arXiv:2601.09372v1 [cs.CR] 14 Jan 2026

使用 NAVe 形式化验证 Noir 零知识程序

Pedro Antonino 和 Namrata Jain

The Blockhouse Technology Ltd.，英国牛津
{pedro,namrata}@tbtl.com www.tbtl.com

摘要。 零知识（ZK）证明系统是一种密码学协议，能够以极高的概率证明一对 (X, W) 满足某种关系 R，同时不泄露有关私有输入 W 的信息。这种成员关系检查通过一个复杂的算术电路来描述：一组定义在有限域上的多项式方程。Noir 等零知识编程语言旨在简化这些电路的描述。开发者可以使用传统的高级结构编写 Noir 程序，并将其编译为较低级别的 ACIR（抽象电路中间表示）；ACIR 本质上是对算术电路的高级描述。本文使用 SMT-LIB 及其扩展的有限域理论，对 ACIR 语言的部分内容进行形式化。我们利用这一形式化，使用 SMT 求解器 cvc5 为 Noir 语言创建了一个开源形式化验证器。该验证器可用于检查 Noir 程序的行为是否符合预期。例如，它可以检查 Noir 程序是否约束正确，也就是说，生成的有限域多项式方程是否确实刻画了预期关系。我们在 4 组不同的 Noir 程序上评估了该验证器，展示了其实用性，并找出一种难以检查的约束类型，为验证框架的改进指明了方向。

关键词： 零知识证明 · 零知识程序 · Noir · 隐私 · 形式化验证

1 引言

零知识（ZK）证明系统最初被引入为交互式协议 [21, 22, 20]，其中证明者 $\mathcal{P}$ 使验证者 $\mathcal{V}$ 相信形如 $R(X, W)$ 的陈述，其中 R 是一个关系（$\mathcal{P}$ 和 $\mathcal{V}$ 已知），X 是 $\mathcal{P}$ 和 $\mathcal{V}$ 共同拥有的输入，而 W 是证明者私有的输入（通常称为见证）；协议的执行除了表明 $R(X, W)$ 成立之外，不得泄露有关 W 的任何信息。这些证明系统后来实现了非交互式 [17, 23]，并且（其生成的证明）具有简洁性 [19, 32, 24, 26]。随着区块链的出现和 Web3 生态系统的发展，零知识证明系统重新受到关注。它们被视为一种能够提供隐私 [1, 12]，甚至提高区块链效率 [10, 7]的技术。

在这些证明系统中，关系 R 通常通过算术电路来表示：算术电路是在有限域上定义的一组约束（即多项式方程），用于描述 X 与 W之间的关系。如果不熟悉这类系统，描述这些电路可能颇具挑战。为了简化这些系统的使用并提高其普及程度，面向算术电路的领域专用语言（DSL）应运而生。这些零知识编程语言提供了高级构造以及将这些构造编译为算术电路的基础设施，而算术电路正是零知识证明系统所使用的；Noir [8], 、Cairo [2], 以及（抽象层次较低的）Circom [3] 都是此类语言的例子。简单来说，它们可用于创建一个高级程序 P 来描述 R：如果 $R(X, W)$ 成立，则该程序 P将使用参数（X, W）成功运行。

Noir 语言 [8] 由 Aztec 社区 [1]提出。它既可用于创建独立的零知识程序，也可用于为 Aztec 网络编写隐私智能合约；智能合约是一种管理数字资产的程序，这些资产有时价值不菲。Noir 程序会被编译为一种名为 ACIR（抽象电路中间表示）的中间语言，随后由零知识证明系统使用。这种中间表示包含用于表达有限域多项式方程的操作码，也包含更高层级的操作码，例如范围检查——一种确保有限域值可用指定的位数表示的约束——以及*brillig* 调用——一种允许执行无约束代码的操作码。无约束代码是这类语言的一项特性，旨在帮助证明者生成证明，但它不会纳入用于描述关系 R的底层约束。程序员应单独加入相应的匹配约束，例如在 Noir 中使用`assert` 命令。

这种新的编程范式容易出现几类微妙的漏洞 [9,18]。例如，Noir 程序缺乏适当的约束；如果它执行了未被适当约束的无约束代码，就可能导致程序 P 与其试图表达的底层关系 R之间产生偏差。这类漏洞已经导致了现实世界中的严重漏洞；例如，请参见 [13], 。零知识程序还可能遭受其他类型的漏洞，例如非确定性——底层关系原本应当是一个函数，但实际上并不是——甚至编译器漏洞——生成的约束无法恰当地捕获高级编程结构的行为。这些漏洞的隐蔽性，以及零知识程序在高风险场景中的日益广泛应用——保护用户的私人信息或管理大量数字资产——使其成为形式化验证的理想目标。

本文中，我们使用 SMT-LIB 对 ACIR 的一个子集进行形式化 [16] ，并提出 NAVe：Noir ACIR（形式化）验证器；请勿将其与底层零知识证明系统的验证者混淆。我们将 ACIR 操作码子集的语义编码为 SMT-LIB。我们提出了两种编码：一种基于标准的整数算术理论，另一种基于 cvc5 求解器中实现的非标准有限域理论 [29, 30] 。 [15]我们使用这些形式化方法，并采用通过 Noir 函数来描述验证目标／证明义务的惯例`verify_assert(b: bool)`，从而创建了我们的开源形式化验证器

NAVe。我们的验证器专注于识别未受约束的漏洞：Noir 开发者可以在程序的退出点调用 `verify_assert` ，指定预期要捕获的关系 R 。我们已将验证器集成到 Noir 的包管理器 Nargo 中，以便开发者使用。我们在 Noir 语言基础设施中的一些测试程序上评估了验证器，以考察其实际效率。我们使用两种编码分析其性能，结果令人鼓舞。此次评估表明：(a) 某些由范围检查等操作码引入的约束类型，可能显著增加验证时间；(b) 我们的编码互为补充，有些程序使用其中一种编码分析更快，另一些则使用另一种更快。因此，我们的评估为未来工作指明了方向：我们需要提出抽象／近似来解决 (a)，并以某种方式同时使用两种编码分析 Noir 程序，从而受益于 (b)。

目前已有一些关于 ZK 程序自动形式化验证的研究 [28, 31, 33], ，但都未针对 Noir 语言及其特性。Noir 具有语义复杂的高级结构。例如，与其他语言不同，Noir/ACIR 支持高级内存操作、无约束函数，以及范围检查、位运算符等位级操作。Lampe [6] 使用 Lean 4 [27]刻画了 Noir 的语义。与我们的自动化验证器不同，Lampe 的用户必须通过交互方式引导对程序属性的证明。我们选择对 ACIR 而非 Noir 本身进行形式化，因为前者应该比后者更加稳定，经历的演进变化也更少。此外，我们的形式化成果也可供其他以 ACIR 为目标的语言使用。

总而言之，我们的贡献如下：

– 使用 SMT-LIB 标准整数算术理论和非标准有限域理论对 ACIR（以及间接地对 Noir）进行形式化 [29]。– 基于我们的形式化成果，为 Noir 编程语言开发开源形式化验证器 NAVe，并将其集成到广受欢迎的 Noir 包管理器（和构建工具）Nargo 中。– 在 Noir 测试程序上评估 NAVe，以展示其实际适用性，并识别一类复杂约束以及求解器/编码组合，为未来改进指明方向。

概要。 在第 2 节中，我们介绍必要的背景知识，使本文内容完整自洽。第 3 节介绍我们的形式化方法、验证者及其评估。第 4 节介绍相关工作，第 5 节给出结论。

2 背景

本节介绍一些概念，使本文内容完整自洽。

2.1 SMT-LIB、CVC5 和有限域理论

理论可满足性（SMT）将布尔可满足性推广到整数算术等其他理论。SMT 公式涉及这些理论中的谓词，

这些谓词可以使用布尔运算符组合。SMT 求解器（如 cvc5 [15], ）是实现判定过程、用于验证此类公式可满足性的工具。SMT-LIB [16]是一个用于标准化其中一些理论的库，为它们提供语义和语法。它已成为表示这些公式的事实标准语法。因此，即使是未纳入 SMT-LIB 标准的理论，也会使用其语法。例如，`(declare-const x Int)` 可用于声明一个类型/排序为 `Int` 的常量 `x`，而 `(assert (or (= x 0) (= x 1)))`则声明约束：`x` 只能是 0 或 1。此外，还可以使用 `(+ x y)`、`(* x y)`和 `(mod x y)` 来构造相应的典型整数运算。

cvc5 求解器还支持 [29] 中描述的非标准素数阶有限域理论 [4] 。素数阶有限域 p 与模 p 的整数同构。对于整数 x 和正整数 y，我们用 $(x \bmod y)$ 表示满足以下条件的唯一整数 r： q 是整数， $x = qy + r$ ，且 $0 \leq r < y$。设 p 为素数， F 为表示素数阶 p 域的排序， N 为整数， x_i 为表示同一域 F 中域值的项， $[x]$ 为有限域项 x 表示的整数。

- `(_` 有限域 p`)` 表示素数阶 p 排序的域，即模 p 的整数集合。`(as ff`N F`)` 表示有限域元素 N，即整数 $(N \bmod p)$。`(ff.add` x_1 ... x_n`)` 表示 n 元域加法运算。此项表示整数 $([x_1] + \ldots + [x_n] \bmod p)$。`(ff.mul` x_1 ... x_n`)` 表示 n 元域乘法运算。此项表示整数 $([x_1] \times \ldots \times [x_n] \bmod p)$。`(ff.bitsum` x_1 ... x_n`)` 表示 n 元位求和运算。此项表示整数 $(2^0[x_1] + \ldots + 2^{n-1}[x_n] \bmod p)$。`(=` x y`)` 表示相等运算符。对于 $[x]$ 和 $[y]$ 表示的整数，当且仅当 $[x] = [y]$ 时，该相等关系成立。

素数阶为 p 的域与模 p 整数之间的同构关系，意味着可以将有限域项编码为非线性整数算术项，并使用该理论的求解器检查其可满足性；然而，这个问题通常是不可判定的，而对于有限域项则是可判定的。

2.2 Noir

Noir [8] 是一种由 Aztec [1]社区提出的零知识编程语言。它既可用于构建独立程序，也可用于为 Aztec 网络构建智能合约。Noir 是一种高级编程语言，其语法受 Rust [11]启发。与 Rust 一样，可以通过 结构体 及其相应实现来创建抽象数据类型。不同于传统编程语言，Noir 内置了表示有限域的 域 类型，以及加法和乘法等域运算。实际使用的域取决于所采用的（后端）证明系统。对于 Noir 的默认后端，一个域元素将是一个跨越 Grumpkin 曲线 [5][1] 的 254 位整数。另一个 pe-

[1] 该有限域的阶为 21888242871839275222246405745257275088548364400416034343698204186575808495617。

```
fn main(x: 域) -> pub 域 { let i
nvx: 域 = inv(x); invx } fn inv(x:
域) -> 域 { let mut cur = x; let
mut y = 1; let rep:
    [1 253] = u ;; for i in
[1,1,...omitted...]
0..253 { if (rep[i] == 0) { cur =
cur * cur; } else { y = y * cur; cur
 = cur * cur; } } y * cur }
```

```
fn main(x: 域)->pub域{letmut
invx = 0;unsafe {invx =
invu(x);}assert(invx *x == 1);
invx}unconstrained fninvu(x: 域)-
> 域{inv(x)}
```

图 1. Noir 程序：受约束与无约束。

零知识编程语言的一个特点是区分受约束代码与无约束代码。前者会产生算术电路约束，后者则不会，因此必须由开发者手动添加单独的约束。无约束代码可以大幅减小生成的底层算术电路的规模和复杂度，从而缩短证明和验证时间。

我们简要介绍 Noir 语言，以及图 1 中 Noir 程序所展示的无约束代码的优势。有限域中非零元素的乘法逆元 x 可以通过计算 x^{p-2}得到，这依据的是费马小定理，其中 p 是该域的素数阶。左右两侧的程序分别由各自的`main` 函数表示，它们都从私有值$\mathtt{x}^{p-2}$计算出公开值`pub Field`返回类型。通过此程序，证明者可以向验证者证明自己知道公开值`x`所对应的`x`，且不公开$\mathtt{x}^{p-2}$。`x`。[2] 左侧的 Noir 程序使用受约束代码执行此计算，而右侧程序使用无约束代码。在 Noir 中，`let` 关键字用于声明变量，`mut` 修饰符表示变量是可变的。`inv` 函数使用迭代平方 [25]来计算 x^{p-2}：一个`for`循环遍历二进制迭代分解rep

[2]这些程序可在 https://github.com/pedrotbtl/nave 找到。它们是用于演示受约束代码与无约束代码之间差异的教学工具。Noir 内置了无约束代码和用于计算 $1/x$ 的约束，验证者可以从 $\mathtt{x}^{p-2}$ 计算出 x，从而绕过程序的隐私保障。

of $p-2$ 以执行此幂运算。在此示例中，我们使用 p，即 Grumpkin 基域的素数阶；为简洁起见，代码中省略了这个 253 位数组。右侧使用相同的`inv` 代码（位于函数`invu`中），但以无约束方式执行，并依靠`assert(invx * x == 1)` 语句确保以无约束方式计算出的输出`invx`是`x`的逆元；`assert` 命令会为底层算术电路生成约束。左侧的 Noir 程序会生成 379 个约束，而右侧程序只生成 2 个。在零知识程序中，无约束代码有助于证明者生成证明。在本例中，在零知识证明系统中，证明者需要计算`x` 的逆元来生成证明。然而，在零知识验证中，验证者只需检查`invx` 是否为`x`的逆元。

3 NAVe：用于 Noir 程序的形式化验证器

本节将对 Noir 的 ACIR 进行形式化，提出一种名为 NAVe 的形式化程序验证器，并开展评估，以考察 NAVe 的实际适用性。

3.1 ACIR 形式化

Noir 程序会被编译为一种中间 *ACIR* 程序，随后再转换为面向特定零知识证明系统的算术电路。ACIR 旨在将前端高级零知识语言与后端零知识证明系统解耦。它由 Aztec 社区为 Noir 创建，但设计上不依赖特定的高级语言。我们选择以 ACIR 为目标，而非直接以 Noir 为目标，原因如下：(a) 可以利用将 Noir 程序编译为 ACIR 所需的基础设施，而且 ACIR 的抽象层级较低，更易于描述其语义；(b) 只要这种中间语言保持稳定（而且它应该比 Noir 更稳定），我们的形式化验证器就应能在 Noir 发展并推出新版本的过程中继续与其兼容；(c) 由于 ACIR 的设计不依赖特定前端，我们未来也可以将 ACIR 语义用于验证其他以 ACIR 为中间表示的高级零知识语言。

ACIR 语言有 6 种操作码：`AssertZero`、`BlackBoxFuncCall`、`MemoryOp`、`MemoryInit`、`BrilligCall` 和 `Call`。在这项工作中，我们只对 ACIR 的一个子集进行形式化。我们不对 `Call` 操作码以及 `BlackBoxFuncCall` 操作码的某些行为进行形式化；这些内容留待未来工作处理。Noir 中的函数通常会被内联，因此 `Call` 操作码仅用于某些特定且略显复杂的证明场景，这些场景会使用递归证明。至于 `BlackBoxFuncCall`，其某些行为相当复杂，后文将对此进行详细说明。

对于给定的有限域，ACIR 程序由一系列作用于域变量（称为见证）的 ACIR 操作码$\langle op_1, \ldots, op_m \rangle$ 以及一系列 brilling 函数组成；后者与本文无关。操作码可以被视为创建约束，以限制见证可以取哪些值，而见证的有效赋值则记录了 ACIR 程序的一次执行。接下来，我们通过 SMT-LIB 形式化定义操作码的语义，即定义刻画每个 ACIR 操作码行为的约束。

我们提出将 ACIR 操作码翻译为非标准的有限域 SMT-LIB 理论（见第 2 节）以及标准的整数理论。

有些操作码作用于 ACIR 表达式：这是一种数据类型，表示次数至多为 2 的多项式。设 r_i、s_i、q 为（字面量）域元素，a_i、b_i、c_i为见证。这种数据类型是一个三元组，包含：一个长度为 n 的三元组列表，列表元素形式为 (r_i, a_i, b_i)；一个长度为 m 的二元组列表，列表元素形式为 (s_i, c_i)；以及一个域元素 q。它表示域多项式 $\sum_{i=1}^{n} r_i a_i b_i + \sum_{j=1}^{m} s_j c_j + q$。我们在 SMT-LIB 中按如下方式表示这一表达式。ACIR 变量与 SMT-LIB 变量都取值于同一个有限域 F。我们用 $[\![t]\!]_f$ 表示通过有限域理论对 ACIR 项 t （无论是操作码还是表达式）的编码。为简化说明，我们省略域元素的编码：值 v 显然编码为`(as ff`v F`)`，其中 F 是相应域的排序定义。

$$[\![(\langle(r_1, a_1, b_1), \ldots, (r_n, a_n, b_n)\rangle, \langle(s_1, c_1), \ldots, (s_m, c_m)\rangle, q) : \texttt{Expression}]\!]_f \equiv$$
$$\texttt{(ff.add (ff.mul } r_1\ a_1\ b_1\texttt{) ... (ff.mul } r_n\ a_n\ b_n\texttt{)}$$
$$\texttt{(ff.mul } s_1\ c_1\texttt{) ... (ff.mul } s_m\ c_m\texttt{) } q\texttt{)}$$

我们利用模整数算术与域算术之间的同构，通过 SMT-LIB 标准的整数理论提出一种替代的 ACIR 编码；我们用 $[\![t]\!]_i$ 表示这种形式化。在以下 SMT-LIB 公式中，r_i、s_i、q是（字面量）整数，而 a_i、b_i、c_i是整数变量。

$$[\![(\langle(r_1, a_1, b_1) \ldots, (r_n, a_n, b_n)\rangle, \langle(s_1, c_1) \ldots, (s_m, c_m)\rangle, q) : \text{表达式}]\!]_i \equiv \text{(模 (+ (*}$$
$$r_1\ a_1\ b_1) \ldots \text{(* } r_n\ a_n\ b_n) \text{(* } s_1\ c_1) \ldots \text{(* } s_m\ c_m)\ q)\ p)$$

`AssertZero` 操作码会产生一个多项式方程约束。对于 ACIR 表达式 exp，此操作码会导出多项式方程约束$\sum_{i=1}^{n} r_i a_i b_i + \sum_{j=1}^{m} s_j c_j + q = 0$。整数翻译相同，但针对的是整数表达式编码。

$$[\![\text{AssertZero}(exp)]\!]_f \equiv \text{(= } [\![exp]\!]_f\ 0\text{)}$$

例如，操作码 `AssertZero`$(\langle(1, w_1, w_2)\rangle, \langle\rangle, -1)$ 表示约束 $w_1 w_2 - 1 = 0$，即 $w2$ 是 w_1 的乘法逆元。这种操作码由 Noir 语句 `assert(invx * x = 1)` 生成。

`BlockBoxFuncCall` 操作码允许执行黑盒函数。这些函数涵盖了通常无法通过多项式约束高效描述的行为，例如`AssertZero` 操作码生成的约束。它们旨在为底层 ZK 证明系统提供一定的灵活性，使其能够提供自己的优化电路来描述这些函数。ACIR 支持 14 种黑盒函数，范围从范围检查到复杂的哈希函数和加密计算。本文仅对$Range$ 黑盒函数进行了形式化；其他函数的形式化将留待未来工作。对于见证 x 和整数 n，$Range(x, n)$ 函数会检查 x 的值是否可编码为 n 位无符号整数。我们使用 x 的二进制表示来实现此检查。我们使用新的域变量

$y_1, \ldots, y_n$ 来为 x 编码位和表示，即 $x = \sum_{i=1}^{n} 2^{i-1} y_i$，并通过约束 $y_i(y_i - 1) = 0$ 强制要求 y_i 表示一个二进制值。

$[\![\text{BlockBoxFuncCall}(\text{范围}(x, n))]\!]_f \equiv (= (\text{ff.bitsum } y_1 \ldots y_n) \ x) \wedge (= (\text{ff.mul } y_1 \ (\text{ff.add } y_1 \ -1)) \ 0) \ \wedge \ldots \wedge (= (\text{ff.mul } y_n \ (\text{ff.add } y_n \ -1)) \ 0)$

类似的黑盒函数，例如与 和异或，分别表示位级逻辑and 与 xor，可以使用这种二进制表示进行形式化。

与有限域编码不同，整数算术形式化可以依靠整数的内在顺序来实现这一检查。

$[\![\text{BlockBoxFuncCall}(\text{范围}(x, n))]\!]_i \equiv (< \ x \ 2^n)$

`MemoryOp` 和`MemoryInit`操作码使 ACIR 能够创建、访问和修改（新）见证数组。这些数组由内存块 *ID*标识。我们的编码还需要一种方法来跟踪数组何时被修改。我们使用 $m_{id,i,j}$ 来表示数组（即内存块） i中元素的值，该值处于经过 j 次修改之后的状态。例如，$m_{2,1,0}$ 表示数组 2 在初始化时索引 1 处的元素，而 $m_{2,1,1}$ 表示同一数组中同一元素在第一次写入后的值。我们按顺序翻译/编码内存操作，并使用映射 mi跟踪数组的变更索引。因此， mi 属于翻译上下文，并会在创建或更新数组时进行更新。为简洁起见，我们不将其加入翻译规则。

`MemoryInit`操作码会创建一个由新见证组成的数组。[3] 给定一个块标识符（无符号 32 位整数） id和见证 $x_1, \ldots, x_n$作为输入，对于每个输入见证 x_i，此操作码都会生成一个新的（全新）见证$m_{id,i,0}$ ，其值与 x_i相同。作为此操作的一部分，变更索引映射会按如下方式更新 $mi' = mi \oplus \{id \to 0\}$，其中 mi'和 mi 分别是翻译之后和之前的映射值，而 $\oplus$会向映射中添加一个映射项；如果已存在具有相同 id的映射项，则会更新该项。

$[\![\texttt{MemoryInit}(id, \langle x_1, \ldots, x_n \rangle)]\!]_f \equiv (= \ m_{id,1,0} \ x_1) \ \wedge \ldots \wedge \ (= \ m_{id,n,0} \ x_n)$

`MemoryOp` 操作码可用于从内存块中读取数据以及向内存块写入数据。其输入包括一个块标识符 id、一个内存操作 op （读取时为 0，写入时为 1）、一个索引 ACIR 表达式 ind以及一个值 ACIR 表达式val。读取操作会将索引 ind 处的值复制到 val给出的见证中。请注意，我们如何使用值 mi（id）来标识表示数组当前状态的见证，以及蕴含关系如何确定要访问的正确索引。数组的状态不会发生变化，因此 mi也不会改变。

$[\![\texttt{MemoryOp}(id, op = 0, ind, val)]\!]_f \equiv (\texttt{=>} \ (= \ [\![ind]\!]_f \ 1) \ (= \ m_{id,1,mi(id)} \ [\![val]\!]_f)) \wedge \ldots \wedge (\texttt{=>} \ (= \ [\![ind]\!]_f \ \text{n}) \ (= \ m_{id,n,mi(id)} \ [\![val]\!]_f))$

[3] ACIR 支持不同的内存块类型。这里，我们只对更通用的$Memory$ 类型进行形式化。

写入会将索引 ind 处的值变更为 val。请注意，我们使用值$mi(id)$ 和 mid $(id) + 1$ 分别标识表示数组当前状态和下一状态的见证。第一组合取式与读取非常相似，不同之处在于，它们约束数组的下一状态 $mi(id)$，并定义被写入元素的变化。第二组合取式定义数组中未被写入元素的变化；也就是说，这些元素保留原值。变异映射 mi 也会更新： $mi' = mi \oplus \{id \to mi(id) + 1\}$。

$[\![\text{MemoryOp}(id, op = 1, ind, val)]\!]_f \equiv$ (=> (= $[\![ind]\!]_f$ 1) (= m_{id},1,$_{mi(id)+1}$ $[\![val]\!]_f$))∧ . . . ∧ (=> (= $[\![ind]\!]_f$ n) (= $m_{id,n,mi(id)+1}$ $[\![val]\!]_f$))∧(=> (not (= $[\![ind]\!]_f$ 1)) (= m_{id},1, $_{mi(id)+1}$ m_{id},1,$_{mi(id)}$))∧. . . ∧ (=> (not (= $[\![ind]\!]_f$ n)) (= $m_{id,n,mi(id)+1}$ $m_{id,n,mi(id)}$)

内存操作码的模整数形式化通过使用相应的整数见证和表达式编码，映照了上述定义。

最后，`BrilligCall` 操作码用于表示无约束代码。由于我们关注的是分析和形式化受约束代码，而其执行完整性由底层零知识证明系统保障，因此我们不会转换这些操作码；更准确地说，它们会被转换为始终满足的约束`true`。

3.2 验证者

我们提出 NAVe：一种基于 SMT-LIB 语义、用于 Noir 程序的形式化验证器——其高层架构如图 2 所示。它依靠 Noir 基础设施将 Noir 程序编译为 ACIR，并使用 cvc5 [8] 求解由我们的语义生成的 SMT 约束（即 SMT-LIB 转换）。我们的验证器已在代码仓库中公开发布。[4]

我们使用图 3 中的基本 Noir 程序来演示我们的形式化验证框架的用法。Noir 程序可以通过`verify_assert(`$cond$`)`进行注释。与会创建约束的内置 Noir `assert` 不同，这个形式化验证函数会生成一项证明义务：条件 $cond$必须在该语句所在的位置成立。 $cond$是一个无副作用的谓词，它作用于程序在该位置的作用域内的变量。目前，这是一个简单的、未量化的 Noir 表达式。此验证函数必须由

[4] https://github.com/pedrotbtl/nave

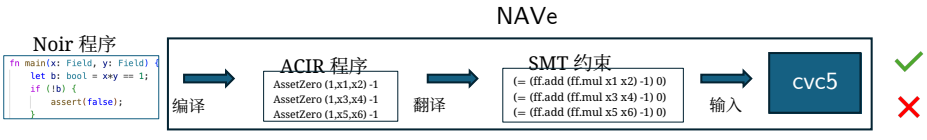


图 2. NAVe 形式化验证器概览。

```
fn main(x: 域) {assert(x * (x- 1)  == 0);unsafe {
verify_assert((x  == 0)  |(x  == 1)); }}


无约束fn 验证_assert(b: 布尔值){}
```

图 3. 基本示例程序。

NAVe 的使用者，例如基本示例程序中的使用者。这种方法最大限度地减少了对 Noir 语言基础设施的改动。我们无需添加新的语言结构（也无需定义其解析、编译等），即可跟踪验证条件。相反，我们在 ACIR 层拦截对这个特定函数的调用，并将其适当地转换为验证条件。此外，该函数完全不会改变 Noir 程序的行为。由于它是一个无约束函数，因此不会向底层算术电路添加约束，而且执行时不会产生任何效果。因此，开发者可以使用这些形式化验证结构注释程序，而不会影响相应的零知识证明生成与验证。

NAVe 作为 Nargo 包管理器的命令（`formal-verify`）实现（位于其公开分支中）。我们希望 Noir 开发者使用熟悉的工具后，会更愿意使用 NAVe。对于 Noir 程序 P，我们使用 Noir 编译器生成 ACIR 程序 A。我们使用形式化方法将 A 转换为 SMT 公式 $[\![A]\!]_f$；按照 ACIR 程序的顺序逐个转换操作码，并将每个操作码生成的约束合取起来。对于 ACIR 程序中的每个验证条件，这些验证条件都是 `BrilligCall` 操作码，用于捕获 Noir 程序中的 `verify_assert(`$cond$`)` 语句。我们还原表示$cond$的布尔 ACIR`Expression` exp_{cond}，并创建公式 $F = [\![A]\!]_f \wedge \neg [\![exp_{cond}]\!]_f$。当然，如果愿意，也可以使用模整数编码来代替有限域编码。随后，我们使用 cvc5 SMT 求解器检查这些 F 公式的可满足性。不可满足意味着验证条件成立。而可满足的公式则表示相应验证条件未通过，也就是说，在给定程序位置存在一次程序执行，使相应谓词被证伪。此时，cvc5 会生成一个模型，即对见证的满足赋值（包括表示 Noir 程序输入参数的见证）。该模型构成一个反例，详细说明了验证条件如何被证伪；我们会将其反馈给开发者，以帮助调试程序。例如，在图 3 中，我们使用 NAVe 验证程序输入 `x` 是否为 0 或 1；它会检查 `x * (x - 1) == 0` 约束是否按预期工作，而`a | b` 则表示 `a` 与`b`之间的析取。不过，如果移除验证条件中的一个析取项，NAVe 就会返回一个模型，证明`x` 也可以取该值；具体来说，如果移除`(x == 1)`，NAVe 就会给出一个反例，其中`x` $= 1$。

3.3 评估

我们在实践中评估了 Noir 程序形式化验证器。我们分析了它在若干测试程序上的性能，这些程序属于 Noir 语言基础设施的一部分。它们来自 Noir 代码仓库，但我们为其添加了验证条件。我们尝试收集复杂程度各异、且符合我们已形式化的 ACIR 子集的程序。我们创建了包含条件成立和被证伪两类测试程序的集合，以分析求解器在这两种不同搜索任务中的表现。此外，我们还创建了包含和不包含范围检查的程序集合——也就是说，包含调用函数 $Range(x, n)$ 的 `BlackBoxFuncCall`操作码。我们希望分析求解器的性能，以及由此反映出的验证器在分析这些操作码所引入的复杂位和约束时的性能。我们在一台专用机器上进行了评估，该机器配备英特尔酷睿 i7-9750H CPU @ 2.60GHz x 12 和 16GB 内存，运行 Ubuntu 22.04.5 LTS。我们使用的测试程序以及复现评估结果的说明，均可在我们的代码仓库中找到。[5]

在表1中，我们展示了在两组不含范围检查约束的程序上运行 NAVe 的结果：一组程序的验证条件已被验证，另一组程序的验证条件则被证伪。我们测试了两种编码。对于有限域编码，我们还测试了两种 cvc5 有限域求解器配置：

[5] https://github.com/pedrotbtl/nave

已验证						
基准测试			**时间**（秒）			
id	**#w**	**#o**	**sp**	**gb**	**int**	
pnv_1	1	1	17.5	20.3	17.8	
pnv_2	1	2	16.0	22.3	17.3	
pnv_3	6	6	18.9	20.2	t	
pnv_4	17	18	16.2	18.8	t	
pnv_5	1	1	17.5	17.1	19.0	
pnv_6	207	139	18.5	17.9	17.2	
pnv_7	1	1	18.2	16.8	20.3	
pnv_8	285	286	19.2	17.1	19.7	
pnv_9	4	4	18.4	16.7	t	
pnv_{10}	1	1	19.0	15.7	17.8	
pnv_{11}	1	2	20.1	17.0	17.5	
pnv_{12}	5	4	17.4	17.4	15.2	

被证伪					
基准测试			**时间**（秒）		
id	**#w**	**#o**	**sp**	**gb**	**int**
pnf_1	189	126	42.5	22.9	t
pnf_2	5	4	18.9	17.9	t
pnf_3	3	2	16.0	18.2	16.6
pnf_4	6	4	17.8	18.0	19.1
pnf_5	9	10	17.3	19.1	t
pnf_6	20	3	18.1	18.9	16.0
pnf_7	4	3	17.9	17.6	15.0
pnf_8	12	6	17.6	18.9	14.8
pnf_9	2	2	16.8	18.7	14.8
pnf_{10}	9	5	17.1	18.4	14.6
pnf_{11}	1	1	17.5	18.8	16.0
pnf_{12}	2	2	16.3	18.5	t
pnf_{13}	63	64	16.6	16.5	17.4

表1. 对于不含范围约束的 Noir 程序，分别给出验证条件已验证和被证伪的结果。列 **标识符** 标识受测程序，列 #**w** 给出程序的见证数量，列 #**o** 给出不包括 BrilligCalls 在内的 ACIR 操作码数量。列 **sp** 和 **gb** 给出采用 ACIR 有限域编码进行程序分析所需的时间，而列 **int** 给出采用整数编码所需的时间；我们用 t表示 120 秒超时。对于列 **sp**，我们使用配置为 --ff=split 的 cvc5 求解器；对于列 **gb**，我们使用 --ff=gb。

`--ff=split` 和默认配置`--ff=gb`。这两种配置采用不同的求解器来检查有限域理论公式的可满足性。简单来说，前一种配置所使用的技术（如 [30] 所述）对后一种配置所使用的技术（如 [29] 所述）进行了近似；这些近似在处理某些位和约束组合时可能非常有效。结果表明，我们的三个 NAVe 后端之间没有明显的优胜者：使用整数理论的编码，以及采用两种 cvc5 配置的有限域理论编码。例如，对于某些测试程序，整数编码速度最快；而对于另一些测试程序，整数编码是唯一会超时的后端。两种有限域求解器的表现非常相似；这符合预期，因为`--ff=split` 求解器旨在改进位和约束，而这些程序不包含范围检查，即不包含会产生这类约束的操作码。

在表 2 中，我们展示了两组包含范围检查约束的程序的结果。与前文一样，一组程序的验证条件已通过验证，另一组的条件则被证伪；我们还针对有限域编码测试了不同的编码和求解器配置。结果再次表明，我们的三个 NAVe 后端之间没有明显的优劣之分。不过，我们确实观察到一个规律：对于范围较大的范围检查——也就是说，对于位数较多的表示——

已验证							
基准测试					**时间**（秒）		
id	**#w**	**#o**	**#r**	$\mathbf{r}_{max}$	**sp**	**gb**	**int**
prv_1	12	21	11	32	17.2	16.1	18.9
prv_2	147	154	52	32	17.6	15.5	17.7
prv_3	15	13	11	8	16.9	15.6	19.6
prv_4	1	3	2	1	16.5	14.5	18.2
prv_5	5	6	3	32	16.4	14.6	18.6
prv_6	20	28	13	8	20.1	14.7	18.9
prv_7	10	14	5	32	17.8	14.8	18.7
prv_8	50	61	15	128	16.6	14.5	19.0
prv_9	10	13	1	32	18.1	14.9	t
prv_{10}	969	984	292	8	17.6	15.9	19.9
prv_{11}	74	86	26	238	19.2	15.7	18.8
prv_{12}	21	37	9	32	18.0	t	t
prv_{13}	8	6	1	1	16.4	18.5	t
prv_{14}	20	31	13	1	17.5	18.9	15.4

被证伪							
基准测试					**时间**（秒）		
id	**#w**	**#o**	**#r**	$\mathbf{r}_{max}$	**sp**	**gb**	**int**
prf_1	24	31	15	1	18.3	18.0	16.3
prf_2	3	4	3	32	18.8	19.3	16.8
prf_3	4	4	2	32	17.2	18.5	t
prf_4	7	6	1	1	17.5	17.8	17.4
prf_5	3	3	3	1	17.7	16.7	17.0
prf_6	5	7	1	1	18.0	17.6	16.0
prf_7	2	3	1	8	18.0	86.2	16.3
prf_8	3	4	1	1	16.5	17.9	16.9
prf_9	32	28	6	32	22.6	19.4	17.0
prf_{10}	9	16	7	32	57.9	t	t
prf_{11}	6	13	7	32	28.6	t	t

表 2. 分别展示了验证条件被验证和被证伪时，不含范围约束的 Noir 程序的结果。列 **标识符** 标识被测试的程序，列 $\#\mathbf{w}$ 给出程序的见证数量，列 $\#\mathbf{o}$ 给出不含 BrilligCalls 的 ACIR 操作码数量，列 $\#\mathbf{r}$给出 BlackBoxFuncCall 范围检查操作码的数量，列 $\mathbf{r}_{max}$ 给出最大范围检查所涉及的位数。列 **sp**和**gb** 给出使用 ACIR 有限域编码时的程序分析时间，而**int** 给出使用整数编码时的时间；我们用 t 表示 120 秒超时。对于列**sp**，我们使用配置为 --ff=split 的 cvc5 求解器；对于列**gb**，我们使用 --ff=gb。

```
fn main(x:域) {
    lety = 不安全{
        empty(x  + 1) };
    令z  = y + x  + 1;
    令z1 =z  + y;
    assert(z + z1 != 3);
    令w  = y + 2 * x  + 3;
    assert(w + z1 !=z);
    不安全{
    verify_assert(
        ((y  != 1/3) | (x != 0))
        & ((y   != 0) | (x != 1/2))
        & ((y   !=-1) | (x != -1/2))
        & ((y !=-1/2) | (x != -1))
    );}
}

无约束函数 空值(_: 域)
    -> 域 {0 }
```

```
fn main(x: 1位无符号整数, y: 1位无符号整数, z: 1位无符号整数)
    -> pub 1位无符号整数 {
    让p  =y - z;
    如果p 不等于= 0 {
        令a  =x / z;
        让b  =a - z;
        如果b 不为= 0 {
            let _ =a / b;
        }
    }
    令u  =x - z;
    令v  = y - u;
    不安全{
    验证_assert(
        ((u  != 0) | (x   == z))
        & ((v != 0) | (u  ==  y))
        & ((p   != 0) | (y   == x))
    );}
    v
}
```

图4. 移除了范围检查操作码的回归测试程序（prv_{13} 的摘录）和包含范围检查操作码的回归测试程序（prv_{14}），分别如下。

耗时会更长，可能超时，也可能比其他测试程序耗时显著更长。此外，我们还可以看到有限域求解器的行为有所不同：在某些情况下，`--ff=split` 求解器的表现显著优于`--ff=gb` 求解器（见prv_{12}, prf_7, prf_{10}, 和prf_{11}）。

我们在图4中展示了评估中使用的两个测试程序。它们都是针对 Noir 语言的回归测试。左侧程序是prv_{13} 的摘录——为简洁起见，我们删除了部分代码。我们添加到该程序中的`verify_assert` 语句，概括了代码中两个`assert`所限定的`x` 和`y` 的有效取值范围；`a & b` 表示`a` 与`b`的合取。`verify_assert` 中的前两个合取项由第一个`assert`强制执行，后两个则由第二个`assert`强制执行。右侧程序（prv_{14}）依赖于 1 位无符号整数`u1` 类型所施加的范围约束。我们添加到该程序中的`verify_assert`语句，概括了对变量`u`、`v`和`p`的赋值所强制执行的一些限制。

我们的结果表明，我们可以互补地使用我们的编码和针对有限域编码的不同求解器。我们可以同时对同一程序运行三个后端，并报告最先完成的那个后端的结果。此外，结果表明，包含大量范围检查的程序可能需要花费大量时间才能完成验证。这说明，为了解决这一问题，我们需要研究抽象（即近似）

。这些结果为未来的研究指明了方向：我们将研究求解器与抽象的并行组合，以应对复杂约束。

4 相关工作

本节将讨论与我们的验证框架相关的文献。

零知识编程语言和编译器。CirC [28]提供了一套基础设施，用于将程序编译为电路。程序首先会转换为一种称为 CirC-IR 的中间表示。该 IR 是一种基于 SMT-LIB 标准的抽象存在量化电路（ EQC ），支持布尔值、数值、位向量和数组。因此，CirC 与 LLVM 有些相似，但其目标是零知识证明领域专用语言。该基础设施支持将加密语言编译为 CirC-IR；不过，集成一种新语言需要实现相应的解析器和解释器。CirC 已在 Circom [3] 上进行了评估：后者可以说是最流行的（低级）算术电路领域专用语言。NAVe 与 CirC 的主要区别在于，NAVe 直接处理 ACIR，而 ACIR 已与 CirC-IR 相似，因此无需转换。此外，ACIR 支持一些复杂的高级结构，例如黑盒函数和内存操作，这些结构支撑着 Noir，而 CirC-IR 并不支持。

与 Noir 类似，Cairo [2] 是另一种流行的高级零知识编程语言。迄今为止，针对 Cairo 程序形式化验证的研究寥寥无几。我们找到的该领域唯一一篇论文 [14] 对将 Cairo 程序转换为零知识证明系统所用代数表示的转换进行了形式化，并证明了其可靠性。我们的验证者和形式化工作似乎是首个针对高级零知识编程语言及其特性开展的研究。例如，在 Noir 中，我们有以黑盒函数形式呈现的复杂构造，以及内存操作的概念。

静态分析与缺陷检测。现有的大多数相关工作都集中于 Circom 及其他电路领域专用语言。一些研究声称其方法可以推广到 Circom 之外，但其评估仅限于 Circom 电路。Noir 还会将其 Rust 风格的领域专用语言编译为 ACIR，从而与证明器无关。ZEQUAL [33] 是一个使用静态分析和演绎验证来验证 ZK 电路的框架。它构建于 Circom 编译器之上，并通过采用有限域约束的模整数算术编码，使用 Z3 执行 SMT 查询。该框架能够分析不需要完整实例化的电路模板。然而，与 NAVe 的关键区别在于，ZEQUAL 同样在领域专用语言层面运行，并且仅在 Circom 电路上进行了评估。ZKAP [34] 也是一个用于检测 ZK 电路中漏洞的静态分析框架。ZKAP 的方法以对 Circom 程序中现有漏洞的人工研究为指导。它使用电路依赖图来捕获关键的电路属性，并将语义漏洞模式表示为类 Datalog 查询。他们实现了九种不同的检测器，每种检测器都用于识别一种特定的反模式。QED^2 [31] 同样使用 SMT 求解器进行自动化验证，但重点关注特定的电路属性（准确地说，是约束不足缺陷），而非完整的功能正确性。

尽管 QED^2 原则上能够分析有限域上的多项式方程，但其评估仅限于 Circom 电路。

Noir 程序的验证。Lampe [6] 是一个旨在支持对 Noir 语言的语义以及用 Noir 编写的程序属性进行形式化验证的项目。它在 Lean [27]编程语言和定理证明器中对 Noir 的语义进行建模，重点在于捕获 Noir 的语义，而不是提供验证框架。由于使用了定理证明器，Lampe 用户必须迭代地推动属性证明，而 NAVe 则使用 SMT 求解自动检查这些属性。

5 结论

受零知识程序日益流行以及高级零知识编程语言不断涌现的推动，我们提出了 NAVe：一个用于形式化验证 Noir 程序的框架。我们通过 ACIR（Noir 程序的中间表示）提出了 Noir 的一种形式化方法。我们提出了两种基于 SMT-LIB 的 ACIR 程序形式化方法：一种使用非标准有限域理论，另一种使用标准整数理论。我们的形式化方法处理了 Noir/ACIR 语言的一些特殊性，而这些特殊性在其他语言中并不存在。例如，ACIR 具有多个行为复杂的黑盒函数。我们已开始处理范围检查黑盒函数，但计划将编码扩展到其他黑盒函数。ACIR 的内存操作是另一项在其他传统语言中不存在的特性。此外，我们还必须处理零知识范式的特殊性，例如受约束代码与无约束代码之间的区别。

我们利用形式化成果构建了 NAVe 形式化验证器。它以 `formal-verify` 命令的形式实现于 Nargo 的一个分支版本中；Nargo 是 Noir 广受欢迎的包管理器。我们希望这一集成能鼓励 Noir 开发者使用我们的工具。我们依赖 Noir 编译器生成程序的 ACIR 表示，随后使用我们的形式化方法将其编码为 SMT-LIB，并由 cvc5 检查其可满足性。我们工具的主要目标是识别约束不足的程序，这是零知识程序的一种常见问题。这类程序缺少某些约束，可能是由于无约束代码的使用不受限制所致。NAVe 用户可以使用 `verify_assert(·)`函数指定验证条件。对于被证伪的验证条件，我们的工具还会向开发者报告反例，帮助他们在此过程中调试代码。我们在 Noir 语言基础设施中的一些测试程序上评估了 NAVe，结果令人鼓舞。这些测试也表明，在某些情况下，需要更好的抽象来处理复杂约束，例如范围检查生成的位求和约束。

作为未来的工作，我们计划扩展形式化验证器，以处理 ACIR 的更大子集。我们将特别研究如何形式化其他类型的黑盒函数；众所周知，这些函数很难编码。此外，我们计划研究新的抽象，以处理复杂的 ACIR 约束/程序。

参考文献

1. Aztec 网络。
https://aztec.network/blog/introducing-aztec-nr-aztecs-private-smart-contract-framework。访问日期：2025-11-24。 2. Cairo 语言。
https://www.cairo-lang.org。访问日期：2025-11-21。 3. Circom 语言。
https://docs.circom.io/。访问日期：2025-11-21。 4. cvc5 有限域理论参考文档。
https://cvc5.github.io/docs/cvc5- 1.3.1/theories/finite_field.html。访问日期：2025-11-21。 5. Grumpkin 曲线。https://aztecprotocol.github.io/aztec-connect/primitives.html#2-grumpkin—a-curve-on-top-of-bn-254-for-snark-efficient-group-operations。访问日期：2025-11-28。 6. Lampe 代码仓库。
https://github.com/reilabs/lampe。访问日期：2025-11-21。 7. Lean 以太坊共识。
https://leanroadmap.org/。访问日期：2025-11-24。 8. Noir 语言。
https://noir-lang.org。访问日期：2025-11-21。 9. Oxparc 零知识漏洞跟踪器。
https://noir-lang.org。访问日期：2025-11-21。 10. Polygon 技术。
https://polygon.technology/。访问日期：2025-11-24。 11. Rust 语言。
https://rust-lang.org/。访问日期：2025-11-28。 12. Starknet。
https://www.starknet.io/。访问日期：2025-11-24。 13. Tornado Cash 解说。
https://tornado-cash.medium.com/tornado-cash-got- hacked-by-us-b1e012a3c9a8。访问日期：2025-11-28。 14. Jeremy Avigad、Lior Goldberg、David Levit、Yoav Seginer 和 Alon Titelman。《Cairo 程序执行的经过验证的代数表示》。载于第*11*届 *ACM SIGPLAN* 国际经认证的程序与证明会议论文集，CPP 2022，第153–165页，纽约，美国，2022年。美国计算机协会。 15. Haniel Barbosa、Clark W. Barrett、Martin Brain、Gereon Kremer、Hanna Lach- nitt、Makai Mann、Abdalrhman Mohamed、Mudathir Mohamed、Aina Niemetz、Andres Nötzli、Alex Ozdemir、Mathias Preiner、Andrew Reynolds、Ying Sheng、Cesare Tinelli 和 Yoni Zohar。《cvc5：一款功能多样、达到工业级强度的 SMT 求解器》。载于 Dana Fisman 和 Grigore Rosu 编，*系统构造与分析的工具和算法——第28届国际会议，TACAS 2022，作为欧洲软件理论与实践联合会议 ETAPS 2022 的一部分举办，2022年4月2日至7日，德国慕尼黑，会议论文集，第一部分*，《计算机科学讲义》第13243卷，第415–442页。Springer，2022年。 16. Clark Barrett、Pascal Fontaine 和 Cesare Tinelli。《理论可满足性库（SMT-LIB）》。www.SMT-LIB.org，2016年。 17. Manuel Blum、Paul Feldman 和 Silvio Micali。《非交互式零知识及其应用》。载于第二十届 *ACM* 计算理论年度研讨会论文集，STOC ’88，第103–112页，纽约，美国，1988年。美国计算机协会。 18. Stefanos Chaliasos、Jens Ernstberger、David Theodore、David Wong、Mohammad Jahanara 和 Benjamin Livshits。《综述：我们还不知道什么? 理解 SNARK 中的安全漏洞》。载于 Davide Balzarotti 和 Wenyuan Xu 编，第*33*届 *USENIX 安全研讨会，USENIX Security 2024，2024年8月14日至16日*，美国宾夕法尼亚州费城。USENIX 协会，2024年。 19. Rosario Gennaro、Craig Gentry、Bryan Parno 和 Mariana Raykova。《二次跨度程序与无需 PCP 的简洁非交互式零知识证明》。*IACR Cryptol. ePrint Arch.*，第215页，2012年。

20. Oded Goldreich、Silvio Micali 和 Avi Wigderson。证明除了其有效性之外不会泄露任何信息，或 NP 中的所有语言都具有零知识证明系统。载于 第*27*届 *ACM* 计算理论研讨会年度会议论文集（*STOC*），第174–187页。ACM，1991年。21. Shafi Goldwasser、Silvio Micali 和 Charles Rackoff。交互式证明系统的知识复杂度（扩展摘要）。载于 Robert Sedgewick 编，第*17*届 *ACM* 计算理论研讨会年度会议论文集，*1985*年*5*月*6*日至*8*日，美国罗德岛州普罗维登斯，第291–304页。ACM，1985年。22. Shafi Goldwasser、Silvio Micali 和 Charles Rackoff。交互式证明系统的知识复杂度。*SIAM* 计算学杂志，18(1)：186–208，1989年。23. Jens Groth。简短的基于配对的非交互式零知识论证。载于 Masayuki Abe 编，密码学进展——*ASIACRYPT 2010*——第*16*届密码学与信息安全理论及应用国际会议，新加坡，*2010*年*12*月*5*日至*9*日。会议论文集，第6477卷，《计算机科学讲义》，第321–340页。Springer，2010年。24. Jens Groth。论基于配对的非交互式论证的大小。第305–326页，柏林、海德堡，2016年。Springer-Verlag。25. Donald E. Knuth。《计算机程序设计艺术》第二卷：半数值算法。Addison-Wesley，马萨诸塞州雷丁市，第3版，1997年。26. Mary Maller、Sean Bowe、Markulf Kohlweiss 和 Sarah Meiklejohn。Sonic：基于线性规模、通用且可更新的结构化参考字符串的零知识 SNARK。CCS '19，第2111–2128页，美国纽约州纽约市，2019年。美国计算机协会。27. Leonardo de Moura 和 Sebastian Ullrich。Lean 4 定理证明器与编程语言。载于 自动推理——*CADE 28*：第*28*届自动推理国际会议，线上会议，*2021*年*7*月*12*日至*15*日，会议论文集，第625–635页，柏林、海德堡，2021年。Springer-Verlag。28. Alex Ozdemir、Fraser Brown 和 Riad S. Wahby。CirC：面向证明系统、软件验证等的编译器基础设施。载于 第*43*届 *IEEE* 安全与隐私研讨会，*SP 2022*，美国加利福尼亚州旧金山，*2022*年*5*月*22*日至*26*日，第2248–2266页。IEEE，2022年。29. Alex Ozdemir、Gereon Kremer、Cesare Tinelli 和 Clark W. Barrett。有限域模可满足性。载于 Constantin Enea 和 Akash Lal 编，计算机辅助验证——第*35*届国际会议，*CAV 2023*，法国巴黎，*2023*年*7*月*17*日至*22*日，会议论文集，第二部分，第13965卷，《计算机科学讲义》，第163–186页。Springer，2023年。30. Alex Ozdemir、Shankara Pailoor、Alp Bassa、Kostas Ferles、Clark W. Barrett 和 Isil Dillig。有限域模可满足性的分裂 Gröbner 基。载于 Arie Gurfinkel 和 Vijay Ganesh 编，计算机辅助验证——第*36*届国际会议，*CAV 2024*，加拿大魁北克省蒙特利尔，*2024*年*7*月*24*日至*27*日，会议论文集，第一部分，第14681卷，《计算机科学讲义》，第3–25页。Springer，2024年。31. Shankara Pailoor、Yanju Chen、Franklyn Wang、Clara Rodríguez-Núñez、Jacob Van Geffen、Jason Morton、Michael Chu、Brian Gu、Yu Feng 和 Isil Dillig。自动检测零知识证明中的欠约束电路。*Proc. ACM Pro- gram. Lang.*，7(PLDI)：1510–1532，2023年。32. Bryan Parno、Jon Howell、Craig Gentry 和 Mariana Raykova。Pinocchio：近乎实用的可验证计算。*Commun. ACM*，59(2)：103–112，2016年1月。33. Jon Stephens、Shankara Pailoor 和 Isil Dillig。零知识证明电路中一致性的自动化验证。载于 Ruzica Piskac 和 Zvonimir Rakamaric 编，计算机辅助验证——第*37*届国际会议，*CAV 2025*，

克罗地亚萨格勒布，*2025*年*7*月*23*日至*25*日，会议论文集，第一部分，第15931卷，《计算机科学讲义》，第315–338页。Springer，2025年。34. Hongbo Wen、Jon Stephens、Yanju Chen、Kostas Ferles、Shankara Pailoor、Kyle Charbonnet、Isil Dillig 和 Yu Feng。零知识证明电路的实用安全分析。载于 Davide Balzarotti 和 Wenyuan Xu 编，第*33*届 *USENIX* 安全研讨会，*USENIX Security 2024*，美国宾夕法尼亚州费城，*2024*年*8*月*14*日至*16*日。USENIX 协会，2024年。