Bithoven：面向高表达能力比特币智能合约的形式化安全

Hyunhum Cho, Ik Rae Jeong

arXiv:2601.01436v1 [cs.CR] 4 Jan 2026

摘要—比特币 UTXO 架构严格的安全模型往往以牺牲开发者可用性为代价，迫使开发者依赖手动栈操作，从而导致签名可塑性、不可花费状态和不受约束的执行路径等严重金融漏洞。Miniscript 等行业标准为策略验证提供了必要的抽象，但无法对复杂合约所需的完整命令式逻辑进行建模，因而在状态管理和资源活性方面留下空白。本文介绍了 Bithoven，这是一种旨在弥合表达能力与形式化安全之间差距的高级语言。通过将严格的类型检查器和资源活性分析器与语义控制流分析器相结合，Bithoven 能够在部署前消除我们在故障模型中定义的主要共识缺陷和逻辑缺陷。我们的结果表明，这种安全性只需付出适度代价：Bithoven 编译生成的比特币脚本，其效率可与手工优化的代码相媲美，证明了即使在比特币区块链严格的字节大小限制下，类型安全且对开发者友好的抽象仍然可行。

索引词—比特币、智能合约、安全性、形式化方法。

I. 引言

SMART 合约——在区块链上执行的程序——重新定义了去中心化应用。然而，这一范式仍存在鲜明分歧。一方面，以太坊所代表的基于账户的模型提供了图灵完备的表达能力，但代价是灾难性漏洞、重入攻击和编译器缺陷 [1], [2], [3]。这催生了复杂的事后安全工具生态 [4]。

另一方面，比特币 [5]，开创的未花费交易输出（UTXO）模型优先考虑安全性与并行性。其原生智能合约语言比特币脚本是一种简单、非图灵完备、基于栈的语言。这种刻意的限制最大限度地缩小了攻击面，但简单性也有利有弊。直接编写比特币脚本向来十分困难、容易出错，也难以分析 [6]。对比特币区块链的实证分析表明，这种复杂性会直接导致可证实的资金损失，并发现了数千个存在严重漏洞的有缺陷脚本，例如“未绑定交易标识符”（任何人都可花费）或“无用签名”（混乱逻辑） [7]。

这在设计空间中留下了一个重大空白：开发者如何才能在比特币上构建复杂且具备高保证的合约，同时避免继承图灵完备性的危险或原始脚本中晦涩难懂的陷阱?

研究界提出了若干解决方案。BitML 等形式化语言偏重理论，往往不切实际，因为它们需要复杂谓词或共识变更 [8],[9]。Simplicity 等低级语言旨在完全取代虚拟机，但并非为面向人类可读代码的开发而设计[10]。当前最先进的方案 Miniscript，在可分析性和组合性方面迈出了意义非凡的一步 [11], [12]。然而，Miniscript 主要是一种高度可分析的 汇编语言 和编译器目标，而非以开发者为中心的命令式源语言。

稿件已收到；（通讯作者：Ik Rae Jeong。）Hyunhum Cho、Ik Rae Jeong 任职于韩国首尔 02841 高丽大学网络安全学院（电子邮箱：hyunhum@korea.ac.kr；irjeong@korea.ac.kr）。

我们主张采取务实的折中方案——优先保障安全性与可验证性，同时不牺牲开发者的生产效率。开发者需要一种高级、富有表现力的语言，既直观易写，又能编译为形式上安全且可验证的比特币脚本。

本文介绍 Bithoven：一种面向比特币的新型形式化规范智能合约语言。Bithoven 为常见的密码学操作和时间锁定操作提供高级、易读的抽象，其风格类似于人们熟悉的命令式语言。我们的核心贡献不仅是这门语言，还包括其正确性由构造保证的静态分析器。通过集成严格的类型系统 [13],、活跃性与作用域分析器、递归安全检查器以及语义控制流分析器，Bithoven 编译器可提供先验安全保证，在部署前消除整类漏洞。

我们的贡献如下：

- 我们提出了一种专为原生比特币脚本设计的、具有形式化规范、类型安全的高级命令式语言，并完整定义了其语法、操作语义和类型系统（第三节和第五节）。
- 我们介绍了 Bithoven 多阶段静态分析流水线的设计与实现。该流水线具有独特能力，不仅能检测类型系统和共识层面的缺陷，还能发现 Miniscript 无法阻止的复杂活性错误（例如签名重用）（第六节A）。
- 我们评估了 Bithoven 的安全性保证，证明它能够在编译时防止文献中记录的多种比特币脚本漏洞（例如 [7] 中所有相关类别）（第六节B）。
- 我们提供了一个采用成熟库以 Rust 实现的开源编译器和 Web 集成开发环境[14],[15], [16],，并与 Miniscript 进行了比较分析。我们证明，Bithoven 在保持与优化策略语言的结构同构的同时，能够更好地表达复杂逻辑，且链上开销较低（第六节E）。

II. 相关工作

我们将 Bithoven 置于更广泛的研究领域中。本节围绕三个关键领域展开：(1) 基础区块链编程范式，(2) 比特币语言与编译器的发展现状，

以及 (3) 比特币的替代应用与可扩展性解决方案。

A. 区块链编程范式

智能合约语言的设计从根本上受到其底层账本模型的制约。

基于账户的模型： 以太坊模型最具代表性，它将合约视为全局状态中的有状态对象 [17]。这种图灵完备模型具有很强的表达能力，但其复杂性是安全漏洞的主要来源。由此催生了一个丰富的事后验证研究领域，包括字节码分析 [2], 编译器错误检测 [1], 形式语义 [4], 以及符号执行框架 [3]。

基于 UTXO 的模型： 比特币引入的 UTXO 模型 [5], 是一种无状态、可并行的模型，交易会消耗现有输出并创建新输出[13]。这种设计天然具有更强的可扩展性，也更易于分析，但在有状态计算和数据存储方面面临挑战[18]。比特币交易结构本身的复杂性促使人们开展各种形式化建模研究，以实现稳健的分析 [8], [19]。Bithoven 被设计为一种原生 UTXO 语言，它接受这一模型的约束，以提供更强的安全性。BitVM2 和 Taproot Covenants 等近期提案 [20],[21] 拓展了 UTXO 模型的表达能力边界，也凸显了 Bithoven 这类高层次安全设计工作的紧迫性。

B. 比特币语言与编译器

比特币智能合约面临的首要挑战，是弥合开发者生态与低层级且易出错的比特币脚本[11]之间的鸿沟。实证分析表明，脚本层面的复杂性会直接导致资金损失，并增加审计难度 [7]。事实上，比特币开发者社区自身也认识到原始脚本存在重大风险，并警告其复杂性可能导致显而易见的资金损失[6]。这推动了多种高级解决方案的开发，其中最突出的是Miniscript策略语言 [11],[12]。

比特币脚本分析： 对更完善工具的需求源自对比特币区块链的分析；这些分析发现了数千个存在缺陷的脚本，其中包含“未绑定交易标识符”或“无用签名”等漏洞 [7]。这些缺陷已导致显而易见的资金损失。Bithoven的静态分析器旨在从构造上防止这些特定的已知缺陷类别。此外，Bithoven的分析器还超越了这些语义缺陷，提供全面的安全性模型，防止若干根本性的错误类别，包括类型系统错误（例如对字符串进行算术运算）、共识规则违规（例如格式错误或不在曲线上的公钥）以及控制流缺陷（例如非终止执行路径）。

形式语言与演算： 大量学术研究探讨了比特币合约的形式化方法。例如，BitML 是一种用于描述合约逻辑的进程演算，侧重于交易层面的符号化计算模型 [8]。其他研究则探讨了将有状态合约逻辑安全地编译到 UTXO 模型中，通常使用一系列交易来管理状态 [9]。尽管这些研究奠定了坚实的理论基础，但它们抽象掉了脚本层面的具体细节，因此难以适应快速演进的比特币协议（例如 Taproot 升级），甚至需要扩展共识机制，从而降低了其即时实用性。Bithoven 借鉴了这些理论模型的洞见，同时务实地聚焦于脚本本身，在确保安全性的同时提供更强的表达能力和适应性。其适应性体现在目标编译指示（用于 Legacy、隔离见证或 Taproot 编译）等功能，以及可直接映射到比特币原生操作码的语义模型。

实用语言提案： 已有多种语言被提出，以提升比特币智能合约的安全性或开发者体验。

- **Simplicity** [10] 是一种低级、形式化定义的函数式语言，旨在完全取代比特币脚本。其主要目标是形式化可证明性与验证，但其基于组合子的设计并不适用于直观、人类可读的合约开发。Bithoven 的务实设计与 Simplicity 形成对比。Bithoven 并不取代比特币中对共识至关重要的虚拟机，而是提供可编译为当前已部署且经过优化的比特币脚本的高层抽象。这使得 Bithoven 能够在无需共识变更或比特币脚本扩展的情况下提供先验安全保证。

- **Miniscript** [11], [12] 定义了比特币脚本中一个结构化、可组合的策略子集。它为自动化脚本分析与组合性提供了强有力的支持，但最好将其理解为一种高度可分析的汇编语言和编译器目标，而非面向开发者的源语言。Miniscript 已获得显著采用，包括被集成到 Bitcoin Core[22] 中。其规范被标准化为比特币改进提案（BIP）379 [12],，作为一种建立在现有共识规则之上的信息性标准运作。

- **描述符** [23]: 输出脚本描述符是一种规范语言，钱包使用它来明确描述如何派生 scriptPubKey 并定位相关的未花费交易输出。描述符的主要优势在于提供了统一的抽象层，将钱包的高级花费策略与具体的、带版本的脚本实现（例如 P2SH、P2WPKH 或 P2TR）分离开来。例如，描述符可以包含 Miniscript 策略，将复杂的最终脚本抽象为一个易于人类阅读的字符串。Bithoven 的 pragma target 指令直接受到这一设计的启发，提供了类似的编译器级抽象，确保能够适应未来的协议升级。

- **Ivy** [24], **Clarity** [25], 和 **sCrypt** [26] 为比特币或 UTXO 系统带来了更高层次的可编程性和不同的编程模型。然而，与 Bithoven 不同，这些语言并未提供形式化、构造即正确的静态分析，无法保证在部署前消除脚本级缺陷。

C. 比特币上的应用与可扩展性

尽管比特币脚本存在复杂性和局限性，仍有大量研究探索了扩展比特币功能的方法。

链上应用：研究人员早已证明，比特币上可以实现复杂应用，包括彩票 [27] 和密码学承诺方案[28]。这些应用通常需要复杂、手工编写的脚本——而这正是 Bithoven 旨在通过结构化抽象消除的脆弱逻辑。

扩展与链下逻辑：最著名的扩展方案闪电网络，将大部分交易移至链下 [29]。其他系统，如 FastKitten，则利用可信执行环境 (TEEs) 在链下运行复杂合约，仅将比特币用作最终结算层 [30]。

图灵完备性：BitVM[31], 等近期研究提出了复杂的多交易协议，以在比特币上实现准图灵完备计算。

Bithoven 找到了务实的平衡：它既不追求准图灵完备性，也不依赖外部硬件或链下层。相反，Bithoven 专注于充分发挥 实用链上合约的优势——例如金库、契约、多重签名方案和哈希时间锁合约——使其安全、灵活且易于编写。这种方法直接针对比特币脚本中已有记录的缺陷 [6],[7]。

III. 系统与故障模型

我们首先定义系统模型，该模型遵循标准的比特币 UTXO 架构，然后介绍威胁与故障模型。后者对 Bithoven 旨在防止的由开发者引入的缺陷进行分类。

A. 系统模型

我们的系统模型采用比特币 [5] 使用的标准未花费交易输出（UTXO）模型。在该模型中，值由输出脚本（也称为 scriptPubKey）锁定。要花费 UTXO，用户必须广播一笔新交易，为每个被花费的输出提供有效的输入脚本 （或 scriptSig）。

每笔交易输入的验证都通过执行比特币原生的智能合约语言——比特币脚本 [6] 来完成。虚拟机（VM）是一个简单的、非图灵完备的基于栈的解释器。对于每个输入，VM 将所提供的输入脚本与相应的输出脚本串联起来，并执行组合后的脚本。

执行有两种可能的结果：

- **成功：** 脚本执行完毕，且栈顶的最终值为真（或任意非零值）。该交易被视为有效。
- **失败：** 脚本执行完毕，但栈上的最终值为假（或零）；或者脚本在执行过程中中止（例如，OP_VERIFY 验证失败或遇到无效操作）。该交易将被拒绝。

我们的模型兼容所有标准的比特币脚本系统，包括 Legacy（P2SH）、隔离见证（P2WSH）和 Taproot（P2TR）。由于 Bithoven 编译器的最终输出是原生比特币脚本，它天然保留了比特币共识行为，不会改变底层虚拟机语义。

B. 威胁与故障模型

我们假设开发者是善意但可能犯错的，其目标是编写安全的输出脚本——只有当支付方提供预期的秘密（例如有效签名或原像）时，脚本才会评估为**成功**。

此处考虑的故障不包括网络层或共识层攻击（例如 51% 控制权、双重支付、交易审查）。相反，我们关注开发者在比特币脚本层引入的逻辑和语义缺陷。

对手模型： 我们假设存在一个标准网络攻击者，能够观察内存池和区块链上的所有交易。攻击者可以构造任意输入脚本，试图未经授权地花费资金，但无法破解标准密码学原语（例如椭圆曲线数字签名算法、SHA-256）[7]。攻击者唯一的优势来自利用开发者编写的有缺陷的输出脚本。

根据 [7], 中的分类，我们区分两种主要的缺陷结果：(1) 攻击者可花费缺陷，即脚本无意中允许未经授权的花费；以及 (2) 永不可花费缺陷，即脚本变得无法赎回，即使是合法所有者也无法花费。我们将该分类法扩展至类型系统、控制流、栈纪律和共识规则违规，并在第六节中对它们进行类似评估（见表 II）。

1) 攻击者可花费缺陷：这些脚本在语法上有效，但包含会导致未经授权花费的语义缺陷。

- **语义安全缺陷：**脚本包含一条无需所需密码学证明即可求值为真的逻辑路径。这对应于表 II 中的语义安全失效。一个代表性示例是未绑定交易 ID 缺陷 [7],：其中一条分支缺少 checksig 操作，从而造成“任何人都可花费”的漏洞。类似地，当脚本针对支付方提供的任意公钥验证签名，而不是针对所有者的固定密钥进行验证时，就会出现不确定签名缺陷[7]。

- **密码学操作的误用：**脚本误用了密码学原语，例如反转签名检查的结果（如 OP_CHECKSIG OP_NOT）。这种缺陷称为 [7],无用签名，允许攻击者提供一个无效签名，使其仍被判定为真，从而花费资金。

2）永不可花费缺陷：这些脚本会永久锁定资金，即使合法所有者也无法取回。

- **共识规则违规：**开发者引入了违反比特币共识规则的格式错误常量，例如超出 32 位限制的数字字面量或无效的公钥编码。这些问题对应于 [7]中的共识规则缺陷类别。

- **类型系统与活性错误：**脚本对不兼容的类型执行操作（例如，对字符串进行算术运算），或违反栈纪律（例如，重复使用某个变量）。Miniscript 可以防止许多此类问题[11]，，但其语法不会跟踪有状态的资源消耗，因此仍容易出现活性错误，例如在嵌套逻辑中重用已消耗的变量。

- **不可满足逻辑：**该脚本包含在逻辑上不可能满足的控制流（例如，最后执行时出现 OP_NOT），或者某个分支缺少返回路径。这些情况对应于永不为真缺陷，以及表 II 中的“控制流”错误 [7]和。

Bithoven 的类型系统、栈跟踪器和语义分析器共同确保在编译时预防该模型中的所有缺陷类别，如表 II 所总结。

IV. BITHOVEN 语法与语义

本节提供 Bithoven 语言的完整形式化规范。这些形式体系为正确构造编译器和静态分析器提供了精确蓝图。我们不仅给出规则，还阐释每种形式体系所提供的直观含义和安全性保证。

A. Bithoven 语法

我们使用以下记法： $P \in$ **程序**， $p \in$**编译指示**， $\Sigma \in$ **栈声明**， $s \in$ **语句**， $e \in$**表达式**， $f \in$ **签名因子**， $x \in$ **标识符**， $\tau \in$ **类型**，$v \in$ **版本**， $t \in$ **目标**， $n \in \mathbb{Z}$， $b \in \{$**真**，**假**$\}$，以及$str \in$**字符串**。

Bithoven 的抽象语法采用层级结构，从顶层程序定义开始，逐步细化语句、表达式以及密码学操作的专用语法形式，如图 1 所示。

程序 (P**)** 是完整的编译单元。它以必需的**编译指示 (**p**)**开头，这些编译器指令用于指定语言版本或编译目标（例如 Legacy、隔离见证或 Taproot）等必要元数据。这些信息对于生成针对特定目标优化的比特币脚本至关重要。编译指示之后，必需的 **栈声明 (**Σ**)**定义脚本在开始执行时预期在栈中找到的带类型变量（$x : \tau$）。此声明构成合约的公共接口，也是静态类型检查和活性分析的基础。为了提供直观、类似函数签名的接口，Bithoven 的栈声明按逆序指定，随后由编译器映射到比特币的后进先出栈。程序主体是一个代码块，其中包含一系列实现合约逻辑的语句（s^*）。

语句（s）表示操作和控制流。该语言包含用于分支逻辑的标准 if/else 条件语句，以及用于运行时断言的 verify 语句；若条件为假，它会停止执行并使交易失败。Bithoven 通过两个专用语句直接支持比特币的时间锁定功能：用于相对时间锁的 older（CheckSequenceVerify）和用于绝对时间锁的 after（CheckLockTimeVerify），其设计灵感来自 miniscript[11]。

表 I BITHOVEN 语法与比特币操作码的映射

类别	Bithoven 语法	比特币操作码
数学	`+`	OP ADD_
	`-`	OP_SUB
	`++`	OP_1ADD
	`--`	OP_1SUB
	`max`	OP 最大值_
	`min`	OP_MIN
	取反	OP_NEGATE
	`abs`	OP 绝对值_
逻辑与比较	`&&`	OP BOOL与_
	`||`	OP BOOL或_
	`==`	OP_(数值)EQUAL
	`!=`	OP_(数值)NOTEQUAL
	`<`	OP LESSTHAN_
	`>`	OP GREATERTHAN_
	`<=`	OP_小于或等于
	`>=`	OP_大于或等于
	`!`	OP 非_
密码学	SHA-256	OP SHA256_
	RIPEMD-160	OP RIPEMD160_
字节操作	`len`	OP SIZE_

最后，return 语句是一种终止操作，它会对表达式求值，以定义脚本的最终成功状态，从而有效地结束程序执行路径。通过规定只有 return 语句才能生成脚本的最终输出，其他语句（如 verify 或 older）均不会在栈上留下中间值。这种设计大幅简化了栈管理，并消除了原始比特币脚本中常见的一整类栈操作错误。

表达式（e）是可计算出一个值并将其压入栈中的构造。它们包括字面量（整数、布尔值、字符串）以及在输入栈声明中定义的变量。该语言支持一组用于常见操作的二元（$\oplus$）和一元（$\ominus$）运算符，这些运算符会直接映射到比特币操作码，如表 I 所示。

比特币授权逻辑的核心封装在 checksig(f) 表达式中。

为清晰处理签名检查中不同的参数结构，我们引入了**签名因子**（f）。因子是一种语法结构，用于组合简单的单签名检查（一个签名和一个公钥对）或复杂的 m-of-n 多重签名检查所需的参数。这种抽象使 checksig 能够作为不同密码学验证方案的统一接口，从而提升可读性和可维护性。

B. Bithoven 的语义

我们使用小步操作语义定义 Bithoven 程序的含义。这种形式化方法描述抽象机的状态如何仅基于语言结构发生演变，而不涉及任何特定的交易上下文。

语义域： 我们的抽象机状态由以下组件定义：

- **值**（v）：运行时值的集合，包括整数、布尔值、字符串、签名和公钥。 $v \in$值。

程序（P）： Bithoven 程序是顶层构造，由编译指示、输入栈声明和脚本组成。

$$P \quad ::= \quad p^{+}\Sigma^{+}\{s^{*}\}$$

编译指示（p）编译指示用于向编译器声明元数据，例如语言版本和编译目标 .

p ::= **编译指示版本** v (语言版本)| **目标编译指示** t (编译目标，例如 Taproot)

栈声明（Σ）： 定义执行开始时栈中预期作为输入的带类型变量。

$$\Sigma \quad ::= \quad \textbf{Stack}(x_1 : \tau_1, \ldots, x_n : \tau_n)$$

签名因子（f）：

f	::=	(e_{sig}, e_{pk})	(Single Signature Pair)
	|	$[m, (e_{sig_1}, e_{pk_1}), \ldots, (e_{sig_n}, e_{pk_n})]$	(Multi-Signature Structure)

表达式（e）： Bithoven 中的表达式会求值得到一个值，并将其推入栈中。

e	::=	$n \mid b \mid str \mid x$	$n \in \mathbb{Z}$, $b \in \{\textbf{true}, \textbf{false}\}$, $str \in$ **String**, $x \in$ **Identifier**
	|	$e_1 \oplus_{math} e_2$	$\oplus_{math} \in$ {**+, -, max, min**}
	|	$e_1 \oplus_{compare} e_2$	$\oplus_{compare} \in$ {**==, !=,>, >=, <, <=**}
	|	$e_1 \oplus_{logical} e_2$	$\oplus_{logical} \in$ {**&&, ||**}
	|	$\ominus_{math}\, e$	$\ominus_{math} \in$ {**negate, abs, ++, --**}
	|	$\ominus_{logical}\, e$	$\ominus_{logical} \in$ {**!**}
	|	$\ominus_{crypto}\, e$	$\ominus_{crypto} \in$ {**sha256, ripemd160**}
	|	$\ominus_{byte}\, e$	$\ominus_{byte} \in$ {**len**}
	|	**checksig**(f)	(Signature Check)

语句 (s)： 语句 会执行操作并控制执行流程，但不一定会产生值 e.

s	::=	**if** e $\{s^{*}\}$ **else** $\{s^{*}\}$	(条件结构)
	|	**验证** e	(断言)
	|	**older** n	(相对时间锁)
	|	**after** n	(绝对时间锁)
	|	**返回** e	(终止表达式)

图 1. Bithoven 的抽象语法

- **栈（σ）**：由一系列值组成，$\sigma \in$值*。将值 v 压入栈 σ 记作 $v ::$ σ。
- **运行时环境（ρ）**：从标识符映射到其运行时值的映射，ρ：标识符 $\rightarrow$ 值。此环境在执行开始时，根据栈声明中的类型化参数进行设置。
- **配置**：一个对 $\langle K, \sigma \rangle$，其中 K 是待执行的代码（表达式或语句），σ 是当前栈。
- **终止状态**：执行可能以最终栈状态σ_f 成功终止，也可能进入一种特殊的失败状态，记作 $\perp$。

表达式的操作语义：表达式的求值由以下判断定义：$\rho \vdash \langle e, \sigma \rangle \rightarrow \sigma'$。这表示："在运行时环境 ρ 中，对带有栈 σ 的表达式 e 求值后，得到新栈 σ'。"

字面量与变量：

$$\frac{}{\rho \vdash \langle v, \sigma \rangle \rightarrow v :: \sigma} \qquad \frac{\rho(x) = v}{\rho \vdash \langle x, \sigma \rangle \rightarrow v :: \sigma}$$

一元运算和二元运算：

$$\frac{\rho \vdash \langle e, \sigma \rangle \rightarrow v' :: \sigma' \quad v = (\ominus, v')}{\rho \vdash \langle \ominus e, \sigma \rangle \rightarrow v :: \sigma'}$$

$$\frac{\rho \vdash \langle e_1, \sigma \rangle \rightarrow v_1 :: \sigma_1 \quad \rho \vdash \langle e_2, \sigma_1 \rangle \rightarrow v_2 :: \sigma_2 \quad v = (\oplus, v_1, v_2)}{\rho \vdash \langle e_1 \oplus e_2, \sigma \rangle \rightarrow v :: \sigma_2}$$

签名检查操作： checksig 表达式会对其参数求值，并将一个布尔值压入栈中。由于实际的密码学验证取决于上下文，因此不在这些语义的讨论范围内 [32],[33], ，我们通过非确定性地产生真或假来对其进行建模。

$$\frac{\rho \vdash \langle f, \sigma \rangle \rightarrow \langle v_{args}, \sigma' \rangle \quad b \in \{\textbf{true}, \textbf{false}\}}{\rho \vdash \langle \textbf{checksig}(f), \sigma \rangle \rightarrow b :: \sigma'}$$

语句的操作语义：语句的执行由以下判断定义：$\rho \vdash \langle s, \sigma \rangle \to \sigma'$ 或 $\perp$。其含义是："在环境 ρ中，执行语句 s时，栈 σ 会转换为一个新栈 σ'，或者执行失败。"

条件语句与验证语句：

$$\frac{\rho \vdash \langle e, \sigma \rangle \to \textbf{true} :: \sigma' \quad \rho \vdash \langle S_{if}, \sigma' \rangle \to \sigma''}{\rho \vdash \langle \textbf{if } e \ \{S_{if}\} \textbf{ else } \{S_{else}\}, \sigma \rangle \to \sigma''}$$

$$\frac{\rho \vdash \langle e, \sigma \rangle \to v :: \sigma'}{\rho \vdash \langle \textbf{verify } e, \sigma \rangle \to \begin{cases} \sigma' & \text{if } v = \textbf{true} \\ \perp & \text{if } v = \textbf{false} \end{cases}}$$

为简洁起见，此处省略了 If 语句中假分支的规则，但它与真分支的规则对称。

锁定时间语句：与 checksig 类似，older 和 after 锁定时间语句都是断言，其结果取决于外部交易上下文[34], [35]。我们将其行为建模为在成功（栈保持不变）和失败（终止执行）之间进行非确定性选择。

$$\frac{op_{lock} \in \{\textbf{older}, \textbf{after}\}}{\rho \vdash \langle op_{lock} \ n, \sigma \rangle \to \sigma \text{ or } \perp}$$

Return 语句： 此语句是终止语句。它会对表达式求值，并用表达式求值后得到的栈替换当前栈。

$$\frac{\rho \vdash \langle e, \sigma \rangle \to \sigma'}{\rho \vdash \langle \textbf{return } e, \sigma \rangle \to \sigma'}$$

V. BITHOVEN 类型系统

形式化类型系统是一项至关重要的贡献，因为原生比特币脚本并没有类型系统。在比特币基于栈的虚拟机中，每个项都只是一个字节数组。值的类型仅由解释它的操作码决定（例如，OP ADD 将字节数组视为数字，而 OP VERIFY 将其视为布尔值）[13]。这种模糊性是开发者错误的主要来源，可能导致诸如对字符串执行算术运算之类的缺陷。此类类型不匹配可能导致脚本执行失败，造成"永远无法花费"的缺陷，永久锁定资金。Bithoven 的类型系统旨在于编译时消除此类漏洞，提供原始脚本无法实现的先验安全保证。

Bithoven 中的类型集合记作 τ，其来源于语法中可用的基本类型。我们还为公钥引入了一个独立的类型，它在语义上不同于一般的字符串或数字。同样，我们将 sig 类型引入为一种符号类型，仅用于栈声明；这对于语义分析器强制执行高级安全不变量至关重要，例如防止出现"未绑定交易标识符"的花费路径。

$$\tau \quad ::= \quad \textbf{num} \mid \textbf{bool} \mid \textbf{string} \mid \textbf{sig} \mid \textbf{pubkey}$$

A. 类型环境

类型环境 Γ 将变量标识符映射到其类型。它由 Bithoven 程序的输入栈声明中提供的显式类型注解构成。

$$\Gamma \quad ::= \quad \emptyset \mid \Gamma, x : \tau$$

记号 $\Gamma(x)$ 表示环境 Γ 中变量 x 的类型。

B. 类型判断式

我们定义两种形式的类型判断式：

1) $\Gamma \vdash e : \tau$ 表示在环境 Γ 中，表达式 e 类型正确，且类型为 τ。2) $\Gamma \vdash s \Rightarrow$ **正确** 表示在环境 Γ 中，语句 s 类型正确。

C. 表达式和因子的类型规则

图 2 展示了定义 Bithoven 静态类型系统的形式化类型规则。这些规则是 Bithoven 编译时安全性的核心，确保每个表达式在执行前都符合类型要求。它们施加了原始比特币脚本中不存在的严格约束，例如将数学运算符限制为数值类型，将逻辑运算符限制为布尔类型。该系统还包含专门规则，用于验证密码学原语的结构，确保 checksig 始终与格式正确的签名和公钥对一起使用。

D. 语句的类型规则

If 语句： 条件必须是布尔值，且两个分支都必须符合类型要求。

$$\text{(T-If)} \quad \frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash B_{if} \Rightarrow \textbf{ok} \quad \Gamma \vdash B_{else} \Rightarrow \textbf{ok}}{\Gamma \vdash \textbf{if } e \textbf{ then } B_{if} \textbf{ else } B_{else} \Rightarrow \textbf{ok}}$$

Verify 语句： 待验证的表达式必须是布尔值。

$$\text{(T-Verify)} \quad \frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash \textbf{verify } e \Rightarrow \textbf{ok}}$$

锁定时间语句： 为了用一条规则形式化 older（CSV）和 after（CLTV）语句的类型规则，我们引入记号 op_{lock}，它可以表示任一运算符。Bithoven 静态分析的一项关键功能是防止共识层缺陷。因此，这些语句的类型规则不仅必须验证运算，还必须强制满足比特币共识约束，即数值字面量 n 是有效的 32 位无符号整数（$0 \le n < 2^{32}$）[34], [35]，具体定义如下规则：

$$\frac{op_{lock} \in \{\textbf{older}, \textbf{after}\} \quad n \ge 0 \quad n < 2^{32}}{\Gamma \vdash op_{lock} \ n \Rightarrow \textbf{ok}} \quad \text{(T-Locktime)}$$

Return 语句： 内部表达式必须具有正确的类型。

$$\text{(T-Return)} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{return } e \Rightarrow \textbf{ok}}$$

VI. 实现与求值

为验证我们的方法，我们为 Bithoven 语言实现了一个完整的编译器和静态分析器。本节介绍系统的实现细节，并评估其在防止故障模型（第 III-B 节）中定义的缺陷类别方面的有效性。

$$\text{(T-数值)}\ \frac{}{\Gamma \vdash n\text{: }\textbf{数值}} \qquad \text{(T-布尔值)}\ \frac{}{\Gamma \vdash b\text{: }\textbf{布尔值}} \qquad \text{(T-Str)}\ \frac{}{\Gamma \vdash str\text{: }\textbf{字符串}}$$

$$\text{(T-Var)}\ \frac{\Gamma\ (x)\ = \tau}{\Gamma \vdash x\text{: }\ \tau}$$

$$\text{(T-BINARY MATH)}\ \frac{\Gamma \vdash e_1\text{: }\ \tau \quad \Gamma \vdash e_2\text{: }\ \tau \quad \tau \in \{\textbf{数值}\}}{\Gamma \vdash e_1 \oplus_{math} e_2\text{: }\ \textbf{数值}} \qquad \text{(逻辑规则)}\ \frac{\Gamma \vdash e_1\text{: }\ \tau \quad \Gamma \vdash e_2\text{: }\ \tau \quad \tau \in \{\textbf{布尔值}\}}{\Gamma \vdash e_1 \oplus_{logical} e_2\text{: }\ \textbf{布尔值}}$$

$$\text{(比较规则)}\ \frac{\Gamma \vdash e_1\text{: }\tau \quad \Gamma \vdash e_2\text{: }\tau \quad \tau \in \{\textbf{数值},\ \textbf{字符串},\ \textbf{布尔值}\}}{\Gamma \vdash e_1 \oplus_{compare} e_2\text{: }\ \textbf{布尔值}}$$

$$\text{(一元数学规则)}\ \frac{\Gamma \vdash e\text{: }\tau \quad \tau\text{:}\textbf{数值}}{\Gamma \vdash \ominus_{math} e\text{ : }\textbf{数值}} \qquad \text{(T-一元字节)}\ \frac{\Gamma \vdash e\text{: }\tau \quad \tau\text{: }\textbf{字符串}}{\Gamma \vdash \ominus_{byte} e\text{: }\ \textbf{数值}}$$

$$\text{(T-一元加密)}\ \frac{\Gamma \vdash e\text{: }\ \tau \quad \tau \in \{\textbf{数值},\ \textbf{字符串},\ \textbf{布尔值},\ \textbf{签名},\ \textbf{公钥}\}}{\Gamma \vdash \ominus_{crypto} e\text{: }\ \textbf{字符串}}$$

$$\text{(T-单因子)}\ \frac{\Gamma \vdash e_{sig}\text{ :}\textbf{签名} \quad \Gamma \vdash e_{pk}\text{ : }\textbf{公钥}}{\Gamma \vdash (e_{sig}, e_{pk})\ \Rightarrow \textbf{因子}} \qquad \text{(T-检查签名)}\ \frac{\Gamma \vdash f\ \Rightarrow \textbf{因子}}{\Gamma \vdash \textbf{checksig}(f)\text{ : }\textbf{布尔值}}$$

$$\text{(T-多因子)}\ \frac{\Gamma \vdash m\text{: }\textbf{数值} \quad \forall i \in \{1..n\},\ (\Gamma \vdash e_{s_i}\text{ : }\textbf{签名} \quad \Gamma \vdash e_{p_i}\text{ : }\textbf{公钥})}{\Gamma \vdash [m, (e_{s_1},\ \ e_{p^1}) \ldots, (e_{s_n}, e\ \ _{p_n})] \Rightarrow \textbf{因子}}$$

图 2. 表达式和因子的类型规则

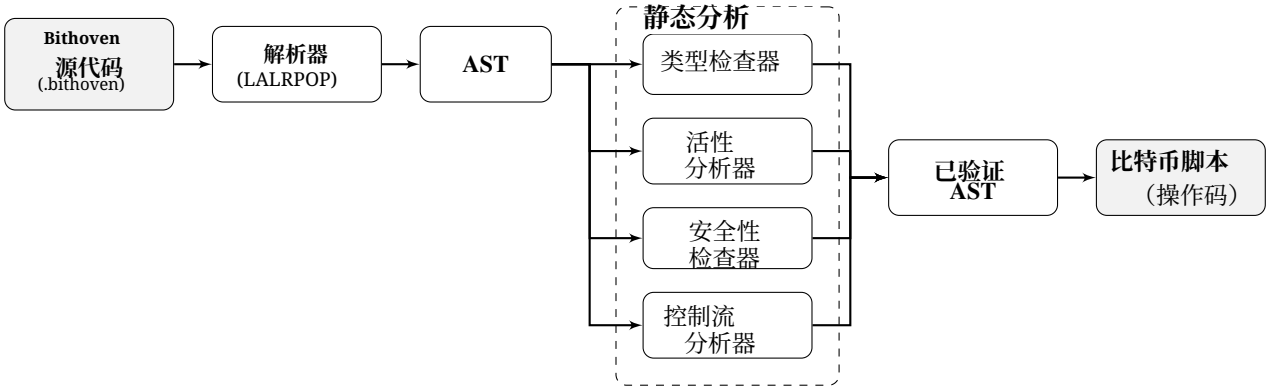

图 3。Bithoven 编译流水线。源代码经过解析生成抽象语法树，随后进行包括类型检查、资源活跃性分析、控制流分析和安全验证在内的多阶段静态分析。只有经过完全验证的抽象语法树才会传递给操作码生成器。

A. 实现

Bithoven 编译器使用 Rust 和 LALRPOP 解析器生成器实现[15],，后者可直接根据形式文法生成经过形式化规范的解析器。清单 1 展示了该文法的一个节选，其中包含定义 Bithoven 抽象语法树（AST）的核心语句和表达式规则。用于输出比特币脚本的编译后端采用 rust-bitcoin 实现[16],，这是比特币生态系统中使用最广泛的软件之一。

解析器输出类型化抽象语法树，随后将其传递给由四个核心组件组成的静态分析流水线，如图 3 所示：

1) 类型检查器：强制执行第 III 节定义的类型系统，确保所有操作的类型正确（例如，不得对字符串进行算术运算），并确保所有密码学原语格式正确（例如，验证 secp256k1 曲线上的公钥）。

2) 活跃性与作用域分析器：执行活性分析，以强制实施变量作用域和资源使用约束，确保资源（例如签名）在任何有效执行路径上都恰好被消耗一次。

3)安全检查器：对抽象语法树执行无状态的自顶向下遍历，以检测局部反模式漏洞。这包括共识规则违规，例如整数溢出和篡改操作（例如无用签名漏洞）。

4）语义与控制流分析器：遍历

```
...
公开 语句: 语句 ={
    <条件语句>      ,
    <锁定时间语句>          ,
    <验证语句>,
    <表达式语句>,
}

条件语句: 语句 = {
    <l:@L>"如果" <c:表达式0> <b1:BlockStatement>
    "否则" <b2:BlockStatement> <r:@R> =>     ...
};

锁定时间语句: 语句 =
  <l:@L> <op:锁定时间操作符> <operand:无符号整数>
  <r:@R> <s:分号> => ...
验证语句: 语句 =
  <l:@L> "验证" <e:表达式0> <r:@R> <s:
      分号> =>        ...
表达式语句: 语句 =
  <l:@L> "返回" <e:表达式0> <r:@R> <s:
      分号> =>...

检查签名表达式: 表达式 = {
    <l:@L> <op:检查签名操作符> <operand: 单一签名因子
      > <r:@R> => ...
    <l:@L> <op:检查签名操作符> <operand: 多重签名因子>
        <r:@R> => ...
}

单一签名因子: 因子 ={
    <l:@L> <o: 左括号> <sig:表达式4> <逗号:
      逗号>
    <公钥:表达式4> <c: 右括号> <r:@R> =>
        ...
}
...
```

清单 1. BithovenLALRPOP 解析器语法

程序的控制流图（CFG），以强制执行复杂的全局不变量，这些不变量无法在局部发现。这包括验证每条执行路径都要求签名（从而防止未绑定交易标识符缺陷），以及查找控制流缺陷，例如缺少返回路径的代码分支。

验证成功后，经过验证的抽象语法树会传递给代码生成器，由其生成经过优化、针对特定目标的比特币脚本（例如，多重签名隔离见证使用 OP_CHECKMULTISIG，而 Taproot 使用 OP_CHECKSIGADD）。

B. 求值：静态漏洞检测

这项求值为 Bithoven 的安全保证提供了核心实证依据。我们通过两个方面的比较，展示 Bithoven 的静态分析提供了更优越的安全模型：（1）与最先进的事后检测工具 BSHunter 相比；（2）与最先进的受限策略语言 Miniscript 相比。表 II 总结了这项分析，展示了相较于这两种替代方案，Bithoven 如何在编译时预防更多种类的漏洞。

表 II 中的分析得出了两项关键见解。首先，分析表明，Bithoven 明显优于 BSHunter 这类基于检测的工具；后者只能发现现有缺陷，而 Bithoven 从设计上就能防止缺陷。其次，分析凸显了 Miniscript 的根

```
编译指示 Bithoven 版本 0.0.1;
编译指示 Bithoven 目标 隔离见证;

(条件: 布尔值)
{
    返回 条件 == 真;
}
```

清单 2。未经身份验证的“任何人都可花费”路径。

本局限，而 Bithoven 正是为克服这些局限而专门设计的：

1) 状态与活性： Miniscript 的语法是无状态且专注于组合性的，这是其主要优势。然而，这意味着它无法跟踪资源消耗。正如表格所示，Miniscript 无法识别活性错误，例如重复使用已消耗的签名。Bithoven 新颖的活性分析器是 有状态的，通过强制执行严格的单次使用约束，在编译时消除整类基于状态的错误。

2) 保障表达能力： Miniscript 的不适用条目并非缺陷，而是其有意采用限制性设计的体现。Bithoven 的核心贡献在于，它安全地引入了这些强大且富有表达力的命令式结构——例如通用算术和 if/else 控制流。Bithoven 的静态分析器正是保障这种新增表达能力安全的关键，它能捕获这些强大功能可能引入的类型安全和控制流缺陷。

C. 缺陷预防示例 *n*

现在，我们通过故障模型中的代表性漏洞来演示 Bithoven 的静态分析器。

1) 语义安全：最严重的缺陷是存在一条缺少签名认证的“任何人都可花费”路径。这对应于[7]中指出的未绑定交易 ID 缺陷。在清单 2 中，开发者错误地让合约是否成功仅取决于一个布尔标志。

错误： 第 4 行第 2 列错误：NoSigRequired("栈至少需要一个签名，但： `[StackParamloc:Location start: 64, end: 79,line: 4, column: 2,identifier:Identifier(condition), ty: Boolean ]`。")

Bithoven 编译器拒绝这段代码，因为其语义安全分析器检测到一条有效的执行路径（条件 == 真），该路径没有消耗任何签名类型的输入。

2) 状态与活性：Bithoven 相较于 Miniscript 的一项重要贡献是其活性分析。变量，尤其是签名，是一种线性资源，必须且只能消耗一次。在代码清单 3 中，开发者错误地在二次检查中重复使用了 sig_alice 变量。

错误： 第 9 行第 21 列出错：VariableConsumed("已消耗的变量：sig_alice。")

编译器正确地将其报告为错误，因为活跃性分析器会追踪到，sig_alice 已被第一个 verify 语句消耗。

表 II 静态漏洞检测比较分析。

漏洞类别	示例/场景	BSHunter	Miniscript	Bithoven	Bithoven 的关键特性
共识规则	整数字面量超出 32 位限制。	×	✓	✓	编译时字面量验证
	格式错误的公钥常量。	✓	✓	✓	密码学类型检查
类型系统安全	对字符串进行算术运算。	×	*N/A**	✓	严格类型系统
	比较数字与字符串。	×	*N/A**	✓	严格类型系统
活性与作用域	引用未定义变量。	×	✓	✓	静态活跃性与作用域分析
	在变量被消耗后使用它。	×	×	✓	静态活跃性分析
语义安全	某条代码路径可以返回真，但没有一个未绑定交易标识符签名。	✓	✓	✓	语义安全分析器
	checksig 的结果被忽略或被篡改为无用签名）。	✓	*N/A**	✓	语义安全分析器
	checksig 使用由以下对象提供的公钥支付方（不确定签名）。	✓	✓	✓	语义安全分析器
控制流	Return 语句之后仍有代码。	×	*N/A**	✓	控制流图分析
	某个代码分支没有返回路径。	×	*N/A**	✓	控制流图分析

*不适用（不适 用）：*Miniscript* 的策略语言不支持某些功能（例如一般算术运算、命令式控制流）。

```
编译指示 Bithoven 版本 0.0.1;
编译指示 Bithoven 目标 隔离见证;

(签名_alice: 签名)
{
    验证 checksig(签名_alice, <公钥_alice>);

    // 错误：签名_alice 被重复使用
    返回 checksig(签名_alice, <公钥_bob>);
}
```

清单 3. 签名变量被重复使用。

```
编译指示 bithoven 版本 0.0.1;
编译指示 bithoven 目标 隔离见证;

(签名_alice: 签名)
{
    // 错误：语法有效的 33 字节格式，
    但该点不在 secp256k1 曲线上。
    返回 checksig(签名_alice, "0345
    a6b3f8eeab8e88501a9a25391318dc
    e9bf35e24c377ee82799543606bf5211");
}
```

图 4. 格式错误的公钥字面量。

3）共识规则：Bithoven 的类型检查器会根据比特币共识规则验证所有字面量，从而防止出现无效密钥缺陷 [7]。在清单 4 中，开发者提供了一个不是有效压缩公钥的字符串。

```
错误： 第 7 行第 32 列错误：类型不匹配("公钥格式错误：¨
0
345a6b3f8eeab8e88501a9a25391318dce9bf35e24c377ee82799543606bf
5211¨。"))
```

编译器会拒绝此合约，并检测出该公钥违反了比特币共识所要求的 secp256k1 曲线数学约束。这些示例共同表明，Bithoven 提供先验安全保证，可在部署前消除主要类别的漏洞。

D. 比较分析：语义与链上成本

为评估 Bithoven 高级抽象的成本，我们将其命令式语法与 Miniscript 策略片段进行比较。表 III 全面列出了语义原语、密码学检查和控制流结构的对应关系，并对编译开销进行了定量分析。

1) 结构同构与零逻辑开销：表 III 中“逻辑开销”一栏表明，绝大多数 Bithoven 构造不会产生链上开销。高级控制流结构（例如 if/else 代码块和逻辑运算符（&&、 ||））会编译为与 Miniscript 的优化组合器（例如 andor、or_i）在功能上等价的操作码序列。同样，复杂的多重签名和阈值检查可直接映射到标准 OP_CHECKMULTISIG 或 Tapscript 等价结构。这表明，对于大部分智能合约逻辑而言，Bithoven 的命令式语法是一种零成本抽象，在不引入“虚拟机税”的情况下保留底层虚拟机的效率。

2）安全性与效率权衡：对比凸显了 Bithoven 引入少量开销的两个具体方面。这些都是经过审慎考量的设计选择，体现了该语言优先保障形式化安全，而非追求原始字节效率：

- **显式资源管理与栈复制：**标准比特币脚本经常依赖 OP_DUP 来复用栈项（例如，复制公钥以进行 pk_h 检查）。相比之下，Bithoven 严格的活性分析 要求每个资源都必须恰好使用一次。因此，在逻辑上需要两次使用同一个值的操作——例如，对密钥进行哈希后再验证其签名——需要提供不同的见证输入（例如 k 和 k’）。虽然这会使见证大小增加一个栈项，但它消除了整类栈操作漏洞，并确保了引用透明性。

•栈卫生：Bithoven 中的时间锁语句（older、after）会编译为显式的 OP_丢弃。与可能在栈上留下验证结果的策略片段不同，

表 III 全面对比：语义、Miniscript 与 Bithoven

语义	Miniscript 片段	Bithoven 表达式	逻辑开销*
原语			
假	0	假	None
True	1	true	None
关键检查			
检查（密钥）	pk_k(密钥)	签名检查（密钥）	None
检查(key 哈希)**	pk_h(key)	验证 ripemd160(sha256(k))==<H>; 返回 checksig(s,k');	+1 见证项
时间锁***			
nSequence ≥ n	较早时间（n）	older n	+1 操作码（丢弃）
nLockTime ≥ n	after(n)	after n	+1 操作码（丢弃）
哈希锁**			
SHA-256	SHA-256(h)	验证 len(h)==32; 返回 SHA-256(h')==<H>;	+1见证项 +1 操作码（丢弃）
HASH256	哈希256(h)	验证 len(h)==32; 返回 sha256(sha256(h'))==<H>;	+1见证项 +1 操作码 (丢弃)
RIPEMD-160	RIPEMD-160(h)	验证 len(h)==20; 返回 RIPEMD-160(h')==<H>;	+1见证项 +1 操作码（丢弃）
HASH160	hash160(h)	验证 len(h)==20; 返回 RIPEMD-160(SHA-256(h'))==<H>;	+1见证项 +1 操作码（丢弃）
组合器：与			
条件结构	andor(X,Y,Z)	如果(X){Y}否则{Z}	None
验证序列	与_v(X,Y)	验证 X；Y	None
布尔值	与_b(X,Y)	X && Y	None
包装器	与_n(X,Y)	如果(X){Y}否则{返回 假;}	None
组合器：或			
布尔值	或_b(X,Z)	X || Z	None
控制	或_c(X,Z)	如果 !X {Z}	None
不满足	或_d(X,Z)	None	N/A
If-Else	或_i(X,Z)	如果 (Var) {X} 否则 {Z}	None
多重签名与阈值			
代数	thresh(k,X..)	(X_1 + X_2 + ..) == k	None
多重签名****	multi(k,密钥..)	checksig(k, [密钥..])	None
Tapscript****	multi_a(k,密钥..)	checksig(k, [密钥..])	None
包装器			
备用栈*****	a:X	编译器优化。	None
交换*****	s:X	编译器优化。	None
checksig	c:X	checksig(X)	None
True	t:X	X；返回 1；	None
Dup	d:X	None	N/A
验证	v:X	验证 X	None
包装器	j:X	如果 (len(Var) != 0) {X};	+1 操作码（丢弃）
非零	n:X	X != 0	None
可能	l:X	if {返回 0;} else {X}	None
不太可能	u:X	if (Var) {X} else {返回 0;}	None

* 逻辑开销指抽象本身所需的功能性操作码。它不包括栈管理指令（例如 OP_SWAP、OP_TOALTSTACK），这些指令会根据变量深度自动生成，以确保操作数对齐和类型安全。** Bithoven 的线性资源系统要求为重复使用的变量显式提供见证输入（例如“h”和“h”），因此与“OP DUP”相比会增加见证开销。*** Bithoven 的时间锁会额外输出一个“OP DROP”（1 个虚拟字节），以确保栈卫生。**** 多重签名的编译输出因目标（隔离见证或 Taproot）而异。***** 栈包装器由编译器自动处理；生成的脚本大小与手动优化的 Miniscript 相当。

Bithoven 的命令式语句会强制执行“栈干净”这一不变量。这可确保栈状态在连续语句之间保持可预测，从而降低静态分析的复杂性。

总而言之，存在的开销主要局限于见证数据——在隔离见证和 Taproot 交易权重计算中，这部分数据会被折扣计算——并用于强制执行形式化安全保证，以防止各类缺陷，这些缺陷已在第六节B中讨论。

E. 求值：表达能力与可用性

为评估 Bithoven 的表达能力和可用性，我们以标准哈希时间锁合约（HTLC）为案例研究，对比其在原始比特币脚本（清单5）、Miniscript（清单6）和 Bithoven（清单7）中的实现。

表 IV 语言范式与表达能力的比较分析

功能 / 属性	比特币脚本	Miniscript	Bithoven
语言范式	基于栈的虚拟机	可组合策略语言	命令式高级语言
主要设计目标	虚拟机执行	静态可分析性	开发者易用性与安全性
目标用户	虚拟机 / 专家	编译器 / 分析器	人类开发者
静态类型系统	None	基于子集的类型系统	显式（数值、字符串、签名）
显式状态（变量）	无（仅栈）	None	是（通过栈声明）
通用控制流（if/else）	有限（例如，OP_IF）	无（基于策略）	原生支持
算术与比较	原生（不安全）	不支持（仅策略）	原生（类型安全）
任意表达式嵌套	不支持	可组合片段	原生（例如数学、逻辑）
活性／状态分析	N/A	不支持	原生（在静态分析器中）
特定目标编译	N/A	N/A	原生（目标编译指示）

```
OP IF_OP PUSHBYTES N  <锁定时间>  OP CSV OP DROP
_          _          _          _OP_PUSHBYTES_33 <
pk_alice>  OP_CHECKSIG OP ELSE_OP HASH256 OP TOALTSTACK
OP PUSHBYTES 32_          _          _          _<
H>  OP_FROMALTSTACK OP_SWAP OP_EQUALVERIFY OP_
PUSHBYTES_33 <pk_bob>OP_CHECKSIG OP ENDIF_
```

清单5. 在比特币脚本中编译的哈希时间锁合约

```
或（与（pk(pk_bob),    SHA-256(H)),
与(pk(pk_alice),        older(<锁定时间>)))
```

清单6。Miniscript策略语言中的哈希时间锁合约

```
编译指示 Bithoven 版本 0.0.1;
编译指示 Bithoven 目标 隔离见证;

(条件：  布尔值，签名_alice：  签名)
(条件：  布尔值，原像：字符串，签名_bob:
     签名)
{
     如果 条件 {
          older   <锁定时间>;
          返回 checksig (签名_alice,        <pk_alice>);
     } 否则 {
          验证 SHA-256 SHA-256 原像  == <H>;
          返回 checksig (签名_bob,   <密钥_bob>);
     }
}
```

清单7：HTLC Bithoven。

原始比特币脚本实现是一串不透明且容易出错的栈操作操作码，准确展现了导致开发者引入缺陷的高层复杂性。Miniscript策略则是一次巨大的改进，为合约的 策略提供了可读、可分析且可组合的抽象。

Bithoven 弥合了这一差距，兼具高层策略的清晰性与命令式程序的强大功能和直观结构。哈希时间锁合约的两条花费路径并非定义为扁平的单一策略，而是使用熟悉的 if/else 控制流块进行建模。至关重要的是，Bithoven 的语言级设计提供了其他系统所不具备的一项功能：能够正式声明每条不同的花费路径，并为其分别设置独立的类型化输入栈。如清单7所示，开发者可以将“退款”路径（接收 sig_alice）和“赎回”路径（接收原像和 sig_bob）视为两种不同的、自文档化且静态可验证的函数签名。

此案例研究展示了 Bithoven 能够以开发者易于理解的方式表达复杂的多路径逻辑，这一优势总结于表 IV。Bithoven 添加了命令式控制流和状态感知活性分析等关键功能，而这些功能有意不纳入 Miniscript 以策略为先的设计范围。

VII. 结论

智能合约设计领域长期以来都由一个鲜明的权衡所定义：以太坊以灾难性漏洞为代价换取高层次的表达能力 [1], [2], ，而比特币则具备稳健的安全性，但这种安全性被锁在“出了名地困难”且容易出错的比特币脚本 [6], [7]之后。这为一种兼具开发者友好性与形式化安全性的务实解决方案留下了空白。

本文介绍了 Bithoven，这是一种面向比特币的新型、高级命令式语言，并对其进行了形式化规范，填补了这一空白。我们证明，Bithoven 的核心贡献不仅在于其直观的语法，还在于其正确构造的静态分析流水线。该多阶段分析器集成了严格类型系统、新颖的活性分析器和语义控制流检查器，可提供先验安全保证。

我们的评估得出了三个关键结果。首先，我们证明，Bithoven 的静态分析器可在编译时阻止多类已记录的、会导致资金损失的漏洞，包括 Miniscript 未能处理的缺陷，例如基于状态的活性错误（如重复消耗签名）。其次，如表 III 所示，我们消除了人们对高级抽象会造成高昂链上成本的担忧。我们的分析证明，对于绝大多数操作，Bithoven 在结构上与优化后的 Miniscript 同构，且零逻辑开销。对于少量额外开销，这是为确保栈卫生和资源安全而有意作出的权衡。最后，我们在表 IV 中的比较突显了 Bithoven 更强的表达能力：它原生支持算术运算和命令式控制流，而基于策略的语言则不具备这些能力。

这项工作的核心贡献之一，是 Bithoven 能够在编译器层面吸收逻辑与基于状态的复杂性

。通过从构造上管理这种复杂性，Bithoven 证明了无需为了表达能力而牺牲安全性或效率。它弥合了安全性与可用性之间长期存在的鸿沟，为比特币智能合约的未来开发提供了一条安全、实用且有形式化依据的路径。

致谢

作者谨此说明，曾使用谷歌的 Gemini[36], 大语言模型协助润色稿件的语法结构。作者已审阅并核实本文中的所有内容，包括代码和最终文本，并对其负全部责任。

参考文献

[1] H. Ma、W. Zhang、Q. Shen、Y. Tian、J. Chen 和 S. C. Cheung，《深入理解 Solidity 编译器中的漏洞》，载于第 *33* 届 *ACM SIGSOFT* 软件测试与分析国际研讨会（*ISSTA*）论文集，2024年，第 1312–1324 页。[2] J. Chen、X. Xia、D. Lo、J. Grundy、X. Luo 和 T. Chen，《DefectChecker：通过分析 EVM 字节码自动检测智能合约缺陷》，*IEEE* 软件工程汇刊，第 48 卷，第 7 期，第 2189–2207 页，2021年。[3] Dxo、M. Soos、Z. Paraskevopoulou、M. Lundfall 和 M. Brockman，《Hevm：面向 EVM 字节码的快速符号执行框架》，载于计算机辅助验证国际会议（*CAV*），计算机科学讲义，第 14681 卷。Springer Nature Switzerland，2024年，第 453–465 页。[4] F. Cassez、J. Fuller、M. K. Ghale、D. J. Pearce 和 H. M. Quiles，《以 Dafny 描述以太坊虚拟机的形式化语义与可执行语义》，载于形式化方法国际研讨会（*FM*），计算机科学讲义，第 14000 卷。Springer International Publishing，2023年，第 571–583 页。[5] S. Nakamoto，《比特币：一种点对点电子现金系统》，白皮书，2008年，访问日期：2025年10月27日。 [在线]。网址：https: // bitcoin.org/bitcoin.pdf [6] Bitcoin Core 开发者，《交易 - 比特币开发者指南》，网站，访问日期：2025年10月27日。 [在线]。网址：https: // developer.bitcoin.org/devguide/transactions.html [7] P. Zheng、X. Luo 和 Z. Zheng，《BSHUNTER：检测并追踪比特币脚本缺陷》，载于*2023 IEEE/ACM* 第 *45* 届软件工程国际会议（*ICSE*）。IEEE，2023年，第 307–318 页。[8] M. Bartoletti 和 R. Zunino，《BitML：一种比特币智能合约演算》，载于*2018* 年 *ACM SIGSAC* 计算机与通信安全会议（*CCS*）论文集，2018年，第 83–100 页。[9] M. Bartoletti、R. Marchesin 和 R. Zunino，《在贫乏的 UTXO 区块链上安全编译丰富的智能合约》，载于*2024 IEEE* 第 *9* 届欧洲安全与隐私研讨会（*Euro&S&P*）。IEEE，2024年，第 235–267 页。[10] R. O'Connor，《Simplicity：一种面向区块链的新语言》，载于*2017* 年安全编程语言与分析研讨会（*PLAS*）论文集，2017年，第 107–120 页。[11] P. Wuille、A. Poelstra 和 S. Kanjalkar，《Miniscript》，网站，2019年，访问日期：2025年10月27日。 [在线]。网址：https://bitcoin.sipa.be/ miniscript/ [12] P. Wuille、A. Poelstra、S. Kanjalkar、A. Poinsot 和 A. Chow，《BIP 379：Miniscript》，https://github.com/bitcoin/bips/blob/master/bip-0379. mediawiki，2023年10月，比特币改进提案。[13] 比特币维基，《脚本 — 比特币维基》，https://en.bitcoin.it/wiki/Script，访问日期：2025年11月11日。[14] Bithoven，《Bithoven：一种形式化规范的比特币智能合约语言》，https://github.com/ChrisCho-H/bithoven，2025年，访问日期：2025-11-14。[15]LALRPOP，《Lalrpop》，https://github.com/lalrpop/lalrpop，访问日期：2025年11月9日。[16] rust-bitcoin，《rust-bitcoin》，https://github.com/rust-bitcoin/rust-bitcoin，访问日期：2025年11月9日。[17] L. Br¨unjes 和 M. J. Gabbay，《UTxO 与账户模型智能合约区块链编程范式》，载于形式化方法应用国际研讨会（*ISoLA*），计算机科学讲义，第 12478 卷。Springer International Publishing，2020年，第 73–88 页。[18] S. Jiang、J. Li、S. Gong、J. Yan、G. Yan、Y. Sun 和 X. Li，《BZIP：一种面向基于 UTXO 的区块链的紧凑型数据内存系统》，《系统架构杂志》，第 109 卷，文章编号 101809，2020年。[19] N. Atzei、M. Bartoletti、S. Lande 和 R. Zunino，《比特币交易的形式化模型》，载于国际金融密码学与数据安全会议（*FC*），计算机科学讲义，第 10957 卷。Springer Berlin Heidelberg，2018年，第 541–560 页。[20] R. Linus、L. Aumayr、A. Zamyatin、A. Pelosi、Z. Avarikioti 和 M. Maffei，《BitVM2：将比特币连接到二层网络》，2024年。[21] E. Heilman、V. I. Kolobov、A. M. Levy 和 A. Poelstra，《Colliderscript：通过 160 位哈希碰撞在比特币中实现契约》，密码学电子印本档案，2024年。[22] 比特币，《比特币》，https://github.com/bitcoin/bitcoin，2009年，访问日期：2025年11月9日。[23] P. Wuille 和 A. Chow，《BIP 380：输出脚本描述符通用操作》，https://github.com/bitcoin/bips/blob/master/bip-0380. mediawiki，2021年6月，比特币改进提案。[24] D. Robinson 和 Chain.com，《Ivy for bitcoin：一种面向智能合约的高级语言》，https://docs.ivylang.org，2017年，访问日期：2025年10月27日。[25] Clarity，《Clarity 智能合约语言》，https://clarity-lang.org，2020年，访问日期：2025年10月27日。[26] sCrypt，《scrypt：一种面向比特币的高级智能合约语言》，https: //docs.scrypt.io，2019年，访问日期：2025年10月27日。[27] M. Bartoletti 和 R. Zunino，《比特币上的固定存款多方彩票》，载于国际金融密码学与数据安全会议（*FC*），计算机科学讲义，第 10323 卷。Springer International Publishing，2017年，第 231–247 页。[28] K. Crary 和 M. J. Sullivan，《使用比特币实现点对点仿射承诺》，载于第 *36* 届 *ACM SIGPLAN* 编程语言设计与实现会议（*PLDI*）论文集，2015年，第 479–488 页。[29] J. Poon 和 T. Dryja，《比特币闪电网络：可扩展的链下即时支付》，白皮书，2016年1月。 [在线]。网址：https://lightning.network/lightning-network-paper.pdf [30] P. Das、L. Eckey、T. Frassetto、D. Gens、K. Host´akov´a、P. Jauernig、S. Faust 和 A.-R. Sadeghi，《FastKitten：比特币上的实用智能合约》，载于第 *28* 届 *USENIX* 安全研讨会（*USENIX Security 19*），2019年，第 801–818 页。[31] L. Aumayr、Z. Avarikioti、R. Linus、M. Maffei、A. Pelosi、C. Stefo 和 A. Zamyatin，《BitVM：比特币上的准图灵完备计算》，密码学电子印本档案，报告 2023/1618，2024年，https://eprint.iacr. org/2023/1618。[32] J. Lau 和 P. Wuille，《BIP 143：版本 0 见证程序的交易签名验证》，https://github.com/bitcoin/bips/blob/ master/bip-0143.mediawiki，2016年1月，比特币改进提案。[33] P. Wuille、J. Nick 和 A. Towns，《BIP 341：Taproot：隔离见证版本 1 花费规则》，https://github.com/bitcoin/bips/blob/master/bip-0341. mediawiki，2020年，比特币改进提案。[34] P. Todd，《BIP 65：OP CHECKLOCKTIMEVERIFY》，https://github. com/bitcoin/bips/blob/master/bip-0065.mediawiki，2014年10月，比特币改进提案。[35] BtcDrak、M. Friedenbach 和 E. Lombrozo，《BIP 112：CHECK- SEQUENCEVERIFY》，https://github.com/bitcoin/bips/blob/master/bip- 0112.mediawiki，2015年8月，比特币改进提案。[36] 谷歌，《Gemini（大语言模型）》， 2025，[访问日期：12月。2025]。 [在线]。网址：https://gemini.google.com



Hyunhum Cho 于2019年获得韩国首尔延世大学商学院工商管理学士学位，并于2022年获得韩国首尔大学工业工程理学硕士学位。目前，他正在韩国首尔高丽大学攻读信息安全博士学位。其主要研究兴趣包括区块链、形式化方法和密码学。



Ik Rae Jeong 分别于1998年、2000年和2004年获得韩国高丽大学计算机科学理学学士学位、理学硕士学位和信息安全博士学位。2006年6月至2008年2月，他在韩国电子与电信研究院（ETRI）担任高级工程师。目前，他是高丽大学网络安全学院的教职人员。其当前研究兴趣包括密码学、理论计算机科学、区块链和生物识别。

附录 A 类型安全性证明

为了从形式上证实 Bithoven 的安全性主张，我们给出连接类型系统（第 V 节）与操作语义（第 IV 节）的可靠性证明。我们证明，任何通过静态分析器检查的 Bithoven 程序，在比特币虚拟机上执行时都不会到达未定义状态。

A. 定义

首先，我们定义运行时栈与静态类型之间的一致性。

定义 *1*：**良类型值：** 运行时值 v 的类型为 τ，记作 $\models v : \tau$，当且仅当：

- 如果 $\tau =$**数值**，则 $v \in \mathbb{Z}$。
- 若 $\tau =$ **布尔值**，则 $v \in \{$**真**，**假**$\}$。
- 若 $\tau =$ **字符串**、**签名**或**公钥**，则 v 是满足相应格式的字节序列。

定义 *2*：**良类型栈：** 一个栈 $\sigma = v_1 :: v_2 :: \cdots :: v_k$ 对应于类型序列 $\vec{\tau} = \tau_1, \tau_2, \ldots, \tau_k$，记作 $\models \sigma : \vec{\tau}$，当且仅当对所有 $i \in \{1..k\}$, $\models v_i : \tau_i$。

B. 安全性定理

我们断言两个基本性质：进展性 （良类型程序永远不会“卡住”）和保持性 （执行保持类型一致性）。

定理 *1*：**进展性：** 设 s 为满足$\Gamma \vdash s \Rightarrow$ **正确**的语句。设 σ 为满足 $\models \sigma$ 的栈：$\sum$（该栈与输入声明匹配）。那么，执行配置 $\langle s, \sigma \rangle$ 不会陷入停滞。它要么：

1）以结果栈 σ'成功终止；2）在有效失败状态 $\perp$ 中终止（例如通过 verify 假）；或 3）可以转换为中间配置$\langle s', \sigma' \rangle$。

证明概要： 我们对语句 s的结构进行归纳。

- **案例（表达式求值）：** 对于在 s 中使用的任意表达式 e，类型规则（图 2）确保运算符作用于兼容的类型。例如，如果代码包含 $e_1 + e_2$，规则 T-BI NARYMATH 确保e_1， e_2 : **数值**。根据规范形式引理，值v_1， v_2 必须是整数。底层的比特币操作码 OP_ ADD 对所有整数都有定义。因此，该操作不会未定义。
- **案例（If-Else）：** 对于**如果** $e \ldots$，规则 T-IF 确保 e : **布尔值**。操作语义为**真**和**假**两种情况都定义了转换。因此，控制流永远不会未定义。
- **情况（锁定时间）：** 对于 **older** n，规则 T-LOCKTIME 确保 n 是有效的 32 位无符号整数。语义规定该操作将转换为 σ （成功）或 $\perp$（基于交易 nSequence 的失败），二者都是有效状态。

■

定理 *2*：**保持性与主题归约：** 如果程序状态具有良好类型，并且执行执行一步，则所得状态仍具有良好类型。形式化地说，如果 $\Gamma \vdash s \Rightarrow$ **正确**且 $\models \sigma : \vec{\tau}_{in}$，并且 $\rho \vdash \langle s, \sigma \rangle \to \sigma'$，那么存在一个类型序列 $\vec{\tau}_{out}$，使得 $\models \sigma' : \vec{\tau}_{out}$。

证明概要： 我们假设归纳假设对所有子语句都满足保持性。

- **基础情形（返回）： 返回** e。如果 $\Gamma \vdash e : \tau$，那么根据表达式语义， e 求值得到一个值 v，其中$\models v : \tau$。最终栈项变为 $v :: \sigma'$（或根据上下文有所不同），其类型良好。
- **归纳步骤（条件语句）：** 考虑**如果** e $\{s_1\}$ **否则** $\{s_2\}$。类型检查器强制要求$\Gamma \vdash s_1 \Rightarrow$**ok** 和 $\Gamma \vdash s_2 \Rightarrow$**ok**。如果 e 求值为**真**，我们就进入 s_1。根据归纳假设，由于 s_1类型良好，其执行结果会得到一个类型良好的栈σ'。如果 e为**假**，则对 s_2同样适用。
- **归纳步骤（验证）： 验证** e。语义规定，如果 $e \to$ **真**，栈保持为 σ' （弹出 e后的状态）。由于在压入/弹出布尔值之前栈的类型良好，因此之后仍然类型良好。如果 $e \to$**假**，状态变为 $\perp$，这是一个有效的终止状态。

因此，Bithoven 程序在执行过程中不会出现类型混淆错误。

■

附录 B 已预防缺陷目录

本附录提供了 Bithoven 静态分析器在编译时预防的缺陷类别的全面目录。每个示例对应于表 II 中识别出的一类漏洞，展示了分析器的稳健性。

A. 漏洞类别：类型系统安全

1) 缺陷：非数值类型上的算术运算：开发者尝试对字符串字面量应用数学运算符（+和 -），而比特币脚本不允许这样做。

```
编译指示 Bithoven 版本 0.0.1;编译指示
Bithoven 目标 隔离见证;

    (签名_alice: 签名){验证("MATH" + "仅适
用于"- "数值");}
```

列表 8：对字符串应用数学运算符。

类型检查器正确识别出，对字符串执行 + 与 - 运算是错误的，因为字符串不是算术运算所需的正确类型。

错误： 第 6 行第 13 列错误：无效操作("操作数必须是数字或布尔值，但实际为：字符串字面量(位置 起点：100，终点：106，行：6，列：13，“MATH”)")

2) 缺陷：类型不匹配比较：开发者试图将数字字面量 2 与字符串字面量进行比较。

```
编译指示 Bithoven 版本 0.0.1;编译指示
Bithoven 目标 隔离见证;

    (签名_alice: 签名){返回 (
2 < "类型错误");}
```

清单 9. 比较数字与字符串。

类型检查器会拒绝此操作，从而确保比较只能在类型相同的值之间进行。**错误：** 第 6 行第 13 列错误：

B. 漏洞类别：活性与作用域

1) 缺陷：引用未定义变量：栈声明定义了 sig_alice，但代码主体错误地引用了 sig_bob。

```
编译指示 Bithoven 版本 0.0.1;编译指示
Bithoven 目标 隔离见证;

(sig_alice: 签名)
{
返回 checksig (sig_bob, "0245...bf5212");}
```

清单 10. 使用未定义变量。

活跃性与作用域分析器会立即标记 sig_bob，因为它尚未在当前作用域中声明。

错误： 第 6 行第 22 列错误：未定义变量("未定义变量："sig_bob"。")

C. 漏洞类别：语义安全

1) 缺陷：混乱逻辑（无效签名检查）： 这对应于[7]中的 useless-sig 漏洞类别。开发者反转了 checksig 的结果。这是一个严重缺陷，因为攻击者现在可以通过提供无效签名来花费资金；无效签名会使 checksig 返回假，而 ! 运算符又会将其反转为真。

```
编译指示 Bithoven 版本 0.0.1;编译指示
Bithoven 目标 隔离见证;

    (sig_alice: 签名){返回 ! checksig (sig_
alice, "0245...bf5212");}
```

清单 11. 反转 checksig 的结果。

Bithoven 的语义安全分析器旨在发现这一特定反模式，从源头上避免密码逻辑遭到破坏。

错误： 第6行第12列错误：无效签名("! 使签名检查操作无用：一元数学表达式 loc：位置 start：99,end：198,line：6, column：12 ,operand: 检查签名表达式 loc:位置 start: 101, end: 198,line:6, column: 14, operand: 单一签名因子 loc: 位置 start: 110,end: 198, line:6, column: 23 , sig:变量(位置 start: 111,end: 120, line:6, column: 24 , 标识符("sig_alice")), 公钥：字符串字面量 (位置 start: 123, end:197, line:7,column: 7, "0245a...f5212") ,op: 签名检查 , op: 非 .")

2) 缺陷：不确定签名（*uncertain-sig*）：开发者试图使用栈上动态提供的公钥（pubkey_alice）验证签名，而不是使用静态常量。这使攻击者能够提供自己的密钥来通过检查。

```
编译指示bithoven 版本0.0.1;编译指示
bithoven 目标隔离见证;

(签名_alice: 签名, 公钥_alice: 字符串){返回
checksig(签名_alice, 公钥_alice);}
```

清单 12. 尝试使用动态公钥。

类型检查器拒绝此操作，强制要求公钥必须是静态字面量，以防止密钥替换攻击。**错误：** 第 6 行第 33 列错误：

D. 漏洞类别：共识规则

1) 缺陷：整数溢出（*32* 位限制）：比特币脚本存在隐式的 32 位整数限制。开发者提供了一个数字字面量（21474836478），其值超过了 32 位有符号幅值整数的最大值（2147483647）。

```
编译指示 bithoven 版本 0.0.1;编译指示
bithoven 目标 隔离见证;

    (sig_alice: 签名)
{返回 21474836478;}
```

清单 13。超出范围的整数文字。

编译器的字面量验证可在部署前阻止这种共识层面的缺陷。

错误： 第 6 行第 12 列错误：整数溢出（"数字是 32 位带符号数值：21474836478"）

E. 漏洞类别：控制流

1) 缺陷：缺少返回语句： 这对应于 [7], 中的永不为真缺陷类别，因为脚本可以在未返回真值的情况下结束。该代码路径包含验证和 older 语句，但缺少终止返回语句。脚本会执行后失败，因为终止时栈将为空。

```
编译指示 bithoven 版本 0.0.1;编译指示
bithoven 目标 隔离见证;

    (签名_alice: 签名){验证 checksig(签名_alice, "
0245...bf5212");older 1000;}
```

清单 14. 缺少终止的返回语句。

控制流分析器遍历程序的 CFG，并检测到存在一条有效路径不会以返回语句终止。

错误： 第 7:5 行错误：NoReturn("每条可能的执行路径都必须存在 Return 语句：锁定时间语句 loc: 位置 start: 195, end: 211, line: 7, column: 5 , operand: 1000, op: Csv .")

2) 缺陷：不可达代码：开发者在终止返回语句后放置了一条 older 语句，因此无法执行。

```
编译指示 Bithoven 版本 0.0.1;编译指示
Bithoven 目标 隔离见证;

    (签名_alice: 签名){返回 checksig (签名_alice, "
0245...bf5212");older 1000;}
```

清单 15. 放置在 Return 语句之后的代码。

控制流分析器会将同一代码块中终止操作之后的所有代码识别为不可达代码。

错误： 第 7:5 行错误：不可达代码（"Return 语句后的不可达代码：锁定时间语句 位置：起始位置：195，结束位置：211，行：7，列：5，操作数：1000，操作码：Csv。将 Return 语句移至执行路径的最后一个作用域"）