# SafeJIT：一种经过机器码级验证的推测式 JIT 引擎

Gabriel Ebner 微软研究院
美国
gabrielebner
@microsoft.com

arXiv:2611.28436v1 [cs.PL] 21 Nov 2026

## 摘要

即时编译器是现代软件中最常遭受攻击的组件之一。它们编译由攻击者选择的程序，根据受攻击者影响的类型反馈进行推测，并以完整的原生权限运行编译结果。一个守卫被错误地消除，或缺少反优化机制，就会导致内存损坏。我们介绍 SafeJIT：一款带有推测式基线 JIT 的小型 JavaScript 引擎，并给出 Lean 4 中经机器检查的证明，证明其静态链接的 RISC-V 和 x86-64 可执行文件在 Linux 用户进程模型下不会因任何输入而崩溃。该定理针对实际发布的二进制文件，逐条指令在 Sail ISA 模型（RISC-V 使用官方模型）上执行，并涵盖其中所有内容：解析器和字节码编译器、解释器及其类型反馈记录机制、JIT 编译器、JIT 对任意程序和任意反馈都可能生成的所有机器码（类型守卫、守卫消除、循环类型事实、带反优化元数据的寄存器分配、内联缓存、反优化和栈上替换）、标记-清扫垃圾回收器、运行时以及独立运行 libc。C 编译器、汇编器和链接器均不可信，浮点运算也不可信。该证明采用分离逻辑（iris-lean），基于 Linux 用户进程模型；模型中的卡住状态恰好涵盖进程可能崩溃的所有情形，包括取指执行尚未设置为可执行或其指令缓存尚未刷新的代码。

该引擎及其证明由 AI 编码代理在一位研究人员的指导下编写，本报告也是如此。最初的 RISC-V 版本耗时约十个工作日；完整开发历时四周，包含约 6,500 行 C 语言代码和 485,000 行 Lean 代码。我们介绍该引擎、定理及其可信基、证明架构、证明发现的错误以及采用的方法。我们还指出，二进制级验证并不会让编译器验证过时：即时编译器是二进制文件中的编译器，而大部分证明工作都用于证明它生成的代码对所有输入都是安全的。

## 1 引言

即时编译（JIT）编译器旨在尽可能快速地运行不可信的代码：来自任意网页的 JavaScript 和 WebAssembly，以及加载到 Linux 内核中的 eBPF 程序。它们的输入由攻击者选择，输出则以完整模式的原生权限运行，不受任何类型系统约束。优化器中的逻辑错误，例如错误地移除了守卫，或某项类型假设从未被撤销，不会导致答案错误，而会导致内存破坏；浏览器渲染进程中的内存破坏是典型漏洞利用链的第一步。

数据印证了这一点。2021 至 2023 年间，所有在实际攻击中被发现的 Chrome 漏洞利用都始于渲染器内存损坏，其中 60% 是 V8 的漏洞；从历史上看，被利用的 V8 漏洞中约有一半出自其某个编译器 [16]。Microsoft 发现，自 2019 年以来，V8 的 CVE 漏洞中约有 45% 与即时编译有关 [33]。Google 在 2023 年追踪到的 97 个实际发生的零日漏洞中，19 个出现在浏览器中，9 个出现在 JavaScript 引擎中 [14], ；2021 年，Chrome 的 14 个零日漏洞中有 6 个出现在 V8 中，其中包括由遗漏去优化导致的 TurboFan 类型混淆漏洞（CVE-2021-30632） [38]。对于面临风险的用户，通常的缓解措施是关闭即时编译；Apple 锁定模式和 Edge 增强安全模式的严格级别允许用户为自行选择的网站例外，平衡级别则允许对不熟悉的网站关闭即时编译 [1, 29], ，代价是在计算密集型代码上性能下降 1.5 至10× 更多 [16]。采用内存安全的实现语言也无济于事：“如果编译器本身直接暴露在攻击面上，编译器就无法保证内存安全” [16]。模糊测试 [5, 17] 和沙箱 [16] 可以减轻损害，而证明则能消除这一类漏洞。

先前的形式化验证工作已覆盖这一问题的重要部分（第 12节）。Jitawa [31] 对 Lisp 运行时及其即时编译器进行了验证，直至 x86-64 机器码，但它在执行前编译，且不使用反馈或推测执行。Icarus [37] 则依据宏汇编器的模型，对 SpiderMonkey 的 CacheIR生成器中的类型特化内联缓存桩进行了验证。关于推测执行与反优化的研究，从 sourir [13] 到 CoreJIT [3, 4],证明了推测执行层级之间的语义保持；CoreJIT 的最新版本将推理推进到 Com- pCert 的 x86 汇编语义，但仍信任从汇编到已安装字节码的转换步骤以及运行时原语。针对 eBPF 和 WebAssembly 的即时编译器研究（例如 Jitk [43], Jitter-bug [32] 和 [6, 19, 41], ）则涉及机器码和生产级编译器，但没有对包含反优化的推测执行进行验证。这些工作都没有在攻击者实际操作的层面上，对动态语言推测执行引擎的整个二进制文件证明一个定理。

本我们介绍 SafeJIT——一种配备推测式 JIT 编译器的小型 JavaScript 引擎，以及一份经过机器检查的

Lean 证明 4 [10]，证明其编译生成的 RISC-V（RV64）和 x86-64 可执行文件在任何输入下都不会卡住：

**定义** BinarySafe : 命题 :=∀ argv fs $\sigma_0$, Init Image.entry Image.segments argv fs $\sigma_0$ →∀ c, Reach (Loop, $\sigma_0$) c → PrimStep.NotStuck c

**定理**内存_安全性 : BinarySafe --RISC-V**定理**内存_安全性_最终 : BinarySafe --x86-64

这里映像是可执行文件，初始化表示 $\sigma_0$ 是 Linux 使用参数 argv 和文件系统fs之后立即得到的状态，Reach 表示进程操作语义中的可达性，而NotStuck c表示配置c 已经退出或可以执行一步（第 4 节）。

对于任何命令行（包括调试标志 --trace-jit、--dump-jit 和 --dump-bytecode）以及任何文件系统，这个静态链接的 ELF 可执行文件在 Sail ISA 模型 [2, 34, 40]（RISC-V 的官方模型）上逐条指令运行时，绝不会执行非法指令或触发陷阱，绝不会访问未映射或保护属性不正确的内存（包括从尚未设为可执行的 JIT 代码中取指），绝不会取指其字节自上次指令缓存刷新后被写入的指令（在 RISC-V 上，这是必要的），并且只会进行格式良好的系统调用。对于用户提供的任何 JavaScript 程序及该程序产生的任何类型反馈，这些性质都成立。证明中不含 sorry 占位符，也不含 Lean 标准公理之外的任何公理，唯一例外是 bv_decide 为其经过验证的 LRAT 检查器引入的辅助公理。

SafeJIT 是一个演示系统：该引擎规模很小。但这个定理具备生产环境引擎所需的形式，并且覆盖了即时编译中最难正确实现的部分。SafeJIT 在其解释器中记录类型反馈，将热点函数编译为会基于观测到的类型进行推测的本机代码，使用抽象类型消除冗余守卫（在基本块内部，以及在 RISC-V 上跨越循环回边），将局部变量和虚拟操作数栈分配到寄存器中，在隐藏类上缓存属性查找，当守卫失败时反优化回解释器，并通过栈上替换（OSR）从循环中间进入已编译代码。CVE-2021-30632，即映射变更 [38], 后缺少去优化，正是这些机制中的一个漏洞。

贡献。

- 针对完整动态语言即时编译引擎的二进制文件，提出一个端到端安全性定理；该定理以 Sail ISA 模型为依据，适用于所有输入，并且不将 C 工具链或浮点纳入可信基（第 4 节）。据我们所知，这是首个针对会进行推测、反优化和栈上替换的引擎提出的此类定理。

- 使其变得可处理的证明架构（第 5 节和第 6 节）：从 Sail 到抽象指令集架构；一种针对完整指令的分离逻辑，其中代码（包括 JIT 代码）是一种持久资源；可并行构建的模块化规约；用于经验证的垃圾收集器的垃圾回收纪元；以及双层 JIT 证明：其中，JIT 的 Lean 模型被证明能够为每个格式良好的函数和每个反馈向量生成安全代码，而二进制文件中的 JIT 则被证明能够准确写入该模型的代码。

- 面向指令集架构截然不同的两个目标平台——RISC-V 和 x86-64，同时共享逻辑中与体系结构无关的部分（第 7 节）。

- 验证过程中发现的漏洞和缺口（第8节），其中包括未经检查的指令缓存刷新，以及对地址空间大小未经检查的假设。

- 系统构建过程的介绍：几乎完全由 AI 编码代理完成，在一名研究人员的指导下，用时四周（第 10 节）。证明的规模约为 C 代码的 75 倍；对人类而言这将难以承受，但对代理而言却是可负担的。

- 论证二进制级验证并不会使编译器验证过时，尽管前者将 C 编译器排除在可信基之外。即时编译器就是二进制文件中的编译器。我们的工作大部分属于编译器验证，且是在二进制级证明内部完成的（第 11 节）。

安全性，而非正确性。定理陈述的是安全性，而非功能正确性：我们没有证明 SafeJIT 计算出的结果符合 ECMAScript 规范。NotStuck本身排除了执行不可执行或过期字节、访问未映射或保护错误的内存，以及执行格式错误的系统调用。它并不能排除在可写页面上向错误对象存储数据，或执行数据错误但格式正确的write。证明在内部确实建立了更多性质：在其不变量中，每个值都带有正确的标签，每个指针都指向一个存活对象，且对象类型与记录的类型一致；每次内存访问都有由访问代码拥有的指向事实作为依据；即时编译器的输出也正是经过验证的模型所生成的结果（第8节和6）。这些性质排除了即时编译器漏洞利用所依赖的内存破坏，但它们是证明的不变量，而非定理陈述的一部分。

制品。我们计划将引擎、证明和工具链作为开源项目发布；目前它们尚未公开。本文中的 C 语言路径均相对于仓库根目录，Lean 路径则相对于 proofs/SafeJIT/（RISC-V）或 proofs/x86/SafeJITX86/（x86-64）。

生成式 ***AI*** 工具的使用。本工作全程使用了生成式 AI。引擎、证明和工具均由 AI 代理编写（第10节）。本报告也由 AI 智能体 Claude Opus 5.5 在 GitHub Copilot CLI 中根据代码库、其历史记录和代理会话日志，并遵照作者的指示撰写；之后又根据

对审查意见作出回应，这些审查意见本身也是由 AI 代理生成的。

## 2 背景

### 2.1 推测式即时编译引擎

生产环境中的 JavaScript 引擎（V8、SpiderMonkey、JavaScript- Core）首先在解释器中执行函数。在解释过程中，它们会记录类型反馈：每次算术运算、比较或属性访问所遇到的类型。一旦函数变得足够热门，JIT 编译器就会将其转换为机器码，并推测未来情况与过去相似。如果某次加法始终只遇到小整数，编译器就会生成 32 位整数加法指令，其前面附带用于检查操作数标签的守卫，后面则跟随溢出检查。失败的守卫会触发去优化：它根据机器状态重建解释器的栈帧，并在相应的字节码指令处恢复解释器执行 [18]。反之，长时间运行循环中的函数可以在循环头处从解释器切换到新近生成的代码，这一过程称为栈上替换（OSR）。对象上的属性访问则通过隐藏类（形状、映射）和内联缓存得到加速；内联缓存会记住访问位置所见的形状以及属性所在的槽位 [9, 12]。

这些机制中的每一个都有一个很容易被破坏的不变量。分析错误地得出某个守卫是多余的结论时，可能会将其移除。去优化元数据负责将机器寄存器映射回解释器槽位，它可能在某个特定的 exit 处出错。OSR 入口可能假定某些事实在正常入口路径上成立，但从解释器进入时却并不成立。缓存形状可能在支撑它的假设失效后仍然存在。这样生成的代码随后会基于一个被错误判断类型的值来读取或写入内存。这就是类型混淆，也是 JIT 漏洞利用通常采取的第一步。

### 2.2 Sail ISA 模型

Sail [2] 是一种用于描述指令集架构的语言。官方 RISC-V 模型 [34] 由 RISC-V International 维护，是 ISA 的黄金参考；对于 x86-64，我们使用一个独立的 Sail 模型，该模型根据英特尔的手册和硬件行为编写而成 [40]。Sail 可以将模型编译为定理证明器中的模型，其中包括 Lean 4。在我们使用的 Lean 后端中（这是继承自 xv6iris [21]的 lean-sail 的一个自由单子分支），该模型是自由单子中的 Lean 程序，其效应（内存读取和写入、未定义值、屏障）由调用方处理。一个机器步（周期）会取指、解码并执行一条指令，还可能引发异常；在我们的设定中，这些异常就是崩溃。

### 2.3 分离逻辑与 Iris

我们使用 Iris [20],这一高阶并发分离逻辑进行推理，并通过其 Lean 4 移植版 iris-lean [25]开展工作。Iris 命题描述资源：$\ell \mapsto v$ 拥有位于 $\ell$处的内存； $P * Q$ 成立，当资源可以拆分为满足 $P$和 $Q$的部分； $P \mathrel{-\!\!*} Q$（“魔杖”）则将 $P$ 转化为 $Q$。持久命题 $\square P$描述了可以自由复制的知识：在我们的证明中，这指的是某段代码位于某个地址，或某种类型的对象存放在某个指针处。Iris 的最弱前置条件 WP $e\{\{\Phi\}\}$表示 $e$ 可以安全执行，且其结果满足 $\Phi$； 充分性 则将最弱前置条件的封闭证明转化为关于操作语义的陈述，即执行永远不会陷入停滞。后置模态 $\triangleright P$与 Löb 归纳（$(\triangleright P \mathrel{-\!\!*} P) \vdash P$）结合，可在没有显式终止度量的情况下处理递归和循环。

### 2.4 MachCSL 与 xv6iris

Kaashoek 和 Zeldovich 使用 MachCSL 在 RISC-V 二进制文件层面验证了 xv6 内核 [21] 。MachCSL 是基于 Sail RISC-V 模型的并发分离逻辑，这次证明工作也主要由 AI 代理完成。SafeJIT 建立在他们的方案之上：我们复用了他们的 Sail配置、自由单子lean-sail分支，以及 MachCSL 的一些部分，例如启动寄存器解释器。两个项目验证的对象有所不同：xv6iris 验证的是证明完成时代码已固定的内核，而 SafeJIT 执行的代码中有很大一部分要到运行时才会生成。

## 3 SafeJIT 引擎

SafeJIT 是一个用 C 语言编写的 JavaScript 引擎：编译器、解释器、运行时和垃圾回收器约有 2,500 行代码，RISC-V 即时编译后端有 1,400 行，x86-64 即时编译后端有 1,100 行，独立C库有 1,500 行。图 1展示了其结构。单遍编译器将源代码转换为栈字节码。解释器运行字节码并记录类型反馈。热点函数会被编译为推测执行这些反馈的本机代码；若推测失败，则返回解释器。所有内容，包括 libc，都会链接到一个静态ELF可执行文件中，该文件会发起原始系统调用。本节将以工程师的视角介绍该引擎；第 4 节至第 6 节将介绍关于该引擎已证明的内容。

### 3.1 语言

SafeJIT 支持 JavaScript 的一个子集，其功能足以编写逼真的基准测试：let/const/var；数字（32 位整数和双精度数）、字符串、布尔值、null 和 undefined；带有字面量、索引、.length 和 push 的数组；带有字面量以及属性读取和写入的对象；顶层递归函数；if、while、for、break、continue 和 return；以及所有算术运算、位运算、比较运算和逻辑运算符、?:、复合赋值和 ++/--。内置函数包括 print、sqrt、floor、Array(n) 和 clock。不支持闭包、异常或 eval。测试套件会在四种分层配置下将引擎的输出与 Node.js 进行比较：仅解释器、默认阈值、阈值为一，以及阈值为二且去优化次数实际上不受限制。

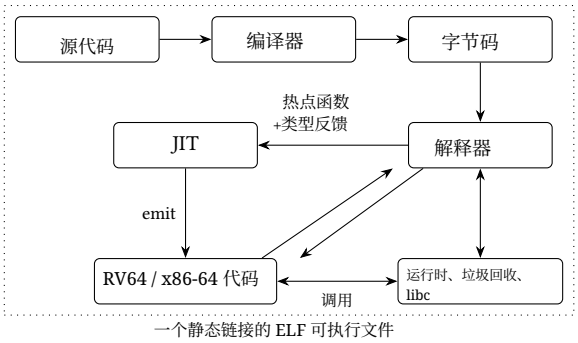


**图 1.** SafeJIT 的体系结构。定理涵盖整个方框，包括即时编译器在运行时生成的机器代码。

**表 1**。NaN 装箱值表示。

| 高 16 位 | 含义 |
|---|---|
| < 0xFFF9 | 双精度数（任意位模式） |
| 0xFFF9 | 低 32 位中的 32 位整数 |
| 0xFFFA | 假 = 0，真 = 1，空值 = 2，未定义 = 3 |
| 0xFFFC | 指向堆对象的指针 |

所有资源限制都会以明确的错误处理，而不会产生未定义行为：语法嵌套深度（1,000）、每个函数的字节码、常量和局部变量数量（各 65,535）、操作数栈深度（30,000）、调用深度（20,000）、字符串长度（$2^{30} - 25$）、嵌套的数组到字符串转换，以及源代码大小（1 GiB）。内存不足也会触发明确的退出。这些限制对证明至关重要：每一项都构成一项义务，否则证明将无法完成。

### 3.2 值

每个值都是一个 64 位字，采用 NaN 装箱（表 1）。双精度数按原样存储。其他类型位于高位 NaN 空间中，并通过其最高 16 位加以区分，因此类型检查只需进行移位和比较。指针宽度为 48 位，因此每个堆地址都必须低于 $2^{47}$（第8节 描述了证明如何发现这一点并不总能得到保证）。

低于 $\texttt{0xFFF9} \ll 48$ 的每种位模式都是有效的双精度数，包括正规范 NaN 0x7FF8. . . 和 x86-64 的默认 NaN 0xFFF8. . . 。只有符号位为负且第 48–50 位中有效载荷非零的 NaN 会与标签冲突，而位模式读回后变成另一种类型的双精度数会导致类型混淆。因此，每个双精度数结果都会通过一次整数测试：将大于或等于 $\texttt{0xFFF9} \ll 48$ 的位模式替换为规范 NaN。即时编译器会在每条浮点指令之后生成这项测试；C 语言中的$_{\texttt{mk_双精度数}}$也会将每个 NaN 规范化。RISC-V 算术运算只会产生规范 NaN，但 x86-64 会传播 NaN 载荷，因此在真实硬件上必须进行这项测试。在证明中，ISA 模型的浮点运算仍是黑盒函数，可能返回任意位模式；正是整数测试使安全性不依赖于浮点语义。

### 3.3 字节码

编译器是一个 Pratt 解析器，直接生成栈字节码，每次处理一个函数。在生成每条指令时，它还会记录该指令之前的操作数栈深度，以及是否有跳转指令指向该指令。运行时不会重新检查字节码。解释器和即时编译都依赖字节码格式良好：每个操作数都在有效范围内，每条路径都能到达具有相同栈深度的指令，栈永远不会下溢，也不会超过函数记录的最大深度。在证明中，这体现为一个谓词 `CodeWF` （第 5.4 节），并且已经证明编译器能够建立该谓词，而解释器和即时编译则对此作出假设。

### 3.4 解释器、反馈与分层编译

解释器是一个围绕栈帧运行的分派循环，其槽数组依次存放局部变量和操作数栈。算术、比较、条件跳转和元素访问指令会将一个位或运算到 `fn->反馈[程序计数器]` 中：整数、双精度数 （当整数操作数溢出时也会设置）、布尔值、数组或其他。调用 200 次后，或循环回边执行 200 次后，函数会被编译。函数调用随后进入程序计数器为 0 的即时编译代码，而回边则通过栈上替换在循环头进入该代码。当即时编译代码发生去优化时，解释器会丢弃该代码，并从发生失败的指令处恢复执行；该指令现在会记录原本可以避免此次去优化的反馈；函数再次变热后会重新编译。函数发生 $N$ 次去优化后会保持由解释器执行，其中 $N$ 由 --max-deopts 设置，默认值为 8。

函数之间的调用（包括来自即时编译代码的调用）都会经过 `vm_调用`。它会检查调用深度，在单独的虚拟机栈上为被调用函数设置栈帧，并在存在即时编译代码时进入该代码，否则进入解释器。因此，即时编译代码与解释器会相互递归。

### 3.5 JIT 编译器

RISC-V JIT（src/jit.c，共 1,400 行）将 RV64IMD 代码生成到缓冲区中，将其映射为可读写，将代码复制进去，使用 mprotect 将映射切换为可读执行，并刷新指令缓存。系统会检查刷新结果；如果失败，则解除代码映射并使编译失败。

共享栈帧布局。JIT 代码使用解释器栈帧布局。由于每个程序计数器位置的栈深度都是静态的，每个操作数栈槽相对于栈帧基址都有固定偏移量，并保存在一个固定寄存器中。其他固定寄存器则保存整数标签、装箱后的 false、栈帧、运行时辅助函数表以及全局变量。在字节码边界处，如果没有任何内容缓存在寄存器中，内存帧就与解释器应有的内存帧完全一致。反优化随后只需将程序计数器和栈深度存入栈帧并返回，而栈上替换则是跳转到循环头的本机地址。

**表 2。** 根据类型反馈选择的策略。

| 反馈 | 策略 | code |
|---|---|---|
| 仅整数<br>整数、双精度数 | int<br>双精度数 | 标签守卫、32位算术逻辑单元、溢出/−0 检查<br>数值守卫、转换、浮点运算单元操作、规范化 |
| 仅布尔值 | bool | 基于原始位进行分支 |
| 仅数组 | 数组 | 标签、类型和边界守卫；直接访问 |
| none | 去优化 | 无条件反优化 |
| 其他 | 通用 | 调用 C 运行时辅助函数 |

推测执行。 对于每条指令，都会根据其反馈（表 *2*）选择一种策略。整数代码会检查两个操作数的标签，执行 32 位运算，并检查溢出和 −0。双精度代码会检查操作数是否为数字，将整数转换为双精度数，并使用浮点运算单元。数组代码会检查标签、对象类型和边界，然后直接加载或存储。没有任何反馈的指令从未被解释器执行过。它们的代码会无条件地反优化（即“软去优化”），这样在编译时较冷的分支就不会永远保持通用形式。在筛法基准测试中，软去优化将加速比从 1。3× 提升至约 20×。

守卫与去优化桩。 守卫失败后会跳转到一个位于代码外的小型桩。该桩将寄存器中缓存的所有状态写回，加载恢复执行时的字节码程序计数器和栈深度，然后跳转到共享的去优化出口。该出口将寄存器分配的局部变量溢出保存，把程序计数器和深度存入栈帧，并向即时编译_进入的调用者返回 1。

守卫消除。 编译器会为每个栈槽和每个局部变量跟踪一个抽象类型（未知、整数、双精度数、布尔值、数组），并跟踪每个栈槽最初复制自哪个局部变量（如果有）。在 i < n 已被检查为整数之后，后续的 i + 1不会再次检查 i。整数运算的结果已知为整数，a.length 已知为整数，依此类推。只有当抽象类型尚不能推断出相关类型时，才会生成守卫。

循环类型事实。 在合并点，编译器通常会忘记所有抽象类型。在 RISC-V 上，合并点也可以从对最多 16 个局部变量的检查 假定类型开始；如果检查失败，就会反优化。检查之后的代码（循环头部）知道这些类型。来自解释器的入口、顺序落空和前向跳转都会执行这些检查。对于后向跳转，如果其自身的抽象状态已经隐含了这些事实，则会跳过检查，直接跳转到头部。即时编译_编译 器通过第一次代码生成遍来推断这些事实：该过程会记录每个跳转进入各合并点时的抽象状态；随后，编译器使用这些状态的交集进行第二次编译。这些事实仅是提示：任何选择都是安全的，因为检查会强制确保这些事实。如果第二遍失败，编译器会退回到不使用这些事实进行编译。

寄存器分配。 在 RISC-V 上，前五个局部变量位于被调用者保存寄存器中。它们的栈帧槽位均为过时状态，只有在溢出之后立即恢复有效；而溢出会发生在每次辅助函数调用之前（辅助函数可能会回收垃圾或检查栈帧），以及在去优化出口处。直线代码中，操作数栈的栈顶是虚拟的：待处理槽位存放在八个寄存器池中的某个寄存器中，或者它是某个局部变量或常量的副本。任何可能读取栈帧或离开直线代码的操作之前，都会将待处理槽位刷新到栈帧中。某条指令的反优化桩会准确物化该指令起始位置存在的待处理槽位。这就是生产级即时编译器所保留的最简单形式的去优化元数据。比较后跟随条件跳转会融合为单条原生分支。

对象与内联缓存。 对象具有隐藏类：形状是转换树中的一个节点，每个子节点都会添加一个属性。形状是永久存在的，因此缓存形状始终有效。每个属性访问点都有一个单态内联缓存，其中保存着一个形状和一个槽位（对于添加属性的存储操作，还会保存添加后的形状）。这些缓存是数据驱动的：即时编译器将该访问点缓存的地址嵌入代码中，代码在运行时读取缓存。它会检查接收者是否为对象，以及其形状是否等于缓存的形状，然后直接加载或存储缓存的槽位；如果未命中，则调用运行时属性辅助函数，由其更新缓存。代码从不被修补，缓存由解释器和 JIT 代码共享。

一个示例。图2展示了 --dump-jit 为循环 for (let i = 0; i < 数组长度; i++) s = s + 数组[i]; 的一部分所打印的代码。局部变量数组和 i 位于寄存器 s0 和 s9 中。数组长度会检查数组是否为数组，其结果会写入池寄存器（a5），而不是栈帧。比较不会检查 i 的标签，因为循环事实表明 i 是整数，并且比较与循环的 exit 分支融合在一起。元素访问不会检查数组是否为数组，因为同一代码块中已经进行过检查。位于 0x150 和 0x1b8 的去优化桩会在离开之前将待处理状态写回栈帧。

### 3.6 运行时、垃圾回收器和 libc

运行时实现了通用语义：转换、+字符串操作、比较、数字打印、数组、对象和内置函数。字符串、数组和对象由非移动式标记-清除回收器（src/gc.c，约 100 行）进行回收。它仅在 虚拟机_调用中的安全点运行，即自上次回收以来分配了足够多对象之后运行。在该时刻，每个存活值要么是被调用函数、虚拟机栈上的槽位，要么是全局变量；NaN 装箱使指针能够被精确识别，并且没有任何 C 语言局部变量、机器寄存器或即时编译代码会在调用期间持有可回收对象。函数、原生函数和形状是永久存在的。即时编译代码从不释放：每个函数最多编译 $N$ 次，因此去优化泄漏的代码量最多为程序大小的常数倍乘以 $N$。

libc（src/libc）旨在易于验证：直线代码、有界循环、定长缓冲区，且数字转换过程中不使用堆。_start 会切换到静态的 64 MiB 栈。malloc 是一个基于 64 MiB 的大小类分配器

```
; 18 GET_LENGTH 反馈={arr} -> 数组 114: mv a0, s0 # 局部变量 0 (a) 位于 s0
118: srli t0, a0 , 48 # 标签 11c: addi t1 , s4, 3 # s4 = 0xFFF9; +3 =对象标记
120: bne t0, t1 , 150 # 守卫：对象 124: slli a0, a0 , 16 # 解包指针 128: srli a0,
a0 , 16 12c: lwu t0, 0(a0) # 对象类型 130: li t1, 1 134: bne t0, t1 , 150 # 守卫：数
组 138: lwu a3, 4(a0) # 长度 13c: ... # 装箱为 32 位整数 148: mv a5, a3 # a5 中的
待处理栈槽 14c: j 164 150: sd s9, 24(s1) # 反优化桩：物化 154: sd s0, 32(s1) # 待
处理槽位，158: li t0, 18 # 程序计数器 = 18, 15c: li t1 , 2 # 深度 = 2, 160: j
78 # 到去优化出口；19 LT 反馈={int} -> 整数（i 已知为整数）164: mv a0, s9
168: mv a1, a5 16c: sext.w a0 , a0 170: sext.w a1, a1 174: blt a0, a1 , 17c # 与
JMP_IF_FALSE 融合 178: j 260 # 循环出口；23..29 GET_LOCAL i；GET_
LOCAL a；GET_LOCAL i（虚拟）; 32 GET_ELEM 反馈={arr} -> 数组（a 已知
为数组）17c: mv a0, s0 180: mv a1, s9 ... 边界检查，加载 a[i] 到 a5 ... 1b8: sd
s8 , 24(s1) # GET_ELEM 的去优化桩...
```

**图2.** sum(a) 循环的带注释 --dump-jit 输出（RISC-V）。注释由我们编写。

带有真正 `free` 的 mmap 内存区域。`strtod` 和 `printf("%e")` 的结果是精确的（正确舍入），使用固定的 4096 位大整数。C 编译器为 GCC，参数为 -march=rv64g：不使用压缩指令，也不使用 libgcc，因此二进制文件和即时编译输出都仅由 32 位指令字组成。

### 3.7 x86-64 后端

src/x86_64/jit.c 是将 RISC-V JIT 移植到 x86-64 的版本。它沿用了相同的栈帧布局、策略、守卫、守卫消除、去优化协议和栈上替换，但没有寄存器分配或循环事实。该移植版本必须处理 x86 的特殊之处。idiv 在除以零以及 INT_MIN / -1 时会触发陷阱，因此会先检查除数，并以 64 位执行除法。溢出通过 OF 标志检测。对于无序操作数，ucomisd 会设置 ZF/PF/CF，因此选择比较方式，使 NaN 操数得到 false。SSE 算术会传播 NaN 载荷，因此每次浮点运算后的检查在真实硬件上都是必需的，而不只是防御性措施。调用 C 辅助函数时，栈必须按 16 字节对齐。其他部分（编译器、解释器、运行时、垃圾回收器以及大部分 libc）都与之前相同，仍是用 GCC 为 x86-64-v1 编译的 C 代码。

**表 3.** 崩溃及进程模型对它们的表示方式。

| 事件 | Linux | 模型 |
|---|---|---|
| 陷入 U 模式之外（非法或未对齐指令、ebreak，…） | 致命信号 | 不执行单步 |
| 访问映射内存之外或违反其保护属性 | SIGSEGV | 无步骤 |
| 执行未被制成可执行的 | SIGSEGV | no step （取指需要 X） |
| 取指自上一次指令缓存刷新 | 陈旧代码 | 无步骤 |
| Sail 断言失败 | — | 无步骤 |
| 超出契约范围的系统调用（未知编号、无效指针、无效标志位） | -EFAULT，..... | 无步骤 |
| exit_组(n) | exit | 最终值 *n* |

## 4 定理

本节说明哪些内容经过了证明，哪些内容是基于信任的。我们介绍 RISC-V 的开发过程； 第 7 节介绍 x86-64 版本的不同之处。

### 4.1 一种将卡住视为崩溃的进程语言

定理基于 Linux 用户进程的小步语义（`Proc/Lang.lean`）。进程状态 $\sigma$ 由 Sail 寄存器文件、具有按页读/写/执行保护属性的字节寻址内存，以及抽象的内核状态（打开的文件及其偏移量和内容，以及文件系统）组成。每一步要么执行一个 Sail 周期，要么在程序计数器处的指令为 `ecall`时，按照契约原子地执行系统调用。该语言经过构造，使真实进程可能崩溃的每一种情况都对应于一个没有后续步骤的状态（表 3）。

指令缓存一致性。Sail RISC-V 模型将 fence.i 作为空操作执行，并将指令缓存一致性交由内存模型处理。在 SMP Linux 上，单靠 fence.i 无论如何都不够，因为线程可能迁移到另一个 hart；用户程序必须调用 riscv_flush_icache 系统调用。因此，进程内存会跟踪脏字节。每次存储都会标记其写入的字节，而对脏字节进行指令取指时则没有后续步骤。只有返回值为 *0* 的riscv_flush_icache 才会清除脏位，初始映像则是干净的。因此，在模型中，若即时编译器写入代码后未成功刷新就运行这些代码，进程便会卡住；真实硬件上也可能如此。

地址空间。 采用 Sv57 的 Linux 可分配最高达 $2^{56}$ 的地址：一旦低位窗口已满，mmap 就会回退到 $2^{47}$ 以上；而在 Linux 6.6 之前，栈位于地址空间顶部之下。因此，该模型允许内核将栈和 mmap 结果放置在 $2^{56}$，以下的任意位置，但

平台的 CLINT 区域除外，因为 Sail配置要求将其取消映射。程序本身不得假定超出其检查范围的内容。

### 4.2 该陈述

令 Init entry segs argv fs $\sigma_0$ 表示 $\sigma_0$ 一种状态，在该状态下，内核刚刚执行了入口点为 entry、可加载段为 segs 的 ELF 映像，并使用参数向量 argv 和文件系统 fs。安全性表明，每个可达配置都可以执行一步或已经 exit：

**定义** Safe（entry：Addr）（segs：List Segment）：Prop：=∀ argv fs $\sigma_0$，Init entry segs argv fs $\sigma_0$ →∀ c，Reach（Loop， $\sigma_0$） c → PrimStep.NotStuck c

**定义** BinarySafe：=Safe Image.entry Image.segments**定理** memory_safe：BinarySafe

**定义**ElfSafe：=Elf.Wf file ∧ Safe（Elf.entry file）（ Elf.segments file）**定理** memory_safe_elf：ElfSafe

argv 遍历所有字节字符串列表，因此该定理涵盖每个 JavaScript 程序（程序是命令行中指定的文件）、每个标志位（包括 --jit-threshold=N、--max-deopts=N，以及用于反汇编 JIT 代码的调试标志），以及每种文件系统状态。memory_safe_elf 在文件层面重述了该定理：文件是 safejit 可执行文件的原始字节，由一个小型形式化 ELF 加载器（Proc/Elf.lean）读取。证明所用的映像是 Image.lean 中的解码指令列表，由一个不可信脚本生成；decide +kernel 会检查该列表是否与加载器从原始字节读取的内容一致。

该定理对成功执行模型化 exec 后的每个状态进行量化，因此其意义取决于初始化是否可满足。我们对此进行部分检查：initRegs_nonempty 表明，对于每个入口点和栈指针，初始化中的寄存器文件部分都有一个见证；这一点通过运行 Sail 模型自身的 initialize_registers，再执行 Linux 在 exec 时进行的写入来确立。我们尚未为实际映像上的完整谓词构造见证（其中包括已加载的段、有效栈和可接受的参数向量）；初始化是对 Linux 实际操作的直接转录，而参数是 C 字符串，因此不能包含 NUL 字节。

即使一个引擎毫无实际作用，例如始终失败或始终去优化的即时编译器，也能满足安全性定理。SafeJIT 通过两点排除了这种情况：二进制文件的编译器已被证明会精确写入可执行模型 `Jit.compile`（第6节）所定义的代码；该模型也经过测试，与真实引擎记录的编译结果相符；而基准测试（第9节）表明，JIT 代码能够运行，且速度比解释器快 2–25× 倍。

### 4.3 可信基

可信基由以下部分组成。

- **Sail RISC-V 模型**，其中包含 F 和 D 扩展，由 Sail 的 Lean 后端编译为 Lean，以及 free 单子 Sail 库。我们的重新生成脚本还修复了该后端中的一个错误：flen = flen_bytes * 8 输出时没有加括号，因此每个浮点寄存器都只有八位宽。
- **进程抽象**：一个 U 模式 hart，satp 为 = Bare 模式；内存为顺序一致内存，具有逐页保护属性；没有信号、没有中断，并遵循指令取指的脏字节规则。
- **十二种系统调用的契约**（Proc/Syscall.lean）：read、write、openat、close、lseek、mmap（仅匿名映射）、mprotect、munmap、clock_gettime、riscv_flush_icache、exit 和 exit_group。除两个 exit 调用外，其他每个调用都可能以任意错误码失败，因此程序必须在所有情况下处理失败。成功的 mmap 会在 $2_{56}$ 以下的任意位置返回新的页。flush 失败时，指令缓存会保持过期。
- **exec 状态**：Linux 设置的寄存器，以及 System V 初始栈；它们可以位于$[2^{37}, 2^{56})$中的任意位置。
- **ELF 加载器** (Proc/Elf.lean)及其良构条件（不带解释器的 RISC-V 可执行文件、位于文件内且互不重叠的段、不会发生回绕、页同余偏移），以及原始文件转储 ElfFile.lean；另一个 Lean 程序会逐字节检查它是否与可执行文件一致。
- **Lean 内核**，以及 bv_decide 的已编译 LRAT 检查器；其用途会以辅助公理的形式出现。没有任何证明使用原生对象_decide。

可信基不 包括 C 编译器、汇编器或链接器，因为证明针对的是它们的输出；也不包括 libc，因为它是二进制文件的一部分。它也不包括浮点：模型中的 softfloat 运算是不透明函数，且没有任何安全性论证依赖其结果。NaN 装箱会通过整数测试规范化每个双精度数，而任何源自浮点结果的值（例如由 fcvt 得到的数组索引）都会在使用前进行边界检查。构建是可复现的：编译源代码会精确地产生 ElfFile.lean 中的字节。

定理没有说明什么。该定理并未说明 SafeJIT 能正确实现 JavaScript 、一定会终止，或不会泄漏内存。内存不足、嵌套限制和 JavaScript 错误都会导致正常退出，因此也符合安全性要求。无限循环的程序也是安全的。这些都是有意为之的选择。安全性是攻击者需要破坏的属性，而且无需 JavaScript 的形式语义即可表述。上述更强的属性均属于扩展，而非替代方案（第 11 节）。

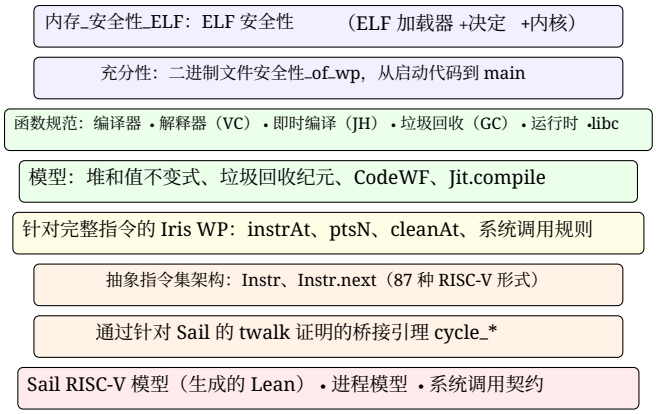


**图 3**。 RISC-V 证明的各层。红色部分是可信的，橙色部分是指令集架构抽象，黄色部分是程序逻辑，绿色部分是程序证明，蓝色部分是陈述。

## 5 证明架构

图 3 展示了证明的结构。在最底层，Sail 模型将一条指令执行为一个由寄存器读取、内存事件和位向量操作组成的长程序。每种指令形式都对应一个桥接引理，它将这个程序转换为抽象的单步语义。其上是一个针对完整指令的 Iris 程序逻辑，而二进制文件中各函数的分离逻辑规约则在该逻辑中得到证明。该逻辑的充分性进而推出安全性。本节将自底向上介绍这些层次。第6节介绍即时编译器。

### 5.1 从 Sail 到抽象指令集架构

Sail 模型的周期函数编译为 Lean 后，是一个自由单子中的程序，其效应包括寄存器读写、内存事件和非确定性选择。为了对它进行推理，我们使用符号执行器 twalk——一种 Lean 策略，它可以证明：对于任意效应的任意结果，只要从满足某些假设的符号寄存器文件开始执行一个周期，最终状态就满足给定的后置条件。它一次执行一个效应。它通过内核归约折叠封闭子项，并依次使用 decide、bv_decide 和 bv_omega 判定分支条件，且只使用涉及条件变量的假设。若无法判定某个条件，就对该条件进行分支。使用符号寄存器字段解码符号指令字时，需要特殊的化简过程，根据宽度选取指令字段。

每个 桥接引理 都会针对所有操作数证明一种指令形式。例如，cycle_addi 涵盖所有 rd、rs1 和 imm 对应的 addi rd, rs1, imm。它需要 487 步，耗时约 25 秒。该引理指出，周期会取指该指令，保持配置不变式 Cfg（U 模式、无地址转换、物理内存保护（PMP）和物理内存属性（PMA）设置不变，且禁用中断），将 PC 设置为 $pc{+}4$，并更新寄存器文件。加载与存储还要求地址位于主存中且自然对齐并记录数据访问。87 个桥接引理涵盖二进制文件中的每一种指令形式，以及即时编译所生成的每一种形式：30 种整数算术逻辑单元形式、8 种控制转移形式、11 种加载与存储、13 种 RV64M 形式、24 种 F/D 形式和 fence.i。对于乘除指令，另有一个单独的文件证明 Sail 的 Int 算术（带溢出钳位的截断除法）等于相应的 BitVec 运算。浮点结果是模型中不透明的 softfloat 函数。

脚本读取引理陈述并生成 Proc/ISA.lean：其中包含归纳类型 Instr，每个引理对应一个构造子；此外还包括其编码、内存前置条件 Instr.Pre，以及函数式下一状态函数 Instr.next。通用引理 step_instr 表示，如果程序计数器指向编码为 i 的可执行字节，且 i.Pre 成立，那么每个周期结束时的状态都由 i.next 给出。在这一层之上，没有任何证明会提及 *Sail*。脚本会将 .text 中的每个字解码为一个 Instr（或 ecall），并检查解码结果：每个字都必须能够唯一解码，并且重新编码后完全一致。

### 5.2 指令级分离逻辑

我们将 iris-lean 的语言接口实例化为一个进程：表达式是 AExpr.run r，即一个即将执行的寄存器文件；一步操作则是一条指令或一次系统调用。状态解释是一个关于内存字节的权威幽灵映射，其中包含每个字节的保护属性、内核状态的幽灵副本（osIs），以及一个由干净字节构成的幽灵集合。基本资源如下。

- ptsN a dq p n w： $a$ 处的 $n$ 字节持有 $w$，其份额为 dq（也可以是已丢弃的份额），且页面保护属性包含 p。bytesAt、slots 和 cstrQ 分别表示字节字符串、内容任意的栈字，以及以 NUL 结尾的字符串。
- cleanAt a：字节 $a$ 是只读且干净的，因此可以取指。它具有持久性，只有启动规则和成功执行 riscv_icache 刷新_的规则才能产生它。
- instrAt pc w：位于$pc$处的一个持久、可执行、干净且只读的字 $w$。因此，同样的规则既可从 .text 取指，也可从 JIT 代码取指。

每一类指令对应一条规则。例如，加载指令要求满足 instrAt r.pc i.encode 以及针对地址 i.access r 所计算出的指向关系，并在之后的 with 中以如下下一个状态继续：

$$\texttt{instrAt}\ pc\ (enc\ i) \mathrel{-\!\!*} \texttt{ptsN}\ a\ q\ p\ n\ w \mathrel{-\!\!*} \\ \triangleright\big(\texttt{ptsN}\ a\ q\ p\ n\ w \mathrel{-\!\!*} \text{WP}\ next(i,r,w)\ \{\{\Phi\}\}\big) \\ \mathrel{-\!\!*} \text{WP}\ r\ \{\{\Phi\}\} \qquad \text{if } access(i,r) = load\ a\ n.$$

系统调用规则遵循契约，并且既涵盖成功情况，也涵盖失败情况。例如，wp_sys_mmap 在成功时会在 $2^{56}$ 以下分配新的零填充页面。充分性定理，再加上对启动代码 _start 以及 __libc_start_main 序言的证明，表明它们能够携带

预期资源到达 main，从而将 BinarySafe 归约为 main 处的最弱前置条件。

函数规约。 函数规约是函数入口地址处、采用续延传递风格的最弱前置条件。这里是 strlen：

**规约_定理** strlen_规约 (x f : Nat → 位向量 64) (dq : DFrac) (p : Perm) (hpr : p.r = true) (s : 列表 Byte) (hx0 : x 0 = 0) (Φ: Byte → IP) ⊢textCode -∗cstrQ (x 10) dq p s -∗(cstrQ (x 10) dq p s -∗ ∀ x', ⌜ Clobbers x x' [10, 14,15] ∧x' 10 =BitVec.ofNat 64 s.length⌝ -∗ WP (AExpr.run x 1 &&& ~~~#64, x', f) {{ Φ}}) -∗WP (AExpr.run & strlen#64, x, f) {{ Φ}}

textCode 是对全部 .text 的持久所有权。延续是返回地址 ra（x 1）处、使用新寄存器文件 x' 的最弱前置条件；新寄存器文件只能在列出的寄存器中与 x 不同。因此，规约精确说明了某个函数可能破坏哪些寄存器。这就是编译器实际采用的应用二进制接口，其范围可能比标准应用二进制接口更窄。&strlen#64 是一个符号地址：一种从 ELF 符号表生成的表示法，会展开为此构建中 strlen 的地址。证明只以这种形式引用代码地址，例如 &fmod+0xdc 或 &"(null)"#64。重新链接后，大多数证明仍会通过检查，除非函数内部的代码发生了移动。

执行代码。 策略运行直线代码：sl_step 执行一条指令，根据该指令访问的地址，在上下文中找到正确的指向关系；sl_steps 则运行至下一个分支、跳转或 ecall。栈帧是从代码中读取的，而不是写入证明中的：frameSaves &fn 会扫描 fn 的序言，查找其寄存器溢出保存，因此证明无需硬编码编译器所选择的栈帧布局。循环通过归纳或 Löb 归纳进行证明。

模块化规约。 有数百个函数和规约，证明必须并行检查；对某个函数证明的修改，也不能触发其调用方证明的重新构建。为此，我们构建了一个小型 Lean 扩展（Holds.lean）。接口模块使用 spec_theorem 陈述规约，如上所述，但不提供证明。这会定义其陈述 foo_spec.Stmt，以及定理 foo_spec；后者将 [Holds foo_spec.Stmt] 作为实例参数。证明模块只导入其被调用函数的接口，将它们的陈述作为节变量，并通过 spec_holds foo_spec := by ... 证明自身规约；这会注册一个 Holds 实例，该实例以被调用函数的 Holds 假设为参数。顶层模块导入所有证明模块，并在 callee 规约完成后，先用 spec_link 将其关闭，随后由实例解析组装出一个封闭的证明项。这不涉及任何公理：如果缺少证明，实例便无法解析，而主定理也无法通过类型检查。RISC-V 证明包含 376 个规约_证明 声明，x86 证明则包含 222 个。除了构建并行性之外，这种拆分还使多个 AI 代理能够同时参与证明工作（第10节）。

### 5.3 运行时：堆、值与垃圾回收

分配器。 malloc 规范揭示了大小类设计：不变量 allocInv 拥有当前内存区域和 95 个空闲链表，而 malloc(n) 返回 0 或一个 mblock——即位于头部之后、经零填充且按 16 字节对齐的块，并拥有其一直延伸至类别容量的填充空间。free 接收一个 mblock 并将其回收。释放内存的客户端（例如即时编译的临时缓冲区和 realloc）会保留 mblock，而不会将其遗忘。

堆和值。堆不变式 heapInv 拥有一个从地址到对象类型（字符串、数组、对象、函数、原生对象）的权威幽灵映射。每个存活对象都有一个持久片段：

**定义** objAt (p : Addr) (k : Kind) :IP :=⌜p.toNat % 16 = 0 ∧ 16 ≤ p.toNat ∧ p.toNat < 2 ^ 47⌝ ∗tyIs p k ∗ objRo p k**定义** valOk (v : 位向量 64) : IP :=⌜IsNum v⌝ ∨ ⌜IsSpecial v⌝ ∨ ∃ k, ⌜v >>> 48 = 0 xFFFC#64⌝

∗ objAt（v &&& ptrMask）k

objRo 提供对对象不可变部分的分片只读访问权限：字符串的字节，或函数的字节码、常量和元数据。可变部分（数组的长度、容量和元素块，对象的形状和槽位，函数的反馈字节、计数器和 fn->jit 指针）保存在堆不变式中，并从中借用。对于函数，只读部分还包括以下事实：其字节码格式正确（见下文的 CodeWF），跳转目标表与之相符，并且每个属性访问操作码都指向一个现有的内联缓存。valOk 是值不变式：虚拟机栈、全局变量、数组元素、对象槽位和返回值都持有有效值。其指针情形将地址限制在 $2^{47}$, 以下；证明正是在此处发现了地址空间漏洞（第8节）。

内联缓存是单元格 icCell a n，它们要么为空，要么存储属性 n 对应的形状和槽位索引，或者存储一个会添加 n 的形状转换。形状是永久且持久的（shapeAt sh ns），而单元格的不变式 icOk 规定，缓存的槽位索引小于缓存形状的属性数量，因此具有该形状的对象必定有这个槽位。即时编译代码会在运行时读取缓存，因此无论缓存当时存储什么，这个不变式都能确保从缓存槽位内联加载是安全的。运行时的属性辅助函数更新缓存时，幽灵映射 *loans* 会记录该单元格是从其所属函数的函数体中借出的。

垃圾回收纪元。 objAt 是持久的，但对象会被垃圾回收器释放。我们通过纪元来处理这一问题。堆幽灵

映射不是全局常量：objAt、valOk 以及提及它们的每个不变式都以实例 HG : HeapGS（当前纪元的幽灵名称）为参数。垃圾回收会分配一个新的幽灵映射，其中只包含已标记对象。旧纪元的持久事实都会失去效用，只有那些被转移的事实例外。转移过程

**定义** epT (HG HG' :HeapGS) (R : 列表 (位向量 64)) : IP :=□ ∀ p k, tyIs (HG :=HG) p k -∗(⌜k.Imm ∨RootP R p⌝ ∨ ∃ q fi, ⌜RootP fi.consts p⌝ ∗tyIs (HG :=HG) q (.fn fi)) -∗tyIs (HG := HG') p k

也就是说，在 HG 中存活且可从根 R、存活函数的常量或不朽对象到达的每个对象，在 HG' 中仍然存活。垃圾回收器的规约 gc_spec 将被调用函数、虚拟机栈和全局变量作为根，并返回一个满足 epT HG HG' R 的新纪元 HG'。每个可能在垃圾回收后恢复执行的延续，例如 vm_call 调用者的延续或即时编译代码的 jKont，都会量化其恢复执行时所在的纪元，并在调用期间携带自己的根 (vstk B rs)。解释器只在一个确切的位置切换纪元，即 vm_call 开始处的安全点。垃圾回收器的证明分为标记（其中包含关于标记栈和已标记集合的不变量）、根循环、排空、清扫（释放未标记的字符串、数组和对象），以及新纪元的构造。

### 5.4 字节码编译器

解释器和即时编译的安全性取决于字节码格式正确：解释器不会在运行时检查栈深度或操作数范围。因此，证明分为一个关于字节码的谓词和两个定理：编译器建立该谓词，而对于任何满足该谓词的代码，解释器和即时编译都是安全的。CodeWF 要求

- 每个指令起始位置（即记录有深度≥ 0的字节）都包含有效的操作码，其操作数位于代码范围内，且不带有记录的深度；
- 操作数必须在有效范围内：常量索引小于 nconsts，局部变量索引小于 nlocals，全局变量索引小于 4096；
- 记录的深度 depth[pc] 沿每条边保持一致：跳转目标以及指令顺序执行时的后继位置，都具有该指令执行后留下的深度，因此控制流不会从代码末尾越界；
- 任何指令都不会导致栈下溢，并且指令执行前后的深度都至多为 max_栈 ≤ 30,001；
- 执行从 程序计数器 = 0 处开始，且栈为空。

第二个谓词 WalkWF 表示，从 0 开始线性解码代码（正如即时编译器所做的那样）时，只会遇到指令起始位置，包括不可达代码中的指令起始位置。编译器在生成代码时会写入深度[程序计数器]和跳转目标标志，而不是单独运行分析；证明则跟踪一个模型，其中

**定义** VC : IP := □ ∀ (HG : HeapGS) x f nj d n Φ pm E ob vs,⌜x 0 = 0 ∧ x 3 = gpAddr ∧ (x 2).toNat % 16 = 0 ∧8 * vcW nj d ≤ (x 2).toNat ∧ nj ≤ 1000 ∧ d ≤ 20000 ∧x 12 = vs.长度 ∧ ...⌝ -∗□ rtCtx pm -∗ □ Φ 1 -∗ □ 虚拟机运行时世界 E ob -∗ □ valOk (x 10) -∗ vstk (x 11) vs -∗slots (x 11 + 8 * vs.长度) n -∗slots (x 2 - 8 * vcW nj d) (vcW nj d) -∗虚拟机运行时世界 nj d -∗vcKR x f nj d (vs.长度 +n) Φ -∗WP (AExpr.run ⟨&vm_call#64, x, f⟩) {{ Φ }}

**图 4.** vm_call 的规约（简化版）。a0（x 10）是被调用函数，a1 是虚拟机栈上的参数 vs，而 sp（x 2）必须为 vcW nj d 个字留出空间。vmWorld 是运行时世界（分配器、堆、内核状态、全局变量），vcKR 是延续，它会在新纪元中接收结果。通过 JavaScript 错误退出满足 Φ 1。

经由递归下降解析器生成的代码。编译器的规约 cp_spec 声明，compile_program 接收任意字节串作为源代码，要么因语法错误而退出，要么返回一个有效的函数对象，因此对它创建的每个函数，CodeWF 都成立。编译器证明有 38,457 行，是除即时编译器之外最大的单个组件。原因之一是词法分析器、解析器和代码生成器在对源代码进行单次遍历时彼此交织。

### 5.5 解释器

解释器的规约VC(图 4) 表示vm_call(callee, args, argc)在虚拟机栈上接收到有效的被调用函数和有效参数后，会在取决于当前调用深度d的本机栈预算内，返回有效值或因 JavaScript 错误而退出。解释器通过三个入口点调用即时编译器：jit_compile、jit_has_entry 和 jit_enter。它们的规约构成命题 JH，而解释器的证明以此为假设：

```
vm_call_spec : ⊢ □ JH -∗ VC.
```

证明采用 Löb 归纳，因为 vm_call 是递归的：CALL 操作码处理程序会调用 vm_call，而归纳假设在一个后续模态下可用，该模态会被 call 指令消去。证明涵盖了每个操作码处理程序、分派循环、参数设置、层级提升、OSR 入口、去优化后的恢复，以及对原生函数的调用。JIT 代码可以回调 vm_call，因此 JH 对 jit_enter 的规约反过来将 ▷VC 作为假设（第6节）。由于 JH 只以这种形式假设 VC，即时编译证明 jh_holds : textCode ⊢ JH 不依赖于解释器的证明，而 vc_holds : textCode⊢ VC 则将二者结合起来。

本机栈。栈溢出会导致崩溃，因此每种规范都要考虑本机栈使用量：slots (x 2 - 8 * w) w 表示函数占用的栈字数。解释器的预算 vcW nj d 取决于 JS 调用深度 d（上限为 20,000）以及 nj，即 to_string 当前正在拼接的数组数量（最多 1000，用于检测循环引用）。C 代码会

在递归前检查这些限制，而二进制文件的静态 64 MiB 栈足以应对最坏情况。libc 的启动代码会切换到这个栈，因此内核的栈限制无关紧要。

### 5.6 将各部分整合起来

顶层模块 Sep/Safe.lean 链接了所有具体说明，并整合了启动证明、main 的选项解析（包括每个调试标志）、文件读取、cp_spec 和 vc_holds。结果是 memory_安全：BinarySafe。最后，ElfCheck.lean 通过 decide +内核证明，ELF 加载器应用于原始文件后，能够准确得到生成映像的入口点和各个段。此步骤大约需要 80 秒，逐字比较数据。它给出了 memory_安全_elf。

## 6 验证即时编译

即时编译是本项目的核心，其证明也是规模最大的部分：RISC-V 上有 51,095 行，x86 上有 101,323 行。本节将说明其结构。困难的核心在于，即时编译的输出并不是一段固定的代码。它取决于函数的字节码（由攻击者控制）、类型反馈（同样由攻击者控制）以及即时编译自身的分析结果。我们必须证明，jit_compile 能生成的每一段代码都 具备安全性，并且二进制文件中的 jit_compile 只会生成此类代码。

### 6.1 两个层次

我们通过一个编译器的 Lean 模型来拆解这个问题。

1. Jit.compile : JIn → JS 是对 jit_compile 的函数式转录，按函数逐一对应，运行于一个状态单子中；该单子操作的记录镜像了 C 语言的结构体 J：指令缓冲区、字节码到代码的程序计数器 pc_映射、跳转修正列表、局部变量和栈槽的抽象类型、待处理的（虚拟）栈，以及失败标志。输入 JIn 包含字节码、常量、跳转目标、反馈字节和循环事实。C 语言中的整数变为 Int，循环变为结构化递归，而指令字则通过与 C 编码器相同的位公式构造。2. **代码安全性**：对于每个格式良好的函数和每个输入，如果编译没有失败，那么无论将其放置在何处，它生成的代码都能安全地在偏移量 0 处以及 pc_映射所列出的每个跳转目标处进入。3. **精化**：二进制文件中的 jit_compile 在函数对象上运行时，会将 compile i 的 buf 中的各个指令字以及 compile i 的 pc_映射，准确地写入新分配的可执行内存；其中，i 是该函数当前的字节码和反馈，以及第一遍计算出的循环事实（jit_compile_spec）。

模型并不可信：精化证明将其与二进制文件联系起来。不过，如果模型与 C 代码不一致，精化证明就会失败，而且失败方式会令人困惑。

因此，我们采用差分测试：tests/jitmodel/run.sh 会从插桩构建中转储测试套件和基准测试的每次编译结果，通过 Lean 编译的 compile 逐一重放，并检查代码字、程序计数器映射和失败标志位是否完全一致。每种体系结构都有一个这样的测试。

即时编译与其余证明之间的接口是 JitModel 类，堆不变式使用它来描述 fn->jit：

**定义** jitCode (p : Addr) (fi : FnInfo) (j : Addr) : IP :=∃ (i : J.In) (mem pmA : Addr), ⌜J.ok fi i ∧...⌝ ∗ptsN j .discard rwPerm 8 mem ∗ptsN (j + 16) .discard rwPerm 8 pmA ∗J.代码 mem i ∗ □ (∀ 程序计数器, ⌜程序计数器 < fi.fc.len⌝ →ptsN (pmA + 4 * 程序计数器) .discard rwPerm 4 (J.pcMap i 程序计数器))

函数的即时编译代码是一个持久事实：代码 J.code mem i（对于 RISC-V，即 codeAt mem (compile i).buf，也就是每个字对应的 instrAt）及其程序计数器映射永远是只读的。即时编译代码永远不会被释放：当函数去优化时，fn->jit 会被清除，但其他激活实例可能仍在运行旧代码。堆幽灵状态、解释器和垃圾回收可与任意 JitModel 协同工作，这正是它们能够在两种体系结构之间共享的原因。

### 6.2 安装代码

jit_compile 会编译到 malloc 分配的缓冲区中，然后通过 mmap 创建新的页（读写权限），复制代码，并调用 mprotect 将其设为可读可执行。在 RISC-V 上，随后会调用 riscv_flush_icache；如果任何步骤失败，则会解除页映射并返回 NULL。在逻辑中，mmap 返回具有写入权限的 bytesAt，mprotect 将其转换为具有执行权限的已丢弃（只读）分数，而只有刷新规则才能生成 instrAt 所需的 cleanAt 令牌。这就是 W⊕X：代码页绝不会同时可写和可执行，不会再次变为可写，也绝不会被修补。因此，代码的只读所有权可以是持久且可复制的：任意数量的激活实例都可以共享代码，没有任何东西能写入其中，而且证明无需推理自修改代码。

### 6.3 安全不变式

代码安全性针对固定环境 $\varepsilon$ 进行陈述：JEnv：模型输入、代码地址、栈帧、虚拟机栈区域、入口处的寄存器以及延续。持久部分是 jCtx $\varepsilon$ Φ：模型未失败，代码位于 $\varepsilon$.B，函数对象处于存活状态，并且各种布局事实均成立。所有权部分 j Res $\varepsilon$ Φ ls st 拥有包含局部变量 ls 和操作数栈 st 的解释器栈帧、栈帧中未使用的槽位、运行时世界、保存的寄存器和本机栈，以及延续 jKont；后者是 jit_enter 的后置条件。ls 和 st 中的所有值都满足 valOk。

寄存器分配的局部变量。在 RISC-V 上，前五个局部变量存放在被调用者保存寄存器中。它们对应的栈帧槽位已过时。因此，jRes 拥有一个保存 lm 的栈帧；lm 除寄存器局部变量外均与 ls 一致，而纯事实 RegsOk $\varepsilon$ x ls 表示寄存器中保存着当前值。每次从 JIT 代码发起的调用都会保留这些值，因为它们位于被调用者保存寄存器中；被调用函数的规范通过 Clobbers 对此作出说明。在可能触发垃圾回收的调用之前，代码会将寄存器局部变量溢出保存到栈帧中，使其成为根。

虚拟栈。RISC-V JIT 会延迟压栈：如果某个常量或局部变量入栈后，紧接着就被下一条指令消费，那么它就不会被存储。对于待处理基址 *pb* 之上的每个槽位，编译器状态都会记录它位于寄存器、局部变量还是常量中。不变量 jResP 表示，从 pb 起向上的槽位都位于编译器状态所指示的位置，并且内存中只保存 pb 以下的栈内容。

类型事实。为进行守卫消除，编译器会跟踪每个局部变量和栈槽的抽象类型，以及某个槽位是否是局部变量的副本。这些事实构成一个 Iris 命题：

```
def tyV (t : Nat) (v : 位向量 64): IP :=(if t = AT_ARRAYthen arrV v
else emp) ∗ ⌜t = AT_BOOL →v = V_TRUE ∨v = V_FALSE⌝def
tyOk (s : JS) (ls st : 列表 (位向量 64)) : IP :=⌜∀ k <st.length, 0 ≤
s.ssrc[k] →st[k]? =ls[s.ssrc[k]]?⌝ ∗□(∀ k v, ⌜st[k]? =some v⌝ →
tyV s.stype[k]v) ∗□(∀ j v, ⌜ls[j]? =some v⌝ →tyV s.ltype[j]v)
```

注意 tyV 未包含的内容。整数和双精度事实没有说明任何事情，因为它们与安全性无关：错误地将某个值当作整数处理的代码会用垃圾位进行计算，但会将结果重新装箱为有效整数。另一方面，数组事实意味着即时编译器会不经检查就读取数组的长度和数据指针，因此必须有 objAt 作为依据；布尔事实也很重要，因为代码通过翻转低位来计算 !b。安全不变式弱于完整的类型不变量，因此证明规模更小。事实必须在垃圾回收后仍然成立，因为数组事实会在当前纪元中提及 objAt。只要栈帧中的每个值都是根，tyOk_epoch 就会通过 epT 传递这些事实。

主循环。编译器会线性遍历字节码：轨迹 traj i m 表示经过 $m$ 步后所处的字节码程序计数器和编译器状态。规约 SS $\varepsilon$ $\Phi$ m 表示在与第 $m$步之前的编译器状态一致的任意寄存器文件和栈帧下，第 $m$ 步生成的代码都是安全的：

$$\begin{aligned}\Box \forall x\ f\ ls\ st.\ \ulcorner pc_m < len \wedge depth(pc_m) = |st| \wedge \\ \texttt{RegsOk}\ \varepsilon\ x\ ls \wedge \ldots \urcorner \mathrel{-\!\!*} \texttt{tyOk}\ s_m\ ls\ st \mathrel{-\!\!*} \\ \texttt{jResP}\ \varepsilon\ \Phi\ s_m\ x\ ls\ st \mathrel{-\!\!*} \triangleright \texttt{VC} \mathrel{-\!\!*} \\ \text{WP}\ \langle B + 4 | s_m.buf |, x, f \rangle\ \{\{\Phi\}\}\end{aligned}$$

SS_all : jCtx $\varepsilon$ $\Phi$ $\vdash$ $\forall$ m, SS $\varepsilon$ $\Phi$ m 可通过 Löb 归纳，并结合从轨迹末尾开始对 $m$进行的反向归纳来证明。某一步的代码会继续执行下一步（由反向归纳提供），或者跳转到任意一步（由跳转指令之后的 Löb 假设提供）。Löb 归纳涵盖所有堆纪元，因为调用的代码可以在新纪元中恢复执行。

单步证明是对操作码和代码生成策略进行分类讨论，代码生成策略由反馈字节选择。每种情况都是一个模板引理：例如，从满足 tyOk s 和 jResP 的栈帧及编译器状态 $s$出发，针对带溢出检查的整数加法所生成的指令，会运行到模板代码末尾，并得到一个满足该步骤之后状态不变量的栈帧；或者跳转到该指令的去优化桩。模板引理是针对模型生成器的输出 陈述的，例如 exec (genInsn pc op d) i s，因此对偏移量、寄存器编号和立即数的算术运算会以符号方式求值，而这些引理可以一次性适用于所有这些值。各操作码的证明分布在约 100 个模块中，并行构建。

跳转和栈上替换。 在跳转目标处，编译器会忘记所有类型事实并物化虚拟栈，因此目标处的不变量是平凡的不变量（tyOk_pre_tgt）。这样，任何跳转都可以在此继续执行。栈上替换入口正是这些位置：jit_enter(jc, f, pc) 搭配一个其 pc_map 条目已设置的 pc 运行序言；序言会保存被调用者保存寄存器并加载寄存器局部变量，然后跳转到该指令的代码。其前置条件 frameAt 是解释器的栈帧不变式，它在没有类型事实的情况下蕴含 jRes。

循环事实。 循环事实使情况变得复杂。带有事实的循环头会先检查假定的局部变量类型；检查之后的代码，也就是循环头部，便可据此进行推理。状态蕴含这些事实的后向跳转会直接跳到循环头部，跳过检查。所有其他入口（包括栈上替换）都会经过这些检查。事实 lf 是模型的输入，而证明对每个 lf 都成立：C 代码会在第一遍计算这些事实，但安全性并不取决于计算方式。因此，循环事实分析未经验证，也无需验证。如果分析结果有误，代码就会去优化。

退出。 每条指令的守卫都会分支到该指令之后的一个共享反优化桩。该桩会将虚拟栈还原为指令开始时的状态，将字节码程序计数器加载到 t0，并将栈深度加载到 t1，然后跳转到公共退出点。退出点会将寄存器局部变量溢出保存到栈中，将 pc 和深度存入栈帧，恢复被调用者保存寄存器，并返回 1（wp_stub，wp_exit）。解释器会从此处恢复执行，重新执行失败的指令。要确保这一过程安全，栈帧必须满足该 pc 处解释器的不变量 frameAt。这一点成立，因为即时编译器的栈帧始终包含有效值，且其深度符合 CodeWF 为该 pc 指定的深度。由于守卫失败时该指令尚未产生任何可见效果，因此栈帧仍处于指令执行前的状态。请注意，这个证明并未说明去优化是否

正确，也就是说，解释器计算出的结果是否与即时编译器计算出的结果相同；它只说明解释器能够安全地继续执行。

调用。 来自即时编译代码的 JavaScript 调用会进入运行时的 rt_call，后者会调用 vm_call；因此，即时编译代码是可重入的。jit_enter 的规约假设 ▷VC，即时编译证明会在调用辅助函数时使用该规约。对其他辅助函数（rt_binop、rt_get_prop 等）的调用则直接使用它们各自的规约。每次调用之前，代码都会将寄存器局部变量和虚拟栈写回栈帧，这也使它们成为垃圾回收的根。调用之后，模板引理会将类型事实和延续迁移到新纪元。

### 6.4 精化：jit_compile 计算编译

精化证明是一个常规的（尽管规模较大的）分离逻辑证明，证明对象是一个包含 malloc、realloc、free 和循环的 C 语言程序：jit.c 中的每个 C 函数都有一条规约，说明对于一个存有模型状态 $s$的缓冲区，该函数会使缓冲区存有执行相应模型函数后的状态。由于模型是逐字转写而成，每个函数的证明都遵循其代码。模型中的算术运算基于无界整数，因此证明也表明 C 语言算术运算不会溢出，这一结论利用了 CodeWF 的界限。jit_compile 最多执行三轮（第一轮用于计算循环事实，第二轮为正式执行；若失败，则进行不使用事实的回退执行），并且可能返回 NULL，因为 mmap、mprotect 或刷新操作可能失败。堆内存耗尽会导致正常退出。最后，jh_holds 将 jit_compile_spec、jit_has_entry 的规约以及 jit_enter 的规约（该函数运行序言，然后使用 SS_all）封装到 JH 中。

### 6.5 为何这项工作可行

有三项选择让 JIT 证明保持在可处理的范围内。首先，这两个层次将两种截然不同的推理分开：一种关于生成代码的语义，另一种关于操作缓冲区的 C 语言程序；由于它们只在编译时相接，因此可以分别开发和修改。其次，JIT 的设计考虑了证明需求：只有一个编译器状态，每条指令各有一个去优化桩，而不是每个守卫各有一个，并且类型事实会在每个跳转目标处重置。第三，不变量只涉及安全性。证明无需说明 JIT 代码计算出哪个值，只需证明它有效；正因如此，才能证明无论反馈如何，代码都是安全的。

## 7 x86-64 移植版

为了测试这种方法是否依赖于 RISC-V 的简单性，我们将引擎和证明移植到了 x86-64。x86 后端（src/x86_64/jit.c，共 1,063 行）是 RISC-V 即时编译器的移植版本，保留了其设计：相同的栈帧布局、策略、抽象类型、守卫消除和去优化存根，但没有寄存器分配、虚拟栈或循环事实。除此之外，C 代码共用，二进制文件使用 gcc 的默认选项 -march=x86-64 构建。x86 证明针对 Sail x86-64 模型陈述了相同的定理 memory_safe_final : BinarySafe，适用于所有命令行，包括调试选项。有一处关联比 RISC-V 上弱：x86 没有 memory_safe_elf 的对应项。该定理针对 Image.entry 和 Image.segments；脚本 proofs/tools/elf2lean_x86.py 从可执行文件中提取这些内容，而在 x86-64 上，这个脚本属于可信基。从 x86 模型的首次提交到无条件定理，共花了九天；调试选项又花了两天。

### 7.1 模型

我们从 sail-x86 模型生成 Lean [40], ，此前该模型尚未被编译为 Lean。为此，我们必须修补模型并编写后处理脚本，因为我们在 Sail 的 Lean 后端中遇到了十个错误（例如，`return` 内的 `try/catch` 处理程序、未转义且被用作标识符的 Lean 关键字，以及无法解析的记录更新），并遇到了两个与 Lean 4.32 相关的问题：其新的 `do` elaborator 会在某些连接点后跟循环时失败，而一些递归的 Sail 函数没有终止度量。该模型还需要一些 RISC-V 模型不需要的假设：

- **CPU 配置**。 通过将 CPUID 特性标志设为常量，将处理器固定为 x86-64-v1（SSE2，不支持后续扩展）。这正是 gcc 和我们的即时编译器所生成的内容，而且引擎从不执行 cpuid。

- **浮点**。 模型中的 162 个浮点外部项是不透明常量。只要 MXCSR 保持默认值（舍入到最近值，屏蔽异常），且二进制文件中不包含更改 MXCSR 的指令，这就是可靠的。

- **循环**。 Sail while 循环会被编译为 Lean 的 Loop 类型，该类型在自由单子中未作规定。我们提供了一个燃料实例，最多运行 $2_{64}$ 次迭代，然后触发断言失败。计数为 64 位的 rep movsb 会在此上限内终止。

进程模型采用 RISC-V 的模型，但不包括`riscv_刷新_指令缓存`。在 x86 Linux 上，单个线程能在指令取指时看到自己的存储，因此不需要进行脏字节跟踪。地址空间遵循 Linux 对 4 级分页的默认设置：`mmap` 的结果和初始栈位于 $2^{47} - 4096$ (第 8 节）以下。 `mprotect` 和 `munmap` 契约都要求相应范围已映射；而 RISC-V 与 Linux 一样，也允许对未映射的页执行 `munmap` ； 这对程序提出了更严格的义务。模型会将处理器异常记录在待处理故障标志中，而不会传递异常。写入此标志、更改特权级别或模式，或断言失败，都不会产生步进。内核以原子方式执行系统调用 指令。

### 7.2 衔接变长指令

在 RISC-V 上，我们将 .text 解码为指令列表。x86 指令长度可变，因此映像是由原始字节页组成的列表。证明会使用 decide +kernel 从 .text 中截取函数的字节，模型再对其进行解码。一个桥接引理涵盖一种编码，其中 ModRM 字节、REX 前缀和操作数均为符号，并给出 RIP、通用寄存器和标志位的新值，以及精确的内存事件列表。标志位以位向量运算表示，例如用 BitVec.saddOverflow 表示 OF。146 个文件中共有 474 个桥接引理。其中大多数由脚本生成（按运算、操作数大小和前缀分类的算术逻辑单元形式；按寻址模式分类的加载与存储；SSE 移动、算术运算、比较和转换；移位；条件码），其余则手工编写。覆盖率脚本会反汇编二进制文件，并报告每条未被任何引理匹配的指令。某些指令需要特别处理：

- idiv 遇到零除数以及 INT_MIN / -1 时会触发陷阱；其引理要求除数非零且商能够表示，而即时编译器会检查除数并以 64 位进行除法，因此不会出现第二种情况。
- rep movs 和 rep stos 在模型中是循环；其引理通过对带燃料的循环进行归纳证明，并为每次迭代设定一个不变量。
- gcc 使用 x87 fprem 指令内联实现 fmod，因此需要为若干 x87 指令编写引理，并假设 x87 标签字满足一定条件，以排除栈错误。
- `ucomisd` 会为无序操作数设置 ZF、PF 和 CF，因此即时编译器会选择在 NaN 时结果为 false 的比较。x86 算术运算会传播 NaN 载荷，因此结果可能与 NaN 装箱标签冲突，即时编译器会检查每个浮点结果（第 3 节）。defaultNaN `0xFFF8...` 本身是有效的双精度数，因此不作处理。

### 7.3 共享证明

x86 程序逻辑采用相同的 Iris 实例化，只是使用针对字节范围而非指令字的 codeAt a bs 资源。体系结构无关的模块（幽灵状态、指向关系、字节字符串、栈槽）在两种开发中共享。运行时数据结构的模型（分配器的大小类别和 mblock、堆和值不变量、形状和内联缓存）也同样共享；它们只通过 Layout 类引用全局地址，只通过 JitModel 引用 JIT 代码，而这两者均由各目标分别实例化。

函数的证明本身并不共享，因为它们针对的是不同的机器码。不过，其结构与 RISC-V 上的相同，规约也相同，只有调用约定不同，因此 RISC-V 证明为其提供了蓝图。x86 即时编译器有自己的 Lean 模型（x86Jit）、自己的差分测试，以及相同的双层证明。x86 即时编译器的证明规模大于 RISC-V 的证明（10.1 万行对 5.1 万行），

部分原因是 x86 指令比 RISC-V 指令执行的操作更多：一条带内存操作数的加法指令会读取内存、进行计算并设置六个标志位，而证明必须跟踪后续分支所依赖的标志位。

## 8 证明发现了什么

SafeJIT 从一开始就是为形式化验证而编写的，并且在证明开始前经过了测试和加固：针对 Node.js 的差分测试套件、包含约 200 个格式错误或对抗性程序的健壮性测试套件，以及一轮审查和模糊测试。这些工作促成了对深层嵌套、循环数组、词法分析器中的越界读取、长度溢出，以及未检查 realloc 和 mprotect 返回结果等问题的修复。尽管如此，形式化验证期间还是发现了两个测试未能找到的错误。这两个错误都不是单靠证明发现的。我们请一个基于不同模型的智能体评估安全性主张的力度，以及其中哪些假设不切实际；正是在这一审查中发现了这两个错误。审查指出，可信模型有两处过于乐观：指令缓存和地址空间大小。使模型符合实际后，现有代码的证明便告失败，而失败之处也指出了必须修改的代码。

未检查的指令缓存刷新。Sail RISC-V 模型将 `fence.i` 作为空操作执行，而进程模型的第一个版本完全没有对指令缓存建模。在该模型下，即时编译器被证明是安全的，但模型本身是错的：真实的 RISC-V 核心在即时编译器写入新代码后，可能会取到过时的指令。我们在进程模型中加入了脏字节跟踪（第 4 节），而对即时编译器_编译 的证明失败了。（审查时已经预料到会失败。）C 语言代码调用了 gcc 的 __内置___清除_缓存函数，该函数在 Linux/RISC-V 上会调用 `riscv_`刷新_指令缓存 系统调用，但会丢弃其返回结果。如果系统调用失败，引擎就会运行 hart 可能看不到的代码。修复方法是直接调用 `__riscv_`刷新_指令缓存，并在返回错误时取消映射该代码并使编译失败。该系统调用在实际中很少失败，但这恰恰是形式化验证旨在消除的假设：契约说明它可能失败，因此证明必须处理这种情况。

高于 $2^{47}$的地址。 SafeJIT 的 NaN 装箱会将指针存储在值的低 48 位中。可信模型的第一个版本假定内核将所有内存放置在 $2^{38}$,以下，而这正是 Sv39 上的用户地址上限。Sv48 上并非如此，在 Sv57 上也不是：在 Linux 6.6 之前，初始栈可能被放置在 $2^{56}$,正下方，而当地址空间的低位部分耗尽后，mmap 会返回高于 $2^{47}$ 的地址。高于 $2^{48}$ 的指针在装箱时会被截断，导致引擎访问错误的内存。当我们修改模型，允许内核使用低于 $2^{56}$, 的任意地址后，启动代码及 mmap 调用者的证明失败了。现在，__libc_启动时，如果初始栈或任何 argv 字符串位于

或高于 $2^{47}$,，mmap 封装器会取消映射任何延伸至 $2^{47}$ 以上的映射，并以 ENOMEM 失败，从而使 malloc 和即时编译_能够干净地失败。这些检查位于共享 libc 中，因此 x86-64 构建也包含这些检查。不过，x86-64 模型遵循 Linux 默认设置的采用四级分页的 47 位地址空间：其 mmap 契约和初始栈均保持在$2^{47} - 4096$以下，因此其证明会经过这些检查，但无需处理成功的高地址分配或高地址初始栈。

关于可信基的教训。 这两个漏洞一度都躲过了机械化证明，因为可信模型编码了与程序员（此处为智能体）相同的乐观假设。可信基和代码一样值得审查。对于真实系统中模型抽象掉的每项特性（缓存、地址空间布局、失败的系统调用），都应当追问这种抽象是否保守。请一个基于不同模型构建的智能体来审查，是一种低成本的做法；它在几分钟内就发现了这两个问题。现在，我们明确规定系统调用契约：每次调用都可能失败，每次 mmap 都可能返回任何低于 $2^{56}$, 的地址，并且只有成功刷新后，新代码才可被取指。

工具。 生成模型的过程也暴露了工具中的漏洞。Sail 的 Lean 后端输出 `flen  = flen_bytes *8` 时没有加括号，导致 RISC-V 模型中的每个浮点寄存器都只有八位宽。x86 模型在 Lean 后端中又发现了十个漏洞，并发现 Lean 中的两个问题（第 7 节）。这类漏洞提醒我们，生成的模型也是可信基的一部分：如果模型中的漏洞让某条指令的行为比硬件更宽松，就可能掩盖引擎中的真实漏洞。

为便于验证而进行的引擎更改。 有些更改并非用于修复错误，而是为了让引擎更易于验证。编译器在生成代码时记录栈深度和跳转目标，取代了一个在加载时运行的独立字节码验证器；证明编译器输出格式正确，比证明验证器正确要简单，而且还省去了一次遍历。即时编译器中的每个代码生成器都标记为 noipa，因此 gcc 会单独编译它；对某个生成器的更改不会改变其他生成器的机器码，因而也不会改变相应的证明。引擎针对不含压缩指令的 RV64G 进行编译，因此指令按 32 位字对齐。libc 采用独立运行方式，使用原始系统调用和静态 64MiB 栈，因此规约无需依赖 glibc 或资源限制。

## 9 评估

我们评估三件事：即时编译是否值得采用（性能）、除了证明之外如何测试引擎，以及证明的成本（大小和检查时间）。

**表 4**。 运行时间（单位：秒；取 10 次运行中的最佳值）。RISC-V 使用 qemu-user，x86-64 使用原生对象。Node.js 18 的时间包含约 0.09 秒的启动时间。

| | RISC-V (qemu) | | | x86-64 | | | Node.js | |
|---|---|---|---|---|---|---|---|---|
| | 解释器 | JIT | × | 解释器 | JIT | × | 无即时编译 | full |
| fib | 0.29 | 0.13 | 2.2 | 0.037 | 0.014 | 2.6 | 0.141 | 0.117 |
| loop | 7.75 | 2.27 | 3.4 | 0.438 | 0.148 | 3.0 | 0.301 | 0.141 |
| mandel | 5.10 | 0.22 | 23.1 | 0.362 | 0.039 | 9.3 | 0.250 | 0.119 |
| 筛法 | 3.77 | 0.15 | 25.3 | 0.459 | 0.057 | 8.1 | 0.373 | 0.149 |

### 9.1 性能

SafeJIT 是一个基线 JIT，其基准测试是用于检验它所优化的四类操作的小型内核程序：调用（`fib`，对 27 求值的朴素递归斐波那契数列）、整数循环（`loop`，执行五百万次整数更新）、浮点循环（`mandel`，一个 $160 \times 160$ 每个点最多迭代 200 次的曼德博集合）以及数组访问（`sieve`，对不超过 200,000 的范围运行十次埃拉托斯特尼筛法）。表 4展示了运行时间。RISC-V 构建版本在 x86-64 主机上通过用户模式`qemu-riscv64` 运行，因此其绝对时间没有太大意义，但即时编译相对于解释器的加速比是有意义的：二者都在同一个模拟器下运行。x86-64 构建版本则原生运行于 AMD EPYC 7V12。我们报告十次运行中的最佳结果，测量范围涵盖端到端过程；SafeJIT 的启动时间约为半毫秒。

在 RISC-V 上，即时编译的速度是解释器的 2–25× 倍。推测执行消除的工作越多，性能提升就越显著：在 `mandel` 和 `sieve`中，循环事实让即时编译器可以省去对局部变量的类型检查；双精度数和数组元素也由内联代码处理，而不是通过运行时辅助函数处理。 斐波那契 的提升最小，因为每次调用都要经过运行时的 虚拟机_调用，而这会设置一个解释器栈帧。x86-64 即时编译器没有寄存器分配、虚拟栈或循环事实（第 7 节），并且每个局部变量都存放在内存中，因此其加速比较小，为 2.6–9。3×。（模拟器上的加速比不一定与 RISC-V 硬件上的一致；我们并未测量后者。）

为了让这些数字更有背景，表 4 还展示了 Node.js 18 在相同程序上的表现（并加入了一小段定义 `print` 的前置代码）。从端到端的表现来看，SafeJIT 的 x86-64 即时编译比 Node 在 斐波那契、`mandel` 和 筛法 上更快，在 循环上则大致相当；而 V8 的解释器（`--jitless`）在这四项上都比 SafeJIT 的即时编译慢。不过，Node 的用时包含约 0.09 秒的启动时间，对于如此短的运行而言，这一时间占据主导；而 SafeJIT 的启动时间不到 1 毫秒。扣除启动时间后，可以粗略估计 V8 优化层级的表现：在 9×循环 上比 SafeJIT 的 x86-64 即时编译快 约 23× 倍，在 `mandel` 上快约 1 倍，在 筛法 上不相上下，而在 2× 斐波那契 上慢约，因为启动时间占据了 Node 用时的大部分。精确的比较应当在进程内对这些内核程序计时。我们并未做出超出以下范围的主张：SafeJIT 的即时编译确实是即时编译；对于基线层所针对的那类代码，它能够带来足以证明采用基线层合理性的加速比。

### 9.2 测试

该证明没有涉及功能正确性（第 4 节），因此也对引擎进行了测试：

- **与 Node.js 进行差分测试**。运行 11 个测试程序（算术运算、数组、位运算、调用、比较、循环结构、双精度数、错误、循环、对象、字符串），分别采用四种模式：仅使用解释器、采用默认阈值 200、首次调用时编译（`--jit-threshold=1`），以及第二次调用时编译并允许最多 1,000 次去优化（`--jit-threshold=2 --max-deopts=1000`）。后一种模式会根据单次调用的类型反馈进行编译，因此经常触发去优化。每项输出都会与 Node 的输出进行比较。基准测试脚本还会检查解释器和即时编译是否打印相同的结果。

- **鲁棒性**。生成器会产生 204 个恶意输入：在每个语法位置嵌套至 200,000 层深、接近或超过限制的操作数栈、词法分析器边界情况、超大字符串和数组，以及 150 个随机令牌序列。每个输入都会在四种模式下运行；如果任何一次运行因收到信号而被终止或超时，测试就会失败。这套测试及其促成的修复，都早于该证明。

- **libc**。 独立运行的 printf、strtod 和 mathfunc- tions 会与 glibc 进行对比测试。

- **JIT 模型**。 RISC-V 引擎的插桩构建会记录每次编译（字节码、反馈、生成代码）。一个 Lean 程序会对每个记录的输入运行模型`Jit.compile`，并检查其生成的代码是否逐字节完全相同。该测试是编写模型的主要工具（第6节）。x86-64 模型也采用相同方式进行测试。

该证明使健壮性测试套件中的一部分测试变得多余：任何输入都不会导致引擎崩溃，因为崩溃就是一种卡住状态（但仍有可能超时）。不过，这些测试成本很低，而且在开发过程中极具价值：一次失败的差分测试能在几分钟内指出错误所在，而从卡住证明中寻找该错误可能需要数小时。

### 9.3 证明大小和成本

表 5展示了开发工作的规模。RISC-V 证明约有 221k 行 Lean，其中 57k 行是生成的：包括 ELF 映像及其符号的转储，以及指令数据类型。生成的 Sail 模型未计入其中。x86-64 证明约有 264k 行，其中 12k 行是生成的：包括映像、符号以及由脚本生成的桥接引理。总计而言，这两个证明包含约 485k 行 Lean，对应 6.5k 行 C 语言，比例约为 75:1（不含生成代码时为 64:1）。对于初始版本（4.4k 行 C 语言和 194k 行 Lean），该比例为 45:1；之后加入的即时编译优化（寄存器分配、虚拟栈、循环事实）是证明规模增长最快的部分。

检查证明所需的时间主要集中在少数几个步骤上。编译 Sail RISC-V 模型大约需要45

**表 5.** 按行数统计。缩进行包含在上一行中；生成 的内容计入总计。分离逻辑层 包含程序证明；即时编译 是即时编译模型及其证明。生成的 Sail 模型不计入。

| C 语言（引擎 + libc） | | Lean、RISC-V | |
|---|---|---|---|
| 引擎（源代码） | 3,868 | *Sep* | 103,540 |
| RISC-V jit.c | 1,376 | 编译器 | 38,457 |
| libc | 1,545 | 解释器 | 7,870 |
| x86-64 jit.c | 1,063 | GC | 4,145 |
| | | *Jit* | 51,095 |
| | | 进程模型 | 7,748 |
| | | ELF 映像、符号 | 55,783 |
| | | 其他 | 2,655 |
| | | 总计 | 220,821 |
| | | 生成 | 57,488 |
| 总计 C 语言 | 6,476 | Lean、x86-64 | |
| | | *Sep* | 138,829 |
| | | *Jit* | 101,323 |
| | | 桥接 | 16,197 |
| | | 其他 | 7,466 |
| | | 总计 | 263,815 |
| | | 生成 | 12,022 |

CPU分钟。每个浮点桥接引理大约需要五分钟，并且可以并行构建。在控制流图中以符号方式运行函数的直线代码（`twalk`），每个周期需要5–25秒。检查 ELF 文件是否与映像匹配（`ElfCheck`，通过 `decide`+内核）大约需要80秒。由于 Lean 构建在模块级别采用增量方式，因此对某个函数的代码进行更改时，通常只需重新检查该函数的证明；这就是每个代码生成器都作为独立函数编译的原因（第8节）。

## 10 使用 AI 代理进行开发

SafeJIT 的全部内容（包括引擎、libc、测试、Lean 模型、证明和证明工具）均由 AI 代理编写：Claude Opus 5.5 在 GitHub Copilot CLI 中运行，由一名研究人员提供指导。在本报告发布前主分支上的 1,411 个非合并提交中，有 1,395 个带有某个代理的会话标识符；研究人员本人只提交了少量 README 修改和日常维护内容。本节介绍了工作的组织方式、成本以及我们的收获。我们认为，这是该项目最重要的成果：并不是小型即时编译器能够得到验证——这一点从未受到质疑——而是整个过程只花了四周时间。

### 10.1 设置与时间线

该项目始于一个简单的提示，要求构建“一个带有 RISC-V 即时编译器的简单 JavaScript 引擎”，并实现“一些基本的类型推测执行”。智能体在

**表 6.** 里程碑（2026年）。在11月16日至18日并行工作进程运行期间，每日提交数达到峰值，分别为339、344和250次。

| | |
|---|---|
| 10月23日 | 引擎编写完成；验证会话开始 |
| 10月28日 | 研究人员审查：证明未使用分离式——逻辑；迁移至 Iris |
| 11月6日 | 针对 RISC-V 二进制文件的首个无条件安全性定理 二进制文件 |
| 11月9日 | 带有经验证 free 操作的分配器；初始版本 com- 完成（404 次提交） |
| 11月10日 | 对象与内联缓存、垃圾回收；x86-64 专题开始 |
| 11月13日 | 指令缓存和地址空间模型修正（第8节） |
| 11月14日 | 调试选项已验证 |
| 11月15日 | RISC-V 寄存器分配 |
| 11月16日 | ELF 交叉验证；x86-64 证明已并行化 |
| 11月17日 | 循环事实 |
| 11月18日 | 针对 x86-64 二进制文件的无条件定理 |
| 11月20日 | x86-64 调试选项已验证 |

不到两小时内完成了引擎。随后，另一个代理会话受命在该引擎上复现 xv6iris 的方法，目标是证明其具备内存安全性，“尤其要证明优化的 JIT 生成代码也不会破坏这种安全性”，并允许修改引擎的构建方式。该会话持续了四周，并完成了 RISC-V 证明。表 6展示了各个里程碑。

最初的 RISC-V 版本（包括引擎、解释器、带推测执行的基线 JIT、栈上替换和去优化、分配器以及证明）耗时约十个工作日。之后的功能（对象、垃圾回收、寄存器分配、虚拟栈和循环事实）都在同一会话中添加，每项功能都在各自的分支上开发，并在证明再次覆盖该功能后合并。

对于 x86-64 移植版本，研究人员启动了第二个并行会话。第一周里，该会话独立开展工作：移植引擎、修补 Sail x86 模型的 Lean 输出、生成桥接引理，并构建共享模块。11 月 16 日，研究人员建议先移植规范文件，再并行完成证明。随后，智能体充当协调者：编写规范，将各组函数分配给工作代理（整个会话中共 22 个子代理，通常同时约六个），让每个代理在各自的分支上工作，并审查、合并这些分支，还在工作代理之间传递修复方案。三天内，x86-64 定理便在没有前提的情况下得到了证明。

### 10.2 研究人员做了什么

表 7 总结了这一互动过程。在所有会话中，研究人员编写了约 180 条提示，总计约 17,000 个字符，大致相当于本节的篇幅。其中，68 条是“出色的工作，请继续！”的某种变体，另有 14 条要求智能体合并一个分支。其余提示可分为以下几类：

**表 7。** 工作量。模型时间是模型请求持续时长的总和，包括并发请求。令牌包括已缓存的输入。全部列还包括初始引擎、对抗性审查和工具相关的会话，因此其数值超过另外两列之和。

| | RISC-V | x86-64 | all |
|---|---|---|---|
| 日历天 | 29 | 11 | 29 |
| 研究人员提示 | 56 | 69 | 182 |
| 子代理 | 2 | 22 | 33 |
| 模型请求 | 33.6k | 41.5k | 79.5k |
| 输入词元 | 3.6 G | 4.4 G | 8.4 G |
| 其中已缓存 | | | 97% |
| 输出标记 | 33 M | 34 M | 71 M |
| 模型时间 | 91 h | 94 h | 196 小时 |

- **设定目标**。最初的提示，以及对每个后续功能的选择：“我们现在的形状是否已经足够好，可以添加垃圾回收？”、“接下来我们来处理跨循环回边推断类型事实吧”、“Sail 有了一个全新的 x86 模型……我们应该试试它。”

- **引导设计方向**。影响最深远的一次干预发生在第五天：“我们似乎根本没有使用分离逻辑。它比 xv6 慢得多。”在此之前，智能体直接推理执行过程，采用全机器安全性的概念，且没有框架规则；因此，约 2.8 万行证明都必须迁移到基于 Iris 的逻辑。其他人则要求证明性质，而不是在运行时检查性质（由此促成了通过构造生成字节码元数据，见 第8节）；要求证明不受重新编译影响（由此引入了符号地址）；要求 x86-64 证明继续使用分离逻辑，而不是重复 RISC-V 证明中“历史遗留产物”的做法；还要求 x86-64 的规约模式与 RISC-V 的模式保持一致。

- **让智能体保持诚实**。研究人员要求智能体移除一个 sorry 占位符，它只是在策略中作为后备方案；研究人员还质疑了一份状态报告，其中把 RISC-V JIT 证明说成一个假设，但它实际上并非如此。研究人员还询问了另一个智能体，安全性主张究竟有多强，由此促成了 第8节 中的两项模型修正。

- **运维**。机器重启、构建卡住，以及并行构建超出文件描述符限制。

研究人员无需编写证明，也不必阅读其中的大部分内容：Lean 内核会检查这些证明，人类只需检查定理的陈述、可信基，以及该证明所依赖的公理（第 4 节）。

### 10.3 经验教训

证明检查器就是审阅者。证明助手是智能体的理想环境：每项主张要么

通过检查，要么不通过，智能体无法靠言辞蒙混过内核。智能体在报告中确实会犯错：有一次，某个智能体漏掉了一个引理中的 sorry 占位符，却报告说该引理已经证明（并在下一步中发现并修正了这个问题）。这类错误通过查看定理陈述和 #print axioms 就能发现，而不必阅读数十万行证明。每个里程碑的检查清单都很简短：nosorry、不使用原生对象_decide、只包含预期的公理，以及定理陈述未被更改。

规约是并行工作的基本单位。因为每个函数的证明只依赖于其被调用函数的规约（第5节），所以证明可以按任意顺序进行，也可以由不同的代理完成。在 x86-64 工作的第一周，一名代理构建了模型、桥接引理和共享基础设施，并验证了最初的几个函数。规约就位并实现并行化后，其余函数（包括即时编译器）仅用了三天便完成。协调者的工作主要是合并分支、重新构建，并在不同工作者之间传递修复方案（例如，针对 elaborator 接受但内核拒绝的证明所采用的变通方案）。

机械化规模不成问题。证明的篇幅约为代码的 75 倍。对人类而言，这样的工作量难以承受：seL4 的形式化验证耗费了约 20 人年，其中约 11 年用于 seL4 专属证明，其余用于框架和工具 [22]。同样由代理构建的 xv6iris 连同其框架共耗时 102 日历天 [21]。（人年和日历天是不同的单位；SafeJIT 耗时 29 日历天。）对代理而言，证明的规模远不如难度重要，而 SafeJIT 的许多证明虽然冗长，却很简单：对直线代码进行符号执行，每种指令形式对应一个引理，每种操作码对应一个模板引理。能够让证明更机械化的投入（例如，生成桥接引理的脚本、twalk 之类的策略，以及能让证明在重新链接后仍然有效的符号地址），比让证明变短的投入更有成效。

可执行模型能够及早发现错误。JIT 模型是一个可以运行的 Lean 函数，在测试套件的每次编译中都将其与 C 代码进行测试（第9节），在几秒钟内发现了模型中的大多数错误，远早于尝试进行证明。

可信基需要进行对抗性审查。编写证明的智能体也编写了进程模型，第8节中的两个错误都隐藏在该模型中。无论证明作者是人类还是智能体，由证明作者选择的模型都是薄弱环节。让另一个智能体攻击这一主张，成本低且效果显著；我们会将其作为常规步骤。

资源。 构建时间始终是一个令人担忧的问题。Lean 构建规模增长到超过 1,400 个作业，有些单独的证明检查起来需要几十个 CPU分钟。智能体在构建性能方面投入了相当多的精力（拆分模块、减少导入、替换开销较大的策略调用），通常是在研究人员的提示下进行的。随着多个智能体同时进行构建，这台机器（128 个核心）遇到了其文件描述符上限。

导入、替换开销较大的策略调用），通常是在研究人员的提示下进行的。随着多个智能体同时进行构建，这台机器（128 个核心）遇到了其文件描述符上限。

# 11 讨论

## 11.1 二进制文件的形式化验证并不会使编译器过时

Kaashoek 和 Zeldovich 认为，验证二进制文件“使 MachCSL 能够将编译器、链接器、汇编器以及构建过程的其他部分排除在可信计算基之外” [21], ，并且证明免费吸收了 gcc 的优化。SafeJIT 依赖于同一理念：gcc 不受信任，而我们从不对 C 语言进行推理。

即时编译展示了这一论证的边界。即时编译运行的机器码在证明完成时尚不存在。它是在之后根据攻击者选择的程序和类型反馈生成的。二进制文件可以经过验证，但其中有一个函数是编译器，而其规约必须涵盖它可能接收的每个程序以及它可能生成的每份代码。“对于每个格式良好的函数和每个反馈向量，生成代码都保持不变量”是一条编译器验证定理，而 SafeJIT 证明的大部分工作都花在这里：为每个操作码建立模板引理，证明能够移除守卫的抽象类型具有可靠性，以及证明去优化和 OSR出口与解释器保持一致。二进制级验证将 gcc 排除在可信基之外；但它并没有消除验证作为系统组成部分的编译器的必要性。即时编译引擎正是这类系统，而且它们是现实中遭受攻击最频繁的软件之一。经过验证的编译器研究仍然不可或缺，只是现在它位于二进制级验证之中。

JIT 证明的双层结构（第6节）是我们将两者结合起来的方式：一个即时编译的模型，它是一个 Lean 函数，使用编译器验证的常规工具证明其安全性；以及一个证明，表明 即时编译_编译所得的二进制文件 精化了该模型。细化层是机械化的（RISC-V JIT 约 1.2 万行），其规模不会随着分析复杂度的提升而扩大，只取决于代码生成器的规模。

## 11.2 局限性

安全性，而非正确性。定理表明，引擎不会陷入停滞。但它并未保证引擎计算出正确答案、最终会终止，或不存在侧信道。推测执行攻击不在模型涵盖范围内，大多数经过验证的系统也是如此；V8 的开发者认为，代码级缓解措施无法完全遏制 Spectre，因此转而采用进程隔离 [28]。JIT 给出错误答案仍是严重的缺陷，但就第1节所说的可利用性而言，并非如此：在生产环境引擎中，类型混淆之所以危险，是因为它会导致内存破坏，而定理排除的正是内存破坏。

一个小型引擎。 SafeJIT 缺少闭包、异常、Eval、大多数标准库、正则表达式、类型化数组和并发。其即时编译器是一个带有简单

寄存器分配的基线 JIT；它没有优化层。我们认为，缺失的语言特性属于“同类扩展”：每种新增的堆类型都会扩展堆不变式和解释器的操作码证明，而针对这些特性的 JIT 快速路径与现有的数组和对象路径具有相同的形状（标签保护、类型保护、边界或形状检查，然后直接访问）。原生库函数根据固定接口单独进行证明，不会触及 JIT 证明。优化层则是另一回事。大多数生产环境中的 JIT 错误都出现在这里，而验证一个优化层需要大规模的经验证编译器工作。我们的证明结构可以沿用，但工作量会更大。

内存回收。 垃圾回收器会释放字符串、数组和对象，但不会释放函数或 JIT 代码。函数本来就是永久存在的，而且由于一个函数最多只能被编译$N$次（--max-deopts 限制，默认为 8），去优化泄漏的代码最多是程序大小的常数倍 $N$。一旦函数可以消亡，回收代码就成为必要之举，而这是一项跨领域的变更：代码所有权（codeAt）是证明中的持久事实，因此运行中的激活记录需要非持久证据来证明其代码仍保持映射。我们计划将存活代码保存在垃圾回收器开启的每历元不变量中，并统计每段代码的激活实例数量。

单线程。 没有后台编译，也没有并发垃圾收集。该逻辑支持并发（MachCSL 验证了多核 xv6），但引擎将需要进行最大幅度的修改。

可信基。 该定理信任 Sail 模型、我们对 Linux 的进程模型、Lean 的内核，以及 `bv_decide`(第 4 节) 的 LRAT 检查器。进程模型由我们构建，而第8节说明了它可能以一些会产生实际影响的方式出错。在 x86-64 上，浮点指令是不透明的，并且证明假设 MXCSR 寄存器保持其默认值（第 7 节）。

### 11.3 未来工作

按价值排序，接下来的步骤是：

1. **边界检查消除**。数组访问始终会检查索引。当已知索引处于有效范围内时（例如将循环计数器与长度进行比较），要消除该检查，除了循环类型事实之外，还需要范围事实，以及数组在此期间不会缩短这一不变量。这一领域的错误很隐蔽：在 CVE-2019-8518 中，循环不变代码外提将一次数组访问提升到了保护它的边界检查之前 [15]。

2. **更广泛的寄存器分配**。 只有五个局部变量存活在寄存器中，待处理的栈槽会在每个合并点刷新。在合并点之间保留待处理状态，就能消除循环中剩余的大部分栈帧流量。

3. **x86-64 的功能对等**。x86-64 即时编译器没有寄存器分配、虚拟栈或循环事实。移植这些功能主要是移植 RISC-V 模型和证明，而这类工作正是智能体擅长的。4. **代码回收与闭包**，如上所述。5. **eBPF**。 在同一框架中实现 eBPF 验证器和即时编译器，将是另一个具有直接实际意义的演示项目；验证器的范围分析也正是边界检查消除所需的分析类型。

## 12 相关工作

表8 将 SafeJIT 与最接近的既有工作进行了比较。

推测执行与反优化。 带反优化的推测执行可以追溯到 Smalltalk 和 Self [9, 12, 18]。Guo 和 Palsberg [44] 对基于轨迹的编译进行了形式化，包括热路径记录和侧出口，并采用双模拟判据。Flückiger 等人 [13] 对带有显式假设的推测执行以及跨程序版本的反优化给出了形式化描述：sourir 是一种带有假设指令的中间表示，这些指令会将程序反优化到优化程度较低的版本；他们证明了在存在这些指令的情况下，一组优化是正确的。D’Elia 和 Demetrescu [11] 给出了栈上替换的正确性论证，该机制能够跨程序版本重构状态。在转换过程中保持程序的活跃值，比我们所证明的内容更强（我们证明的是目标栈帧可以安全恢复执行），但它并不是关于引擎二进制文件的定理。CoreJIT [3] 是一个在 Coq 中验证的即时编译器，能够在 CompCert 风格的中间表示上执行推测、反优化和动态优化。它证明了整个动态编译过程的语义保持。其后继工作 [4] 通过 CompCert 增加了原生后端，并将解释代码与生成代码之间的转换推理延伸至 CompCert 的 x86 汇编语义；但它信任从汇编语法到已安装机器字节的步骤，以及运行时原语的实现。这些都是正确性证明，在这一意义上比我们的证明更强。SafeJIT 的定理较弱（仅保证安全性），但其信任边界是指令集架构模型：解释器、运行时、垃圾回收器、libc 以及生成代码的字节，全部都是经过验证的二进制文件的一部分。我们的即时编译器所使用的抽象类型、循环事实和去优化元数据，都比这些工作所处理的优化简单得多。

面向真实代码的经验证即时编译器。 Myreen [30] 使用其将反编译转化为逻辑的方法，在 HOL4 中验证了一个面向小型基于栈的字节码语言的即时编译器，直至 x86 机器码，并涵盖自修改代码。Jitk [43] 使用 CompCert 验证了一个经典 BPF 即时编译器；Jitterbug [32] 逐条指令验证了 Linux 内核的 eBPF JIT，发现了许多错误。CertrBPF [45] 是面向嵌入式操作系统的经验证 eBPF 解释器；Agni [42] 和 VeRA [7] 则分别验证了 Linux eBPF 验证器和 Firefox 的 JIT 中的范围分析。Tur- boTV [24]验证了单独的 TurboFan 中间表示变换

**表8**。 经验证的即时编译器和相关系统。“推测执行”是指针对动态观测类型设置守卫，并在必要时反优化到较低层级。

| Work | 经验证的内容 | 层级 | 定理 | 推测执行 |
|---|---|---|---|---|
| Myreen [30] | 小型字节码语言的即时编译 | x86 机器码 | 功能正确性 | no |
| Jitawa [31] | Lisp 运行时、垃圾回收和即时编译 | x86-64 机器码 | 功能正确性 | no |
| Jitk [43] | 经典 BPF 即时编译器 | Coq，使用 CompCert 提取 | 功能正确性 | no |
| Jitterbug [32] | Linux eBPF 即时编译器，逐条指令 | C 语言的 Rosette 模型 | 功能正确性 | no |
| CertrBPF [45] | 适用于 RIOT 的 eBPF 虚拟机 | Coq、C 语言 | 正确性、隔离 | no |
| Agni [42] | Linux eBPF 验证器范围分析 | SMT | 可靠性 | – |
| VeRA [7] | Firefox 即时编译范围分析 | C 语言++ 遍，SMT | 单项分析的可靠性 | – |
| Icarus [37] | CacheIR 存根生成器 | 宏汇编器模型 | 桩代码的安全性 | 内联缓存守卫 |
| sourir [13] | 推测性优化 | IR | 语义保持 | yes |
| CoreJIT [3, 4] | 带推测执行和去优化的即时编译 | Coq 模型、CompCert 后端 | 语义保持 | yes |
| VeriWasm [19] | WebAssembly 提前编译输出 | x86 二进制检查器 | 沙箱安全性 | no |
| vWasm [6] | WebAssembly 编译器 | F* | 沙箱安全性 | no |
| Crocus [41] | Cranelift 降低规则 | SMT | 规则正确性 | no |
| CakeML [23, 36, 39] | ML 编译器、运行时和 Eval | 机器码 | 功能正确性 | no |
| Islaris [35] | 机器码片段 | Sail Arm/RISC-V | 功能正确性 | – |
| xv6iris [21] | xv6 内核和用户程序 | RISC-V 二进制文件，Sail | 功能正确性 | – |
| SafeJIT | 整个引擎，包括 JIT 输出 | RISC-V 和 x86-64 二进制文件、Sail | 对所有输入的安全性 | 是，另包括栈上替换 |

在 V8 中借助 SMT；对单次转换的检查既不是测试判定基准，也不是对所有生成程序的证明。这些系统针对的是不具备动态类型的语言，或只验证某个组件，而且它们都不采用带反优化的推测执行。

Jitawa [31] 是与其即时编译器一同在机器码级别完成验证的动态语言运行时中最接近的先例：它在 HOL4 中对一种 Lisp 的解析器、打印器、复制式垃圾收集器和 JIT 编译器进行了验证，直至 x86-64 代码，并给出了一个函数正确性定理，其前提包括初始内存和源程序的执行。它在运行每个函数之前对其进行编译，不使用反馈、守卫、去优化或栈上替换。Icarus [37] 是实际 JavaScript 引擎中采用类型特化快速路径的最接近先例。它通过基于宏汇编器模型进行符号元执行，验证 SpiderMonkey 的 CacheIR生成器能够生成的内联缓存存根的安全性，并将生成的代码集成回 Firefox。其验证单元是存根生成器和建模的操作；而我们的验证单元是引擎映像、生成的指令字节以及层级之间的转换。内联缓存未命中会回退到通用路径，这也比整帧去优化更简单。

***WebAssembly***。 对于 WebAssembly，VeriWasm [19] 会检查提前编译器生成的 x86 输出是否始终处于沙箱内；vWasm [6] 是一个经过 F* 验证、能生成沙箱化代码的编译器；而 Crocus [41] 则验证 Cranelift 的指令降级规则。与 SafeJIT 一样，VeriWasm 和 vWasm 证明的是安全性属性，而非正确性；对于沙箱而言，重要的正是这一属性。这些系统都没有验证推测执行；生产环境引擎确实会进行推测执行以 WebAssembly 为例，V8 会根据反馈内联间接调用，并在假设不成立时进行去优化 [26]。

机器码验证。CompCert [27] 和 CakeML [23, 39] 都是经过形式化验证的编译器；CakeML 的证明一直延伸到机器码，并涵盖其运行时和垃圾回收器。CakeML 还支持运行时编译：其经过验证的交互式 REPL [23] 及其经过验证的 `Eval` [36]会编译并安装与现有代码交换值的新代码，因此仅凭运行时代码生成并不能区分 SafeJIT；真正起区分作用的是反馈导向推测。Cai、Shao 和 Vaynberg [8] 提出了一个 Coq 框架，用于对存储在可写内存中的代码以及生成的代码进行局部推理。我们的`codeAt` 资源将同一思想应用于具体的引擎代码，并结合页面权限和 RISC-V 缓存刷新约定。Islaris [35] 在 Sail ISA 模型之上引入了分离逻辑，使用 Isla 计算指令轨迹，并使用 Iris 进行推理。xv6iris [21] 将这一思想应用于整个系统：它在 Lean 中，根据 Sail 模型，使用构建于 iris-lean 之上的并发分离逻辑 MachCSL，验证 xv6 内核和用户程序的 RISC-V 二进制文件 [20, 25]。其证明也借助 AI 代理完成，并在 xv6 中发现了 11 个错误。SafeJIT 直接构建于 xv6iris 之上：我们复用其 Sail 模型配置、其 Sail Lean 后端分支，以及 MachCSL 的部分内容（访问类型、启动序列和板级写入）。SafeJIT 的新颖之处在于 JIT 在此基础上所需的内容：作为资源、在运行时创建并通过 `mprotect` 和缓存刷新安装的代码；其输出针对每个输入都经过验证的 JIT；以及连接这两者的 第 5.4 节和第 6 节中的证明结构。seL4 [22] 仍然是衡量完整功能正确性证明对系统组件所需工作量的参考标准

这是对系统组件进行完整功能正确性证明所需工作量的参考标准。

测试 ***JIT*** 编译器。诸如 Fuzzilli 的模糊测试器 [17] 和 JIT-Picking [5] 能够发现许多 JIT 缺陷，而将 JIT 与其自身的解释器进行差分测试是一种标准的判定基准。我们在开发过程中也采用了相同的技术（第9节）。它们能够发现缺陷；证明则能证明缺陷不存在。

## 13 结论

SafeJIT 是一个带有推测式 JIT 编译器的 JavaScript 引擎，其 RISC-V 和 x86-64 可执行文件均已在 Lean 中针对 Sail ISA 模型完成证明，确保其在任何输入上都不会卡住。该证明涵盖所有会运行的部分：解析器和字节码编译器、解释器、运行时和垃圾回收器、独立运行 libc、JIT 编译器，以及 JIT 可能针对任何程序和任何类型反馈生成的每一段代码，包括其守卫、反优化退出和栈上替换入口。C 编译器和浮点运算均不在信任范围内。

该引擎规模很小，其即时编译器也是基线 JIT。但使 JIT 编译器变得危险的部分——对攻击者可控反馈进行推测执行、基于抽象类型移除守卫，以及层级之间的转换——都存在于其中，并且都在攻击者实际操作的层级上完成了形式化验证。其中有两点尤为重要。第一，当二进制文件中包含编译器时，二进制级验证并不能免除对编译器进行验证的必要；二者必须结合起来，而构建一个经过安全性证明、再由二进制文件细化的 JIT 模型，是将二者结合起来的有效方式。第二，AI 代理改变了这类工作的成本效益。证明的规模是代码的 75 倍，耗时四周完成；期间只有一名研究人员负责设定目标、引导设计，并检查论断和可信基。我们认为，这表明生产级即时编译引擎，以及其他长期以来被认为难以企及的安全关键系统，都有可能在其二进制文件层级上得到形式化验证。

## 致谢

SafeJIT 构建于 M. Frans Kaashoek 和 Nickolai Zeldovich 开发的 xv6iris 之上：包括其 Sail 模型配置和 Lean 后端分支，以及其 MachCSL 框架的部分内容（MIT 许可证）。我们感谢 Sail、Sail RISC-V 和 x86 模型、Lean 以及 iris-lean 的作者；本工作依赖于这些项目。

## 参考文献

[1] Apple。2024。《通过锁定模式强化您的设备》。Apple 个人安全用户指南。https://support.apple.com/guide/personal-safety/harden-your-devices-with-lockdown-mode-ipsd5baf79d0/web[2] Alasdair Armstrong、Thomas Bauereiss、BrianCampbell、Alastair Reid、Kathryn E. Gray、Robert M. Norton、Prashanth Mundkur、Mark Wassell、Jon French、Christopher Pulte、Shaked Flur、Ian Stark、Neel Krishnaswami 和 Peter Sewell。2019。ARMv8-A 指令集架构语义，RISC-V 和 CHERI-MIPS。《ACM 编程语言》3，POPL（2019）。https://doi.org/10.1145/3290384[3] Aurèle Barrière、Sandrine Blazy、Olivier Flückiger、David Pichardie 和 Jan Vitek。2021。JIT 编译器中推测执行与去优化的形式化验证。《ACM 编程语言》5，POPL（2021）。https: //doi.org/ 10 1145 3434327 . / Aurèle Barrière、Sandrine Blazy 和 David Pichardie。2023。有效应的即时编译器中的本机代码生成形式化验证：将 CompCert 后端转变为经形式化验证的 JIT 编译器。《ACM 编程语言》7，POPL（2023）。https://doi.org/10.1145/3571202[4] [5] Lukas Bernhard、Tobias Scharnowski、Moritz Schloegel、Tim Blazytko 和 Thorsten Holz。2022。JIT-Picking：JavaScript 引擎的差分模糊测试。载于 ACM SIGSAC 计算机与通信安全会议（CCS）。351–364。https://doi.org/10.1145/3548606. 3560624[6] Jay Bosamiya、Wen Shih Lim 和 Bryan Parno。2022。使用 WebAssembly 实现可证明安全的多语言软件沙箱隔离。载于 USENIX 安全研讨会。1975–1992。https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya[7] Fraser Brown、John Renner、Andres Nötzli、Sorin Lerner、Hovav Shacham 和 Deian Stefan。2020。迈向 JavaScript 即时编译器的经验证范围分析。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。135–150。https: // doi.org/ 10 1145 3385412 3385968 . / . Hongxu Cai、Zhong Shao 和 Alexander Vaynberg。2007。经认证的自修改代码。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。66–77。https://doi.org/10.1145/1250734.1250743[8] [9] Craig Chambers、David Ungar 和 Elgin Lee。1989。SELF 的高效实现：一种基于原型的动态类型面向对象语言。载于面向对象编程系统、语言与应用会议（OOPSLA）。49–70。https://doi.org/ 10 1145 74877 74884 . / . Leonardo de Moura 和 Sebastian Ullrich。2021。Lean 4 定理证明器与编程语言。载于自动演绎——CADE 28（LNCS，第 12699 卷）。625–635。https://doi.org/10.1007/978-3-030- 79876-5 37[10] [11] Daniele Cono D’Elia 和 Camil Demetrescu。2018。精简解读栈上替换。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。166–180。https://doi.org/10.1145/3192366.3192396[12] L. Peter Deutsch 和 Allan M. Schiffman。1984。Smalltalk-80 系统的高效实现。载于 ACM SIGACT-SIGPLAN 编程语言原理研讨会（POPL）。297–302。https://doi.org/ 10 1145 800017 800542 . / . Olivier Flückiger、Gabriel Scherer、Ming-Ho Yee、Aviral Goel、Amal Ahmed 和 Jan Vitek。2018。通过动态去优化确保推测性优化的正确性。《ACM 编程语言》2，POPL（ 2018 49 1 49 28 10 1145 3158137 ），：–：。https://doi.org/[13] [14] 。/ Google 威胁分析小组和 Mandiant。2024。。我们同在：2023 年野外利用零日漏洞年度回顾。https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/Year_in_Review_of_ZeroDays.pdf[15] Samuel Groß。2019。JavaScriptCore：因循环不变代码外提将数组访问移至边界检查之前而导致 FTL 即时编译器越界访问。Project Zero 问题 42450850（CVE-2019-8518）。https://project-zero.issues.chromium.org/issues/42450850[16] 2024 8 Samuel Groß。。V 沙箱。V8 博客。https://v8.dev/blog/ sandbox[17] Samuel Groß、Simon Koch、Lukas Bernhard、Thorsten Holz 和 Martin Johns。2023。FUZZILLI：针对 JavaScript JIT 编译器漏洞的模糊测试。载于网络与分布式系统安全研讨会（NDSS）。https://doi.org/10.14722/ndss.2023.24290

[18] Urs Hölzle、Craig Chambers 和 David Ungar。1992。使用动态去优化调试优化代码。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。32–43。https://doi.org/10.1145/143095.143114[19] Evan Johnson、David Thien、Yousef Alhessi、Shravan Narayan、Fraser Brown、Sorin Lerner、Tyler McMullen、Stefan Savage 和 Deian Stefan。2021。Doveryai, no Proveryai：原生编译 WebAssembly 的 SFI 安全性。载于网络与分布式系统安全研讨会（NDSS）。https: //doi.org/ 10 14722 2021 24078 . /ndss. 。Ralf Jung、Robbert [20] Krebbers、Jacques-Henri Jourdan、Aleš Bizjak、Lars Birkedal 和 Derek Dreyer。2018。Iris 从零开始：高阶并发分离逻辑的模块化基础。《函数式编程杂志》28（2018），e20。https://doi.org/10.1017/S0956796818000151[21] M. Frans Kaashoek 和 Nickolai Zeldovich。2026年。将并发分离逻辑扩展至硬件层，以使用 AI 代理验证 RISC-V 上的 xv6 操作系统内核。https://doi.org/10.48550/arXiv. 2609.04043 arXiv:2609.04043v3 [cs.LO] 版本 3。[22] Gerwin Klein、Kevin Elphinstone、Gernot Heiser、June Andronick、David Cock、Philip Derrin、Dhammika Elkaduwe、Kai Engelhardt、Rafal Kolanski、Michael Norrish、Thomas Sewell、Harvey Tuch 和 Simon Winwood。2009。seL4：操作系统内核的形式化验证。载于 ACM 操作系统原理研讨会（SOSP）。207–220。https://doi.org/ 10 1145 1629575 1629596 . / . Ramana Kumar、Magnus [23] O. Myreen、Michael Norrish 和 Scott Owens。2014。CakeML：ML 的经验证实现。载于 ACM SIGPLAN-SIGACT 编程语言原理研讨会（POPL）。179–191。https://doi.org/10.1145/2535838.2535841[24] Seungwan Kwon、Jaeseong Kwon、Wooseok Kang、Juneyoung Lee 和 Kihong Heo。2024。V8 JavaScript 引擎中 JIT 编译器的翻译验证。载于 IEEE/ACM 国际软件工程会议（ICSE）。 1 12 10 1145 3597503 3639189 – 。 [25] 2026 https://doi.org/ 。/ 。Lean 社区。。iris-lean：Lean 中的 Iris。GitHub 代码仓库。https://github.com/leanprover-community/iris-lean [26] 4 Daniel Lehmann 和 Matthias Liedtke。2025。使用去优化和内联为 WebAssembly 实现推测性优化。V8 博客。https://v8.dev/blog/wasm-speculative-optimizations [27] Xavier Leroy。2009。现实编译器的形式化验证。《ACM 通讯》52，7（2009），107–115。https://doi.org/10.1145/1538788. 1538814[28] Ross Mcilroy、Jaroslav Sevcik、Tobias Tebbi、Ben L. Titzer 和 Toon Verwaest。2019。Spectre 将长期存在：侧信道与推测执行分析。arXiv: 1902 05178 [ ] . cs.PL Microsoft。。 [29] 2025 借助 Microsoft Edge 增强网络安全性。Microsoft 支持。https://support.microsoft.com/en-us/edge/enhance-your-security-on-the-web-with-microsoft-edge [30] Magnus O. Myreen。2010。x86 上经验证的即时编译器。载于 ACM SIGPLAN-SIGACT 编程语言原理研讨会（POPL）。 107 118 10 1145 1706299 1706313 – 。 [31] 2011 https://doi.org/ 。/ 。Magnus O. Myreen 和 Jared Davis。。经验证定理证明器的经验证运行时。载于交互式定理证明（ITP）（LNCS，第 *6898* 265 280 10 1007 978 3 642 22863 6_20 卷）。– 。 [32] https://doi.org/ 。/ - - - - Luke Nelson、Jacob Van Geffen、Emina Torlak 和 Xi Wang。2020。实践中的规约与验证：将形式化方法应用于 Linux 内核中的 BPF 即时编译器。载于 USENIX 操作系统设计与实现研讨会（OSDI）。41–61。[33] Johnathan Norman。2021。超级安全模式。Microsoft 浏览器漏洞研究博客。https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode/ [34] RISC-V International。 [不详。]。Sail RISC-V 模型。GitHub 代码仓库。https://github.com/riscv/sail-riscv

[35] Michael Sammler、Angus Hammond、Rodolphe Lepigre、Brian Camp- bell、Jean Pichon-Pharabod、Derek Dreyer、Deepak Garg 和 Pe- ter Sewell。2022 年。Islaris：依据权威指令集架构语义验证机器码。载于 ACM SIGPLAN 编程语言设计与实现大会（PLDI）。825–840。https: // doi.org/ 10 1145 3519939 3523434 。/ 。Thomas Sewell、 [36] Magnus O. Myreen、Yong Kiam Tan、Ramana Kumar、Alexander Mihajlovic、Oskar Abrahamsson 和 Scott Owens。2023 年。会烘焙蛋糕的蛋糕：CakeML 中的动态计算。《ACM 编程语言汇刊》7，PLDI（2023），1121–1144。https://doi.org/10. 1145/3591266[37] Naomi Smith、Abhishek Sharma、John Renner、David Thien、Fraser Brown、Hovav Shacham、Ranjit Jhala 和 Deian Stefan。2024 年。Icarus：借助符号元执行构建可信的即时编译器。载于 ACM 操作系统原理研讨会（SOSP）。473–487。https://doi.org/ 10 1145 3694715 3695949 。/ 。 [38] 2022 Maddie Stone。 。知之愈多，愈知所不知：2021 年在野外利用的零日漏洞年度回顾。Google Project Zero 博客。https://projectzero.google/2022/04/the-more-you-know-more-you-know-you.html [39] Yong Kiam Tan、Magnus O. Myreen、Ramana Kumar、Anthony Fox、Scott Owens 和 Michael Norrish。2019 年。经过验证的 CakeML 编译器后端。《函数式编程杂志》29（2019），e2。https://doi.org/ 10 1017 0956796818000229 。/S Rui Ueyama。 。sail-x 。 [40] 2026 86 https://github.com/shina-lab/sail-x86/tree/1829cb6ecc5f924b0fd920582953a2402357a12a.[41] Alexa VanHattum、Monica Pardeshi、Chris Fallin、Adrian Sampson 和 Fraser Brown。2024 年。面向 WebAssembly 到本机指令选择的轻量级模块化验证。载于 ACM 编程语言与操作系统架构支持国际会议（ASPLOS）。https://doi.org/10.1145/3617232.3624862[42] Harishankar Vishwanathan、Matan Shachnai、Srinivas Narayana 和 Santosh Nagarakatte。2023 年。验证验证器：eBPF 范围分析验证。载于计算机辅助验证（CAV）（LNCS，第 13966 卷）。https://doi.org/10.1007/978-3-031-37709-9_12[43] Xi Wang、David Lazar、Nickolai Zeldovich、Adam Chlipala 和 Zachary Tatlock。2014 年。Jitk：可信的内核内解释器基础设施。载于 USENIX 操作系统设计与实现研讨会（OSDI）。33–47。[44] Shu yu Guo 和 Jens Palsberg。2011 年。基于轨迹的编译的本质。载于 ACM SIGPLAN-SIGACT 编程语言原理研讨会（POPL）。563–574。https://doi.org/10.1145/ 1926385.1926450[45] Shenghao Yuan、Frédéric Besson、Jean-Pierre Talpin、Samuel Hym、Koen Zandberg 和 Emmanuel Baccelli。2022 年。面向微控制器的 eBPF 虚拟机端到端机械化证明。载于计算机辅助验证（CAV）（LNCS，第 13372 卷）。https://doi.org/ 10.1007/978-3-031-13188-2 15_