定制为理论接口, 以解决迭代开发与重复的数据结构定义带来的冗余验证问题。具体内容如下:

定义 1. consts:

sp::*reg*;

*t*0::*reg*;

…//其他状态。

定义 1 对 SBI 固件初始化时的汇编实现的硬件层系统常量进行形式化规约。其中, *reg* 表示寄存器类型, *sp* 表示栈指针寄存器, *t*0 表示临时寄存器, 通常用于临时计算或数据传输。此外还定义: *mscratch* 为机器模式暂存寄存器, 用于异常处理; *mie* 表示机器中断使能寄存器; *zero* 表示 RISC-V 常用的零寄存器。这些变量将在 SeSBI 的汇编层中出现。

定义 2. type_synonym:

reg=int;

state="*reg*→int";

…//其他状态。

在定义 2 中, *state* 表示系统状态类型。省略掉的部分重要定义还包括: 1)*reg_index* 索引寄存器, 该变量与 RISC-V 架构的寄存器编号方案一致, 常用于 M 模式下的初始化函数; 2)*address* 内存地址, 在这里为方便验证运算, 直接将其定义为自然数; 3)*size* 内存区域大小。然后, 本文结合 *address*, *size*, *protection* 组成 RISC-V 架构中 PMP 的典型结构, 定义 *pmp_entry* 为 PMP 条目三元组: (地址, 大小, 保护级别)。此外, 在本文省略的其他定义过程中, 首先, 表示入口地址, 将物理内存保护级别 *protection* 抽象为自然数, 允许形式化不同的访问权限或安全级别; 其次, 定义 *m_reg* 为 M 模式下寄存器; 然后, 定义 *stacks_start* 为栈的起始地址, 这样便于在后续验证过程中进行灵

活内存布局表示。

4.2 启动文件形式规约及验证

根据 SBI 规范[23], SBI 固件执行启动环境准备进行 C 环境跳转, 该行为基于汇编语言实现。本文首先采用 Isabelle/HOL 对 SBI 固件抽象层的 RISC-V 汇编语义进行形式规约与验证。

定义 3. *csrw*:: "*reg*→ *reg*→ *state* →*state*" where *csrw csr rs*≡*s*(*csr*:=*rs*)。

定义 3 阐述控制寄存器写入指令"*csrw*"。对于给定的控制状态寄存器 *csr*、源寄存器 *rs* 和初始状态 *s*, 将源寄存器 *rs* 的值写入 *csr*, 从而产生一个新的状态。

定义 4. *add*:: "*reg*→ *reg*→ *reg*→ *state*→ *state*" where "*add rd rs*1 *rs*2 *s*≡*s*(*rd*:=*s rs*1+*s rs*2)"。

定义 4 表示加法指令"*add*": 对于给定的目标寄存器 *rd*、源寄存器 *rs*1 和 *rs*2 以及系统初始状态 *s*, 计算 *rs*1 和 *rs*2 中存储的值之和, 并将结果存储在 *rd* 中。

定义 5. *li*:: "*reg*→*int*→*state*→*state*"

where "*li rd im s*≡*s* (*rd*:=*im*)"。

定义 5 表示立即数加载指令"*li*": 对于给定的目标寄存器 *rd*、立即数 *im* 和初始状态 *s*, 将立即数 *im* 直接加载到寄存器 *rd* 之中。

定义 6. *init_sequence*:: "*state*→*state*"

where "let *s*=*csrw mie zero s* in

let *s*=*lisp stacks_start s* in

let *s*=*add sp t*0 *s* in

let *s*=*csrw mscratch sp s* in *s*"。

定义 6 规定函数 *init_sequence* 的初始化序列, 主要是定义了 RISC-V 系统启动过程的关键步骤: 1)*csrw* 写指令将零寄存器的值写入机器来中断使能控制状态寄存器。通过禁用所有机器级中断, 以确保在系统完全准备好之前不会被中断干扰。2)将 *stacks_start* 的值加载到堆栈指针 *sp* 寄存器中。设置系统堆栈的起始地址, 为后续的函数调用和局部变量分配准备内存空间。3)将 *t*0 寄存器的值加到堆栈指针 *sp* 上, 预留 4KB 空间内存资源。4)设置 *mscratch* 为后续模式切换做好准备。

验证 1. 启动文件功能正确性。

引理 1. 寄存器恒等。

P_1: {*t*0=4 096∧0≤*stacks_start*};

C_1: {执行定义 6 中 *init_sequence*};

Q_1: {*sp s*3= *stacks_start* + 4 096}。

引理 2. *mscratch* 写入。

P_2: {*csrw mscratch sp s*3} ;

C_2: {*mscratch*(*init_sequence s*)};

Q_2: {*sp s*3}。

定理 1. 证明初始化正确性属性。

*t*0 = 4 096 ∧ 0 ≤ *stacks_start* ∧ *reg x*0 *s* = 0

→*mscratch*(*init_sequence s*)≠ 0。

证明.

1)由引理 1 得, *sp s*3 = *stacks_start* + 4 096 →

*s*3 >0;

2)由引理 2 得, *mscratch* 被写入非 0 值;

3)由 simp 得证。

Theorem 1.correct_initialization:

assumes "*s t*0=4 096"

and"0≤*stacks_start*"

and "*s zero*=0"

shows "*init_sequences mscratch* ≠ 0"。

证毕.

通过执行 Isabelle/HOL 的自动推理策略 auto, 本文通过展开假设条件下的所有行为定义, 成功证明了初始序列的逻辑结构 *mscratch* 必然非零。此外针对边界条件约束等属性进行了推理, 源码展示于 GitHub[8]。

4.3 功能初始化文件形式规约及验证

根据 SBI 规范[18], SBI 固件进行部分底层设备的初始化。而这些初始化主要基于 C 语言来实现。因此, 若实现对该层的形式化验证, 需要实现对 C 语言语义的形式化规约。这一过程包括形式化定义指令和函数的执行序列, 然后便是设计验证规则对 SeSBI 固件的功能正确性进行推理验证。

定义 7. *update_m_status*:: "*mreg*→*reg_index*→ *nat*→*m_reg*" where

"*update_mstatus m reg val*"。

定义 8. *insert_field*:: "*m_reg*→*reg_index*→ *nat*→*m_reg*" where

"*insert_field m f v*=*update_mstatus m f v*"。

在定义 7 和定义 8 中, *insert_field* 通过 *update_m_status* 实现, 用于以字段方式更新 *mstatus*, 避免影响其他位。

定义 9. *valid_pmp_entry* :: "*pmp_entry*→*bool*"

where "*valid_pmp_entry e* ≡

case *e* of(*addr*, *size*, *protection*)→

addr+*size*≤2×64∧

protection≤*pmp_RWX*∧

is_power_of_two size∧

addr mod *size*= 0 ∧

size≥ 4"。

定义 9 通过规定地址范围以防止内存溢出漏洞, 同时, 按照 RISC-V SBI 规范规定, 规定 PMP 条目地址 *addr* 与 *size* 大小对齐以及保护级别不得超过 *pmp_RWX*(最大允许保护设置的位掩码, 具有读写执行特权)。此外定义了 *is_power_of_two* 以确保内存尺寸 *size* 是 2 的幂。

定义 10. *is_power_of_two* :: "*nat* → *bool*"
where "*is_power_of_two n* ≡ *n* > 0 ∧
(*n* & (*n* — 1)) = 0"。

定义 11. *sbi_set_pmp*:: "*reg_index*→*address*→*size*→*protection*→*pmp_entry* list→*pmp_entry* list"
where
"*sbi_set_pmp idx addr size protection pmps*=
(if *idx*≤*length pmps* ∧
valid_pmp_entry(*addr*, *size*, *protection*)
then *pmps*[*idx*:=(*addr*, *size*, *protection*)] else *pmps*)"。

定义 11 说明函数 *sbi_set_pmp* 为索引 *idx*、地址 *addr*、大小 *size*、保护属性 *protection* 以及当前的 PMP 条目列表 *pmps*。在定义 11 中, 条件表达式含义为当且仅当确保索引不超边界, 且新条目通过了有效性检验, 函数才会执行更新 PMP 寄存器操作。与 C 代码一致, 通过索引检查和有效性验证, 函数 *sbi_set_pmp* 防止了潜在的越界访问和无效条目设置。

验证 2. MPP 设置正确性。

引理 3. MPP 位段写后读一致。

P_3: {*λs*.True};

C_3: {update *MSTATUS_MPP PRV_S*} ;

Q_3: {*λs.read s* = *PRV_S*}。

定理 2. MPP 初始化正确性。

⟨*λs.mstatus.mpp* = set *s*⟩→
(*λs.update s* = *MSTATUS_MPP PRV_S*)→
⟨*λs′.read s′* = *PRV_S*⟩。

证明.

1)根据引理 3 可得, *read s*=*PRV_S*;

2)前提 MPP 配置与步骤 1 结果一致, 不引起冲突;

3)调用 simp 自动化即得证。

Theorem 2.mstatus_mpp_set_correct:
assumes "*mstatus_mpp_set mstatus*"
shows "*read_csr*
(*insert_field mstatus MSTATUS_MPP PRV_S*)
=*PRV_S*" 。

证毕.

由验证 2 可知, 展开定义 8 和定义 11, 成功证明了如下推断: (*read csr mstatus*) ∧ (*mstatus_mpp_set mstatus*) ∧ (*insert_field mstatus MSTATUS_MPP*)→*mstatus*:=*PRV_S*。定理 2 说明, 如果 *mstatus_mpp_set* 在某 *mstatus* 寄存器状态下为真(状态寄存器已将 MPP 字段设置为 *PRV_S*), 在此状态下读取修改后的 *mstatus* 寄存器将得到 *PRV_S*。

验证 3. PMP 初始化正确性。

引理 4. PMP 对齐与越界。

P_4: {*λs.mstatus.mpp* = set *s*};

C_4: {*update MSTATUS_MPP PRV_S*};

Q_4: {*λs′.read s′* = *PRV_S*}。

定理 3. PMP 初始化正确性。

λs.pmps ≠ [] ∧ (*λ.sbi_set_pmp* 0 0x1000000 4 096 *pmp_RWX pmps*)→*λs′.valid_pmp_entry*(*hd s′*)。

证明.

1)引理 4 ∧ 定义 9=*idx* ∧ *addr* 合法→PMP 地址合法。

2)定义 9 前提=PMP 地址合法→PMP 初始化正确。

3)simp 展开 *sbi_set_pmp* 和 *valid_pmp_entry*, 自动化即可得证。

Theorem 3.pmp_correctly_configured:
assumes "*pmps*≠[]"
shows "*valid_pmp_entry* (*hd*(*sbi_set_pmp* 0 0x1000000 *pmp_RWX pmps*))"。

证毕.

证明表明, 用某些参数更新 PMP 条目列表的第 1 个元素可产生有效的 PMP 条目。

本文假设列表 *pmps* 不为空(*pmps*=[]), 能够确保访问列表首部(*hd*)和更新特定索引(此处为 0)上的元素是有效的。本文创新地利用这一点分解了 *pmps* 的这些列表案例, 并采用自动证明策略成功自动地解决了生成的每个案例。

5 基于 Dafny 的可执行规范验证

Dafny 形式验证器具备高效率自动推理特性, 支持复杂数据类型与霍尔逻辑。因此, Dafny 十分符合可执行规范层的形式化验证特点。本文根据 SBI 标准 v3.0 对 SBI 固件进行实现, 基于 Dafny 形式验证器, 通过 HOL 逻辑来抽象硬件系统状态, 对 SeSBI 的可执行规范层进行建模以描述并验证其属性。

由于 Dafny 无法直接与底层数据结构交互, 本文采用 RISC-V 硬件系统状态模拟策略, 在 Dafny 编程语言中模拟与硬件交互时的硬件状态变化, 填补 Dafny 无法直接与硬件交互的缺陷。然后, 基于 Dafny 内置

定理证明器验证固件功能正确性。本文验证的主要功能包括物理内存保护配置和时钟初始化。

5.1 RISC-V 硬件系统模拟

SBI 固件代码位于 RISC-V 操作系统与硬件设备之间，通过接口功能实现交互。因此，不像其他的基础软件，SBI 固件与硬件的交互性较强。而 Dafny 编程语言无法像 C 语言那样，或以指针形式与内存等硬件进行交互。因此，若能在 Dafny 编写形式化规约时实现对 RISC-V 硬件状态的模拟，则能够突破 Dafny 在硬件方面的验证局限性。本文采用硬件系统模拟策略，利用 Dafny 编写可执行规范，并基于内置定理证明器进行自动形式化验证。

根据 SBI 实现（实现源码部分描述于第 6 节）以及各实现功能涉及硬件，硬件系统形式化说明如下所示：

1）时钟初始化涉及硬件系统状态模拟

const *SBI_SET_TIMER*: nat := 0x00;

SBI_SET_TIMER 是根据 SBI 规范设定的 SBI 调用序号，用作参数传递给函数 *SBI_CALL* 以进行时钟初始化。接下来，便是抽象出时钟初始化的硬件状态。

class *HardwareState*{

var *timer*: nat;

var *memory*: map⟨nat, int⟩;

…//其他状态}。

通过 *HardwareState* 类定义抽象的硬件状态模型。在定义中，*timer* 变量代表系统时钟，*memory* 映射则抽象表示了系统内存，从而简化验证过程，增强验证结果的普适性，使其可以应用于不同的硬件平台。

2）物理内存保护配置涉及硬件系统状态模拟

假设处于 64 b 长度下的 RISC-V 硬件系统。首先是对常量进行定义，这部分定义了一系列与 PMP 相关的常量，包括 PMP 寄存器的最大数量、地址偏移、RISC-V 字长等。这些常量反映了 RISC-V 架构规范中的具体参数。

const *MAX_CSR_PMP*: int := 63;

const *PMP_SHIFT*: int := 2 ;

…//其他比特。

在上述定义中，*MAX_CSR_PMP* 为 PMP 条目最大数量。*PMP_SHIFT* 定义 PMP 地址的对齐偏移量为 2，即 4 B 对齐。设置 *RISCV_XLEN* 字长 64 b。然后按照 RISC-V 架构规范定义 PMP 配置寄存器和地址寄存器的 CSR 编号 *EINVAL* 位错误码，并按照 SBI 规范所述用作 SBI 固件运行时的错误码。

SBI 固件的物理内存配置中，对执行行为进行权限检查是保障安全性的关键步骤，因此，本文定义了 PMP 配置寄存器中各个位的含义，如读（R）、写（W）、执行（X）权限等。这些位组合构成了 PMP 的访问控制策略。

const *PMP_R*: bv8 := 0x01;

const *PMP_W*: bv8 := 0x02;

…//其他比特。

由上而下分别为定义的 PMP 配置位，定义读、写、执行权限。此外，在源码[8]中，本文通过 VALID_PMP_FLAGS 定义有效的 PMP 配置位掩码，及其由权限配置位所组合而成。通过 datatype *pmp_entry*=*pmp_entry*(*addr*: bv64, *cfg*: bv8) 定义 PMP 条目的结构，包含地址 *addr* 和配置 *cfg* 这 2 个字段，对应 RISC-V 中的 *pmpaddr* 和 *pmpcfg* 寄存器。

另外，与时钟初始化类似，该模型通过 *HardwareState* 类抽象表示 RISC-V 处理器的 PMP 相关硬件状态，包含变量 *pmp_entries* 等。

3）地址跳转涉及硬件系统状态模拟

跳转行为主要包含输出、插入寄存器值，涉及对 *mstatus* 寄存器等与 RISC-V 地址跳转相关的寄存器状态的模拟。

首先是对一些寄存器位的定义，具体如下所示：

const *BANNER*: string;

const *PRV_S*: bv64;

…//其他比特。

在该变量定义中，*BANNER* 表示输出界面，*PRV_S* 代表权限位，此外，省略的变量定义还包含了 *MSTATUS_MPP*, *MSTATUS_MPIE*, 这些为 *mstatus* 中的 MPP 与 *MPIE* 配置位；*MAX_CSR_PMP* 表示最大 PMP 条目数为 16; *PMP_SHIFT* 表示 PMP 地址对齐偏移量为 2（4 位对齐）；*RISCV_XLEN* 指代 64 位 RISC-V 架构；*FW_JUMP_ADDR* 表示固件跳转地址至后续 C 语言实现的功能初始化文件。由于正确设置了异常向量和程序计数器的地址，当异常发生时，SeSBI 能够实现跳转。

其次，采用 *CSR_state* 模拟 RISC-V 的控制和状态寄存器 *mstatus*, *mepc*, *stvec*, *sie*, *satp*。定义 *pmpcfg*, *pmpaddr* 数组用于存储 PMP 配置和地址。构造函数确保 *pmpcfg* 和 *pmpaddr* 数组被正确初始化及长度符合规范。

接着，采用 *System_state* 类表示整个系统的状态。其中，定义 *currentMode* 为当前的特权模式（取值为 0,

1, 2)。构造函数初始化 CSR 状态，并将初始模式设置为 M 模式。

5.2 时钟初始化正确性

method *SBI_CALL*

(*hw*: HardwareState, *which*: nat, *arg*0: nat, *arg*1: nat, *arg*2: nat) returns (*result*: SBIResult)

ensures

which == *SBI_SET_TIMER* ==>

(*hw.timer* == *arg*0 && *result* == Success)

…//后置条件。

函数 *SBI_CALL* 包含 SBI 调用逻辑、接受硬件状态 *hw*、调用类型 *which* 以及参数 *arg*0, *arg*1, *arg*2。SBI 标准通过 *arg* 的不同区分具体的 SBI 调用函数。函数体中，ensures 为后置条件，保证当调用类型为 *SBI_SET_TIMER* 时定时器被正确设置，并返回成功；否则，定时器保持不变，返回失败。

method *sbi_set_timer*

(*hw*: *HardwareState*, *which*: nat, *stime_value*: nat)

returns (*result*: SBIResult)…//后置。

函数 *sbi_set_timer* 直接调用函数 *SBI_CALL*。保证定时器被设置为参数指定值 *stime_value*。当且仅当指定参数正确时，调用 *SBI_CALL* 函数执行时钟初始化。具体验证过程展示在验证 4、引理 5、引理 6 和定理 4。

验证 4. 时钟初始化正确性。

引理 5. 调用语义分支。

P_5: {true}；

C_5: {执行 *SBI_CALL*}；

Q_5: {$hw'.timer = stime_value \wedge result$ = Success}。

引理 6. 包装函数等价。

P_6: {true}；

C_6: {*sbi_set_timer*(*hw*, *stime_value*)}；

Q_6: {$hw'.timer = stime_value \wedge result$ = Success}。

定理 4. 时钟初始化正确性。

$timer$ (*sbi_set_timer*, *hw*, *stime_value*) = *stime_value* →*result* = Success 。

证明.

1)由引理 1 得，*SBI_CALL* 调用在 *SBI_SET_TIMER* 分支下赋值：

$hw'.timer = stime_value$ $hw'.timer > 0$；

2)由引理 2 得，函数 *sbi_set_timer* 封装等价该 *SBI_CALL* 调用，故其输出同样满足 $hw'.timer = stime_value \wedge result$ = Success。

3)由 simp 证明器，即主要霍尔规则如下所示。

$$\frac{\begin{array}{c}\{\text{true}\}\\ SBI_CALL(hw, SBI_SET_TIMER, stime_value, 0, 0)\\ \{which == SBI_SET_TIMER \Rightarrow (hw.timer == arg0\\ \wedge result == \text{Success})\}\end{array}}{\left\{\begin{array}{c}\{\text{true}\}\\ SBI_CALL(hw, SBI_SET_TIMER, stime_value, 0, 0)\\ \{hw.timer == stime_value \wedge\\ result == \text{Success}\}\end{array}\right.}$$

$$\rightarrow P[e/x]x{:=}eP \rightarrow$$

$$\frac{\left\{\begin{array}{c}\{\text{true}\}\\ SBI_CALL(hw, SBI_SET_TIMER, stime_value, 0, 0)\\ \{hw.timer == stime_value\\ \wedge result == \text{Success}\}\end{array}\right.}{\left\{\begin{array}{c}\{hw.timer == stime_value \wedge\\ SBI_CALL(\cdots) == \text{Success}\}\\ result := SBI_CALL\\ (hw, SBI_SET_TIMER, stime_value, 0, 0)\\ Q : \{hw.timer == stime_value\\ \wedge result == \text{Success}\}\end{array}\right.}$$

证毕.

5.3 物理内存保护配置正确性

在 SeSBI 中，函数 *sbi_set_pmp* 主要功能为配置 PMP 条目：设置其地址范围和访问权限。该函数接受 PMP 寄存器索引、起始地址、内存区域大小和保护标志 4 个参数。函数动作为：首先进行参数有效性检查，确保寄存器索引在合法范围内，并计算内存区域的对数大小；接下来，计算 PMP 配置寄存器(*pmpcfg*)和地址寄存器(*pmpaddr*)的具体 CSR(控制和状态寄存器)编号；然后，根据内存区域大小选择适当的 PMP 地址匹配模式(NA4 或 NAPOT)，并相应地设置配置寄存器中的字段 *A*。对于 NAPOT 模式，函数实现了复杂的位操作来正确设置地址掩码，确保 PMP 条目能够精确覆盖指定的内存区域。函数通过写入相应的 CSR 寄存器配置 PMP 条目。

经过上述函数 *sbi_set_pmp* 的行为分析后，可以得知该函数的主要功能为：通过输入的有效参数，对 PMP 条目进行正确配置、处理边界输入情况以及通过严格的控制内存访问范围和权限来保障系统安全。

method *sbi_set_pmp* (*hw*: *HardwareState*, *reg_idx*: int, *start*: bv64, *size*: bv64, *prot*: bv8)returns (*r*: int)

/*后置条件*/{

/*输入验证*/

var *pmpaddr*: bv64 := *start* ≫ *PMP_SHIFT*;

…/*配置计算*/

if (*order* == *PMP_SHIFT*)

{*pmpcfg*:= *pmpcfg* | *PMP_A_NA4*;

…/*特殊情况*/

hw.pmp_entries := *hw.pmp_entries*[*reg_idx* := *pmp_entry*(*pmpaddr*, *pmpcfg*)];

…/*更新硬件状态*/

return 0; }

由上述关键代码可知，函数 *sbi_set_pmp* 接受硬件状态、寄存器索引、起始地址、大小和保护标志作为输入。前置条件 requires 子句设定了输入大小的有效范围。后置条件 ensures 子句保证如果返回非零值(表示错误)，硬件状态不会改变。方法形式说明：

1)输入验证/检查寄存器索引的有效性。验证内存区域不会溢出；检查保护标志的合法性；计算并验证内存区域的对数大小。

2)计算 PMP 地址寄存器值(*pmpaddr*)。设置 PMP 配置寄存器值(*pmpcfg*)；根据内存区域大小选择 *NA*4(4 字节对齐)或 NAPOT(自然对齐的 2 次幂)模式。

3)特殊情况处理。对于覆盖整个地址空间的情况进行特殊处理。使用位操作计算复杂的 NAPOT 配置。

4)创建新的 PMP 条目并更新硬件状态。

5)成功时返回 0。各种错误情况返回不同的非零值。函数 *sbi_set_pmp* 通过严格的输入验证和错误处理来确保配置的正确性和系统的安全性。

具体验证过程如验证 5 所示。

验证 5. 物理内存保护配置(PMP 配置)正确性。

引理 7. 参数有效分支。

P_7: {$size \geqslant 4 \wedge size \leqslant 2^{64} \wedge is_power_of_two\,size \wedge start \bmod size = 0 \wedge 0 \leqslant reg_idx \leqslant \max \wedge prot \leqslant pmp_RWX$ }；

C_7: {执行函数 *sbi_set_pmp*} ；

Q_7: {$r = 0 \wedge hw'.pmp_entries[reg_idx] = pmp_entry(start \gg PMP_SHIFT, pmpcfg)$}。

定理 5. 证明物理内存保护配置正确性。

{P_7}if(*reg_idex*<0){return −1; }{$P_7 \wedge RI \geqslant 0$} 。

证明.

1)给定前置条件 P_7。

$1 \leqslant size \leqslant 2^{64}$;

$0 \leqslant reg_idex \leqslant \max$。

2)给定后置条件 Q_7。

如果 r=0，则新条目正确设置；

如果 $r \neq 0$，则条目未修改(*PE*=*old*(*PE*))。

3)使用 auto 自动策略，基于多次霍尔顺序推导可得：

$$\left\{\begin{array}{l}\left\{\begin{array}{c}\{P_7 \wedge \cdots\} \\ order := log2roudup(SZ) \\ \{P_7 \wedge \cdots \wedge order = log2roundup(SZ)\}\end{array}\right. \\ \{P_7 \wedge \cdots \wedge order = log2roundup(SZ)\} \\ \text{if}(order < PMP_SHIFT)\{\text{return} -6;\} \\ \{P_7 \wedge \cdots \wedge order \geqslant PMP_SHIFT\} \\ \{P_7 \wedge \cdots\} \\ hw.pmp_entries[reg_idx] := \\ PMP_Entry(pmpaddr, pmpcfg)\{Q_7\}\end{array}\right.$$

证毕.

6 SBI 固件实现

SeSBI 的函数调用过程如图 3 所示。

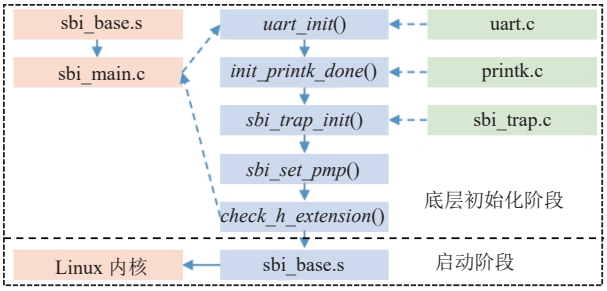


Fig. 3 The workflow of formal verification

图 3 SeSBI 形式化验证工作流程

由图 3 可知，SeSBI 的流程主要包括启动文件 base.s 和底层设备初始化文件 sbi_main.c。其中，base.s 基于汇编语言实现，主要完成从汇编环境到 sbi_main.c 入口地址的跳转功能以及进入 C 语言环境前的准备工作。主要包括 4 个步骤：

1)关闭 M 模式下的所有中断。在 SeSBI 初始化阶段，当系统尚未准备就绪时，防止中断。

2)设置堆栈。为即将执行的程序分配堆栈空间。首先，利用指向堆栈起点的 *la*(加载地址)指令将堆栈起点地址加载到 *sp* 寄存器中。然后，将 *sp* 加上 4 KB 大小以确定堆栈的空间范围，可供程序使用。

3)保存 M 模式寄存器。

4)将控制权转移到名为 *sbi_main* 的函数。转到用 C 语言编写的更高级底层设备初始化代码。

sbi_main.c 对硬件进行初始化，例如 UART(通用异步接收器/发送器)、定时器和执行环境中的一些关键组件。该文件的执行流程主要包括 3 个部分：

1)UART 初始化。系统可以通过串行端口输出日志信息，完成对 timer 初始化的功能。

2)日志系统初始化和陷阱处理初始化。函数 *putchar* 使 SeSBI 固件打印系统横幅的欢迎信息；函

数 *sbi_trap_init* 则初始化系统的陷阱处理机制。

3)PMP 配置。确保无内存保护限制，修改 *mstatus* 寄存器和保存 M 模式环境，实现全地址空间访问。

SeSBI 已实现 RISC-V 平台启动链所必需的关键接口，并配套最小引导文件，可将处理器从加电稳态推进到操作系统入口。虽功能尚不及主流 SBI 固件，但 SeSBI 是借助 Dafny 与 Isabelle/HOL 完成全栈形式化证明的 SBI 固件，无历史冗余，关键行为均有机器可检的数学保证。目前正向 RAS 等安全场景扩展功能，并为每一项新增代码同步生成模板化证明脚本，提高覆盖率，确立可落地的可信固件开发范式。

总的来说，本文所提出的 SeSBI 固件，为 RISC-V SBI 固件提供了一条通用的分层形式化验证路线：先以高阶逻辑规约 SBI 抽象接口，再用 Dafny 生成可执行语义，对不同层次实现 SBI 固件特性定制了安全性、功能正确性的形式验证，适用于所有 SBI 固件。本文工作将有助于推动 RISC-V 系统软件朝向安全性进一步发展。

7 实验分析

7.1 实验环境

实验环境基于 VMware Workstation Player 虚拟机软件配备 8 GB 内存、4 核处理器和 512 GB 硬盘。同时，虚拟机采用 64 位 Linux Ubuntu20.04LTS 版本。此外，为了模拟硬件环境，还使用了 QEMU4.2.0 的 gdb。gdb 仿真器能够复制 ARM 和 RISC-V 等多种现有流行硬件平台，并允许对代码进行单步调试，常被应用于嵌入式计算机的仿真及测试。另外，实验基于 Isabelle2023 与 Dafny4.0.0 工具实现对 SeSBI 固件的形式化验证。

7.2 实验数据

SeSBI 基于 SBI 规范 v3.0 实现。该规范是 RISC-V 社区最新更新的在线规范。本文的实验数据主要为 SeSBI 固件代码，如表 1 所示。

Table 1 Formal Verification of Firmware Data

表 1 固件数据的形式验证

固件名称	语言	文件大小/kb	（主要）代码行数	功能
SeSBI	汇编	0.411	25	SeSBI 启动文件
	C	16.9	764	功能初始化文件

由表 1 可知，目前，SeSBI 固件的启动文件代码模块由 25 行代码的汇编语言实现，主要包括固件的基本功能初始化和地址跳转操作。SeSBI 的功能初始化文件代码目前由基于 C 语言的 764 行代码实现，包括串口初始化功能和处理部分系统调用。SeSBI 将根据接口标准继续实现，代码量将会进一步增加。

7.3 形式化验证结果

如表 2 所示，SeSBI 的验证工作分别在抽象规范层、可执行规范层验证了汇编启动文件 base.s 与 C 语言启动文件 sbi_main.c 的功能正确性，其中包含串口初始化、PMP 初始化、时钟初始化的配置正确性等。具体验证源码已公开于 GitHub 主页[8]。

Table 2 Verification Results

表 2 验证结果

规范层	Isabelle/HOL 代码行数	Dafny 代码行数	验证属性
抽象	732	0	启动文件正确性 串口配置正确性 PMP 配置正确性
可执行	0	1 264	时钟配置正确性 地址跳转正确性 层间关系一致性

此外，在层间关系一致性方面，本文总结 SeSBI 固件抽象规范层与可执行规范层中的 PMP 正确配置等功能。这些功能是在启动时的重要功能。

目前，SeSBI 基于 SBI 标准进行了基础扩展（必备扩展）的实现。而针对可选扩展（例如核间通信扩展等），SeSBI 工作正在进行之中。而 SeSBI 验证方法是针对 SBI 特性研究的 SBI 固件通用框架，因此 SeSBI 的可选扩展仍能够被拓展验证。

7.4 综合对比

1)与现有固件对比

在现有 SBI 固件中，OpenSBI 与 RustSBI 是 2 种较为广泛使用的实现。OpenSBI 基于 C 语言与汇编语言实现，而 RustSBI 基于 Rust 语言实现。如表 3 所示，本节统计了 OpenSBI，RustSBI 和 SeSBI 之间的综合差异。

Table 3 SeSBI is Compared with OpenSBI and RustSBI

表 3 SeSBI 与 OpenSBI，RustSBI 对比

对比项		SeSBI	OpenSBI	RustSBI
代码复杂度	时间复杂度	$O(N)$	$O(N)$	$O(N)$
	固件镜像大小/kb	21	120	46
	分支数	2	3	1
验证接口覆盖率/%		100	0	0
已验证扩展数		4	0	0
关键扩展数		4	4	4
实现扩展总数		8	17	17

根据表 3，能够得到如下结论：

首先，在代码复杂度方面，本文通过时间复杂度、空间复杂度（固件镜像大小）、开源代码分支数对 3 套不同固件进行对比。在时间复杂度方面，3 套固件并无本质差异，这是因为在 SBI 固件的启动阶段和典型 ecall 路径都需要按照硬件线程数逐一处理。在固件镜像大小方面，OpenSBI 与 RustSBI 大于 SeSBI，牺牲了镜像体积和可验证性；而 SeSBI 规模较小，但支持关键启动步骤，并仍在持续更新中。在分支数目方面，OpenSBI 为满足多板型与快速发布维持 3 条长期分支；RustSBI 主要围绕单一主干开发，只保留 1 条长期分支；SeSBI 则把功能开发与形式化证明各自固化在 2 个分支，方便在每次功能演进时同步更新证明脚本，确保代码与证明的一致性。

其次，在验证相关指标方面，SeSBI 相比 OpenSBI 与 RustSBI 来说具备核心优势。SeSBI 对自身实现的全部调用接口给出了形式证明，关键接口验证覆盖率达到 100%，而对比固件在公开仓库中尚无任何形式化结果。同样地，SeSBI 已完成所有关键扩展功能证明。在这一点上，OpenSBI 与 RustSBI 目前仍未进行。尽管都实现了 RISC-V SBI v3.0 规范要求的全部强制扩展，因此“关键扩展覆盖率”一栏同为 100%，但只有 SeSBI 将这一覆盖落实到了形式化验证层面。功能广度方面，OpenSBI 与 RustSBI 各自包含 17 项扩展，覆盖了大量可选外设和板级功能。本文 SeSBI 则把实现范围限定在与安全目标直接相关的 8 项扩展，使 SBI 固件实现能够在保持可用性的同时，大幅降低验证工作量，从而构建出“小而可证”的固件安全模型。

综合来看，SeSBI 以小体量配合全覆盖的形式证明维持较小攻击面，并且，经过形式验证保证了其具备强数学保障，相对另外 2 套固件，其更适用于对高安全性和高可靠性有硬性要求的简化场景。

2）形式验证方法对比

本文基于交互式定理证明工具 Isabelle/HOL 提出了 SBI 固件理论接口，实现了 SBI 固件的形式化验证过程。本节将统计本文改良的 Isabelle/HOL 固件定制化验证工具和 Isabelle/HOL 之间的差异，如表 4 所示。

Table 4 Compared with Interactive Verification Tool Isabelle/HOL

表 4 与 Isabelle/HOL 交互式验证工具比较

验证工具	硬件接口复用率/%	源代码验证开销（1：验证代码行数）	特性覆盖个数
SeSBI 验证方法	82	1：15	5
Isabelle/HOL	未提供硬件接口	1：20	0

根据表 4 可以看出，SeSBI 验证方法在多个关键指标上优于传统的 Isabelle/HOL 方法，具体在以下几个方面：

首先，关于硬件接口复用率，本文 SeSBI 验证方法的平均值超过 50%。在为新的 SBI 扩展撰写证明脚本时，能够更多地调用既有的统一接口定义，例如 *hart_id*, *pmp_cfg* 等常用硬件变量，而不必重复给出低层次描述，显著减少接口重定义和语义对齐工作量。

其次，在源代码验证开销方面，SeSBI 验证方法在验证每行源代码时，仅需在核心库之外补充约 15 行验证代码，即可驱动 Isabelle/HOL 支持脚本；相比之下，原生 Isabelle/HOL 流程则通常需要约 20 行验证代码以实现等价表达。这是因为，对比 Isabelle/HOL，SeSBI 验证方法提出了 RISC-V 硬件模拟的统一接口层，能够使得为扩展付出的“增量代码”更少，提升开发者的验证效率，解决了当前在 SBI 固件验证时，形式验证工作量随着 SBI 固件代码复杂度上升而带来的时间开销问题。而且，SeSBI 验证方法提出的统一硬件接口适用于支持 RISC-V 架构的系统软件，不仅限于 SBI 固件。

然后，在 SBI 固件特性覆盖情况方面，SeSBI 验证方法给出了 5 个 SBI 固件的功能正确性证明（在抽象层、可执行层上分别进行对启动文件正确性、串口等功能正确性的形式验证）。其中，对影响安全的重要属性（PMP 配置）提供了形式化覆盖；而传统手写 Isabelle/HOL 脚本缺乏对同一接口的安全约束检验。

在引入 SeSBI 验证方法预定义硬件接口库后，验证人员不仅写得更少，而且能够在短周期内把验证范围从“仅证明能工作”扩展到“同时证明足够安全”。随着 SeSBI 的进一步扩展，SeSBI 验证方法更符合工业级固件对可靠性与安全性的双重需求。

8 总　　结

RISC-V SBI 固件作为操作系统与底层硬件之间的关键接口，其正确性对整个系统的稳定性和可靠性至关重要。然而，现有针对 Linux 内核的形式化验证方法无法直接复用于 SBI，因为 SBI 位于操作系统与底层硬件之间，需同时处理硬件通信与系统交互，且 RISC-V 的多特权模式使其行为语义和 C 级实现与内核截然不同。由于这些内核验证方法无法覆盖的独有特性，当前 SBI 固件的实现普遍缺乏严格的形式化验证，其启动过程存在安全漏洞与风险。

为此，本文提出一种 SBI 固件形式化验证框架，

能够在开发早期检测 SBI 固件漏洞。基于此,开发了经过功能正确性验证的 SBI 固件——SeSBI。为了展开对 SeSBI 的全面验证,本文提出了一种基于硬件模拟的分层形式化验证策略,分别从抽象规范层与可执行规范层对 SeSBI 的关键扩展进行闭环验证。由于 SBI 固件与硬件交互,本文采用 RISC-V 架构硬件模拟策略以适应验证 SBI 固件的硬件行为。为保证描述语义一致以及减少验证开发开销以及提升验证效率,在抽象规范层,本文定义了 SBI 固件理论接口,实现了不同执行环境到统一抽象层次的映射,节省了验证时间。为精确描述固件行为,本文将 Dafny 引入可执行规范层的验证过程,利用其编程时自动推理特性对 SBI 固件的行为进行模拟并验证。最终,本文成功验证 SeSBI 固件的安全属性和功能正确性。结果表明,SeSBI 以更简洁的代码实现了关键的 SBI 标准接口,具备高度可靠性,为 RISC-V 生态系统提供了一个初步经过形式化验证的高度可信的 SBI 固件实现。

未来计划进行 SeSBI 的 RAS 级别形式验证,并且增强验证框架的泛化性,希望能够涵盖更多固件等基础软件。希望通过完善固件形式化验证方面的努力,进一步提高 SBI 固件的完整性和稳健性,为 RISC-V 生态系统提供更加安全、可靠的底层软件支持,推动 SBI 固件在高安全性要求场景中的应用。

作者贡献声明:杨叶倩负责方案整体设计并撰写论文;戴鸿君提出指导意见并修改论文。

参 考 文 献

[1] Dorflinger A, Albers M, Kleinbeck B, et al. A comparative survey of open-source application-class RISC-V processor implementations[C]//Proc of the 18th Int Conf on Computing. New York: ACM , 2021: 12−20

[2] Bettayeb M, Nasir Q, Talib M A. Firmware update attacks and security for IoT devices: Survey[C]//Proc of the ArabWIC 6th Annual Int Conf Research Track. New York: ACM, 2019: 1−6

[3] Saiki A, Omori Y, Kimura K. Parallel verification in RISC-V secure boot[C]//Proc of the 16th IEEE Int Symp on Embedded Multicore/Many-core Systems-on-Chip. Piscataway, NJ: IEEE, 2023: 568−575

[4] Tung L L, Mohamad K I, Khalid A. Design and implementation of secure boot architecture on RISC-V using FPGA[J]. Microprocessors and Microsystems, 2023, 101: 104889

[5] Liu Guojun, Huang Jili, Liu Xiaoyan. An OS kernel based on RISC-V architecture[C]//Proc of the Int Conf on Computer Science and Educational Informatization. Berlin: Springer, 2023: 3−16

[6] Sun Haonan, Xie Jiangtao, Lin Bo, et al. Large-scale firmware vulnerability analysis based on code similarity[C]//Proc of the 2021 IEEE Int Conf on Power, Intelligent Computing and Systems. Piscataway, NJ: IEEE, 2021: 184−189

[7] Hähnle R. Dijkstra's Legacy on Program Verification[M]. New York: ACM, 2022: 105−140

[8] Yang Yeqian. FVSeSBI[EB/OL]. 2025[2025-07-30]. https://github.com/AII-SDU/Sesbi

[9] Erbsen A, Philipoom J, Gross J, et al. Simple high-level code for cryptographic arithmetic-with proofs, without compromises[C]//Proc of the 2019 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2019: 1202−1219

[10] Klein G, Andronick J, Elphinstone K, et al. Comprehensive formal verification of an OS microkernel[J]. ACM Transactions on Computer Systems, 2014, 32(1): 1−70

[11] Hofer-Schmitz K, Stojanović B. Towards formal verification of IoT protocols: A review[J]. Computer Networks, 2020, 174: 107233

[12] Seligman E, Schubert T, Kumar M V A K. Formal Verification: An Essential Toolkit for Modern Vlsi Design[M]. Amsterdam: Elsevier, 2023,

[13] Garfatta I, Klai K, Gaaloul W, et al. A survey on formal verification for solidity smart contracts[C]//Proc of the 2021 Australasian Computer Science Week Multiconference. New York: ACM, 2021: 1−10

[14] Xu Fengwei, Fu Ming, Zhang Xiaoran, et al. A practical verification framework for preemptive OS kernels[C]//Proc of the 28th Int Conf on Computer Aided Verification. Berlin: Springer, 2016: 59−79

[15] Gu Ronghui, Shao Zhong, Chen Hao, et al. Certikos: An extensible architecture for building certified concurrent OS kernels[C]//Proc of the 12th USENIX OSDI. Berkeley, CA: USENIX Association, 2016: 653−669

[16] Wang Ji, Zhan Naijun, Feng Xinyu, et al. Overview of formal methods[J]. Journal of Software, 2019, 30(1): 33−61 (in Chinese)
(王戟, 詹乃军, 冯新宇, 等. 形式化方法概貌[J]. 软件学报, 2019, 30(1): 33−61)

[17] Qian Zhenjiang, Huang Hao, Song Fangmin. Method of formal design and verification of OS on assembly layer[J]. Journal of Software, 2016, 27(12): 3143−3157 (in Chinese)
(钱振江, 黄皓, 宋方敏. 操作系统汇编级形式化设计和验证方法[J]. 软件学报, 2016, 27(12): 3143−3157)

[18] Kovalev M. Microkernel-based operating system for RISC-V architecture[D]. Barcelona: Universitat Politècnica de Catalunya, 2024

[19] Li Hongyu, Guo Liwei, Yang Yexuan, et al. An empirical study of rust-for-Linux: The success, dissatisfaction, and compromise[C]//Proc of the 2024 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2024: 425−443

[20] Rui Zhiqing, Mei Yao, Chen Zhenzhe, et al. SeChain: Design and Implementation of RISC-V secure boot mechanism based on domestic cryptographic algorithms[J]. Journal of Computer Research and Development, 2024, 61(6): 1458−1475 (in Chinese)
(芮志清, 梅瑶, 陈振哲, 等. SeChain: 基于国密算法的 RISC-V 安全启动机制设计与实现[J]. 计算机研究与发展, 2024, 61(6):

1458−1475）

[21] Xu Xuezheng, Yang Deheng, Wang Lu, et al. Sequential consistency per location theorem proving in RISC-V memory consistency model[J]. Journal of Software, 2025, 36(9): 3919−3936 (in Chinese)
（徐学政, 杨德亨, 王璐, 等. RISC-V 内存一致性模型的同地址顺序一致性定理证明[J]. 软件学报, 2025, 36(9): 3919−3936）

[22] Tai Yang, Han Changgang, Quan Yu, et al. Porting of openharmony standard system based on QEMU RISC-V architecture[J]. Computer Systems & Applications, 2023, 32(11): 21−28 (in Chinese)
（邰阳, 韩昌刚, 全雨, 等. 基于 QEMU RISC-V 架构的 OpenHarmony 标准系统移植[J]. 计算机系统应用, 2023, 32(11): 21−28）

[23] Constable S D, Sutton R, Sahebolamri A, et al. Formal verification of a modern boot loader[EB/OL]. 2018[2025-07-30]. https://surface.syr.edu/eecs_techreports/183

[24] Bourke T, Brun L, Dagand P É, et al. A formally verified compiler for Lustre[C]//Proc of the 38th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2017: 586−601

[25] Dabbelt P, Patra A. RISC-V Supervisor Binary Interface Specification[S]. Zurich, Switzerland: RISC-V Platform Specification Task Group, 2021

[26] Bulwahn L, Krauss A, Haftmann F, et al. Imperative functional programming with Isabelle/HOL[C]//Proc of the Int Conf on Theorem Proving in Higher Order Logics. Berlin: Springer, 2008: 134−149

[27] Zetzsche S, Różowski W. Well-behaved (co) algebraic semantics of regular expressions in Dafny[C]//Proc of Int Colloquium on Theoretical Aspects of Computing. Berlin: Springer, 2024: 43−61

[28] Yang Kangle, Yu Jianye, Wei Xinshen, et al. Survey of the formal verification of operating systems in power monitoring system[C]// Proc of the 5th Int Conf on Pattern Recognition and Intelligent Systems. New York: ACM, 2023: 65–70



Yang Yeqian, born in 1998. PhD candidate. Her main research interests include RISC-V, system software, and formal verification.
杨叶倩, 1998 年生。博士研究生。主要研究方向为 RISC-V、系统软件、形式化验证。



Dai Hongjun, born in 1981. Professor. His main research interests include RISC-V, computer architecture, and system software.
戴鸿君, 1981 年生。教授。主要研究方向为 RISC-V、计算机体系结构、系统软件。