分析隐蔽故障对区块链可靠性的影响

Fernando Richter Vidal，Naghmeh Ivaki，Nuno Laranjeiro 科英布拉大学 科英布拉大学信息与系统中心 信息工程系，葡萄牙fernandovidal@dei.uc.pt, naghmeh@dei.uc.pt，cnl@dei.uc.pt

arXiv:2304.05520v1 [cs.CR] 11 Apr 2023

摘要

区块链近年来因应用于加密货币以及在各个领域（如零售、医疗保健和保险）的潜在用途而备受关注。智能合约是区块链系统的重要组成部分，用于规定事务参与方之间的协议。如今，智能合约在部署时仍带有残余故障，其中包括会在运行时导致不同类型失效的严重漏洞。故障检测工具可用于检测故障，从而在部署前将其从代码中移除。然而，对于智能合约，人们普遍认为这类工具尚不成熟、效果不佳。在本研究中，我们开展了一项故障注入活动，以实证分析智能合约中现实存在的故障在运行时可能对区块链系统可靠性造成的影响。我们特别关注那些逃过常用智能合约验证工具检测的故障，并展示这些故障是否以及如何导致区块链系统在运行时失效。结果表明，所使用的三种工具总体检测能力较差，且在一定程度上具有互补性。结果还表明，若干隐蔽故障会导致严重的区块链失效。

索引词

区块链、智能合约、软件故障、安全漏洞、故障注入、验证工具

1 引言

区块链系统可以描述为分布式账本的一种实现，其中交易记录通过密码学方法进行存储并相互链接。此类系统由点对点网络提供支持，节点可以加入该网络并获取区块链状态的副本。当新事务到达时，系统节点之间会使用共识协议，通过共识接受或拒绝该事务 [51]。

在区块链系统的核心，我们可以看到智能合约：一种由区块链中间件存储和执行的编程应用，其中包含与特定事务相关的逻辑，包括确保事务成功完成所需满足的前置条件 [3]。智能合约一旦部署到区块链上，就无法修改。事实上，故障合约只能被终止，并且必须另行部署一个新合约，这可能会给相关各方带来严重后果（例如安全性和财务方面） [38]。

智能合约在携带残留漏洞（包括严重的安全漏洞）的情况下被部署。这是由于多种原因造成的，包括开发者缺乏区块链方面的专业知识、使用新的编程语言（例如，Solidity是用于编写智能合约的热门选择）、使用新的编码工具，以及缺乏成熟的缺陷验证工具。所有这些因素共同作用，导致故障合约被部署；这些合约的故障会在某个时间点被激活，并产生各种后果，包括性能下降、账本完整性违规、资源使用增加进而导致 Gas耗尽，以及许多其他后果。

在本文中，我们开展了一项故障注入活动，以展示现实中的故障会对区块链系统的可靠性产生何种影响，并特别关注可能逃过热门智能合约验证工具检测的故障类型。请注意，我们使用术语 区块链可靠性 泛指那些偏离事务正确执行的各种问题。具体而言，我们指的是以下几类问题和失效：回滚、中止、燃料耗尽、正确性、完整性 和潜在完整性。我们用难以察觉的 来指代那些逃过智能合约验证工具检测的特定故障，并在本文末尾分析其影响。总而言之，我们的方案包括以下三个步骤：

1) 步骤 *1*： 我们首先采用此前研究中构建的现有故障模型 [15]，，该模型基于实际智能合约中观察到的故障（也就是说，在这一意义上，它代表了区块链环境中出现的问题），并针对总计 36 种故障实现注入器。

2) 步骤*2*： 我们从 [18]，中随机抽取 400 个智能合约并注入故障，最终得到 15,494 个故障合约（每次注入故障可能会产生多个故障合约，因为某些故障可以注入合约中的多个位置）。

随后，我们部署原始合约和故障合约，并让它们处理用户交易（本研究在数月期间共执行了 10,925,749 笔交易），以分别了解预期的正常行为和可能出现的偏差（异常行为），即失效（例如，事务中止、账本完整性损坏）以及性能下降（例如，内存消耗过高和 CPU 使用率过高）。

3) 步骤 *3*：我们将观察结果映射到三种先进故障检测工具的故障检测能力上，即 Mythril 版本 v0.22.19 [13], 、Slither v0.8.0 [17], 和 Securify v0.0.1 [46]。此举旨在展示这些工具的整体有效性，尤其是指出验证工具往往无法检测到的故障类型及其对区块链系统的影响。

本研究的主要结果表明：i) 注入故障能够在运行时引发多种类型的失效；ii) 验证工具的有效性较低，即使对于看似容易检测的故障（例如，缺失编译器版本）也是如此；在一定程度上，这些工具还具有互补性；iii) 确实存在隐蔽故障，其中一些与区块链系统中的严重失效类型密切相关，包括潜在完整性、完整性和正确性 失效。本文的贡献如下：

- 通过故障注入工具提供的一组包含 36 个智能合约特有故障的实现，见 [47]。
- 对注入初始的400个真实智能合约中的不同类型现实故障，并由此产生的15,494个变异合约进行的大规模影响分析（即，对于每个单独的故障，每个合约至少有一个故障变体）。
- 在存在不同类型故障的情况下，评估三种先进的智能合约验证工具（即 Securify2[46], Mythril [13], 和 Slither [17]）的故障检测效果。
- 分析难以被验证工具发现的故障对区块链系统可靠性的影响。

本文的组织结构如下。第二节介绍背景和相关工作，第三节介绍实验研究的设计。第四节讨论实验结果，第五节介绍本研究有效性的主要威胁。最后，第六节对本文进行总结。

2 背景与相关工作

本节首先介绍智能合约安全问题的背景，随后回顾安全与验证工具及相关技术。最后，介绍评估安全与验证工具的研究工作。

2.1 智能合约的安全漏洞

为了描述智能合约中的漏洞，主要有两种分类，即去中心化应用安全项目（DASP） [35] 和智能合约弱点分类（SWC） [33]。DASP 是由 NCC Group [35] 发起的一个协作项目，旨在描述智能合约漏洞。它包含 9 类漏洞。SWC 是由 MythX 团队支持的一种漏洞分类，旨在描述智能合约漏洞，并以通用弱点枚举（CWE）为基础。它包含 37 类漏洞。

除了这两种分类之外，我们还可以在一些研究论文中找到特定的漏洞集合（包括其他类型）。Harzand Knottenbelt [22] 对智能合约编程语言的安全性进行了综述，介绍了当前用于实现智能合约的编程语言（共 19 种）、它们的安全特性，以及编程范式、指令集、语义和计量等其他信息。Praitheeshan 等人 [38] 列出了 16 种以太坊智能合约漏洞和 19 种软件安全问题。Peng 等人 [37] 回顾了物联网应用中智能合约的使用，并指出程序中已发现的漏洞所带来的主要安全挑战。作者报告的研究结果表明，现实世界中的应用仍处于起步阶段；尽管已有安全审计工具，但它们只能检测出已知漏洞中的一小部分。

2.2 智能合约安全评估工具和技术

在本节中，我们分析了在回顾智能合约安全验证和安全评估领域的最新进展时所发现的技术和工具。所发现的研究主要分为四类：**形式化验证技术**[6], **静态分析技术** [10], **软件测试技术**以及**基于机器学习的技术** [31]。

形式化验证类别包括基于形式化证明或系统（或系统某一部分）的抽象数学模型的方法，用于证明其正确性（例如，主要是功能正确性） [43]。此类别中的具体技术包括模型检测 [25, 27] 和定理证明 [42]。

静态分析类别包括无需执行代码、依靠各种方式检查代码（例如模式识别、污点分析）来发现软件缺陷 [41]的方法。该类别中的具体技术包括抽象解释 [14], 污点分析 [50], 模式识别 [54]。

软件测试类别包括依赖软件执行来发现缺陷的方法[32]。此类别中的具体技术包括模糊测试 [8],变异测试 [49], 或符号执行 [5, 26, 44]。

机器学习类别包括基于人工智能的技术，重点研究能够从经验和历史数据中学习的算法，以检测异常或预测新的输出值（即漏洞）。在此背景下，大多数技术基于监督学习，即使用带标签的数据集训练算法，以对数据进行分类或预测特定输出的结果。此类别中的具体技术包括：经典机器学习模型：经典机器学习模型 [48], 深度学习模型 [55], 以及集成学习模型 [53]。请注意，有些工具会使用多种技术来实现其目标。例如，Smartian [11] 结合使用静态分析和动态分析技术，对智能合约进行模糊测试。

在此，我们重点介绍三种验证工具，即 Slither、Securify 和 Mythril；本研究将对这些工具进行分析。**Slither** [17] 是一种使用污点分析的静态分析工具。该工具将 Solidity 智能合约源代码编译为 Solidity 抽象语法树（AST），以提取合约的继承图、控制流图（CFG）和表达式列表。随后，它将合约代码转换为 SlithIR——其内部表示语言；该语言采用静态单赋值（SSA），以便执行各种代码分析。此外，它还包含用于代码理解和辅助代码审查的图。作者在 1,000 个使用最广泛的智能合约（交易数量最多的合约）上对该工具进行了评估，结果发现，就性能、健壮性（即工具是否能够完成执行）和准确率（即报告的误报数量）而言，它优于另外三种流行工具（Solhint、 [39], SmartCheck [45],Securify [46]）。

Securify [46] 是一种静态分析工具，它使用可扩展的 Datalog 求解器 Soufflé 对智能合约的 EVM 字节码进行符号分析并提取语义事实，然后根据违规模式检查这些语义。因此，该工具基于一组合规与违规安全模式开发，这些模式包含了证明和证伪实际安全属性的充分条件。为增强可扩展性，这些模式采用领域专用语言编写。

Mythril 是一款开源工具，基于符号执行、SMT 求解和污点分析，用于分析智能合约的安全性。它能够检测软件缺陷，即以太坊及其他一些兼容 EVM 的区块链上实现的智能合约中的各种安全漏洞。该工具常用于与智能合约评估相关的研究中，例如 [20], [36], [2]。

2.3 验证工具评估

在 [36] 中，作者使用十个合约评估 Oyente、Securify、Mythril 和 SmartCheck，并通过执行受试者工作特征（ROC）分析来表达这些工具的有效性，同时还分析了准确率，揭示了这些工具在有效性方面的差异与不足。 [2]提出了一个用于分析和测试智能合约的框架。作者使用 Oyente、Securify、Maian、SmartCheck 和 Mythril对其方案进行了评估，测试对象包括 1,838个合约和 8个故障，并利用这些故障生成了 12,866 个变异合约。使用精确率和召回率来刻画这些工具的检测能力。

文献 [52], 提出了一种漏洞基准测试，并通过运行 Oyente 和 Slither，对从 `etherscan.io`中随机生成选取的 1,010 个合约进行检测，展示了这些工具在面对已知漏洞时的检测缺陷。文献 [20] 的作者评估了多种静态分析工具的漏洞检测效果，包括 Oyente、Securify、Mythril、Smartcheck、Manticore 和 Slither。该方法基于已知漏洞模式向合约中注入漏洞。注入的漏洞类型均为工具声称能够检测的类型，并采用假阴性率和假阳性率等常用指标评估工具的性能。

文献 [16]评估了九种智能合约分析工具。作者使用了 47,587 个以太坊智能合约，指出这些工具的检测能力存在明显不足，其中被认为准确度最高的工具 Mythril 也只能检测出 27% 的漏洞。文献 [40], 的作者对 9 种合约验证工具针对 46,186 个智能合约进行了实证评估。主要发现包括：建议采用一组多样化的测试套件；使用配置了合适运行时参数的统一执行环境；以及采用更具量化性和多维度的性能指标。

在我们之前的工作中 [15], 提出了一种故障注入方法，用于分析三个静态验证工具 Mythril、Securify和Slither 的有效性。然而，该工作只是概念验证，且研究仅限于少量合约和少量漏洞。在本工作中，我们考虑了大量具有多种不同类型漏洞的智能合约。

此外，上述用于评估验证工具的研究通常都认为，智能合约验证工具尚不成熟，这一点反映在其检测能力上。然而，这种观点通常未结合智能合约特定故障的运行时影响进行考察。我们在另一项先前的工作 [21], 中曾尝试分析故障对区块链系统的影响，但研究规模非常小（仅涉及 5个合约）。其他研究往往不会深入分析那些逃过智能合约验证工具检测的故障（及其影响），而这正是本工作的主要目标之一。

3 方法

本节介绍本研究采用的方法。接下来的小节将依次介绍以下主要步骤：

1)故障注入方法，即在一组合约中注入故障；2) 分析注入故障对每个合约所产生影响的过程；3)分析智能合约故障检测工具有效性的过程；4) 分析那些往往能够躲过智能合约验证工具的故障所产生影响。

3.1 故障注入方法

这项工作的出发点是能够对智能合约注入故障。为此，我们遵循软件故障注入这一由来已久的传统：基于一个能够代表真实故障的模型（即在实际运行系统中观察到的故障），在更大型系统的某个组件中人为引入“可能发生的”软件故障[34]。这样一来，我们便能够了解某类故障在被激活后可能对系统产生的影响（例如，系统是否会灾难性失效，其性能是否会下降）[30]。此类技术可用于实现多种目标，即评估系统行为 [34], 测试套件的有效性（即在变异测试方法的情况下）[28], 作用于系统的工具，例如漏洞扫描器 [19]，甚至用于故障预测 [24]。我们采用了一种软件实现的故障注入（SWIFI）技术 [34], , 过去曾成功使用过该技术 [21], , 不过当时的应用范围要窄得多（在5个合约中使用了另一组故障）。为完整起见，接下来几段将从概念上概述该技术。

图1展示了我们的故障注入流程。第一阶段（图像左侧）是将原始代码（即 Solidity 格式）转换为 AST 格式。为完成此任务，我们使用 Solidity 编译器的原生函数（即*solc* 及*–ast-json* 参数）先编译代码（步骤1），再将其转换为 AST（步骤2）。这样，只有成功编译的合约才能生成 AST。

第二阶段（中间部分）将之前创建的 AST 转换为有缺陷的智能合约（步骤 3）。对于故障模型中存在的每种故障，都会针对给定的 AST 至少生成一个故障合约。这是因为故障可以通过不同形式注入（例如，错误的算术表达式可能有多种形式），也可以注入到 AST 中的不同位置。步骤 3 会生成一组有缺陷的 AST。在下一步，即步骤 4 中，所有有缺陷的 AST 都会转换回其原始格式（即 Solidity 代码）。为实现这一点，我们编写了应用该转换所需的代码，并通过手动检查生成的文件并将其与原始代码进行比较，验证了其正确性。两位早期职业阶段研究人员参与了这一验证过程。最终，这一步会生成一组合约，称为变异体。

最后一个阶段聚焦于变异体的部署。在步骤 5 中，我们验证变异体是否仍是有效的可执行程序（即通过编译进行验证）。成功编译的程序随后会部署到 Hyperledger Fabric 中（步骤 6），其余合约则从分析中移除（步骤 7）。

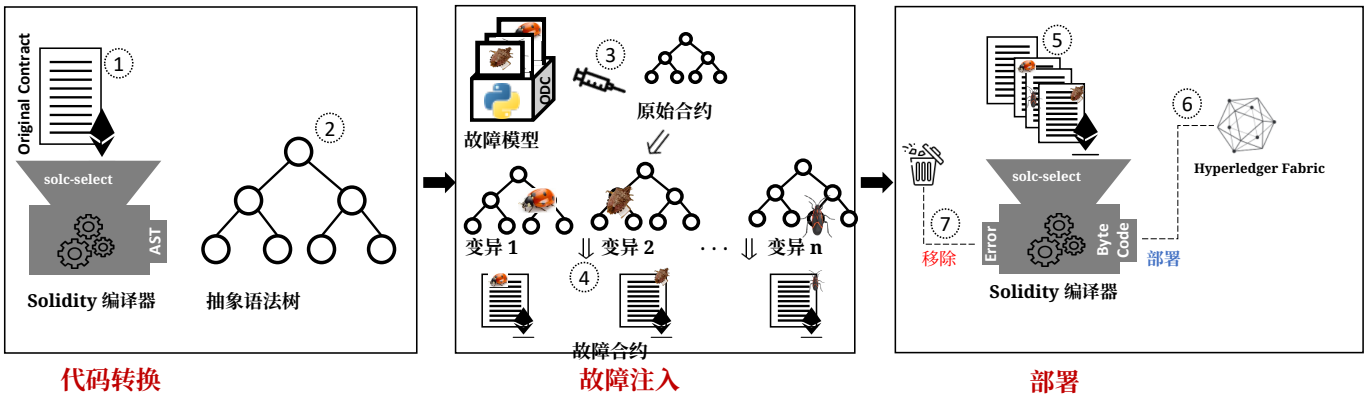


图 1. 故障注入流程。

图 2 展示了一个示例，其中左侧是一个合约（名为 original），右侧是对应的故障版本。该故障版本是对原始合约进行变异后的结果，包含一种名为未初始化的存储指针 （SWC-109） [33] 的特定漏洞。此漏洞指的是局部存储变量未初始化，因此可能被用于指向合约中意料之外的存储位置。这可能导致意外行为，原因可能是攻击者故意为之，也可能是无意造成的。该程序允许经过授权的人员向供应商转账，其设计灵感来自 [4] 中的示例。并发付款由未锁定 变量和*require* 指令控制，只有在合约未锁定时才允许付款。在原始合约中，有一个名为*newTransfer*的局部变量，通过使用内存 属性（第 8 行）显式标记变量的存储位置，从而对其进行了适当初始化。相反，故障合约没有显式标记其存储位置，因此该变量仍是未初始化的局部变量（而不是全局存储变量）。

如前所述，为了注入 SWC-109 漏洞，我们先将原始代码转换为 AST 表示，然后注入漏洞。图 3 展示了一个用于注入此漏洞的 Python 示例。注入器会在 AST 中搜索某个条件（由函数 `condition` 实现）。条件匹配时，会定位树中的节点/属性，然后应用更改（参见函数 `changeTre`），从而生成故障 AST。在这个具体示例中，注入器会查找用于初始化本地存储变量的内存 属性，并将其替换为“storage”。最后，故障 AST 会被转换回故障合约代码。完整的 36 种故障实现可在 [47]获取。

Original Contract

```
contract PaySupplier {
   bool public unlocked = false;
   function TransferMoney(bytes32 _name)
   public {
      Person memory newTransfer;
      newTransfer.name = _name;
      require(unlocked);
   }
}
```

Faulty Contract

```
contract PaySupplier {
   bool public unlocked = false;
   function TransferMoney(bytes32 _name)
   public {
      Person storage newTransfer;
      newTransfer.name = _name;
      [several lines of code]
      require(unlocked);
   }
}
```

Fig. 2. 一个故障合约示例 .

Injector for Uninitialized Storage Pointer vulnerability

```
from common import mainfunc

def condition(root, node):
    return node['attributes']["initialValue"] == None and node['name'] == 'VariableDeclaration'
    and node['attributes']['storageLocation'] == "memory"

def changeTree(root, node):
    node['attributes']['storageLocation'] = "storage"
    node['attributes']['type'] = node['attributes']['type'].replace("memory", "storage")

if __name__ == "__main__":
    mainfunc('Removes memory initialization pointers.', condition, changeTree, '4')
```

图 3. 注入 SWC-109 漏洞的代码示例。

注入故障后，我们将得到一份无故障智能合约（即原始合约，不含已知故障）及其对应的有缺陷的智能合约列表，用于我们的评估。评估由以下研究组成：

- **研究 1 - 故障的影响**：我们评估区块链系统在存在注入故障时的行为。为此，我们在区块链的隔离环境中分别单独运行无故障版本及其对应的故障版本。我们比较无故障运行和故障运行的结果。通过无故障运行，我们可以将其作为参考，评估每种故障的影响。
- **研究 2 -验证工具有效性**：我们评估智能合约验证工具的有效性，即评估这些工具针对故障注入工具生成的一组故障合约的故障检测能力，这些故障合约基于我们的故障模型中所描述的故障。
- **研究3 - 隐蔽故障的影响**：我们分析往往会逃过验证工具检测的故障所造成的影响。

3.1.1 故障模型

在本研究中，我们选择采用先前工作中提出的现有故障模型 [15], ，以选取智能合约特定故障（用于故障注入过程）。这些缺陷依据正交缺陷分类（ODC）方案 [23] 进行分类，我们尝试为每个 ODC 类别和每种缺陷类型至少实现一种故障，同时兼顾所需投入的工作量（同一类别和类型中的某些故障非常相似，因此实现起来也很简单）。最终，我们共实现了 36 种故障，并在表1中列出。

这 36 种已实现故障只是所有可能影响区块链系统的故障的一个子集（例如，目前我们尚未实现重入故障）。尽管如此，值得一提的是，本研究的重点并非定义故障模型，而是提出一种整体方法，最终用于分析一部分容易逃过检测工具的故障所造成的影响。从这个意义上说，所采用的具体故障和工具都可以有所不同。

表1 智能合约缺陷分类 [15]

缺陷类别	缺陷性质	缺陷名称	缺陷标识符
赋值	缺失	存储变量/指针初始化（未初始化的存储指针）**(MISP)**	一个 MISP_
		局部变量初始化**(MILV)**	A_MILV
		状态变量初始化**(MISV)**	A_MISV
		构造函数 **(MC)**	A_MC
		编译器版本（**MCV**）	一个 MCV_
	错误	赋值中使用的算术表达式（**WVAE**）	一个 WVAE_
		整数符号（**WIS**）	一种 WIS_
		整数截断（**WIT**）	一种 WIT_
		赋值时数字过多（**WVATMD**）	一种 WVATMD_
		向合约地址赋值（**WVAA**）	一个 WVAA_
		构造函数名称**(WCN)**	A_WCN
		变量类型（例如，byte[]）**(WVT)**	一个 WVT_
		不变状态变量声明**(WDISV)**	A_WDISV
		变量名（变量遮蔽）（**WVN**）	A_WVN
检查	缺失	“require”关键字 关于交易发送方**(MRTS)**	CH MRTS_
		“require”关键字作用于输入变量**(MRIV)**	CH MRIV_
		交易发送方上的“require”或子表达式**(MROTS)**	CH MROTS_
		“require”关键字或输入变量上的子表达式**(MROIV)**	CH MROIV_
		“require”关键字与交易发送方上的 AND 子表达式（**MRATS**）	CH MRATS_
		“require” AND 输入变量上的子表达式**(MRAIV)**	CH MRAIV_
		检查燃料上限**(MCHGL)**	CH_MCHGL
		算术运算检查**(MCHAO)**	CH MCHAO_
		自毁功能检查**(MCHSF)**	CH MCHSF_
	错误	授权的“require”关键字（通过 tx.origin 进行授权）**(WRA)**	CH WRA_
接口	缺失	状态变量可见性修饰符（隐式可见性）**(MVMSV)**	I MVMSV_
		函数可见性修饰符**(MFVM)**	I MFVM_
	错误	私有/内部函数的可见性（公开）（**WVPF**）	I WVPF_
算法	缺失	交易发送方上的“if”结构加语句**(MITSS)**	AL MITSS_
		“if”结构作用于输入变量加语句**(MIIVS)**	AL MIIVS_
	错误	require、assert 和回滚的使用**(WRAR)**	AL WRAR_
		异常处理**(WEH)**	AL WEH_
	多余的	do-while 语句或for**(ECSWS)** 语句中的 continue 语句	AL_ECSWS
函数	缺失	提款函数**(MWF)**	F MWF_
		继承 **(MINHERITANCE)**	F_MINHERITANCE
	错误	继承及继承顺序**(WIO)**	F WIO_
	多余	继承 **(EINHERITANCE)**	F_EINHERITANCE

3.1.2 智能合约数据集

为了**确定一组合约**，以便将其用作这些实验的输入，我们从 Durieux 等人的工作中使用的合约里随机生成选取了 400 个 [16]，这些合约可在 [47] 获取。接下来，将每个合约传递给故障注入工具，以确定 36 种故障中哪些可以注入，以及在该合约的多少个代码位置进行注入。然后，该工具通过迭代生成相应的故障合约。此过程总共生成了 15,494 个故障合约（每个故障合约恰好携带一种故障）。

3.2 研究1——故障的影响

本节介绍了用于研究我们注入的故障对智能合约可能产生何种影响的方法。图4展示了该方法，实际操作中包括以下步骤：

1）生成智能合约工作负载。2）在私有网络中执行无故障和有缺陷的智能合约。
3）基于一组关注指标分析结果。

步骤 *1*）指的是**智能合约工作负载生成**，通过这种方式，可在调用智能合约操作（即执行交易）时激活注入故障。我们采用简单的工作负载生成流程，目标是能够执行大多数合约；同时也知道，在某些情况下，生成调用需要更复杂的实现。我们强调，目标是生成有效调用（而非生成无效或恶意调用，例如模糊测试器所生成的调用）。在实践中，我们会查找智能合约中的函数，并根据相应输入参数的数据类型、代码中出现的字面量以及随机生成的值来生成参数值，具体如下：

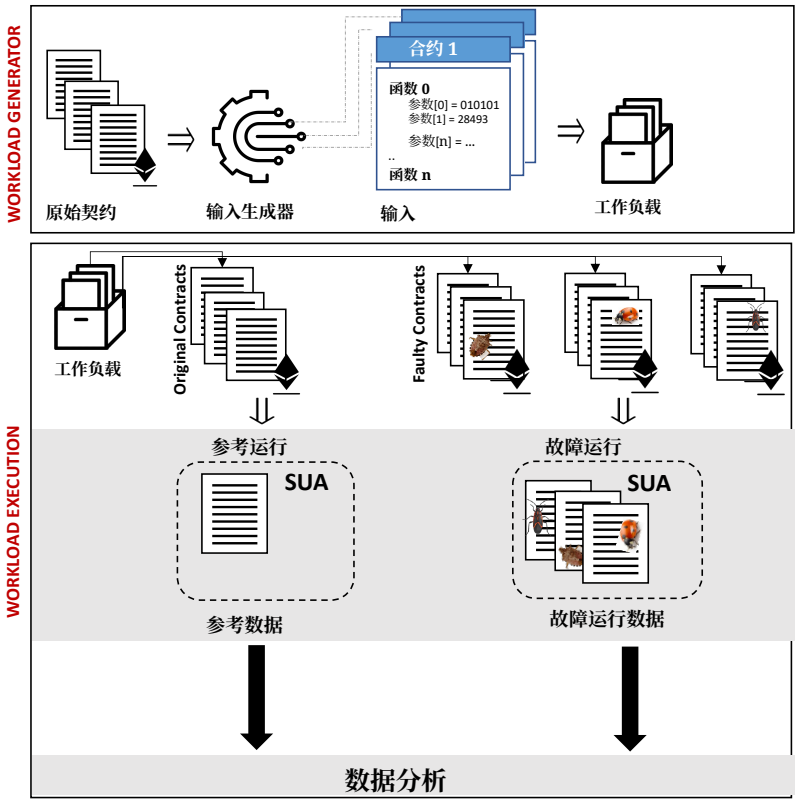


图4. 分析故障对区块链系统影响的方法。

1）基于类型： 我们根据参数的数据类型生成值，例如，布尔值使用 true 和 false，整数使用最小值、最大值或零。对于数组和字符串，我们会递归生成不同长度的值，包括长度为零的情况。2）基于字面量： 函数中出现的字面量会用作与其类型匹配的参数的输入。这样做是因为输入经常会与字面量进行比较，进而决定函数代码中的执行流程。3）随机生成：我们也会生成随机值，例如，为整数类型的参数生成处于最小值和最大值范围内的值，为字符串生成包含随机字符且长度随机的值。数组则由该类型的随机元素构成，长度也各不相同。

目前，我们允许每个函数最多进行1500次函数调用；我们发现，在保持生成的交易总数处于合理水平的同时，这通常足以确保足够的交易多样性（用于数据分析）。这一工作负载生成过程仅针对原始智能合约进行，随后将每个生成的故障合约在针对原始合约生成的工作负载下执行。这样，我们便能够将在故障合约执行期间观察到的运行时行为与其对应的无故障运行（即黄金运行）进行比较。本小节稍后将讨论用于比较的指标。

在 步骤*2* 中，我们**在私有网络中执行智能合约**。在环境方面，我们采用 HyperledgerFabric 以及 Hyperledger Burrow 的以太坊虚拟机（EVM）版本进行部署。随后，我们使用 Hyperledger Caliper执行测试运行。Hyperledger Caliper是一种区块链基准测试工具，可让用户根据一些预定义的用例来衡量区块链实现的性能 [1]。测试运行通过分别在无故障合约和相应生成的故障版本中执行各自的工作负载来完成。一次测试运行的执行顺序如下：

1) Hyperledger被设置为干净的初始状态，这意味着相应的节点（排序节点和背书节点）均已设置（o r
重置）为空区块链。2）将待评估合约部署到区块链上（在本例中，部署到背书节点上）。3）执行为该合约生成的工作负载，并收集有关每个事务的指标。

在测试运行期间，Caliper 会提供多项事务详细信息，例如每个事务阶段的计时、平台返回的副作用以及其他状态信息。最后，对收集的数据进行后处理，以匹配并比较故障合约中发生的每个事务与参考合约中相应的事务。

在实验设置中选择 Hyperledger 平台，主要是因为它提供了便捷的方式来收集所需的各种指标（例如，事务执行时间、回滚的交易、CPU/内存使用情况）。在

关于性能指标，需要注意的是，目标并非获得绝对性能值，而是了解在现实条件下相对影响有多大。因此，该设置与其他研究性能的研究类似 [12, 29]。由于我们的目标是创建一个独立于编程语言的注入器，并且 Hyperledger 包含模块化区块链框架，因此使用 Hyperledger 的决定有利于未来的工作，届时可以轻松复用该基础设施，在不同的区块链中运行程序。

完成运行后，在步骤 *3* 中，我们**分析结果**。我们比较参考数据（即无故障运行的结果）和变异数据（即故障智能合约运行的结果）。为此，我们考察测试用例中事务的成功提交情况，以及事务中出现的差异和失效。在每次执行中，只有满足以下条件，事务才被视为成功：i) 所有背书均成功且相互匹配；ii) 所有背书节点均成功对其排序，并报告其已提交。在之前的工作中，我们识别了几种不同类型的区块链失效 [21], ，这些类型也符合相关工作中讨论的失效类型，例如 [7][9]。基于此，并结合我们在实验期间对不同失效进行的实证分析，我们将观察结果归纳为以下故障模式：

- **回滚失败**：当发生回滚时，事务的执行会停止，所有状态更改都会回滚。回滚的事务会消耗截至回滚时已使用的燃料。在我们的语境中，这种故障模式用于表示故障合约中是否至少有一个事务发生了回滚，而其参考实例却没有回滚。

- **中止失败**：与回滚失败类似，发生中止时，事务的执行会停止，所有状态更改都会回滚。不同之处在于，中止的事务会消耗该事务最大限额内的全部燃料。此失效模式表示，故障合约中是否至少有一个事务被中止，而其参考实例未发生失效。

- **燃料耗尽失败**：表示故障合约中是否至少有一个事务因 Gas 耗尽而失败，而其参考实例中没有发生这种情况。

- **正确性失败**：表示故障合约中是否至少有一个事务的输出结果或返回值与参考无故障合约不同。正确性使我们能够观察到客户端在检查输出不变量时可以发现的失效。

- **完整性失败**：表示故障合约中是否至少有一个事务的输出结果或返回值与参考无故障合约不同，且其读写集也不同（即该事务对区块链状态的修改方式与参考合约中的事务不同）。完整性使我们能够观察到客户端可以发现的账本完整性失效。

- **潜在完整性失败**：表示故障合约中是否至少有一个事务输出了与其参考无故障合约相同的结果或返回值，但其读写集与参考合约不同。此处旨在观察那些隐藏起来且无法被客户端直接看到的错误，因为客户端收到的是预期结果或返回值。

表 2 概述了本研究用于分析结果的故障模型。

表 2 故障模式及其特征

故障模式	**事务 尚未得出结论。**	**不正确 返回值 或 事务结果**	**不正确 账本状态**
中止	●		
回滚	●		
燃料耗尽	●		
正确性		●	
完整性		●	●
潜在完整性			●

为了描述这些失效，我们考察事务是否已完成、事务结果（返回值）是否正确，以及最后账本状态是否正确。如表所示，当中止和回滚失效发生时，不会向客户端返回任何值（事务结果），账本状态也不会发生任何变化。事务执行失败，并向客户端传递某种错误或异常。中止失败 与回滚失败 之间的唯一区别与燃料消耗有关。回滚的事务会消耗截至回滚时已使用的燃料，而中止的事务会消耗该事务允许的最大燃料。

对于燃料耗尽失败，与前述情况类似，不会返回任何值，也不会对账本状态作出任何更改。然而，由于燃料耗尽，事务无法完成（例如，故障可能导致消耗更多资源）。发生正确性失败时，事务成功完成，但事务结果与参考运行不同。在这种情况下，账本状态保持完整。相比之下，发生完整性失败时，除了返回值不正确外，账本状态的完整性也会遭到破坏。最后，潜在完整性失败 指账本状态的完整性发生变化，但事务结果（返回给客户端的值）正确（这意味着

客户端无法检测到问题）。尽管正确性失败和完整性失败都很严重，但由于潜在完整性失败无法被检测到，因此在我们的故障模型中，它是最严重的失效。

3.3 研究 2 - 验证工具的有效性

本节介绍了用于评估验证工具（即 Mythril、Securify2 和 Slither）检测能力的研究，如图 5 所示。实际上，我们执行了以下步骤：

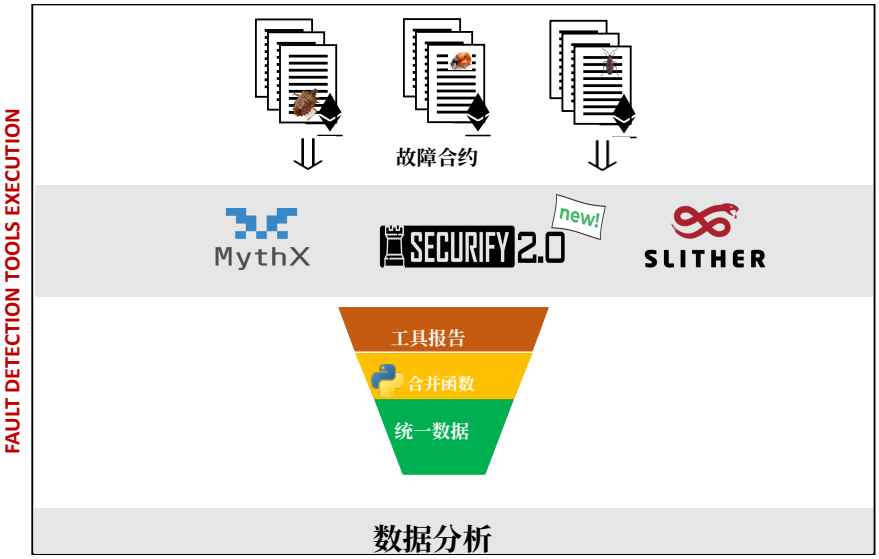


图 5. 分析验证工具有效性的方法。

1)智能合约验证工具的选择；2)针对有缺陷的智能合约（通过故障注入生成）执行这些工具；3) 基于一组相关指标进行结果分析。

步骤 *1*）涉及**智能合约验证工具的选择**。我们的目标是选择受欢迎、积极维护且运行性质不同的工具。具体而言，我们选择了一个抽象解释工具（Securify2版本0.0.1）、一个静态分析工具（Slither版本0.8.0），以及一个使用符号执行的工具（Mythril 版本 0.22.19）。从我们方法的角度来看，这是一组可变的工具，在这一阶段也可以使用其他工具（例如，ZEUS [25], Oyente [26], Smartest [44], Smartian [11]）。工具的具体选择将取决于各种因素，例如运行工具和分析结果的可用时间（例如，某些工具需要更多时间来执行，其他工具的误报率较高）、计算资源，以及执行该方法的用户的总体要求。

在步骤*2)*中，我们**使用工具**检查生成的 15,494 个有缺陷的智能合约，收集工具的输出，然后存储并处理工具生成的结果，将检测到的漏洞映射到所分析的合约。运行工具时使用其默认参数，不针对特定类型的故障进行特别配置。

在步骤*3)*中，**我们分析结果**，但在此之前，会通过我们实现的合并函数将工具的输出报告转换为统一数据格式。这样，添加新工具时无需更改分析流程。在分析结果时，我们会人工验证所有可能的真阳性案例（即工具报告的软件故障确实存在），以检查所报告的缺陷是否确实存在于合约中（同时也借此了解故障注入器是否正确注入了故障）。我们重点评估工具检测注入故障的整体有效性，这些故障应存在于所有受分析的合约中。其他潜在故障（即此前未知的漏洞）不在本研究范围内。

3.4 研究3 -难以捉摸的故障的影响

这项最终研究聚焦于前述研究的结果，并分析了验证工具未能检测到的故障所造成的后果。分析主要旨在了解每种故障类型中故障合约（未被任何工具检测到）的分布情况、不同故障类型所对应的各类失效的发生率，最后重点研究会导致最严重失效的故障。

4 结果与讨论

本节讨论实验评估期间获得的结果。所有实验均在 4 台虚拟机上运行，每台配备 16 个 CPU 和 16 GB 内存，使用 Ubuntu 18.04.5 LTS。运行故障注入流程后，我们为 36 种不同类型的故障分别生成了**至少一个** 故障合约（在 400 个智能合约中），最终

总计生成了 15,494 个（>= 400 *36，因为同一个故障可能被注入到某个合约代码中的多个位置）有缺陷的智能合约。图 6 概述了按缺陷类型划分的生成故障合约分布（蓝色柱形）。

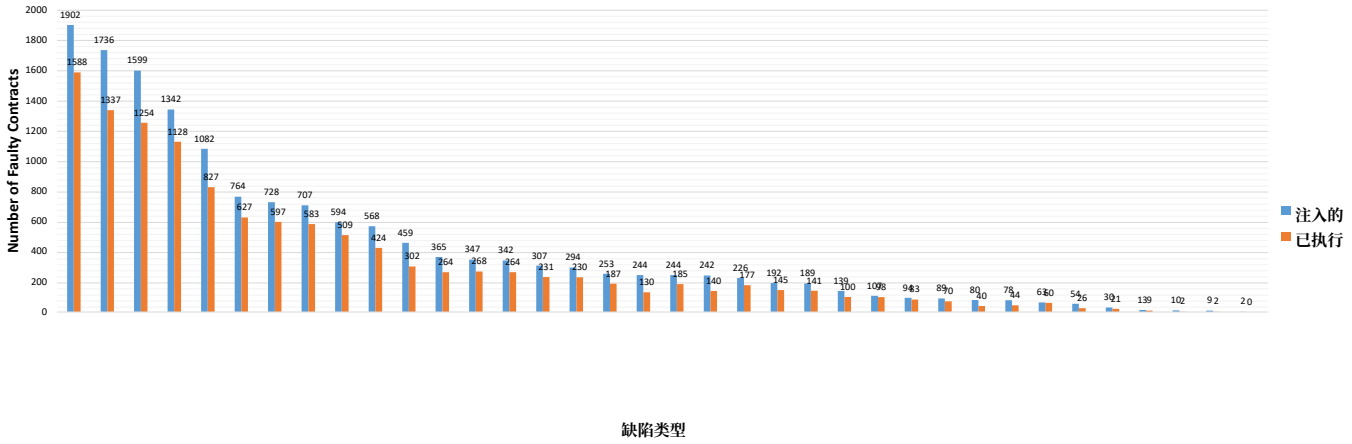


图 6. 按缺陷类型划分的生成的有缺陷智能合约分布。

如图 6 所示，有些故障会导致更多故障合约，例如缺失状态变量可见性修饰符 *(I_MVMSV)*（1902 次）、缺失局部变量初始化 *(A_MILV)*（1736 次），以及缺失对输入值的 *require* 检查 *(CH_MRIV)*（1599 次）。这些数字并不能直接反映它们在现实世界中的出现频率，而仅仅与原始合约代码中符合注入条件的位置数量有关。相反，也有少数故障极少出现，例如缺失燃料限制检查 *(CH_MCHGL)*（2 次），或缺失自毁功能检查 *(CH_MCHSF)*（9 次）。请注意，尽管某种故障可能不常见（即发生概率较低），但其相关风险可能很高，这意味着工具不应忽略此类情况。

图 6 的橙色柱状图还显示了针对每种缺陷类型实际执行的故障智能合约数量。由于工作负载生成工具（第 3 节中介绍）目前无法完全匹配调用全部 15,494 份合约中的所有事务所需的参数类型和数量，因此实际执行的合约数量少于合约总数。尽管如此，我们仍成功运行了所有生成的故障合约中的 83%（12,093 份，共 15,494 份），涵盖全部 36 种故障。

4.1 研究 1 结果——故障的影响

我们在故障智能合约上运行了生成的工作负载，执行的事务总数为 10，925，749，其中 2，782，063 （约 25.46%）成功执行，且未观察到任何影响。其余事务均受到注入故障的影响并导致执行失败。图 7 概述了结果的分布情况。

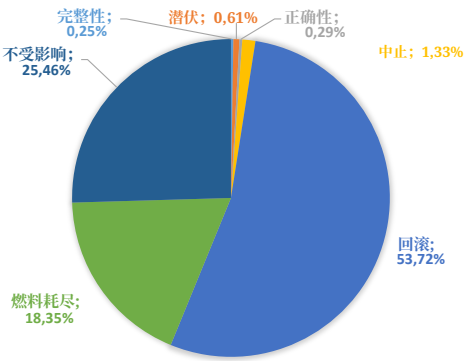


图 7. 故障影响的整体视图。

触发的失效大多属于回滚失败 （约占所有交易的 53.72%），其次是燃料耗尽失败（约占所有交易的 18.35%）。其余失效虽占比不到 2.5%，却是最关键的失效，因为它们会影响燃料消耗、结果的正确性以及账本。

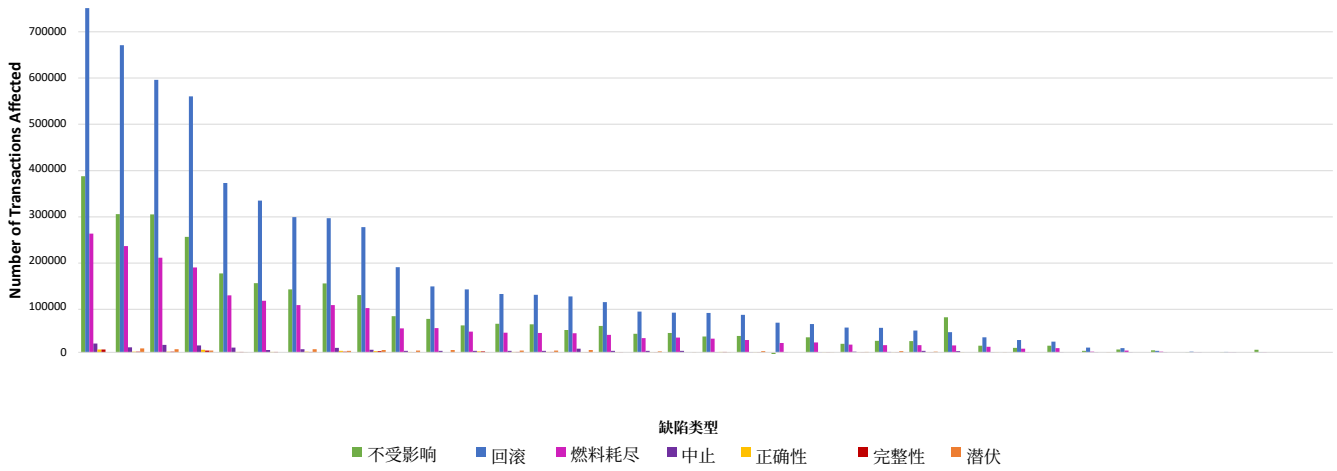


图 8. 各缺陷类型的故障影响。

图8展示了按缺陷类型划分的详细影响结果。值得注意的是，深入到故障类型后，不同类型失效的相对占比在各类故障中总体相同。

图 9 展示了导致严重失效的故障类型的详细结果，即正确性失败、完整性失败和潜在完整性失败。在全部 36 种故障类型中，只有 9 种不会导致上述任何一种失效（例如，*CH_MCHGL* 和*CH_MCHSF* 就是这 9 种情况中的两种）。请注意，在有缺陷的智能合约列表中，除*I_MFVM*之外，这 9 种故障类型出现的频率均最低（参见图 6）。

图 9 所示结果表明，仍有许多情况下，注入的大多数缺陷类型都会导致正确性、完整性，尤其是潜在完整性故障。如图所示，输入变量缺少 *require* 语句 *(CH_MRIV)* 会导致最多的潜伏故障，而状态变量缺少可见性修饰符 *(I_MVMSV)* 会导致最多的完整性和正确性故障。在所有类型中，交易发送方的 *if* 结构及后续语句缺失 *(AL_MITSS)* 和变量名错误 *(A_WVN)*导致潜在完整性故障的比率较高，分别为 3.12% 和 2.75%（该比率为每种缺陷类型导致的故障总数除以执行的事务总数）。

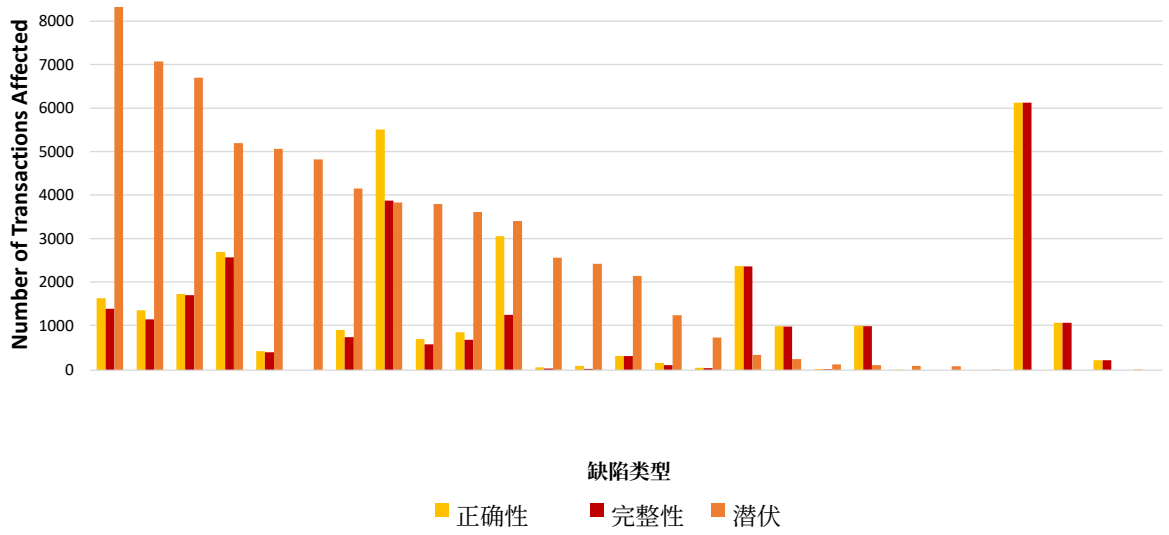


图 9. 引发严重失效的故障类型。

我们还计算了故障合约中注入故障所导致的运行时开销（性能下降），并将其与无故障运行进行比较，比较指标包括 CPU 使用率、内存使用量和交易时间。图 10 概述了结果，其中展示了所有交易中这三类开销值的分布。总体而言，注入故障会使这三项指标都产生一定开销。有些情况下开销较高，具体来说，赋值中使用错误的算术表达式 *(A_WVAE)* 对 CPU 使用率的影响、错误的变量类型 *(A_WVT)* 对交易时间的影响，以及赋值时数字过多 *(A_WVATMD)*对内存使用量的影响。还值得一提的是

某些故障与负开销值相关，因为它们会导致中止 或回滚 交易。

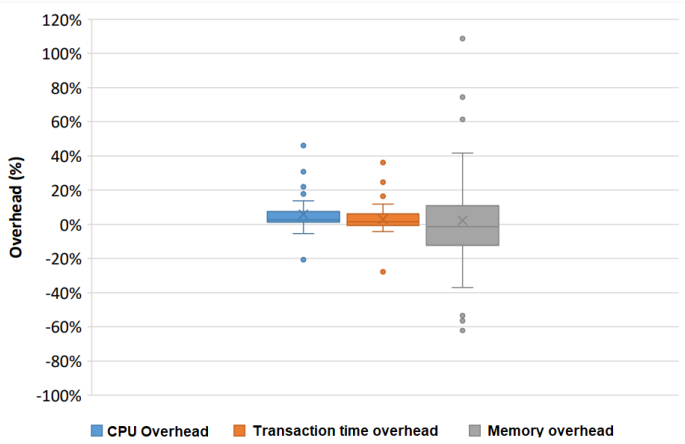


图 10. 故障对 CPU、内存和执行时间造成的开销。

4.2 Study2 的结果——验证工具的有效性

表 3 展示了三种工具（即 Securify、Slither 和 Mythril）声称能够检测的故障。表中列出了这些工具使用的原始名称，并将其映射到我们的故障模型。

图 11 展示了所用三种工具各自检测准确率的概况。具体而言，该图按工具展示了生成的故障合约总数（仅考虑各工具旨在检测的故障类型），以及工具在注入位置检测到问题的合约总数（即真阳性）。

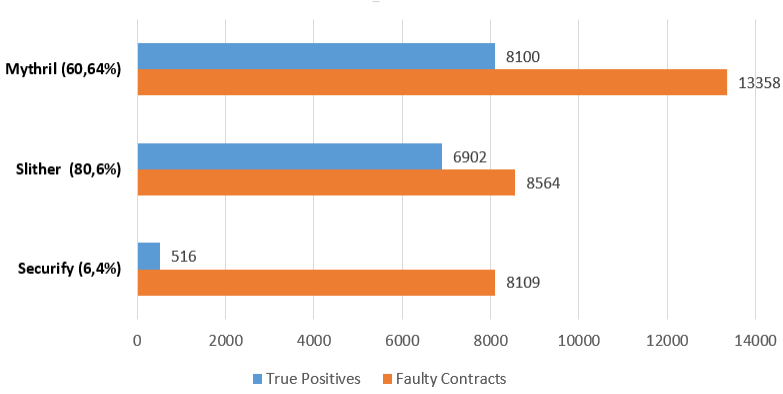


图11. 各工具的检测准确率。

如图 11 所示，Slither 在检测注入缺陷方面更有效（在约 81% 的合约中检测出缺陷），其次是 Mythril，检测准确率约为 61%。Securify 的检测准确率明显较低，仅约为 6%。需要强调的是，这些准确率数据以各工具公布的能力为参照。

需要指出的是，尽管在本研究评估的三种工具中，Slither 似乎是更有效的验证工具，但其生成的警报数量也远高于其他工具。在实验中，仅考虑各工具旨在检测的包含故障的合约，Securify 总共生成了7382 条警报，其中516 条确实是正确警报（即6.99% 的警报属于真阳性）。Mythril 生成了 55090 条警报，其中 8100 条最终被判定为正确警报（14.70%）。Slither 生成了 397236 条警报，其中只有 6902 条是正确警报（1.74%）。

图12展示了验证工具在检测故障合约方面表现出的差异。左侧的图12.a)考虑了工具旨在检测的所有故障（包括仅有一种或两种工具能够检测的故障）。图12.b)仅考虑三种工具共有的故障集合，即三种工具都声称能够检测的故障。

正如我们在图12.a)中所看到的，在11799个故障合约中，只有161个（约1.4%）被全部三个工具正确标记。我们还可以看到，3099个故障合约（约26.3%）同时被Slither和Mythril检测到。其余的故障

表 3 我们的故障模型与工具检测到的缺陷之间的对应关系。

故障	Securify	Slither	Mythril
A_MC	是否调用默认构造函数?	void-cst	SWC-118
A_MCV	-	-	SWC-102
A_MILV	未初始化的局部变量	未初始化的局部变量	SWC-109
一个 MISP_	未初始化的存储	未初始化的存储	SWC-109
A_MISV	未初始化状态变量	未初始化状态	SWC-109
A_WCN	调用默认构造函数?	void-cst	SWC-118
A_WDISV	可更改状态	常量状态	-
A WIS_	-	存储数组	SWC-101
A MIT_	-	先除后乘	SWC-101
一个 WVAA_	-	缺少零值检查	-
A_WVAE	-	-	-
一个 WVATMD_	数字过多	数字过多	SWC-101
A_WVN	被遮蔽的状态变量	状态遮蔽	SWC-119
A WVT_	-	受控数组长度	-
AL ECSWS_	CallInLoop	调用循环	SWC-104
AL_MIIVS	-	-	SWC-123
AL_MITSS	UnrestrictedEtherFlow	未检查的发送	SWC-105
AL WEH_	未处理异常	未检查的底层操作	-
AL WRAR_	-	断言状态变更	SWC-110
CH_MCHAO	-	-	SWC-123
CH MCHGL_	-	高成本循环	SWC-128
CH_MCHSF	不受限制的自毁	有自杀倾向的	SWC-106
CH_MRAIV	-	-	SWC-123
CH_MRATS	-	-	SWC-123
CH_MRIV	-	-	SWC-123
CH_MROIV	-	-	SWC-123
CH_MROTS	-	-	SWC-123
CH_MRTS	-	-	SWC-123
CH WRA_	交易发起方	交易发起方	SWC-115
F继承_	-	缺失继承	SWC-125
F MINHERITANCE_	-	缺失继承	SWC-125
F_MWF	锁定以太币	锁定以太币	-
F WIO_	-	缺失继承	SWC-125
I_MFVM	外部函数	外部函数	-
I MVMSV_	状态变量默认可见性	-	SWC-108
I_WVPF	-	常量函数汇编	-

这些合约由 Slither 或 Mythril 检测到，其中 Mythril 更具优势。尽管 Securify 的检测效果较差，但它实际上能够检测出另外两个工具都无法检测到的 57 个合约中的故障。这清楚地表明，这些工具在故障检测方面具有互补性。在图12.b）中，我们再次观察到这些工具的互补性，不过现在可以看到，Slither 实际上捕获了 Mythril 检测到的大多数故障（在这种情况下，我们将故障范围缩小到三个工具共同检测到的故障集合）。我们还看到，在这种情况下，Securify 并未带来额外的检测价值。

接下来，我们将更详细地考察各工具针对每种故障的能力。图13按故障类型展示了生成的故障合约数量，以及工具标记注入位置存在问题的合约数量（即工具的检测准确率）。

如图13所示，所有故障类型的检测模式似乎相似，只有少数情况例外，即*A_MCV*、*CH_MRAIV*和*CH_MRATS* ，在这些情况下，Mythril能够检测出更多故障合约。对于缺失编译器版本 *(A_MCV)*，大多数故障合约都未被检测到。相比之下，缺陷类型*A_WIS*、*A_WVT*、*AL_WRAR*、*CH_MCHGL*、*CH_MCHSF*和*F_MINTHERITANCE*都能被一个或多个工具完全检测到。

4.3 研究3结果——隐蔽故障的影响

本节聚焦于未被验证工具检测到的故障，并分析其影响。图14展示了按缺陷类型划分、未被任何验证工具检测到的故障合约数量。每个柱形上方的标签表示某一故障类型中未被检测到的故障合约所占的百分比，计算时以针对该故障生成的故障合约总数为基数。总计约有 9% 的合约（15494 个中的 1395 个）未被所有工具检测到。缺失编译器版本这一缺陷类型（*A_MCV*）应该很容易检测（只需在合约开头进行简单检查），但

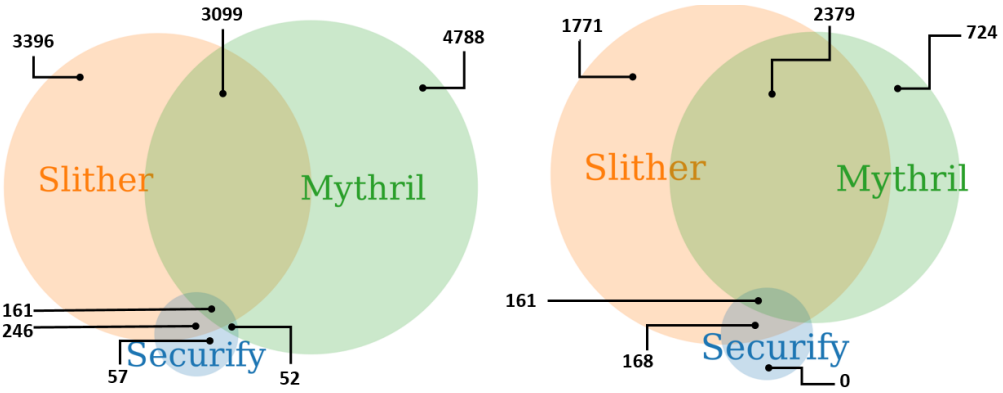


图 12. 工具检测到的缺陷的维恩图。

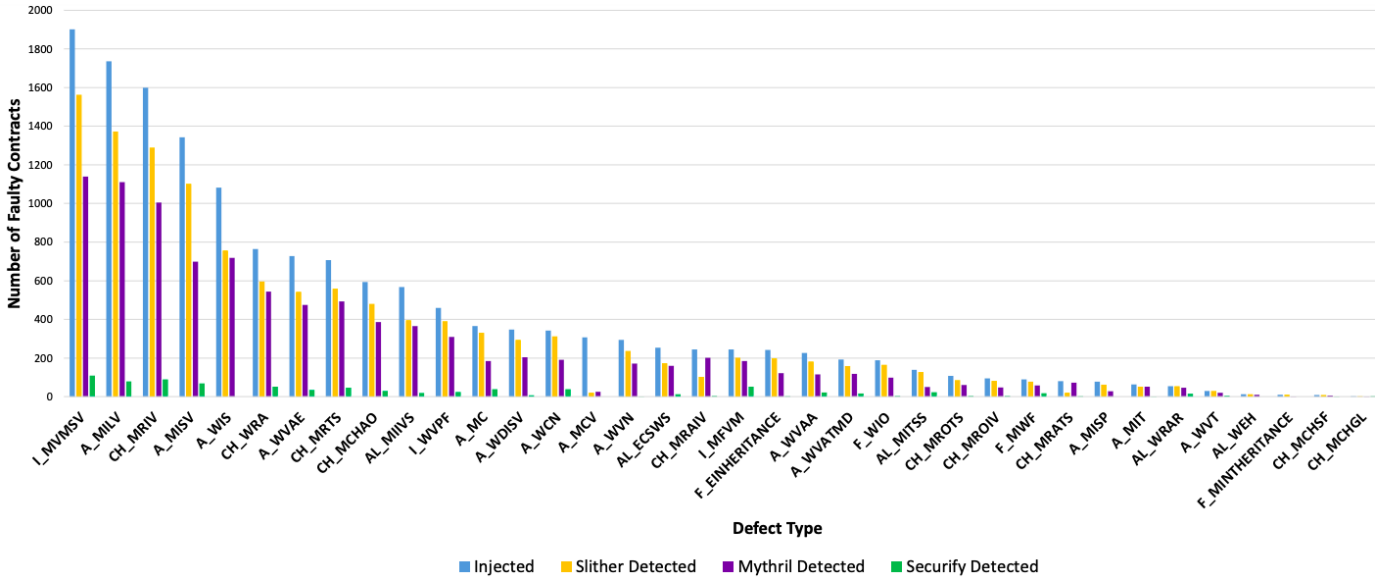


图 13. 验证工具检测到的缺陷。

工具通常无法在大多数情况下检测到它（91.9%）。对于其他缺陷类型，工具的表现往往更好：至少在每 10 个故障合约中，有 9 个注入的故障能被至少一种工具检测到。不过，注入故障的代码位置不同，也会影响工具的检测能力。

为了了解未检测到的缺陷所产生的影响，我们在图 15 中总结了执行未检测到的故障合约的交易所获得的结果。再次，我们观察到，大多数失效属于 回滚失败 和燃料耗尽失败，其次是中止失败。与图 7 中所示的所有交易中的失效分布相比，在隐蔽故障中，不受影响交易的百分比有所下降，导致回滚 和燃料耗尽失效的百分比更高。其他失效的百分比也略有下降。

结果的更详细视图如图 16 所示。在未检测到的缺陷中，*A_MCV* 导致的失效最多。相比之下，未检测到的缺陷类型 *I_MFVM*、*AL_WEH*、*A_WVATMD_2*、*A_MIT*、*AL_ECSWS*、*CH_MROIV*、*F_INHERITANCE*均未导致任何失效。

图 17 深入分析了导致严重失效的未检测到的故障，即 正确性失败、完整性失败和潜在完整性失败。结果表明，即使使用了整套验证工具，残余故障确实仍然存在，并且从区块链的角度来看，会导致严重问题。结果还表明，这些严重缺陷大多属于赋值 或检查类型。

在图17所示的所有类型故障中，*A_MCV*、*CH_MROTS*、*CH_WRA* 的严重程度较低，因为它们不会导致潜伏失效，或只导致少量潜伏失效。在合约*A_WVN* 和*A_WVAE* 中，这些问题可视为智能合约中最严重的问题，因为二者在大多数情况下都会导致潜伏失效。前者有约9.5%的时间未被检测到，后者有约6.0%的时间未被检测到（参见图14）。

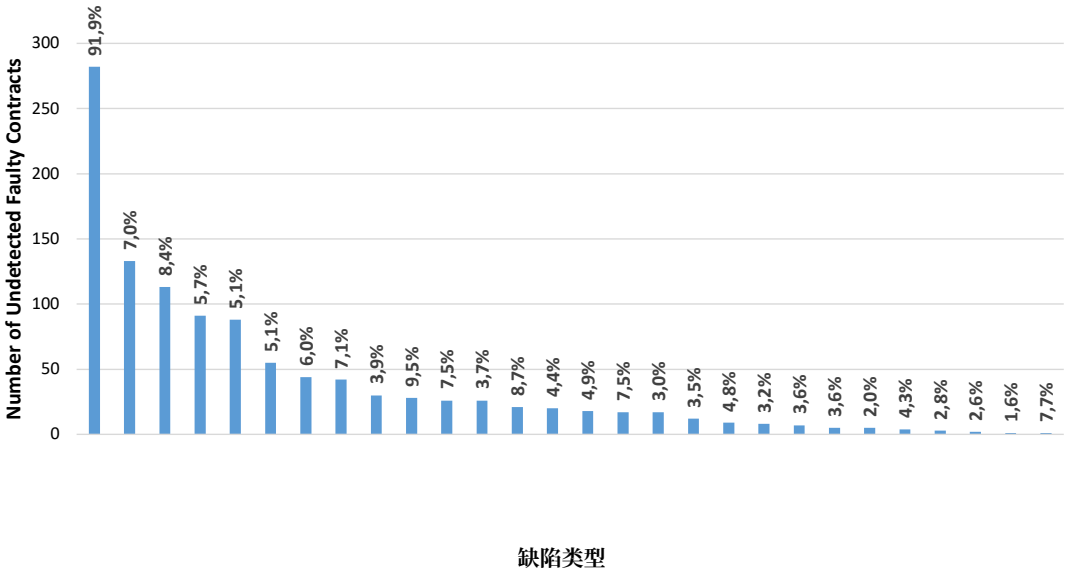


图 14. 未被任何工具检测到的故障合约。

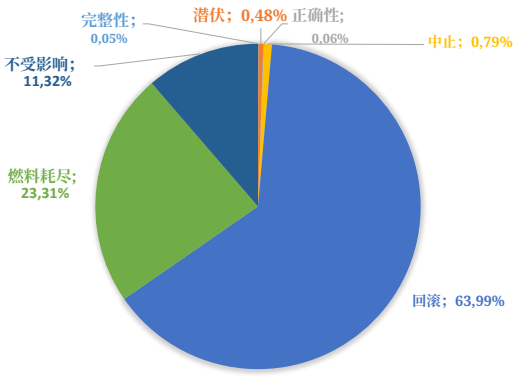


图 15. 未检测到的故障影响的总体视图。

4.4 主要发现

本节重点介绍实验评估的主要发现如下：

- 作为与故障注入过程相关的一般观察，我们发现，几类故障更容易被注入，即缺失状态变量可见性修饰符 *(I_MVMSV)*（1902 次）、缺失局部变量初始化 *(A_MILV)*（1736 次）以及缺失对输入值的 *require* 检查 *(CH_MRIV)*（1599 次），它们导致了更多故障合约。这意味着，注入这些故障所需的条件更频繁地出现。

- 四分之一的故障合约中未观察到任何失效，而在约一半的故障合约中检测到了回滚失败，近五分之一的故障合约中观察到了燃料耗尽失效。这两类失效是这些实验中观察到的最常见失效。

- 与较高注入概率相关的故障（即 *I_MVMSV*、*A_MILV* 和 *CH_MRIV*）也是实验期间观察到的大多数回滚失效 和燃料耗尽失效的诱因。

- 故障*CH_MRIV*是最常见的故障之一，也是大多数潜伏失效的成因，而潜伏失效是最严重的失效模式。*CH_MRTS* 和*A_MISV_2* 不如*CH_MRIV* 常见，但它们也是相当数量潜伏失效案例的成因。

- 智能合约验证工具的有效性相当低，结果显示，与生成的大量警报相比，真阳性的数量很少。这证实了相关工作中的类似观察结果。Slither 似乎更擅长检测注入故障（它能够检测出约 81% 的故障合约中的缺陷），但同时也会生成数量庞大的警报（检测到的缺陷仅占所有生成警报的 1.74%）。如果考虑警报数量，能够检测出约 61% 故障合约中缺陷的 Mythril 是一个值得关注的选择

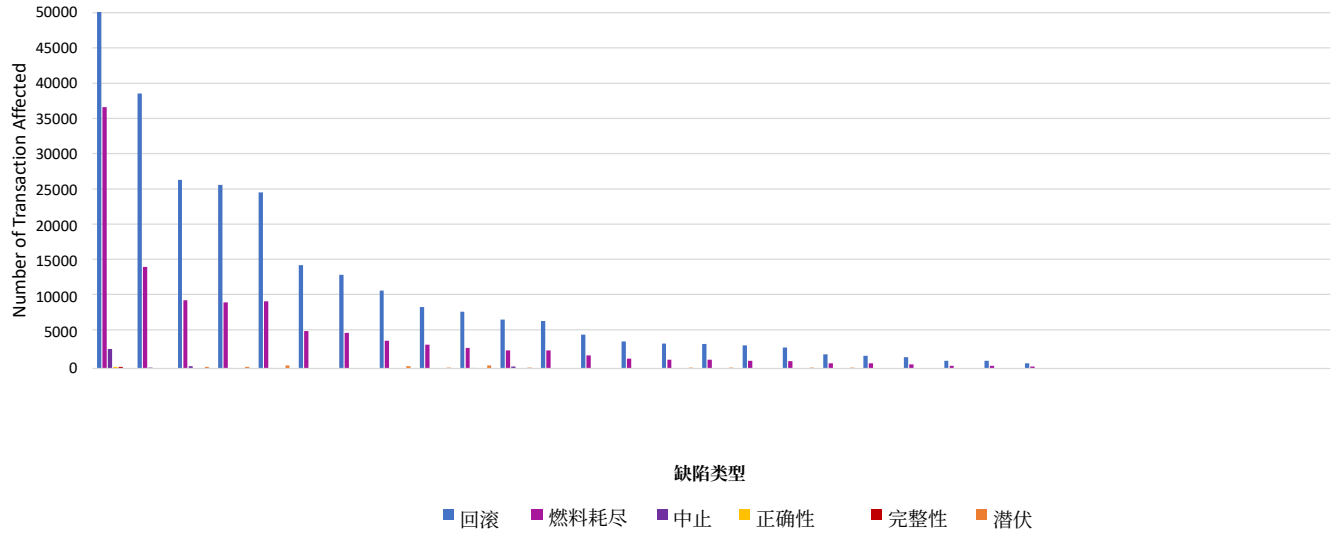


图 16. 故障对未检测到的缺陷的影响。

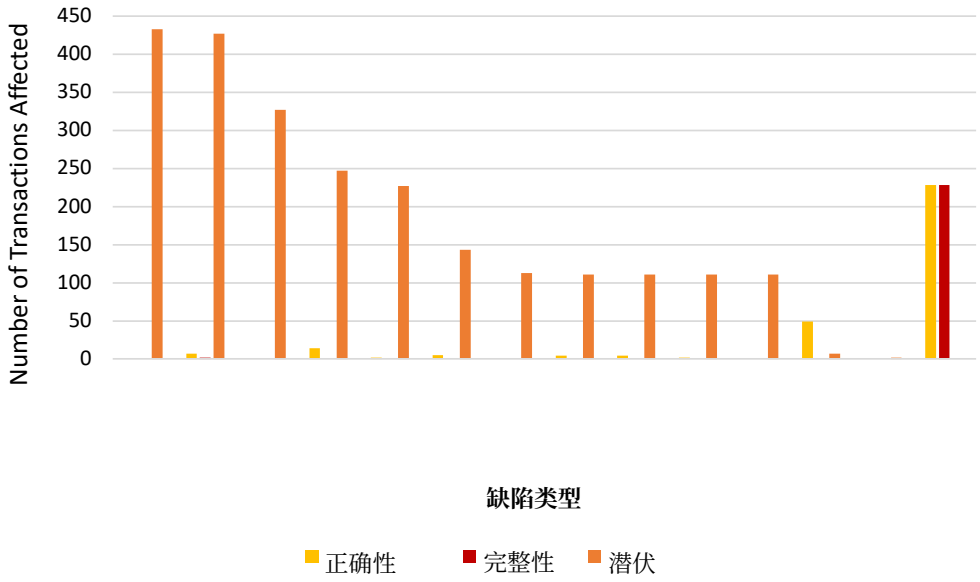


图 17. 故障对未检测到的缺陷造成的关键影响。

生成的（检测到的缺陷仅占所有生成警报的14.70%）。Securify显示能够检测出约6.4%的有故障合约。

- Mythril 和Slither 清楚地展现了互补的能力，尽管它们也共同检测出了许多故障合约。Securify 能够检测出其余工具无法捕获的故障，但规模非常小。因此，开发一种能够利用所涉及的不同技术的工具，可能是提升检测能力的一条途径。

- 使用*A_WIS*、*A_WVT*、*AL_WRAR*、*CH_MCHGL*、*CH_MCHSF*、*F_MINTHERITANCE* 生成的故障合约均至少被一种工具检测到。相反，这些工具大多无法检测缺失编译器版本 *(A_MCV)*。
- 导致最严重失效的故障要么属于赋值 缺陷类型，要么属于检查 缺陷类型。
- 故障对 CPU、内存和交易时间的总体影响相对较小（即 2% 到 6%），不过也存在一些令人担忧的情况：某些故障造成的影响远超正常范围，个别情况下甚至使参考值翻倍（例如内存开销）。
- 就 隐蔽故障 （见第 4.3 节）而言，近四分之三的故障类型（36 种中的 28 种）未被检测到，并且与严重失效（即正确性、完整性和潜伏性）相关。
- *A_WVN*、*A_WVAE*和*CH_MCHAO* 是最严重的问题之一，因为它们共同导致约50%的所有潜伏우우우우우우

未被检测到的故障合约中的交易失效。

- 对 CPU、内存使用量和交易时间的影响总体上并不显著，尽管故障合约中存在*A_MC*和*A_WCN* 分别会导致内存使用量增加约 3 倍和 30 倍。

- 总体而言，关注本研究中被认定为难以检测的缺陷类型，或许有助于提升未来验证工具的检测能力。此外，针对每个故障，更细致地分析工具为何无法在不同代码位置检测到同一故障，对于提升检测能力至关重要。

4.5 对有效性的威胁

本节介绍了本工作的有效性面临的威胁，并讨论了缓解策略。我们首先要指出，所使用的故障模型并未包含所有可能的故障。例如，本工作未使用重入故障，以及其他已知会影响智能合约的故障。这可能会限制对影响和工具有效性的评估，并导致对现实情况中影响和检测有效性的认知产生偏差。无论如何，检测工具及其影响都不能忽视所选故障。在这一限制下，我们确实尝试最终为每种不同类型的故障至少提供一个具有代表性的示例。未来的工作将采用更完整的故障模型，并实现数量更多的不同故障。

考虑到某些故障可能只有在非常特定的输入序列下才会触发，从而掩盖本可能出现的一些有趣失效，因此生成工作负载的方式可能并非最佳选择。此外，Solidity 智能合约的特性可能导致某些调用按规范失败（例如，只有部分地址拥有在智能合约中执行交易的授权和能力）。尽管如此，我们只分析和比较基准参考运行中被判定为成功的交易，以确保故障参考运行确实造成了影响。

所选工具的数量相当少，可能无法全面反映智能合约验证工具的情况。此外，根据具体目标或限制条件（例如，执行该方法可用的时间和资源），也可以使用其他工具；不过，我们选择了文献中经常出现的工具。分析也主要局限于衡量这些工具检测所述漏洞的真正率，这可能无法准确反映工具的能力。尽管如此，我们的目标是让研究结果有助于改进验证工具；研究重点在于注入智能合约故障并生成故障合约，而不论随后使用何种工具进行故障检测。

最后，将所选合约、已实现的故障和所选工具组合起来，可能会导致我们对实际难以发现的故障产生偏颇的认识。尽管如此，考虑到实验规模，我们认为这些选择是合理的；我们还在相关工作中强调了这三个组成部分的存在，以支持其代表性。

5 结论

在这项工作中，我们开展了一系列实验，以展示现实故障可能对区块链系统的可靠性产生的影响。我们使用故障检测工具来了解哪些故障可能逃过检测，以及它们是否会导致区块链系统在运行时发生故障，或者会以何种方式导致其发生故障。未来，我们计划扩展故障集的实现，并使用规模更大、更多样化（在运行特征方面）的验证工具集。基于本文奠定的基础，我们未来的研究方向之一是正式定义一个基准测试，用于评估和比较智能合约漏洞检测工具的有效性。

参考文献

[1] Hyperledger Caliper。网址 https://hyperledger.github.io/caliper/。[2] Sefa Akca、Ajitha Rajan 和 Chao Peng。SolAnalyser：用于分析和测试智能合约的框架。载于*2019*年第*26*届亚太软件工程会议（*APSEC*），第 482–489 页，马来西亚布城，2019年12月。IEEE。ISBN 978-1-7281-4648-5。DOI：10.1109/ APSEC48747.2019.00071。网址 https://ieeexplore.ieee.org/document/8945725/。[3] Maher Alharby 和 Aad van Moorsel。基于区块链的智能合约：系统性映射研究。载于计算机科学与信息技术（*CS & IT*），第 125–140 页，迪拜，2017年8月。学术与产业研究合作中心（AIRCC）。ISBN 9781921987700。DOI：10.5121/csit.2017.71011。网址 http://airccj.org/CSCP/vol7/csit77211.pdf。[4] Andreas Antonopoulos 和 Gavin Wood。精通以太坊：构建智能合约和去中心化应用。O'Reilly Media, Inc.，2018年。ISBN 978-1- 4919-7191-8。[5] Roberto Baldoni、Emilio Coppa、Daniele Cono D'elia、Camil Demetrescu 和 Irene Finocchi。符号执行技术综述。*ACM Computing Surveys*，51(3):1–39，2019年5月。ISSN 0360-0300。DOI：10.1145/3182657。网址 https://dl.acm.org/doi/10.1145/3182657。[6] Karthikeyan Bhargavan、Antoine Delignat-Lavaud、Cédric Fournet、Anitha Gollamudi、Georges Gonthier、Nadim Kobeissi、Natalia Kulatova、Aseem Rastogi、Thomas Sibut-Pinote、Nikhil Swamy 和 Santiago Zanella-Béguelin。智能合约的形式化验证。载于*2016*年*ACM*编程语言与安全分析研讨会论文集，第 91–96 页，美国纽约州纽约，2016年10月。ACM。ISBN 9781450345743。DOI：10.1145/2993600.2993611。网址 https://dl.acm.org/doi/10.1145/2993600.2993611。[7] Jeeta Ann Chacko、Ruben Mayer 和 Hans-Arno Jacobsen。我的区块链交易为何失败？载于*2021*年国际数据管理会议论文集，第 221–234 页，美国纽约州纽约，2021年6月。ACM。ISBN 9781450383431。DOI：10.1145/3448016.3452823。[8] Chen Chen、Baojiang Cui、Jinxin Ma、Runpu Wu、Jianchao Guo 和 Wenqian Liu。模糊测试技术的系统性综述。计算机与安全，75:118–137，2018年6月。ISSN 01674048。DOI：10.1016/j.cose.2018.02.002。网址 https://linkinghub.elsevier.com/retrieve/pii/ S0167404818300658。[9] Y Chen、F Ma、Y Zhou、Y Jiang、T Chen 和 J Sun。Tyr：利用行为分歧模型发现区块链系统中的共识失败漏洞。载于*2023*年*IEEE*安全与隐私研讨会（*SP*）（*SP*），第 1186–1201 页，美国加利福尼亚州洛斯阿拉米托斯，2023年。IEEE Computer Society。DOI：10.1109/SP46215.2023.00068。网址 https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00068。[10] B. Chess 和 G. McGraw。面向安全性的静态分析。*IEEE*安全与隐私杂志，2(6):76–79，2004年11月。ISSN 1540-7993。DOI：10.1109/MSP.2004.111。

[11] Jaeseung Choi、Doyeon Kim、Soomin Kim、Gustavo Grieco、Alex Groce 和 Sang KilCha。《SMARTIAN：通过静态与动态数据流分析增强智能合约模糊测试》。载于《2021年第36届IEEE/ACM自动化软件工程国际会议》（ASE），第227–239页。IEEE，2021年11月。ISBN 978-1-6654-0337-5。DOI：10.1109/ASE51524.2021.9678888。URL https://ieeexplore.ieee.org/document/ 9678888/。[12] Wonseok Choi 和 James Won-KiHong。《使用Hyperledger Caliper对以太坊私有网络和测试网络进行性能评估》。载于《2021年第22届亚太网络运营与管理研讨会》（APNOMS），第325–329页。IEEE，2021年9月。ISBN 978-4-88552-332-8。DOI：10.23919/APNOMS52696.2021.9562684。URL https://ieeexplore.ieee.org/document/9562684/。[13] Consensys。Mythril，2021年。URL https://github.com/ConsenSys/mythril。[14] PatrickCousot。《抽象解释原理》。麻省理工学院出版社，2021年9月。[15] Bruno Dias、Naghmeh Ivaki和NunoLaranjeiro。《智能合约验证工具有效性的实证评估》。载于《第26届IEEE太平洋沿岸可信计算国际研讨会》（PRDC 2021），澳大利亚珀斯，2021年。[16] Thomas Durieux、João F Ferreira、Rui Abreu 和 PedroCruz。《对47,587个以太坊智能合约的自动化分析工具进行实证评审》。载于《第42届ACM/IEEE国际软件工程会议》，ICSE'20，第530–541页，美国纽约州纽约市，2020年。计算机协会。ISBN 9781450371216。DOI：10.1145/3377811.3380364。URLhttps://doi.org/10.1145/3377811. 3380364。[17] Josselin Feist、Gustavo Grieco 和AlexGroce。《Slither：用于智能合约的静态分析框架》。载于《2019年IEEE/ACM区块链软件工程新兴趋势第二届国际研讨会》（WETSEB），WETSEB '19，第8–15页，加拿大魁北克省蒙特利尔，2019年5月。IEEE。ISBN 978-1-7281-2257-1。DOI：10.1109/WETSEB.2019.00008。URL https://doi.org/10.1109/WETSEB.2019.00008。[18] João F. Ferreira、Pedro Cruz、Thomas Durieux 和 Rui Abreu。《SmartBugs》。载于《第35届IEEE/ACM自动化软件工程国际会议论文集》，第1349–1352页，美国纽约州纽约市，2020年12月。ACM。ISBN 9781450367684。DOI：10.1145/3324884.3415298。URL https://dl.acm.org/doi/10.1145/3324884.3415298。[19] Jose Fonseca、Marco Vieira和 Henrique Madeira。《针对SQL注入和跨站脚本攻击测试并比较Web漏洞扫描工具》。载于《第13届太平洋沿岸可信计算国际研讨会》（PRDC 2007），第365–372页。IEEE，2007年12月。ISBN 0-7695-3054-0。DOI：10.1109/PRDC.2007.55。[20] Asem Ghaleb 和 KarthikPattabiraman。《智能合约分析工具的有效性如何？使用缺陷注入评估智能合约静态分析工具》。载于《第29届ACM SIGSOFT国际软件测试与分析研讨会论文集》，ISSTA 2020，第415–427页，美国纽约州纽约市，2020年。计算机协会。ISBN 9781450380089。DOI：10.1145/3395363.3397385。URLhttps://doi.org/10.1145/3395363.3397385。[21] ÁkosHajdu、NaghmehIvaki、Imre Kocsis、Attila Klenik、László Gönczy、Nuno。Laranjeiro、HenriqueMadeira和 AndrásPataricza。《使用故障注入评估存在有缺陷智能合约时的区块链系统》。IEEE Access，8:190760–190783，2020年。ISSN 2169-3536。DOI：10.1109/ACCESS.2020.3032239。[22] DominikHarz 和William JKnottenbelt。《迈向更安全的智能合约：语言和验证方法综述》，2018年。URLhttp://arxiv.org/abs/1809.09805。[23] IBM。《软件设计和代码的正交缺陷分类 v 5.2》，2013年。URL https://researcher.watson.ibm.com/researcher/files/ us-pasanth/ODC-5-2.pdf。[24] Ivano Irrera、Joao Duraes、Marco Vieira和 Henrique Madeira。《通过注入真实软件故障识别用于故障预测的最佳变量》。载于《2010年第16届IEEE太平洋沿岸可信计算国际研讨会》，第3–10页。IEEE，2010年12月。ISBN 978-1-4244-8975-6。DOI：10.1109/PRDC.2010.51。[25] Sukrit Kalra、Seep Goel、Mohan Dhawan 和 SubodhSharma。《ZEUS：分析智能合约安全性》。载于《2018年网络与分布式系统安全研讨会论文集》，第2018–02页，弗吉尼亚州雷斯顿，2018年。互联网协会。ISBN1-891562-49-5。DOI：10.14722/ndss.2018.23082。URL https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-1_Kalra_paper.pdf。[26] LoiLuu、Duc-Hiep Chu、Hrishi Olickel、Prateek Saxena 和 Aquinas Hobor。《让智能合约更加智能》。载于《2016年ACM SIGSAC计算机与通信安全会议论文集》，CCS'16，第254–269页，美国纽约州纽约市，2016年。计算机协会。ISBN 9781450341394。DOI：10.1145/2976749.2978309。URLhttps://doi.org/10.1145/2976749.2978309。[27] Edmund M. Clarke、Edmund M. Jr.Clarke 和 OrnaGrumberg。《模型检测》。麻省理工学院出版社，1999年。ISBN 0262032708。[28] Henrique Marques、NunoLaranjeiro和Jorge Bernardino。《在Python应用程序中注入软件故障》。实证软件工程，27(1):20，2022年1月。ISSN 1382-3256。DOI：10.1007/s10664-021-10047-9。URL https://link.springer.com/10.1007/s10664-021-10047-9。[29] Marco Mazzoni、AntonioCorradi和Vincenzo Di Nicola。《面向金融应用的许可型区块链性能评估：ConsenSys Quorum案例研究》。区块链：研究与应用，3(1):100026，2022年3月。ISSN20967209。DOI：10.1016/j.bcra.2021. 100026。URL https://linkinghub.elsevier.com/retrieve/pii/S209672092100021X。[30] Mei-ChenHsueh、T.K. Tsai 和 R.K. Iyer。《故障注入技术和工具》。Computer，30(4):75–82，1997年4月。ISSN00189162。DOI：10.1109/2.585157。[31] Pouyan Momeni、Yu Wang 和 Reza Samavi。《用于智能合约安全分析的机器学习模型》。载于《2019年第17届隐私、安全与信任国际会议》（PST），第1–6页。IEEE，2019年8月。ISBN 978-1-7281-3265-5。DOI：10.1109/PST47121.2019.8949045。URL https://ieeexplore.ieee.org/document/8949045/。[32] Glenford J. Myers、Tom Badgett 和 CoreySandler，编。《软件测试艺术》。Wiley，2012年1月。ISBN 9781118031964。DOI：10.1002/9781119202486。URLhttps://onlinelibrary.wiley.com/doi/book/10.1002/9781119202486。[33] MythX。SWC，2020年。URL https://swcregistry.io/。[34] Roberto Natella、Domenico Cotroneo和 HenriqueS Madeira。《使用软件故障注入评估可信性：综述》。ACM计算调查，48(3)，2016年2月。ISSN 0360-0300。DOI：10.1145/2841425。URLhttps://doi.org/10.1145/2841425。[35] NCC Group。DASP，2019年。URL https://dasp.co/。[36] Reza Parizi、Ali Dehghantanha、Kim-KwangRaymond Choo 和AmritrajSingh。《区块链上智能合约自动化安全测试的实证漏洞分析》。载于《第28届计算机科学与软件工程年度国际会议论文集》，CASCON '18，第103–113页，美国，2018年。IBM公司。[37] Kai Peng、Meijun Li、Haojun Huang、Chen Wang、Shaohua Wan 和Kim-KwangRaymondChoo。《物联网中智能合约的安全挑战与机遇：综述》。IEEE物联网期刊，8(15):12004–12020，2021年8月。ISSN2327-4662。DOI：10.1109/JIOT.2021.3074544。[38] Purathani Praitheeshan、Lei Pan、JiangshanYu、Joseph K. Liu和 RobinDoss。《以太坊智能合约漏洞的安全分析方法：综述》。CoRR，abs/1908.08605，2019年。URLhttp://arxiv.org/abs/1908.08605。[39] Protofire。Solhint。URL https://protofire.github.io/solhint/。[40] Meng Ren、Zijing Yin、Fuchen Ma、Zhenyang Xu、Yu Jiang、Chengnian Sun、Huizhong Li 和Yan Cai。《智能合约测试的实证评估：最佳选择是什么？》。载于《第30届ACM SIGSOFT国际软件测试与分析研讨会论文集》，第566–579页，美国纽约州纽约市，2021年7月。ACM。ISBN 9781450384599。DOI：10.1145/3460319.3464837。URLhttps://dl.acm.org/doi/10. 1145/3460319.3464837。[41] Xavier Rival，编。《静态分析》，《计算机科学讲义》第9837卷。SpringerBerlin Heidelberg，德国柏林、海德堡，2016年。ISBN 978-3-662-53412-0。DOI：10.1007/978-3-662-53413-7。[42] Johann M. Schumann。《软件工程中的自动定理证明》。SpringerBerlin Heidelberg，德国柏林、海德堡，2001年。ISBN

978-3-642-08759-2。doi：10.1007/978-3-662-22646-9。[43] Erik Seligman、Tom Schubert 和 M V Achutha Kiran Kumar。形式化验证。爱思唯尔，第1版，2015年。ISBN 9780128007273。doi：10.1016/C2013-0-18672-2。[44] So Sunbeom、Hong Seongjoon 和 Oh Hakjoo。Smartest：通过语言模型引导的符号执行，有效搜寻智能合约中的易受攻击的交易序列。载于 第*30*届 *USENIX* 安全研讨会。USENIX 协会，2021年。URL https://www.usenix. org/system/files/sec21-so.pdf。[45] Sergei Tikhomirov、Ekaterina Voskresenskaya、Ivan Ivanitskiy、Ramil Takhaviev、Evgeny Marchenko 和 Yaroslav Alexandrov。SmartCheck：以太坊智能合约的静态分析。载于 区块链软件工程新兴趋势 第*1*届国际研讨会论文集，第9–16页，纽约，纽约州，美国，2018年5月。ACM。ISBN 9781450357265。doi：10.1145/3194113.3194115。URL https://dl.acm.org/ doi/10.1145/3194113.3194115。[46] Petar Tsankov、Andrei Dan、Dana Drachsler-Cohen、Arthur Gervais、Florian Bünzli 和 Martin Vechev。Securify：智能合约的实用安全分析。载于 *2018*年 *ACM SIGSAC* 计算机与通信安全大会论文集，CCS '18，第67–82页，纽约，纽约州，美国，2018年。计算机协会。ISBN 9781450356930。doi：10.1145/3243734.3243780。URL https://doi.org/10.1145/3243734.3243780。[47] Fernando Richter Vidal、Naghmeh Ivaki 和 Nuno Laranjeiro。分析隐蔽故障对区块链可靠性的影响——补充材料，2023年1月。URL https://doi.org/10.5281/zenodo.7589416。[48] Wei Wang、Jingjing Song、Guangquan Xu、Yidong Li、Hao Wang 和 Chunhua Su。ContractWard：以太坊智能合约的自动化漏洞检测模型。*IEEE* 网络科学与工程汇刊，8(2)：1133–1144，2021年4月。ISSN 2327-4697。doi：10.1109/TNSE.2020.2968505。URL https://ieeexplore.ieee.org/document/8967006/。[49] W. Eric Wong，编。新世纪的变异测试。施普林格美国，波士顿，马萨诸塞州，2001年。ISBN 978-1-4419-4888-5。doi：10.1007/ 978-1-4757-5939-6。[50] Zhiwu Xu、Cheng Wen 和 Shengchao Qin。用于检测资源错误的状态污点分析。计算机程序设计科学，162：93–109，2018年9月。ISSN 01676423。doi：10.1016/j.scico.2017.06.010。URL https://linkinghub.elsevier.com/retrieve/pii/S0167642317301314。[51] Dylan Yaga、Peter Mell、Nik Roby 和 Karen Scarfone。区块链技术概览。技术报告，美国国家标准与技术研究院，盖瑟斯堡，马里兰州，2018年10月。URL https://doi.org/10.6028/NIST.IR.8202。[52] Jiaming Ye、Mingliang Ma、Tianyong Peng、Yun Peng 和 Yinxing Xue。面向智能合约的漏洞基准测试自动生成。载于 *2019*年 *IEEE* 软件测试、验证与确认国际会议研讨会（*ICSTW*），第184–187页，2019年。doi：10.1109/ICSTW.2019.00049。[53] Lejun Zhang、Jinlong Wang、Weizheng Wang、Zilong Jin、Chunhui Zhao、Zhennao Cai 和 Huiling Chen。基于信息图和集成学习的新型智能合约漏洞检测方法。传感器，22(9)：3581，2022年5月。ISSN 1424-8220。doi：10.3390/s22093581。[54] Pengcheng Zhang、Feng Xiao 和 Xiapu Luo。SolidityCheck：通过正则表达式快速检测智能合约问题。2019年。URL https://arxiv.org/abs/1911.09425。[55] Yuan Zhuang、Zhenguang Liu、Peng Qian、Qi Liu、Xiang Wang 和 Qinming He。使用图神经网络进行智能合约漏洞检测。载于 第二十九届人工智能国际联合会议论文集，第3283–3290页，加利福尼亚，2020年7月。人工智能国际联合会议组织。ISBN 978-0-9992411-6-5。doi：10.24963/ijcai.2020/454。



Fernando Vidal 是葡萄牙科英布拉大学的一名博士生，研究兴趣与区块链技术相关。Fernando 曾在国际会议上发表论文，介绍他在区块链技术方面的一些研究发现，例如智能合约中的漏洞和撤销。此外，Fernando 受邀担任 Advances in Science, Technology and Engineering Systems Journal (ASTESJ) 杂志和 IEEE Potentials 的审稿人，负责评审有关区块链的投稿。Fernando 还通过咨询，将所学知识应用于帮助企业实施区块链技术。



Naghmeh Ivaki 在葡萄牙科英布拉大学获得博士学位。目前，她受聘担任助理教授，并是科英布拉大学信息工程系信息学与系统中心（CISUC）软件与系统工程组（SSE）的正式成员。她的专业领域为信息工程，尤其专注于计算机系统的可信性、安全性和可信性。在其专业领域，她已发表30多篇同行评审出版物，并参与了多个国内外研究项目。



Nuno Laranjeiro 获得了科英布拉大学（葡萄牙）颁发的博士学位，目前在该校担任助理教- 授。他的研究重点是可信且安全的软件服务，目前领导科英布拉大学信息学与系统中心（CISUC）的软件与系统工程组。他的研究兴趣包括实验性可信性评估、故障注入、软件服务的健壮性、Web 服务互操作性、服务安全性以及企业应用集成。作为作者、审稿人和程序委员会成员，他参与了可信性与服务计算领域顶级会议和期刊的工作。Nuno 参与组织了多项国际活动，包括多届国际软件可靠性工程研讨会、与IEEE 世界服务大会联合组织的可信与安全服务研讨会/专题（担任主要主席）。他参与了多个国际研究项目，包括若干 H2020 项目（例如 ADVANCE、DEVASSES、ATMOSPHERE、EUBrasilCloudFORUM）和 FP7项目（CRITICALSTEP、CECRIS）；目前，他主要致力于开发提高云系统可靠性的新技术，开发相关技术

用于评估区块链智能合约的可靠性和安全性，并运用机器学习技术检测软件故障和漏洞。