ContractGuard：面向以太坊区块链智能合约的入侵检测系统

赵淦森[1,2,3]，谢智健[1,2,3]，王欣明[1,2,3,4,5]，何嘉浩[1,2,3]，张成志[5]，
林成创[1,2,3]，Ziheng Zhou[1,3,6]，陈冰川[3,7]，Chunming Rong[8]

（1. 华南师范大学计算机学院，广东 广州 510000；
2. 广州市云计算安全与测评技术重点实验室，广东 广州 510000；
3. 华南师范大学唯链区块链技术与应用联合实验室，广东 广州 510000；
4. 拉卡拉集团，北京 100080；
5. 香港科技大学，中国 香港 999077；
6. VeChain Foundation Limited, Singapore 238463；
7. 广东财经大学，广东 广州 510000；
8. Stavanger University, Norway Stavanger 4036）

摘　要：以太坊智能合约本质上是一种在网络上由相互间没有信任关系的节点共同执行的已被双方认证程序。目前，大量的智能合约被用于管理数字资产，使智能合约成为黑客的重要攻击对象。常见的攻击方法是通过利用智能合约的漏洞来实现特定操作的入侵攻击。ContractGuard 是首次提出面向以太坊区块链智能合约的入侵检测系统，它能检测智能合约的潜在攻击行为。ContractGuard 的入侵检测主要依赖检测潜在攻击可能引发的异常控制流来实现。由于智能合约运行在去中心化的环境以及在高度受限的环境中运行，现有的IDS技术或者工具等以外部拦截形式的部署架构不适合于以太坊智能合约。为了解决这些问题，通过设计一个嵌入式的架构，实现了把 ContractGuard 直接嵌入智能合约的执行代码中，作为智能合约的一部分。在运行时刻，ContractGuard 通过相应的 context-tagged 无环路径来实现入侵检测，从而保护智能合约。由于嵌入了额外的代码，ContractGuard 一定程度上会增加智能合约的部署开销与运行开销，为了降低这两方面的开销，基于以太坊智能合约的特性对 ContractGuard 进行优化。实验结果显示，可有效地检测 83%的异常行为，其部署开销仅增加了 36.14%，运行开销仅增加了 28.17%。

关键词：区块链；以太坊智能合约；入侵检测系统；异常检测

中图分类号：TP393

文献标识码：A

doi: 10.11959/j.issn.2096−109x.2020025

收稿日期：2020−01−15；**修回日期：**2020−03−25

通信作者：王欣明，wangxinming@lakala.com；张成志，scc@cse.ust.hk

基金项目：中华人民共和国香港特别行政区政府资金资助项目（No.RGC/GRF16202917）；国家重点研发计划基金资助项目（No.2018YFB1404402），广东省重点研发计划基金资助项目（No.2019B010137003）；广东省科技计划基金资助项目（No.2016B030305006, No.2018A07071702, No.201804010314, No.2012224-12）；唯链基金会资金资助项目（No.SCNU-2018-01）；广东省教育厅特色创新项目（自然科学）（No.2017KTSCX074）

Foundation Items: HKSAR (No.RGC/GRF16202917), The National Key R&D Program of China (No.2018YFB1404402), Guangdong Science &Technology Fund (No.2019B010137003), Guangzhou Science & Technology Fund (No.2016B030305006, No.2018A07071702, No.201804010314, No.2012224-12), VeChain Foundation (No.SCNU-2018-01), Guangdong Provincial Department of Education Characteristic Innovation Project (Natural Science) (No. 2017KTSCX074)

论文引用格式：赵淦森, 谢智健, 王欣明, 等. ContractGuard: 面向以太坊区块链智能合约的入侵检测系统[J]. 网络与信息安全学报, 2020, 6(2): 35-55.

ZHAO G S, XIE Z J, WANG X M, et al. ContractGuard: defend Ethereum smart contract with embedded intrusion detection[J]. Chinese Journal of Network and Information Security, 2020, 6(2): 35-55.

ContractGuard: defend Ethereum smart contract with embedded intrusion detection

ZHAO Gansen[1,2,3], XIE Zhijian[1,2,3], WANG Xinming[1,2,3,4,5], HE Jiahao[1,2,3], ZHANG Chengzhi[5], LIN Chengchuang[1,2,3], Ziheng ZHOU[1,3,6], CHEN Bingchuan[3,7], Chunming RONG[8]

（1. South China Normal University School of Computer Science, Guangzhou 510000, China
2. Guangzhou Key Laboratory of Cloud Computing Security and Assessment Technology, Guangzhou 510000, China
3. VeChain blockchain technology and application joint laboratory, Guangzhou 510000, China
4. Lakala Payment Company Limited, Beijing 100080, China
5. HK University of Science and Technology, Hong Kong 999077, China
6. VeChain Foundation Limited, Singapore 238463
7. Guangdong university of finance and economics, Guangzhou 510000, China
8. Stavanger University, Stavanger 4036, Norway）

Abstract: Ethereum smart contracts are programs that can be collectively executed by a network of mutually untrusted nodes. Smart contracts handle and transfer assets of values, offering strong incentives for malicious attacks. Intrusion attacks are a popular type of malicious attacks. ContractGuard, the first intrusion detection system (IDS) was proposed to defend Ethereum smart contracts against such attacks. Like IDSs for conventional programs, ContractGuard detects intrusion attempts as abnormal control flow. However, existing IDS techniques or tools are inapplicable to Ethereum smart contracts due to Ethereum's decentralized nature and its highly restrictive execution environment. To address these issues, ContractGuard was designed by embedding it in the contracts. At runtime, ContractGuard protects the smart contract by monitoring the context-tagged acyclic path of the smart contract. As ContractGuard involves deployment overhead and deployment overhead. It was optimized under the Ethereum Gas-oriented performance model to reduce the overheads. The experimental results show that this work can effectively detect 83% of vulnerabilities, ContractGuard only adds to 36.14% of the deployment overhead and 28.27% of the runtime overhead.

Key words: blockchain, Ethereum smart contract, intrusion detection system, anomaly detection

1　引言

自比特币[1]出现以来，区块链技术作为分布式账本技术中的一种形式，引起了广泛的关注并被推广应用。区块链系统中的参与者共同运行共识协议以维护和保护链上共享账本的数据。区块链在应用过程中逐步超越了单纯的记账和支付，已经扩展为允许以智能合约形式进行可编程交易的平台级技术。以太坊智能合约[2]是一种可以由互不信任的网络节点共同执行的程序，该程序通过特定的共识协议以数字方式强制节点执行：任何节点甚至智能合约的创建者均无法修改智能合约代码、影响代码执行。

由于用户委托智能合约来处理和转让有价值的资产，吸引了大量的黑客关注与大量的恶意攻击。智能合约受到恶意攻击往往导致比常规网络系统上的攻击更加严重的后果，因为智能合约一旦部署在区块链上后不可更改，智能合约的管理者无法对智能合约维护以更正漏洞保护智能合约。以太坊上的智能合约已经发生了许多有据可查的攻击实例[3]。

在已有文献中，静态分析存在大量的研究工作，这些分析工具可以在智能合约部署之前进行漏洞检测[4-11]。但是，相关的调研中目前尚没有发现有文献报道可以在智能合约部署后保护智能合约的相关安全研究成果和进展。

对于传统系统而言，管理员早就认识到，即使开发人员尽了最大的努力来修复缺陷，漏洞仍然会存在。入侵检测系统（IDS）[12]通常是保护已部署系统免受安全攻击的主要手段。根据入侵

检测的方式，IDS 分为基于签名的 IDS 和基于异常的 IDS[12]两种类型。典型的基于异常的 IDS 是对动态程序行为与正常程序行为进行监视，并在检测到异常时发出警报。正常程序行为是通过机器训练和分析来学习的。

本文提出了嵌入式的 IDS 架构 ContractGuard 实现区块链智能合约的安全防护，突破了智能合约在运行时无法获得外在安全监管和防护的限制。图 1 展示了该系统的基本思路，IDS 被嵌入智能合约中，作为智能合约的一部分与智能合约一起在虚拟机中运行。当内嵌的 IDS 检测到智能合约运行时的异常行为，其将回滚所有针对该智能合约状态的更改，并向系统管理员发出警报。这项技术可以弥补因漏洞而带来的损失。例如，在著名的 The DAO 事件中[13]，如果管理员应用了本文所提出的 ContractGuard 技术来保护其智能合约，就可以防止该攻击。该攻击触发了智能合约在内部测试过程中未发现的异常控制流，即可重入控制流。同样地，由于攻击触发了带有异常的库代码，出现了异常的控制流，因此本文所提出的 ContractGuard 也可以防止奇偶校验 Multisig Wallet[14]事件的发生。实验表明，本文所设计的 ContractGuard 可以在发生漏洞攻击时为系统管理员提供有效的防御手段。

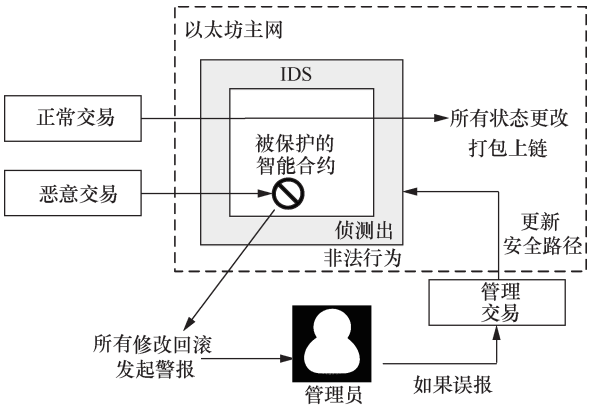


图 1　智能合约入侵检测系统的基本思路
Figure 1　The idea of IDS for smart contract

一般而言，在部署和装配 IDS 的情景下，区块链系统有众多的约束，使传统 IDS 架构并不适用于智能合约，原因如下。

1) 由于以太坊本质上是一个去中心化的应用平台[2]，目前并无现成的技术架构可以实现对一个动态的、开放的、去中心化的区块链网络进行边界或主机的全覆盖。因此，传统的基于主机和基于网络的 IDS[12]技术均不适用。

2) 以太坊智能合约运行在被称为以太坊虚拟机（EVM）的高度受限环境中，该环境缺乏众多常见的功能，这些功能包括硬件寄存器[15]、调用堆栈遍历[16]和事件钩子[17]等，导致传统的 IDS 实现方法在 EVM 中无法实现。此外，EVM 规定了每个智能合约不能大于 24 576 B，使智能合约在设计和实现上其功能和逻辑复杂度受到限制。

3) 智能合约在部署后不能进行修改，在运行时刻不受任何的外部管理和控制。简单而言，智能合约一旦部署后，其不接受任何外部的管理和干预，包括代码维护和补丁以及运行的终止和监控等。传统 IDS 架构需要从外部对受保护对象进行监测、获取运行状态以及对其运行进行干预，明显不适用于智能合约的情景。

4) 以太坊的运行模型与传统系统的运行模型存在着本质上的区别，特别是在成本和性能方面。常规 IDS 优化的目标是提升运行效率、提高响应速度。然而，对于以太坊智能合约来说，执行时间是无关紧要的，重要的是执行所需要的 Gas 开销[18]，这部分开销主要是部署和运行时产生的，因此，这需要以成本为导向的开销优化，这点与常规 IDS 优化思路是不同的。

基于区块链平台，本文提出了嵌入智能合约代码中的基于异常的 IDS 架构 ContractGuard。ContractGuard 作为区块链上第一个基于异常的 IDS，其以嵌入式架构实现在智能合约中融合原有的业务处理逻辑和新的安全防护逻辑于一体，解决上述在以太坊智能合约中存在的挑战。与基于签名的方法相比，ContractGuard 的优点是在检测异常行为时不需要已知漏洞的签名，并且可以发现未知攻击。遵循许多现有的基于异常的 IDS 相关工作[15,19-20]，ContractGuard 通过监视智能合约控制流路径以对执行行为进行分类，通过鉴别异常控制流来发现潜在的攻击行为和恶意行为。其中关键的思路是将以下两种技术结合起来，以满足对智能合约 IDS 有效性和效率的严格要求。

1) 上下文标记无环路径分析：ContractGuard 使用 Ball 和 Larus[21]所提出的经典算法来高效索

引和分析程序内无环路径。为了提高异常检测的效率，ContractGuard 将上下文调用添加到每个程序路径上。由于 EVM 不提供直接遍历调用栈的功能，ContractGuard 引入了一种分析方案来检测智能合约中的上下文调用。

2) 高效 Gas 的自适应路径集存储：EVM 中使用账户存储开销十分高昂。传统的基于内存的实现将会大量耗费 Gas。因此，作为智能合约存储（Storage）的替代品，ContractGuard 自适应地选择 3 个数据结构中的 1 个（嵌入式列表、嵌入式最小完美哈希表[22]和内置映射）以优化存储成本。从本文实验结果可得，使用 ContractGuard 带来的额外开销都是合理可行的。

值得注意的是，本文在 EVM 二进制代码上实现了 ContractGuard 所有技术需求，开发实现了 ContractGuard 的原型。区块链智能合约管理员可以使用 ContractGuard 来保护以太坊智能合约，而无须智能合约的 Solidity 源代码。

为了验证提出的设计，本文研究使用真实的以太坊智能合约进行了 3 组实验。第一组实验关注 ContractGuard 在 Gas 开销和危险警报方面的实用性。本文对在以太坊主网上部署的 8 314 份智能合约上应用 ContractGuard 实施安全防护（收集这些智能合约时主网块高度为 4 200 000，即 2017 年 8 月 24 日前），且收集到的智能合约至少有 100 笔交易。实验结果表明，ContractGuard 的平均部署开销和运行开销仅增加了 36.14%和 28.27%。第二组实验通过将 ContractGuard 应用于以太坊智能合约上发现的 6 个漏洞，以研究 ContractGuard 对安全威胁监控和防护的有效性。该实验的结果表明，ContractGuard 能够成功应对所有这些攻击行为。第三组实验验证 ContractGuard 运用在人工植入漏洞的智能合约上是否有效，实验结果表明，ContractGuard 能检测到其中 83%的漏洞。

从技术角度来看，本文作出了以下贡献。

1) 本文对智能合约漏洞进行详细研究，并提出了一种基于检测异常控制流路径的方法以抵御针对智能合约进行的恶意攻击。

2) 本文提出了一个适用于区块链智能合约技术要求的嵌入式 IDS 架构，实现对智能合约运行时刻的监测，识别因为恶意攻击引起的异常控制流、发现入侵攻击并回滚智能合约状态，实现对智能合约在部署后的保护。

3) 本文对 IDS 的嵌入过程进行了优化并开展了相应的实验。实验结果显示在以太坊智能合约上部署 IDS 是可行的。部署开销和运行开销被控制在一个合理的水平，嵌入的 IDS 可以在发现异常控制流时触发警报。上述各项性能达到了理想的水平。

4) 本文证明 ContractGuard 可以扩展到现实生活中真实的智能合约，可以针对即使没有源代码的智能合约提供有效的防御，同时能够有效地监测到未知漏洞引起的异常攻击。本文同时开展了实验，实验验证 ContractGuard 可以高效检测到现实漏洞以及实验过程中人工植入的漏洞。

2　相关工作

2.1　智能合约分析与验证

目前，在智能合约的相关文献中，已经提出了许多工具来验证智能合约。这些工具可以根据底层技术进行分类，其中一些工具依赖程序分析来发现漏洞，包括 Oyente[23]、Teether[4]、Gasper[5]，这些工具以及 Grossman 等在最近的工作[24]中使用符号执行来挖掘是否存在路径可以触发已知的漏洞。Contract fuzzer[11]使用随机发放来发现潜在的攻击向量。其他的工作依赖于形式化验证和定理证明。例如，Zeus[6]使用抽象解释和带约束的 horn 从句，MadMax[10]使用 Datalog 理论证明器，Grishchenko 等[25]的工作则使用了 F*定理证明器。

尽管这些工具可以检测到智能合约中的漏洞，但仅使用它们还不足以保护智能合约程序的执行。这些工具大多数旨在处理特定的已知漏洞，它们针对未知漏洞将显得力不从心。与之相反，ContractGuard 基于控制流异常检测到的漏洞并不会受限于特定的已知模式。只要这些漏洞触发了未在训练中出现的控制流路径，它就可以防御未知的漏洞。另外，作为在部署后使用的工具，ContractGuard 补充了现有的部署前验证工具。

2.2　执行分析

控制流分析是许多应用程序使用的基础动态分析技术。自从具有开创性的 EPP[21]算法被提出，

许多工作尝试通过添加更多的控制流信息或者进一步减少开销来优化 EPP 算法。例如，全程序路径（WPP）分析[26]将过程内的非循环路径的全部序列进行压缩来分析完整的控制流路径。过程间路径分析[27]尝试将 EPP 算法扩展到系统控制流图来实现相同的目标。这两个工作的执行开销代价非常高，以致于不能应用于以太坊智能合约。D′ Elia 和 Demetrescu[28]最近的工作通过在多个循环迭代中扩展非循环路径段来改进 EPP。尽管运行时开销可与 EPP 相提并论，但这种改进对于带有嵌套循环的函数来说可能会显著增加它们的路径数量。在本文的场景中，这是特别不希望的，因为这样路径集合的大小会不断膨胀，从而增加部署成本。除了路径分析之外，还有一些文献对上下文分析调用进行了研究。例如，Sumner 等[29]最近的工作利用栈深来优化分析开销。然而，栈深的信息在 EVM 中不可用，因此该方法无法在智能合约的情景中应用。

除了控制流，执行过程的其他方面也可以被分析，如数据流[30]或者动态不变性[31]。关于它们对保护智能合约程序有用性的研究留待将来的工作。

2.3　入侵检测系统

传统的入侵检测系统可以分为基于签名或者基于异常的入侵检测。基于签名的入侵检测系统[32-33]尝试识别具有可预先识别的入侵签名来识别已知的入侵模式，而基于异常的入侵检测系统假设入侵的性质是不可知的，但是入侵会偏离训练中描述的程序正常行为。基于签名的入侵检测系统通常更精确，因为它们利用特定的漏洞属性来检测攻击。然而，它们仅适用于已知的漏洞。此外，尽管签名数据库不是传统系统的主要关注点，但是对于入侵检测系统来说，签名数据库很大，无法被存储在智能合约中。这就是 ContractGuard 遵循基于异常的方法的原因之一。

现有的基于异常的入侵检测系统使用不同的模型来定义受保护系统的行为。对于模型来说，主要有两个维度：第一个维度是分析执行信息，包括系统调用[19]、调用栈[16]、输入数据/网络数据包[19]和跳转序列[16]；第二个维度是执行信息如何被抽象成一个模型，除了简单的序列模型，还提出了其他更复杂的模型，如 Dyck 模型[19]、*n* 跳转滑动窗口模型[15]和路径点模型[16]。ContractGuard 通过新的上下文标记的非循环路径模型增加了这一研究范围。该模型旨在满足保护以太坊智能合约程序的严格要求。

3　智能合约程序模型与监控手段

3.1　以太坊智能合约程序模型

以太坊智能合约程序模型如图 2 所示，以太坊智能合约程序由一个或多个部署了智能合约代码实例的合约账户以及一个或多个拥有以太币作为加密货币的用户账户组成。用户依靠客户端与以太坊网络进行交互，以太坊网络由一群将交易打包到区块链中的矿工维护。以下将从以太坊智能合约的程序架构、调用模型和数据管理模型 3 个层面介绍。

3.1.1　程序架构

以太坊智能合约大多使用类似 Javescript 的 Solidity 语言编写[34]。Solidity 语言是类型化的编程语言。在源码层面，用 Solidity 编写的合约类似于面向对象编程语言中的类。智能合约可以包含状态变量的声明、 函数的定义、修饰符以及构造函数。为了能够在以太坊平台上进行部署，开发人员将智能合约源代码编译为 EVM 二进制执行代码。智能合约的入口点是一段被称作函数选择器的代码。每当有函数被调用时，智能合约就会在函数选择器的位置开始执行。函数选择器会解析消息并跳转到恰当的函数。当没有明确调用智能合约的具体函数时，将自动触发回调函数。

3.1.2　调用模型

部署后，智能合约的外部/公共函数可以使用 3 种不同的方法进行调用。第 1 种方法是用户通过客户端发送交易进行调用，该交易的信息包含目标函数的签名哈希和函数所需的参数。这是一种可以更改账户余额和智能合约状态的写入操作。“矿工”将会向发送方收取以太币，以支付交易过程中所产生的 Gas 费用。第 2 种方法是通过智能合约直接调用另一个智能合约的函数。这种操作本质上是智能合约间的消息调用。第 3 种方法是由客户端在本地调用智能合约中的 view

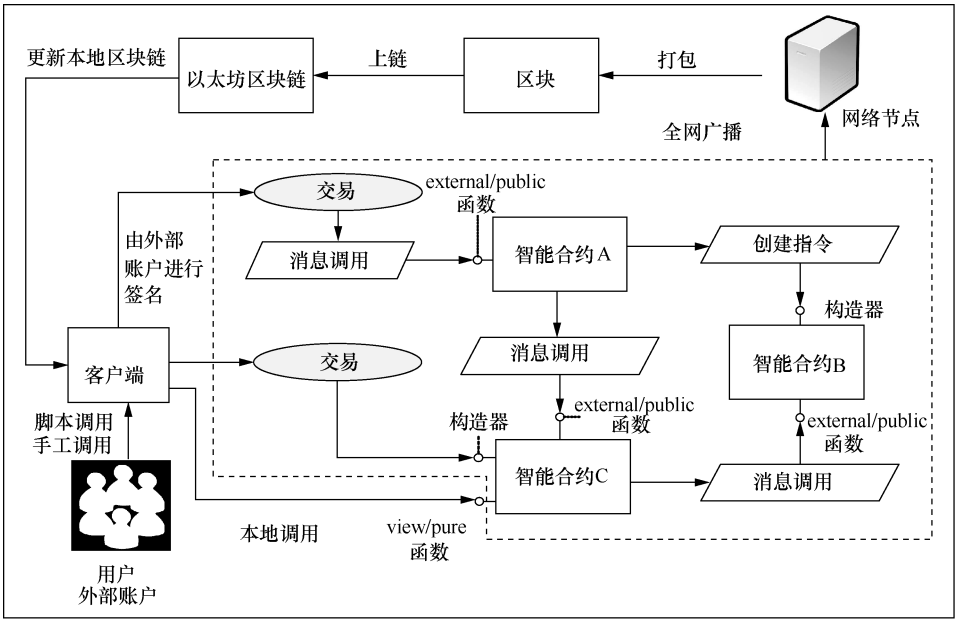


图 2　以太坊智能合约程序模型
Figure 2　Ethereum smart contract program model

函数或 pure 函数，该函数不会修改状态也不需要花费 Gas。

3.1.3　数据管理模型

以太坊虚拟机 EVM 是 256 位的基于堆栈的虚拟机。智能合约可以访问 4 种数据：最大容量为 1 024 的操作堆栈（Stack），用作易失性存储的字寻址的字数组内存（Memory），用于持久性存储的字寻址的字数组存储（Storage）以及作为消息调用数据复制的只读字数组（Calldata）。其中，堆栈的开销为 3，内存的开销至少为 3，存储的开销至少为 20 000，实际运行时情况较为复杂，这里无法给出精确的数值，针对本文研究的额外开销将在第 7.4 节中探讨。

3.2　执行路径和控制流

ContractGuard 通过监视 context-tagged 无环路径来保护智能合约，其中，context-tagged 无环路径标识了智能合约的执行路径和控制流。

3.2.1　context-tagged 无环路径

context-tagged 无环路径是由 Ball 和 Laurus[21] 对过程内无环路径的扩展。context-tagged 无环路径用一个元组<*p*,*c*>表示，其中 *p* 是过程内无环路径，*c* 是调用的上下文信息。通俗地讲，context-tagged 无环路径是控制流图（CFG）无循环版本的路径，即替换回边后的控制流图。通过用两个替代边 entry→v 和 w→exit 替换每个控制流图中所出现的回边 w→v，替换完毕后可以得到控制流图的非循环版本。例如，图 3(a)中的 CFG，在图 3(b) 中两个回边 3→1 和 4→3 被替代为边 Entry_C→3，3→Exit_C 和 4→Exit_C。

context-tagged 无环路径具有两个特性，使其特别适合异常检测任务。首先，非循环 CFG 中不能存在循环路径，因此该 CFG 中路径的数量总是有限的。其次，原始 CFG 中的完整路径可以在回边处分成多个非循环路径。例如，图 3(a)中的路径 Entry_C→1→2→3→4→3→4→3→4→3→1→5→Exit_C 可以被分解为 p1: Entry_C→1→2→3→4→Exit_C，p2: Entry_C→3→4→Exit_C，p3: Entry_C→ 3→4→Exit_C，p4: Entry_C→3→Exit_C，p5: Entry_C→1→5→Exit_C。可以看出，非循环路径对应于循环之前、循环内部或者循环之后的路径段。具有不同循环迭代次数的完整路径可以共享同一组非循环路径。

3.2.2　智能合约执行检测

为了获得智能合约运行过程中的信息，ContractGuard 需要先构建智能合约的函数调用图。

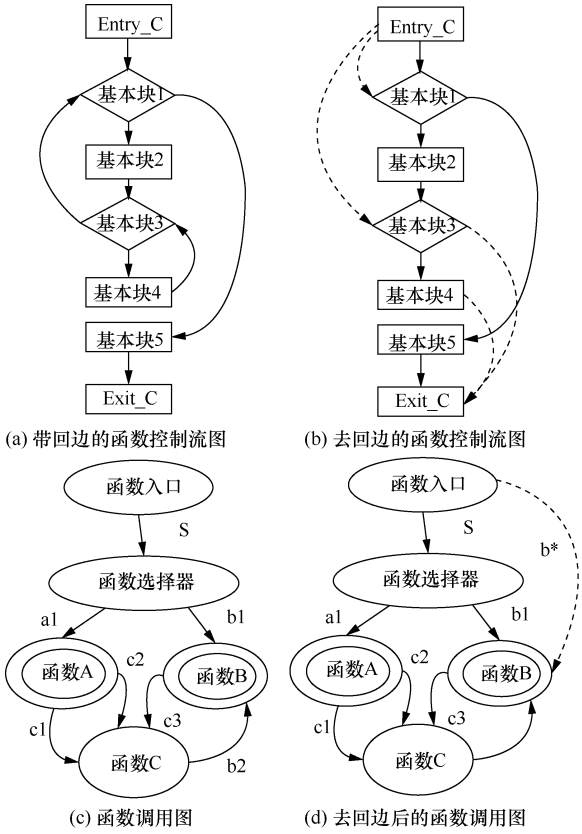


图 3　上下文标记的无环图例子

Figure 3　Examples of context-tagged acyclic path

函数调用图是指在给定的具体智能合约中，将智能合约存在的所有函数作为图中的节点，而调用的上下文信息则表示为调用函数 f 前函数之间的调用序列[35]。例如，在图 3(c)中，函数 B 的调用上下文信息可以为 S→b1，也可以为 S→a1→c1→b2，还可以为 S→a1→c2→b2，具体信息由实际运行情况决定。由于递归，函数调用图可能包含循环，与在 CFG 中将回边替换为代用边一样，每一条递归的边都将换成从智能合约入口到函数调用处的边。例如，将图 3(c)中的智能合约调用图转换为图 3(d)中的，结果中存在函数 C 的 4 个调用上下文信息：S→a1→c1，S→a1→c2，S→b1→c3 和 b*→c3。

为了获得更多智能合约运行过程中的信息，可以通过代码中不存在的虚拟分支对过程内无环路径进行拓展。这些虚拟分支检查每个指令的潜在前提条件，具体来说，对于每个算术运算，检查是否满足上溢/下溢的前提条件。对于每个函数调用，检查后置条件，如返回值是否为空（当返回值是地址类型）、false（当返回值是布尔类型）或 0（当返回值是整数类型）。

4　研究思路

4.1　研究假设

研究发现，异常的交易一般会触发漏洞从而进行攻击。ContractGuard 中有一个关键的假设：通过比较不同交易所触发的控制流来区分正常的交易和异常的交易。具体来说，ContractGuard 对交易的控制流进行监测，并对控制流进行分析和校验，识别异常的控制流，并对引发异常控制流的交易发出告警，认为其可能为异常交易。

一般而言，测试用例所产生的控制流是合法的。测试用例中合法交易类的测试用例应该被智能合约所接受，其控制流反映了合法的交易处理。测试用例中非法交易类的测试用例应该被智能合约所拒绝，其控制流反映了智能合约对非法交易的合法的出错处理。因此，本文认为异常交易所触发的不安全路径不会出现在测试用例所覆盖的安全路径集中。文献[12]表明在传统系统中这一假设是成立的。

4.2　异常控制流分析

为进一步验证这一个假设，本文参考了最近几项关于智能合约漏洞的研究[3,6,11,36-38]，总结出 11 种主要的安全漏洞，并研究这些漏洞是否触发了智能合约中的不安全路径。

(1) 可重入漏洞[12]

智能合约 A 调用智能合约 B 或者向智能合约 B 发送以太币时，外部调用可能被攻击者利用，使智能合约 A 执行一段额外的代码。如果这段额外的代码能回调智能合约 A 本身，而智能合约 A 没有对可重入行为进行检查，继续正常执行，不触发任何异常，则智能合约 A 将很容易被攻击者攻击，随机改变其智能合约状态。著名的 DAO 事件[6]就是利用此漏洞造成了巨大的经济损失。

根据外部调用所调用的函数是否为这个函数本身，本文将可重入漏洞分为相同函数可重入漏洞和交叉函数可重入漏洞[6]。代码 1 是相同函数可重入漏洞的一个例子。在代码 1 的第 6 行，智

能合约向攻击者构造的合约发起调用，这里并没有指明调用的函数，EVM 则会自动触发回调函数（在代码 1 的第 12 行），这时攻击者构造的合约将会触发智能合约的 withdraw 函数，从而造成智能合约的异常状态更改。

代码 1

```
1)  contract InnocentContract{
2)          mapping(address=>uint)private balances;
3)      function withdraw(){
4)              uint amount = balances[msg.sender];
5)          if(amount > 0){
6)  msg.sender.call(balance[msg.sender]);
7)              balances[msg.sender]=0;
8)          }
9)      }
10) }
11)  contract AttackerContract{
12)      function(){
13)          Wallet wallet = msg.sender;
14)          wallet.withdraw();
15)      }
16)  }
```

异常控制流：该漏洞会触发异常的控制流，无论是相同函数可重入漏洞还是交叉函数可重入漏洞都会触发一条异常的可重入控制流路径。当用来提取安全路径的测试用例包含这些控制流时，开发人员可以发现问题并修复此问题。值得注意的是，侦测此类漏洞需要相关的程序执行上下文信息，因此 ContractGuard 需要标识并获取对应的上下文信息。

(2) 危险的 deleagatecall[11]

当一个智能合约使用 delegatecall()调用外部函数时，外部函数执行时的程序上下文信息将不会改变，保持原来的程序上下文信息。这一特性有利于共享库代码的构造。然而，这一特性也会引发不安全的漏洞，当一些库代码在错误的上下文信息中执行时将会发生错误的状态改变。

代码 2 是一个 Parity 多重签名智能合约 delegatecall 漏洞的简化版。在代码 2 的第 11 行，智能合约使用 delegatecall()以及函数签名和函数的输入参数选定库代码中的函数，并对该函数进行调用。在代码 2 的第 15 行，攻击者触发第 11 行的代码，使智能合约触发 init 函数，在该执行过程中由于上下文信息保持不变，即保持 InnocentContract 合约中的所有信息，攻击者可以操纵 InnocentContract 合约中的信息，InnocentContract 合约的拥有者将转换为攻击者。

代码 2

```
1)  contract DelegateLibrary{
2)      address public owner;
3)      function init(){
4)          owner = msg.sender;
5)      }
6)  }
7)  contract DelegateLibrary{
8)      address public owner;
9)      DelegateLibrary _library;
10)     function (){
11)         _library.delegatecall(msg.data);
12)     }
13) }
14) contract DelegateLibrary{
15)     ...innocent.call(
16)         bytes4(keccak256(“init()”)));...
17) }
```

异常控制流：该漏洞会触发异常的控制流。在错误的上下文信息环境中执行智能合约，是在调用者或者开发者意料之外的，因此，这些智能合约的测试用例中肯定不会包含因意外而产生的路径，所以该漏洞所触发的路径都是异常控制流路径。

(3) 算术上溢/下溢[3]

在算术运算中，当操作一个超出固定范围大小的数字时，会出现算术上溢或下溢，如 uint8 的数据经过算术运算后变成 uint256 的数字，则会产生算术上溢。2018 年 4 月，攻击者利用整数溢出的漏洞对美链（BECToken)）进行攻击[39]，几乎将美链中的“数字货币”全部清空。

异常控制流：该漏洞会触发异常的控制流。

在 3.2 节介绍的拓展控制流，添加虚拟分支就是为了处理这种情况。当所有测试用例所产生的交易都是合法时，测试用例所覆盖的控制流路径都不会包含触发上溢/下溢的路径。

(4) 默认类型[37]

在 Solidity 语言中，函数的默认类型都是 public 的，这将允许其他智能合约或者外部合约调用其外部函数。当开发人员因疏忽而暴露原本应该为私有的方法给其他人使用时，可能会给不法分子提供攻击合约的渠道。

异常控制流：该漏洞会触发异常的控制流。当开发人员忘记特别指明函数类型时，即忘记加上 internal 关键字时，将出现异常的控制流。由于测试人员在测试合约时不会直接调用这些函数，因此测试用例不会覆盖此类路径。

(5) 随机种子[37]

在以太坊中，智能合约经常使用区块的时间戳作为随机数的随机种子。此类操作容易被黑客利用、影响和控制，因为这些随机变量受矿工节点决定。

异常控制流：该漏洞不会触发异常控制流。此漏洞与变量的错误使用有关，与控制流无关。

(6) 对外部调用的检查[6]

当一个智能合约出现错误的外部调用时，这条交易将不会进行正常的回滚。当智能合约的外部调用是 call()或者 send()触发时，外部调用返回错误结果，而不是直接进行回滚。很多漏洞是因为没有对 call()和 send()的返回结果进行检查所导致的。例如，在代码 3 中，开发者希望当第 2 行代码发生错误时，程序直接进行回滚，但实际上第 2 行发生错误时，程序继续往下执行，智能合约的状态将随共识的结束而改变。

代码 3

```
1)  if (gameHasEnded && !prizePaidOut){
2)      winner.send(1000);
3)      PrizePaidOut = True;
4)  }
```

异常控制流：该漏洞会触发异常的控制流。在第 3.2 节提到的虚拟分支会对外部调用的返回结果进行检查。假设本文的测试用例都是正确的，它们将不会覆盖使 call()和 send()失败的路径。

(7) 交易顺序[6]

在此类漏洞的攻击中，“矿工”会尝试与调用智能合约的用户竞争，将有问题的交易放在用户交易之前，因此能利用此方法对用户造成损失。

异常控制流：该漏洞不会触发异常的控制流。该漏洞的攻击针对智能合约的运行环境，并不会引起控制流的变化。

当智能合约使用 Tx.Origin 进行身份验证时，该合约通常能够进行钓鱼攻击。攻击者会欺骗用户在这份钓鱼合约中进行身份验证。例如，在代码 4 中，InnocentContract 利用第 3 行的代码确保只有 InnocentContract 的拥有者才能进行转账操作。然而，攻击者欺骗 InnocentContract 的拥有者转账给攻击者设计好的 AttackerContract ，即代码 4 的第 8 行。这时第 9 行的回调函数会被触发，由于使用了 Tx.Origin，第 3 行的代码不会触发异常，代码会继续往下执行，这时 InnocentContract 的拥有者会更改，攻击者成为 InnocentContract 的拥有者。

代码 4

```
1) contract InnocentContract{
2)     function transfer(address dest, uint amount){
3)      if (tx.origin != owner) throw;
4)          dest.send(amount);
5)      }
6)  }
7) }
8) contract AttackerContract{
9)     function (){
10)        Wallet w = Wallet(walletAddr);
11)                    w.transfer(thiefAddr, msg.sender.balance);
12)     }
13) }
```

(8) Tx.Origin[6]

异常控制流：该漏洞会触发异常的控制流。为了让此漏洞的攻击能成功，攻击者必须触发一

次特定的交易。这次交易必须由被欺骗者发起调用，通过攻击者所构造的合约再次调用被欺骗者所持有的合约，并且该合约是使用 Tx.Origin 进行身份验证的。在正常的情况下，测试用例只会直接调用此合约，并不会用其他合约对此进行调用。

(9) 拒绝服务[6]

此类型的攻击定义非常广泛，但从根本上说，它就是使用某些攻击手段造成智能合约在一小段时间甚至永久性失效。

异常控制流：该漏洞会触发异常的控制流。当有控制流造成智能合约失效，说明该控制流本身就是异常的，因为没有测试用例尝试让智能合约失效。

(10) 短地址攻击[37]

当传递给智能合约的参数短于预期长度时，EVM 将会在参数末尾添加 0。当第三方程序没有对交易输入的参数进行检查时，则很容易受到此类攻击。

异常控制流：该漏洞不会触发异常的控制流。这种漏洞的发生与智能合约自身无关，而是发生在与智能合约进行交互的第三方程序。因此，该漏洞不会触发异常的控制流。

(11) 逻辑错误[6]

使用过多复杂的谓词容易出现相关的逻辑错误，如使用过多的逻辑操作符和关系操作符。这些逻辑错误很有可能出现意想不到的漏洞。

异常控制流：有可能会触发异常的控制流。如果引起漏洞的控制流路径和正常的控制流路径没有重叠，就可以检测出异常的控制流[40]。然而，由于这种情况下，很有可能出现路径重叠情况，异常的控制流路径不一定都可以检测出来。

表 1 总结本文的调查和分析结果。如上述分析讨论显示，对大多数漏洞，其攻击是能触发异常的控制流。这初步证实了基于异常控制流的 IDS 对漏洞攻击能进行有效的防御。对于其他漏洞，其攻击的目标并非智能合约本身，而是针对以太坊的运行环境产生的漏洞（#5.随机种子和#7.打包交易顺序），或者针对第三方与智能合约进行交互所产生的漏洞（#10.短地址攻击）。

表 1　漏洞的异常控制流

Table 1　Abnormal control flow of vulnerabilities

漏洞	是否存在异常控制流
1) 可重入漏洞	是，此路径重新进入调用的函数中，然后发起外部调用
2) 危险的 Deleagate call	是，此路径通过 delegate call 调用未知的库代码
3) 算术上溢/下溢	是，在拓展后的控制流图中，用于检查上下溢的分支会返回 true
4) 默认类型	是，这条路径调用了本应该为 internal 属性的函数
5) 随机种子	否
6) 对外部调用的检查	是，在拓展后的控制流图中，用于检测 call()和 send()返回结果的值为 false
7) 打包交易顺序	否
8) Tx.Origin	是，此路径由合约用户在攻击者合约中发起交易，然后触发 Tx.Origin 进行权限更改
9) 拒绝服务	是，此路径使智能合约失效
10) 短地址攻击	否
11) 逻辑错误	不一定存在，取决于安全路径与不安全路径是否进行了重合

5　ContractGurad 智能合约防护方案

5.1　概览

ContractGuard 整个智能合约防护的过程，主要分为训练、保护和审计 3 个阶段如图 4 所示。

在训练阶段，需要在以太坊测试环境中部署未插桩的待保护的智能合约。可选的主流测试环境有 3 种：以太坊私有链、Ropsten 测试链、Ganache 以太坊模拟器[41]。因为 Ganache 提供最为纯净的部署环境以及较快的交易执行速度，本文选择 Ganache 作为测试环境。

智能合约在测试环境中部署后，本文的实验通过合约开发者提供的测试用例向待保护的智能合约发起训练交易。在这些交易执行完毕后，收集交易中合约的执行路径并从中提取出 context-tagged 无环路径。这些路径可视为初始安全路径集合。因为在训练阶段提取安全路径的开销远远小于部署后提取安全路径的开销，开发者在测试阶段提供大量的测试用例可以大量减少 ContractGurad 所带来的额外开销。

在保护阶段，在智能合约的字节码中插入可以起到防卫攻击的两类探针，分别是实时记录

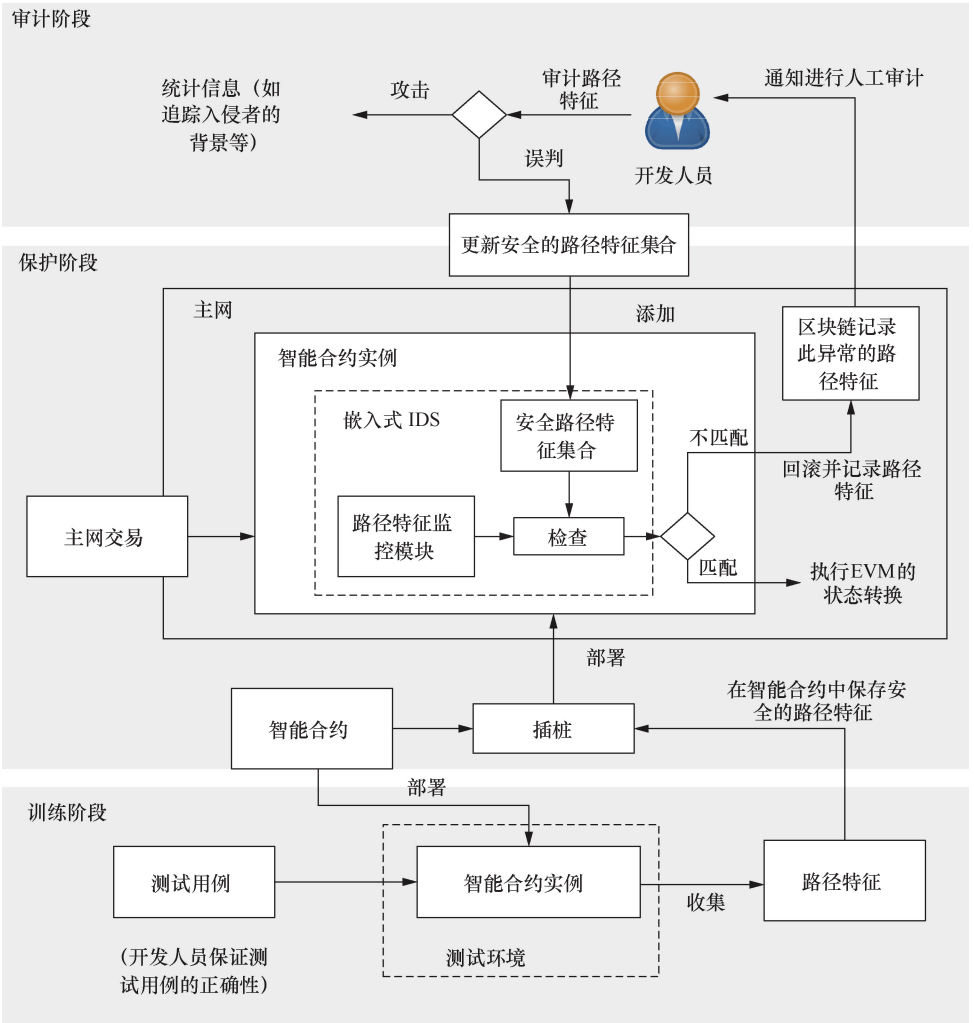


图 4　ContractGuard 的总览
Figure 4　Overview of ContractGuard

context-tagged无环路径以及检查路径是否在安全路径集合中的代码片段。这个过程实际上是对原有智能合约进行插桩的过程。待智能合约管理员在以太坊主网上部署插桩后的智能合约，合约在运行过程中可以自行记录每个交易运行的context-tagged无环路径并且校验其是否在安全集合中。若 ContractGuard 发现至少一条非法路径，则会自动进行回滚取消交易并且对外告警。

在审计阶段，智能合约管理员可以通过监控以太坊主网的交易记录获知智能合约是否发出警报。智能合约管理员有机会查看所有发生回滚的交易，并且在测试环境中重现交易查看智能合约是否正确执行。若交易被智能合约管理员确认是安全的，管理员可以向以太坊主网中发起交易，请求将之前误报的路径添加到安全路径集中。

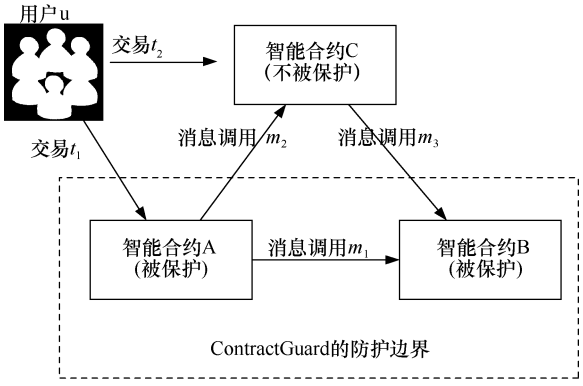
5.2　ContractGuard 的防卫边界

触发 IDS 回滚的情景 1 共有 3 种，如图 5 所示。

情景 1

此次交易全过程都在保护的边界中，整个过程为用户 u 发起交易 t_1 调用智能合约 A，智能合约 A 发起消息 m_1 调用智能合约 B。交易第一个执行的 public/external 函数就在被保护的智能合约 A 中，并且不再对边界外的任何智能合约发起调用。在这种情景下，ContractGuard 记录执行过程中所有函数的 context-tagged 无环路径，并确保这些路径的安全性。当发生跨智能合约的调用时，可以在 calldata 的消息中植入当前 calling-context

的信息，以确保可以建立完整的 calling-context 计算。一旦到达 context-tagged 无环路径的末尾，ContractGuard 会检查路径的安全性。倘若路径异常，则 ContractGuard 把此次交易标记为异常交易。此标记随执行过程通过在 returndata 进行植入的方式返回到此交易各层合约的入口函数。直到整个交易的入口，ContractGuard 通过检查此标记决定进行告警以及整个交易的回滚[18]。



保护过程中可能出现的 3 种情况

情景 1 此次交易都在保护边界中	情景 2 交易在保护合约中发起，但调用了不被保护的合约	情景 3 交易由不被保护的合约发起
用户 u ↓ 交易 t_1 ↓ 智能合约 A ↓ 消息调用 m_1 ↓ 智能合约 B	用户 u ↓ 交易 t_1 ↓ 智能合约 A ↓ 消息调用 m_2 ↓ 智能合约 C ↓ 消息调用 m_3 ↓ 智能合约 B	用户 u ↓ 交易 t_2 ↓ 智能合约 C ↓ 消息调用 m_3 ↓ 智能合约 B

如果警报响起，哪些状态会被改变

情景	配置后的合约	触发警报的合约	回滚的内容
1	A 和 B	A 或者 B	A 和 B 改变的内容
2(a)	A 和 B	A	A, B, C 改变的内容
2(b)	A 和 B	B	B 改变的内容
3	B	B	B 中改变的内容

图 5　ContractGuard 的保护边界和可能保护的情况
Figure 5　Protection boundary of ContractGuard and the possible protection scenarios

情景 2

此次交易首先由用户 u 发起交易 t_1 调用在保护范围中的智能合约 A，然后智能合约 A 发起消息调用 m_2 调用不在保护范围中的智能合约 C，智能合约 C 再发起消息调用 m_3 调用在保护范围中的智能合约 B。交易首个调用的智能合约是被保护的智能合约，但随着智能合约的执行，此次交易会调用到未被保护的智能合约，并且这些未被保护的智能合约有机会调用被保护的智能合约。例如，在图 5 中，智能合约 A 调用智能合约 C，之后智能合约 C 再调用智能合约 B。这种情景下，ContractGuard 会记录 A 和 B 的 context-tagged 无环路径，但 A 和 B 是独立进行异常标记以及回滚的。因此，所造成的结果取决于警报在何处触发。若在 B 触发警报，那只有 B 发生回滚。若在 A 触发警报，则在 A、B、C 这 3 个智能合约都发生回滚。另一种更好的解决方案是保证交易的原子性。但要实现这种回滚的原子性，智能合约与智能合约之间必须使用额外的消息调用以及 storage 存储实现异常标记的同步，因为未被保护的智能合约是不受 ContractGuard 所控制的。这类实现方式导致执行开销太高，因此这种方案是不可行的。

更广泛地说，现有被保护的智能合约 X 和智能合约 Y、未被保护的智能合约 Z，调用顺序为智能合约 X 调用智能合约 Z，接着智能合约 Z 调用智能合约 Y。当被保护的智能合约 X 调用未被保护的智能合约 Z 时，ContractGuard 并不会把当前的 calling-context 传给 Z 调用的 Y。但有一种例外的情况，就是智能合约 X 与智能合约 Y 是同一个合约，这种情况下，ContractGuard 通过智能合约 storage 的方式来传递 calling-context。采用这种实现方式可以防护可重入的攻击，但同时会对每个外部调用造成 5 000 Gas 的额外开销，因而这种实现方式是产生额外运行开销的最大原因。

情景 3

此次交易首先由用户 u 发起交易 t_2 调用不在保护范围内的智能合约 C，然后智能合约 C 发起消息调用 m_3 调用在保护范围内的智能合约 B。这个情景交易的首个智能合约是未保护的合约，是通过调用的方式进入被保护的智能合约中的。在此情形中，保护的智能合约仅有 B，当智能合约

B 触发警报时，智能合约 B 的状态会产生回滚，而智能合约 C 的状态将不受到回滚的影响。

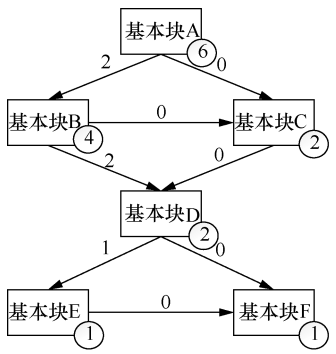
5.3　函数内的路径编码

ContactGuard 采用 Ball 和 Larus[21]提出的 EPP 算法对函数内的路径进行记录和编码。算法的输入是所有回边都被替代后的有向无环图。这种变换已经在第 3.2 节介绍。

图 6 是 EPP 算法示例与描述，核心思路是使用两个数值标记无环控制流图。对于无环控制流图中的每个顶点，使用 NumPaths(v)标记从 v 到 exit 顶点的路径数量。例如，图 6 中的 D 有两条路径可到达 exit 顶点 F（D→F 和 D→E→F），为此标记值为 2。此外，$e_1, e_2, \cdots, e_n$ 为点 v 的出边。E1 可以标记为 0，其余的边可用以下公式计算标记值。

$$\mathrm{Val}(e_i)=\sum_{j<i} NumPaths(end\_vertex(e_j))$$

通过这种标记系统，可以证明对于每个顶点 v，所有从 v 到 exit 的路径都可以用唯一的数值表示，并且这些数值是连续不断的。如图 6(b)的算法所示，EPP[21]可以为每个顶点和边生成准确无误的标记。



(a) CFG例子

```
Foreach在反向拓扑图中
的每一个点v
if v是一个叶子节点
  NumberPaths(v) = 1
else
  NumberPaths(v) = 0
  foreach边  e:v → w
  Val(e)= NumPath(v)
  NumberPath(v)=
  Number(v) +
  NumberPath(w)
```

(b) EPP算法

图 6　EPP 算法示例与描述

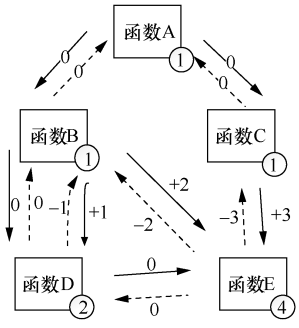
Figure 6　The example and description of EPP algorithm

利用 EPP 算法进行标记后，就可以利用这些标记有效地记录函数内路径：在函数入口，初始化局部整型变量 epp 为 0。当控制流通过边 e 时，若 val(e)不为 0，则累加至 epp 中。在函数结尾，可以在 epp 中得到所覆盖路径的唯一编号。除此之外，在循环的回边中，也可以得到唯一的编号 epp。之后 epp 重置为 0 并重新记录路径。

EPP 算法通过进一步优化，即最小化边标记中非零值可以减少 ContractGuard 带来的额外开销[42]。ContractGuard 使用的是优化后的 EPP 算法。

5.4　调用上下文的记录与编码

除了函数内部的路径外，ContractGuard 还需要调用上下文信息。受 EPP 算法的启发，本文设计了高效的上下文调用信息标识与配置算法，如图 7 所示。



(a) 函数调用图例子

```
Foreach  在拓扑图中的每
一个函数 f
  if f 是一个叶子节点
    NumCCs(f) = 1
  else
    NumCCs(f) = 0
  foreach 边 c:x→f, f→x
    Val(c)= NumCCs(f)
    Val(r)=-NumCCs(f)
    NumCCs(f)=
    NumCCs(x) +
    NumCCs(f)
```

(b) 算法2:调用信息的标识与配置

图 7　上下文标识调用信息标识与配置算法的示例与描述

Figure 7　The example and description of efficient calling-context indexing and profiling algorithm

本文所提算法的输入是所有递归调用边都被替换后的有向无环的函数调用图，其中，程序中的每个 callsite 都被表示为调用边和返回边。

从本质上来说，这个算法是作用在函数调用图中翻转版本的 EPP 算法。与 EPP 类似，这个算法同样对顶点和边进行标记。不同的是，EPP 使用逆拓扑顺序，而此算法使用拓扑排序。此算法利用 NumCCs(f)对每个点 f 进行标记。NumCCs(f)表示程序入口到 f 所有调用路径的数量。接着，$c_1, c_2, \cdots, c_n$ 是所有进入顶点 f 的调用边，$r_1, r_2, \cdots, r_n$ 表示的是返回边。边 c_1 标记为 0，其余的 c_n 利用下面的公式进行标记。

$$\mathrm{Val}(c_i)=\sum_{j<i} NumCCs(start\_vertex(c_j))$$

同时，对于所有的返回边 $r_i=-c_i$。

与 EPP 类似，可以证明顶点 f，每个调用上下文都可以通过对每个 call edge 的 Val(c) 进行累加而被赋上唯一且连续的值。这个证明与 EPP 的证明类似。

为了实现调用上下文记录与编码算法，需要

设置初始化为 0 的全局变量 ctx。对每个 call-site，在函数调用前，利用 Val(c)对 ctx 进行累加，并在 return 后，把 Val(c)的值从 ctx 中减去。在每个函数的入口，ctx 的值表示当前函数所处的调用上下文。

5.5　存储安全路径的方案

ContractGuard 结合了上面两种记录和编码的方式得到可以记录和编码 context-tagged 无环路径的方案：对每个函数 f 的入口 e，其 context-tagged 无环路径可以连续地编码为 0，1，2,⋯,NumPaths(e)*NumCCs(f)−1。当程序在运行阶段，每当到达函数内无环路径的结束点，可以使用 ctx*NumPaths(e)+epp 为此路径进行编码。

使用上述的编码体系，安全路径查询的问题可以形式化为：给定一个整型数 k，如何确定此整型数 k 在集合 S 中？对于这个问题，有很多现成的解决方案，但是基于 EVM 的 Gas 计算体系下，下面 3 种解决方案是最有效的。

1) 嵌入式列表：把查询变成与 S 中常量进行比较的代码并嵌入合约中。

2) 嵌入式完美哈希表（MPHT）：该方案通过把 S 作为 MPHT 嵌入至合约中。ContractGuard 使用 CHD 算法[22]，CHD 算法通过两层 hash 构造哈希表。第一层哈希把 S 映射到不同的槽中，每个槽可以使用第二层哈希映射到槽中的元素。

3) 合约 storage 映射表：使用 Solidity 中的 mapping 数据类型在合约的 storage 中存储 S 集合。

图 8 展示了 ContractGuard 实现上述 3 种存储方案的性能比较。在部署开销上，3 种存储方案都是和集合的大小线性相关的。但是，使用智能合约 storage 映射表的开销是最大的，因为每存储一个值都需要耗费 20 000 Gas。而其余的两种方案，仅需要为每条指令支付额外的 200 Gas 开销。在查询阶段，嵌入式完美哈希表和智能合约 storage 映射表所需要的 Gas 固定，而嵌入式列表的 Gas 是和集合大小线性相关的。但是，当集合元素小于 6 时，嵌入式列表消耗的 Gas 是最小的。

根据上述的分析，可以动态决定选择采用何种存储方案。当在训练阶段后，ContractGuard 可以根据安全路径集合的大小选择使用嵌入式 MPHT 还是嵌入式列表进行安全路径集合的初始化，智能合约 storage 映射表不会使用在此阶段。但是在智能合约部署后，智能合约管理员新添加的安全路径集合大多使用智能合约 storage 映射表来实现。

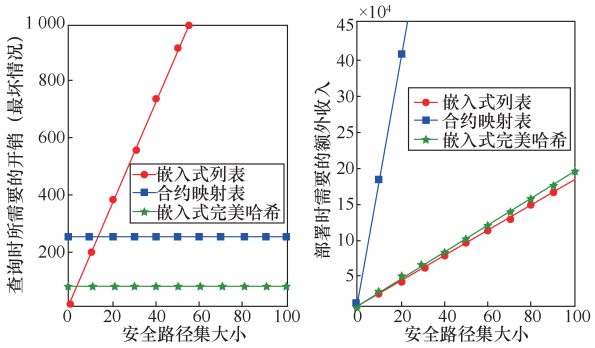


图 8　3 种存储方案的性能比较

Figure 8　The performance comparison of 3 storage solutions

6　基于以太坊主网的实验

为了评估 ContractGuard 的效率，本文进行了仿真实验，主要从开销方面分析将 IDS 嵌入智能合约的实际可行性和可操作性，以及嵌入式 IDS 对异常行为的识别过程是否导致过多无效的错误警报。

6.1　ContractGuard 的开销最小化

一般来说，使用嵌入式入侵检测器 ContractGuard 保护智能合约会涉及两种开销，这两种开销都需要花费额外的以太币。

1) 部署开销：智能合约嵌入带有异常检测功能和初始安全路径集的代码后，智能合约的代码规模会扩大，而智能合约的部署开销与其代码规模有关。因此，以太坊将在智能合约创建时收取额外的 Gas 费用，从而增加部署成本。

2) 运行开销：在交易执行期间，嵌入式 IDS 对控制流路径进行了设置。此外，当路径终止时，IDS 需要检查该路径是否在安全路径集中。该操作作为受保护的智能合约原来的交易处理之外的额外处理，不可避免地会增加智能合约的处理和计算，从而增加交易过程中的 Gas 花费成本。

虽然只有管理员来支付部署开销费用，但是将交易发送到受保护合约的所有用户都需要支付运行开销费用。如果开销不合理地增加，导致智能合约的交易成本过高，合约管理员和用户将不会使用本文所提出 ContractGuard，即使它可以有

效地保护智能合约。

为了调查在现实世界中为智能合约部署 ContractGuard 的开销，本文实验运行了最新版本的以太坊（Geth）全节点（1.9.0）（可重新训练所有历史信息），并与以太坊主网进行了同步，同步的块数量达到 4 200 000（截至 2017 年 8 月 24 日）共有 341 585 个智能合约。其中，332 008 个智能合约的交易少于 100 个，另外 1 173 个智能合约导致控制流程图提取工具崩溃了（本文实验使用了控制流程图提取工具 Vandal [7]）。最后选择了剩余的 8 314 个合约账户作为调查对象，因为它们交易数量大于 100 次，最有可能代表实际中使用的合约。在此期间，用户共向这些合约发送了 19 944 547 笔交易。

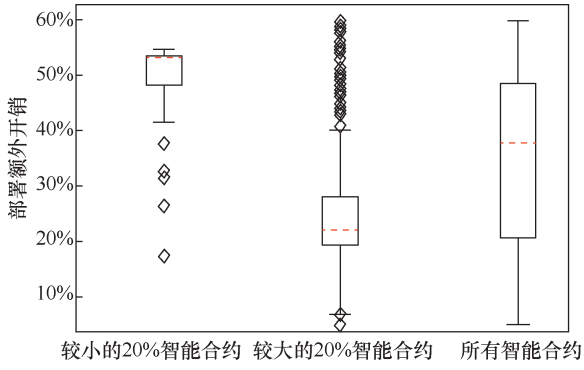
本文比较了这 8 314 个实例的代码，发现有 985 份独特的智能合约，其他智能合约复制至这些独特的智能合约。接下来，本文将报告完整的 8 314 个智能合约和 985 个独特的智能合约的部署额外开销平均值。

本文将 ContractGuard 应用于所有收集到的智能合约中。由于部署智能合约的成本与二进制代码大小（每字节 200 个 Gas[18]）成正比，因此将部署额外开销度量为（检测代码大小）/（原始代码大小）–100%。本文设置每个智能合约的安全路径集，其中包含发送到该智能合约的所有交易覆盖的所有 context-tagged 无环路径。这种处理模拟了部署额外开销的最坏情况。

图 9(a)显示了所有智能合约部署额外开销的分布，较大的 20%智能合约平均部署额外开销为 24.07%，较小的 20%智能合约的平均部署额外开销为 51.84%。总体而言，所有 8 314 个智能合约的平均部署额外开销为 36.14%，而 985 个独特的智能合约的平均部署额外开销为 37.76%。图 9(b)显示了部署额外开销最大的前 5 个智能合约的情况。

图 9 中的结果提供了初步的证据，表明智能合约嵌入 IDS 所需部署开销是切实可行的，其增加的成本在合理范围内：本质上，管理员只需要多支付 36.14%的 Gas 费用，以换取更高的智能合约安全性。值得注意的是，在部署 ContractGuard 后，8 314 个智能合约均未超过 24 576 byte 的限制。

接下来，本文通过实验调查了嵌入 ContractGuard 后的智能合约的运行开销。对于发送给收集到智能合约的所有 19 944 547 笔交易，本文收集了其详细的执行路径。然后，将每条路径与 ContractGuard 所部署的代码对齐，并得到 ContractGuard 所要执行的所有指令。以这种方式，本文通过累积执行额外指令所花费的额外 Gas 成本来获得 ContractGuard 在每个交易上精确的运行开销。



(a) 部署额外开销的分布

#	智能合约在主网上的地址	原始智能合约大小	部署额外开销
1	0x5bde2f1f37f594fd610b1731284f5a204e3ea545	4 295	61.23%
2	0x80aa81029df9afdc70a621c86d7a81d7e9ed7e3a	5 899	59.83%
3	0x051fda7486480dd5abcf5dd742ef002a2ebb9ea0	7 703	59.75%
4	0xc723d744dd32780c6f5cef4705e99919e77879d7	6 273	59.06%
5	0x5c543e7ae0a1104f78406c340e9c64fd9fce5170	2650	58.47%

(b) 部署额外开销最大的 5 份智能合约

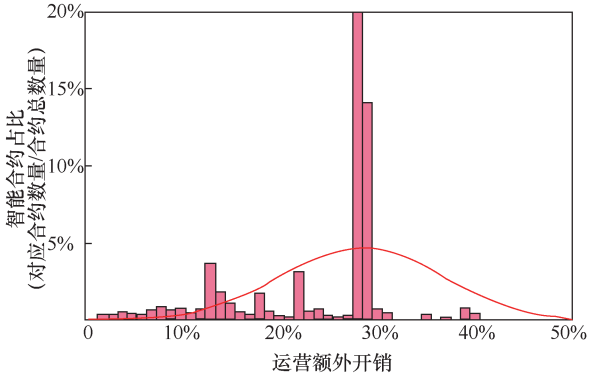
图 9　部署额外开销的测量结果

Figure 9　Measurement of deployment overhead

图 10(a)以 Gas 增加量的方式展示了运行时 Gas 开销的分布情况。平均而言，8 314 个智能合约运行额外开销为 25.98%，985 个独特的智能合约运行额外开销为 17.5%。这与常规系统中基于实际控制流异常的 IDS 相当（如 Giffin 等[19]的工作中，平均执行时间开销为 22%）。对于 94.5%的交易，运行时 Gas 开销都低于 30%。整体而言，嵌入 ContractGuard 后的智能合约运行时开销的增加在较低、合理以及可实际接收的水平。

图 10(b)显示了运行时额外开销最大的前 5 个

智能合约情况。在最坏的情况下，运行额外开销增加了 86.0%。在 7.4 节中，本文将对影响运行开销的因素进行详细分析。经研究发现，对于少量交易而言，运行额外开销如此之高的原因是，它们对外部未受保护的合约进行了大量频繁的调用。发生这种情况时，ContractGuard 必须使用昂贵的智能合约存储来保存调用上下文。目前，ContractGuard 应用于类似情景中成本可能偏高（在 8 314 个主题合约中，这部分情况低于 5%）。本文将在 7.5 节对此问题进行更多的讨论。



(a) 运行额外开销的分布

#	在主网上的交易哈希值	运行额外开销
1	0xda081509e17dcb852c4e12bc435be6d507499993590e0da97aa0da4c1f3d915d	86.0%
2	0xbf7a5b2113966a264dcd213d0e2a1d7d6ce462c1f4f2059a81afe7f36be600f5	85.7%
3	0x832d0282d84221ad21a3e8c43652653b66aea8bedfafc76c357e47da2c02314b	85.4%
4	0xd8626e496a04c8a894094eeb848cf9502d208834fd97bbcf4c86527c10e8555f	84.5%
5	0x41000decd3cd39be9895e3c2a7250deb57c2e8e99a2f59534657d71df2c428de	76.7%

(b) 运行额外开销最大的 5 次交易

图 10　运行开销的测量结果

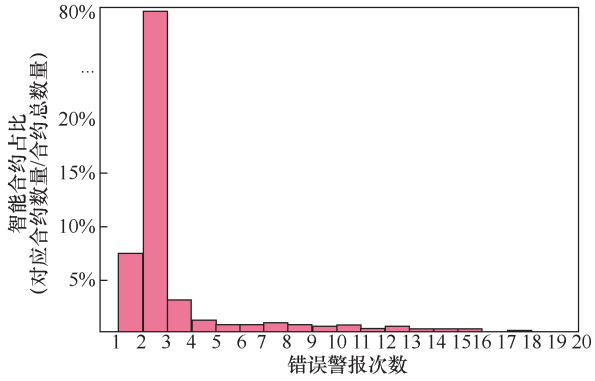
Figure 10　Measurement of runtime overhead

6.2　手动处理错误警报的可行性分析

除了开销之外，另一个可能影响 ContractGuard 实际使用的重要问题是错误警报的触发情况[43]，它对应于被 ContractGuard 误认为是恶意交易的合法交易。为了避免拒绝合法的交易，管理员需要手动查看每个警报，如果是错误警报，则将路径追加到安全路径集中，这可能需要一定的人工干预。

为了调查错误警报的数量，本文使用 8 314 个真实世界的智能合约进行至少 100 交易。假设开发人员部署的智能合约都是没有测试用例的，从而无初始化的安全路径集。ContractGuard 最初将设置一个空的安全路径集。本文进一步假设所有交易都是合法的，并且当该交易被误认为异常交易时，管理员立即将交易的上下文标记的无环路径添加到安全路径集中。因此，ContractGuard 将不会对之前误报的交易发出警报。在这些假设下，本文计算每个合约的误报数量。本质上，这就是估计在现实世界中处理错误警报所需的手动工作量。

图 11(a)显示了错误警报的分布情况，而图 11(b)显示了具有最多错误警报的前 5 个智能合约。在假设的情况下，ContractGuard 会为每个智能合约平均报告 2.88 次错误警报。对于大多数交易（92.2%），错误警报的数量低于 5。此结果为使用 IDS 保护智能合约时错误警报问题是可管理的假设提供有力的证据。



(a) 错误警报的数量分布

#	在主网上的智能合约地址	错误警报次数
1	0x49516fe7bdc54b29a7f95ff55fdd38d9228e55af	60
2	0x2faa316fc4624ec39adc2ef7b5301124cfb68777	59
3	0xda4a4626d3e16e094de3225a751aab7128e96526	56
4	0x3f593a15eb60672687c32492b62ed3e10e149ec6	53
5	0xb2c818252457488728ca62affa8f320247b8a84a	53

(b) 错误警报最多的 5 份智能合约

图 11　错误警报次数的测量结果

Figure 11　Measurement of the false alarm counts

7　基于现实世界合约漏洞的受控实验

本节对 ContractGuard 的有效性进行实验评估，将评估 3 个研究问题。

问题 1：ContractGuard 是否可以有效地检测

漏洞？

问题 2：部署开销、运行开销和错误警报的数量是多少？

问题 3：哪些因素会影响部署和运行时开销？

本节介绍了用于评估所提出的研究问题的实验，并报告了实验结果。

7.1　实验对象

为了研究问题 1 和问题 2，本文研究设计了两组实验。第一组实验（见表 2）的实验对象是 6 个已经被报道存在漏洞并成为攻击受害者的真实智能合约。本文选择这些实验对象是因为它们代表已造成重大经济损失的著名漏洞攻击和典型漏洞攻击。例如，第 1 个实验对象“DAO”合约[6]，黑客从中窃取了价值 6 000 万美元的代币，最终导致了以太坊的硬分叉，产生了以太坊经典（ETC）[6]。第 2 个实验对象“MultiSig”包含一个漏洞[7]，最终导致价值 3 亿美元的加密货币被冻结，并且（可能）将会永远损失。第 4 个实验对象“BecToken”是一个在 2018 年 4 月遭到恶意攻击者攻击的智能合约，黑客将存储在该智能合约的金额几乎全部取走[39]。

第二组实验（见表 3）使用实际在主网运行的去中心化应用（DApp，decentralized application），DApp 实际上是多个智能合约所组成的程序。这些项目都是开源的，并包含开发者使用 Truffle 框架所编写的测试用例。本文选择这些实验对象是因为就代码大小而言，它们是 Github 上存在的最复杂的 DApp。本文研究的所有实验对象、测试用例、漏洞以及其攻击的例子将发布到项目网站。为了调查问题 3，本文使用了以太坊主网上的 8 314 个智能合约实例，对这些实验结果的解释可以在 6.1 节中找到。

7.2　真实漏洞的实验研究

根据 6 个已报告的智能合约漏洞和对应的攻击方式（见表 2），本文实验严格按照这些报告构建能触发漏洞的攻击交易。然后将此交易与从以太坊主网中随机选择的 99 个正常交易混合在一起，以创建一个测试用例，即 100 条交易组成的序列。攻击交易被随机放置在序列末尾 10%位置。为了避免巧合，实验过程中重复此过程以创建 100 个不同的测试用例。智能合约创建部署方式来自主网，并在测试用例之间共享。

由于没有开发者针对这些智能合约所写的测试用例，因此实验过程中省略了智能合约的训练过程，从空的安全路径集开始。一开始假设管理员会检查所有在正常交易中引发的错误警报，并

表 2　应用在现有项目的实验结果

Table 2　Experiments with real-life reported vulnerabilities and attack vectors

项目	漏洞以及其攻击向量	代码大小/Byte	部署额外开销		运行额外开销		平均召回率	平均误报次数
			部署额外开销/原始部署开销	平均开销	运行额外开销/原始运行开销	平均开销		
DAO	可重入漏洞[44]	1 774	11.95%	35 751.43	20.51%	43 845.47	100%	4.48
MultiSig	危险的 Delegatecall[45]	764	29.58%	22 679.66	28.27%	29 085.6	100%	2.0
KoETH	对外部调用的检查[46]	3 617	31.18%	37 359.64	23.82%	44 067.72	100%	13.57
BecToken	算术上溢[47]	5 963	9.52%	39 846.26	15.75%	68 371.95	100%	8.2
OwlWallet	Tx.Origin[48]	561	27.27%	25 032.01	26.50%	31 456.74	100%	3.0
MerdeToken	算术下溢[49]	1 013	21.03%	35 990.97	18.74%	42 397.30	100%	4.0

表 3　人工植入的漏洞实验

Table 3　Experiments with seeded vulnerabilities

项目	代码大小/Byte	Truffle 框架下的测试用例	人工植入漏洞的总数	主要植入的漏洞	部署额外开销	运行额外开销	平均召回率	平均错误警报
Cryptokitties[50]	30 966	516	64(14,7,3,40)	5(2,1,0,2)	14.28%	9.03%	60%	1.5
StatusNetwork[51]	38 631	261	83(28,9,6,40)	2(0,1,1,0)	19.02%	15.79%	100%	3.0
VotingDApp[52]	2 326	14	15(6,2,0,7)	2(2,0,0,0)	12.33%	10.84%	100%	1.0
pool-shark[53]	5 895	69	33(8,6,1,18)	3(2,1,0,0)	10.23%	31.52%	100%	5.9

且这些正常交易的路径将立即插入安全路径集中。在此假设下，将 ContractGuard 应用于每个智能合约并使用 100 个测试用例执行它们，用召回率衡量测试结果，召回率触发警报的测试用例数量占所有测试用例数量的百分比，即触发警报的测试用例数量/所有测试用例数量×100%。

表 2 显示了实验结果。对于所有测试，召回率为 100%。这表明 ContractGuard 可以有效地防御这些已知漏洞。另外，运行开销、部署开销和错误警报与之前在主网上的 8 314 个智能合约上（第 6.1 节）观察到的结果一致。 截至 2019 年 8 月 30 日，平均 Gas 价格为 1.4683×10^{-8}ETH，ETH 的价格为\$ 168.21[54]。如表 2 所示，最大的部署和运行开销是分别为 59 846.26 Gas 和 68 371.95 Gas。因此，如果管理员要在这 6 个现实世界中的智能合约上应用 ContractGuard 来提供入侵检测，最多需要在智能合约部署时为每个智能合约支付额外\$0.147，而用户最多需要为每笔交易支付额外\$0.168 9。

7.3 人工植入漏洞的实验研究

表 3 中的 DApp 来自开源项目并且带有基于 Truffle 框架所写的测试用例。当需要将 ContractGuard 运用到实际中的 DApp 时，这些测试用例是用来收集安全路径的最佳选择，但这些 DApp 不存在任何已报告的漏洞。为了方便实验，本文参考了表 1 中的 11 个主要漏洞，并人工植入了其中的 4 种漏洞。

1) ＃3 算术上溢/下溢：用直接算术运算符替换 SafeMath[55]调用。

2) ＃4 默认可见性：删除关键字“private”。

3) ＃6 未检查的 send：删除整个用于检查 send 和 transfer 返回值的分支。

4) ＃11 逻辑错误：替换关系运算符和逻辑运算符。

植入其余 7 种漏洞是困难的，要么因为它们与智能合约代码（＃5，＃7，＃10）没有直接关系，要么因为这些程序没有可以植入这些漏洞的相关程序结构（＃1，＃2，＃8，＃9）。值得注意的是，第一个实验包含漏洞＃1，＃2，＃8。

对于每个 DApp，本文研究尽可能多地植入安全漏洞。漏洞总数和这 4 个漏洞各自的总数为表 3 的人工植入漏洞总数列中。然后，在每个植入了安全漏洞的 DApp 上执行开发人员所写的测试用例，无法通过测试用例的人工构造版本将被丢弃。

接下来，随机抽取 100 个交易来创建一个测试用例，这 100 个交易来自测试池，测试池主要由主网和手动构建的一批用于触发植入漏洞的交易组成。在原始 DApp 和植入漏洞的 DApp 上都执行测试用例。

如果两份 DApp 都通过了测试用例，但是各自的智能合约存储值在执行交易后存在差别，就将此植入漏洞的 DApp 添加到目标测试集中，并将此交易标记为攻击交易。重复此过程，直到每个目标测试集有了 100 个测试用例。但使用这些目标测试集来模拟开发人员测试用例无法检测部署后可能成为被黑客利用的漏洞。表 3 主要植入的漏洞列中报告了它们的总数以及 4 个漏洞的总数。对于每个目标测试用例，应用 ContractGuard 并使用开发人员提供的测试用例进行训练。然后用 DApp 执行 100 个测试用例。测量哪些测试用例成功引发了警报并在攻击交易中进行回滚，并以该测试用例数量的占比（成功回滚测试用例数量/测试用例总数量×100%）作为该 DApp 召回率。由于存在多个人工植入漏洞的 DApp，因此报告每个 DApp 的平均召回率。

表 3 显示了实验结果。所有 DApp 的平均召回率为 83.3%。对剩余 16.7%的情况，经检查发现，所有 ContractGuard 不能防护人工植入的漏洞都是逻辑错误。ContractGuard 无法阻止逻辑错误是因为出现了一种奇妙的“巧合”：开发人员提供的测试用例与攻击交易触发了智能合约相同路径，但 DApp 运行时并没有出现异常，因此攻击交易的路径被放入了安全路径集中，导致 ContractGuard 无法发现侦察出异常。

7.4 影响开销的因素

为了调查可能影响额外开销的因素，本文将 ContractGuard 的实现分解为各种探针（IP），每种探针表示原始代码中要插入检测代码的点。

表 4 显示了每种探针的部署/运行 Gas 消耗。部署成本很简单，即检测代码的字节大小乘以 200。最高开销来自每份智能合约开始时的智能合

约构造器。这就是为什么较小的智能合约往往具有较高的部署开销。本文实验使用 EVM 的 Gas 模型计算运行时开销[18]。如表 4 所示，除了合同构造的前期成本外，最昂贵的开销是外部调用不受保护的合约。成本高是因为需要将变量 ctx 存入智能合约持久存储：一条在存储中重写 256 bit 的 SSET 指令需要花费 5 000 Gas。这是检测出可重入控制流的代价。请注意，对受保护的智能合约进行外部调用要便宜得多，因为在这种情况下 ctx 可以在消息调用数据内部传递，而无须执行开销昂贵的写入智能合约持久存储操作。

表 4　探针在各类嵌入点中的开销
Table 4　Overhead of various instrument points

嵌入点	部署开销/Gas	运行开销/Gas
合约构造点	16 000	500～5 980
函数入口	4 200	45
分支点	2 600	30
回边	5 200	57
内部函数调用	4 400	48
内部函数返回	6 200	68
外部调用不安全合约	5 000	5 348
外部调用安全合约	6800	70
函数出口	800	9
路径集检查	$1\,600 \cdot n + 2\,800$	378

注：n 是初始的安全路径集大小。

7.5　讨论

本文实验中出现的几个问题值得进一步讨论和分析并作研究。

1) 目前，由于特殊的“巧合”，ContractGuard 可能无法非常有效地防御复杂的逻辑错误。短路求值就是一个典型的例子，本文研究推测可以通过添加针对判断短路求值的虚拟分支来提高 ContractGuard 的有效性。在未来的研究中，计划对此问题进行更深入的探索。

2) 因为外部调用未受保护的智能合约时会产生的昂贵开销，ContractGuard 实际应用于有着丰富特性的智能合约时会产生相当高的额外开销。尽管这似乎是一个限制，但应该注意到调用不信任的合约是一种危险的做法，不鼓励这种编写智能合约的方式[37]。另外 ContractGuard 在外部调用受保护的合约时并不会产生高成本，这是多份智能合约所组成的程序（即 DApp）中常规做法。

3) 尽管在现有的以太坊主网中未发现智能合约大小超过 24 576 B 的情况，但不排除使用 ContractGuard 后智能合约大小会超出这个范围。这种情况原因可能是初始安全路径设置太大，导致其不能作为一个嵌入式列表或嵌入式完美哈希表。在这种情况下，ContractGuard 将使用智能合约存储一些路径。这将大大增加部署开销，因为管理员需要支付额外的费用才能写入智能合约存储，而每条路径将花费 20 000 Gas。但是这不会增加运行开销，因为每次集合成员测试始终包含对智能合约存储的访问，以检查是否有动态添加的路径，并且不论映射大小如何，访问成本都是不变的。

8　结束语

从管理员的角度来看，以太坊智能合约程序在部署后容易受到攻击。一旦合约部署后，任何地方的用户都可以匿名检查它的实现和当前的智能合约状态。而一旦一个漏洞被发现，只要黑客拥有足够的以太币，他们会立即进行入侵攻击。造成的损害一旦被主链接受将会是不可挽回的，也没有机会来修复这个错误。

在本文研究工作中，首要目标是解决这个无法修补损失的现状，并为管理员提供一次弥补损失的机会。本文提出的 ContractGuard 是第一个以太坊智能合约的入侵检测系统，并通过嵌入式架构把入侵检测系统嵌入智能合约中，实现了在以太坊虚拟机的受限和抗干预环境中对智能合约进行异常检测和防护。通过将 ContractGuard 应用在现实世界以太坊主网的智能合约，本文证明了 ContractGuard 只需要较低的部署开销与运行开销即可以有效地防护漏洞。

参考文献：

[1] NAKAMOTO S, Bitcoin: a peer-to-peer electronic cash system[EB].

[2] BUTERIN V. A next-generation smart contract and decentralized application platform[EB].

[3] ATZEI N, BARTOLETTI M, CIMOLI T. A survey of attacks on Ethereum smart contracts (SoK)[C]//International Conference on

Principles of Security and Trust. 2017: 164-186.

[4] KRUPP J, ROSSOW C. Teether: gnawing at Ethereum to automatically exploit smart contracts[C]//27th USENIX Security Symposium (USENIX Security 18). 2018: 1317-1333.

[5] CHEN T, LI X, LUO X, ZHANG X. Under-optimized smart contracts devour your money[C]//Software Analysis, Evolution and Reengineering (SANER). 2017: 442-446.

[6] KALRA S, GOEL S, DHAWAN M, et al. ZEUS: analyzing safety of smart contracts[C]//NDSS. 2018.

[7] BRENT L. Vandal: a scalable security analysis framework for smart contracts[J]. arXiv preprint arXiv:1809.03981, 2018.

[8] ZAKRZEWSKI J. Towards verification of Ethereum smart contracts: a formalization of core of solidity[C]//Working Confer-ence on Verified Software: Theories, Tools, and Experiments. 2018: 229-247.

[9] NIKOLIĆ I, KOLLURI A, SERGEY I, et al. Finding the greedy, prodigal, and suicidal contracts at scale[C]//The 34th Annual Computer Security Applications Conference. 2018: 653-663.

[10] GRECH N, KONG M, JURISEVIC A, et al. MadMax: surviving out-of-gas conditions in Ethereum smart contracts[J]. Proceedings of the ACM on Programming Languages, 2018, 2: 1-27.

[11] JIANG B, LIU Y, CHAN W K. Contract fuzzer: fuzzing smart contracts for vulnerability detection[C]//33rd ACM/IEEE International Conference on Automated Soft-ware Engineering. 2018: 259-269

[12] LIAO H J, LIN C H R, LIN Y C, et al. Intrusion detection system: a comprehensive review[J]. Journal of Network and Computer Applications, 2013(36): 16-24.

[13] Understanding the DAO attack[EB].

[14] Parity multisig hacked again[EB].

[15] ZHANG T, ZHUANG X, PANDE S, et al. Anomalous path detection with hardware support[C]//The 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems. 2005: 43-54.

[16] FENG H H, KOLESNIKOV O M, FOGLA P, et al. Anomaly detection using call stack information[C]//2003 Symposium on Security and Privacy. 2003: 62-75.

[17] GARFINKEL T, ROSENBLUM M. A virtual machine introspection based architecture for intrusion detection[C]//NDSS. 2003: 191-206.

[18] WOOD G. Ethereum yellow paper[EB].

[19] GIFFIN J T, JHA S, MILLER B P. Efficient context-sensitive intrusion detection[C]//NDSS. 2004.

[20] XU H, DU W, CHAPIN S J. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths[C]//International Workshop on Recent Advances in Intrusion Detection. 2004: 21-38.

[21] BALL T, LARUS J R. Efficient path profiling[C]//29th Annual ACM/IEEE International Symposium on Microarchitecture, 1996: 46-57.

[22] BELAZZOUGUID, BOTELHO F C, DIETZFELBINGER M. Hash, displace, and compress[C]//European Symposium on Algorithms. 2009: 682-693.

[23] LUU L, CHU D H, OLICKEL H, et al. Making smart contracts smarter[C]//2016 ACM SIGSAC Conference on Computer and Communications Security. 2016: 254-269.

[24] GROSSMAN S. Online detection of effectively callback free objects with applications to smart contracts[C]//ACM on Programming Languages, 2017(2): 48.

[25] GRISHCHENKO I, MAFFEI M, SCHNEIDEWIND C. a semantic framework for the security analysis of Ethereum smart contracts[C]//International Conference on Principles of Security and Trust, 2018: 243-269.

[26] LARUS J R. Whole program paths[C]//ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI). 1999, 34: 259-269.

[27] MELSKI D, REPS T. Interprocedural path profiling[C]// International Conference on Compiler Construction. 1999: 47-62.

[28] D'ELIA D C, DEMETRESCU C. Ball-larus path profiling across multiple loop iterations[J]. ACM Sigplan Notices, 2013, 48: 373-390.

[29] SUMNER W N, ZHENG Y, WEERATUNGE D, et al. Precise calling context encoding[J]. IEEE Transactions on Software Engineering, 2012, 38(5): 1160-1177.

[30] AMMONS G , LARUS J R. Improving dataflow analysis with path profiles[J]. ACM SIGPLAN Notices, 1998, 33: 72-84.

[31] ERNST M D, COCKRELL J, GRISWOLD W G, et al. Dynamically discovering likely program invariants to support program evolution[J]. IEEE Transactions on Software Engineering, 2001, 27(2): 99-123.

[32] KUMAR V, SANGWAN O P. Signature based intrusion detection system using SNORT[J]. International Journal of Computer Applications & Information Technology, 2012, 1(3): 35-41.

[33] CUI W, PEINADO M, WANG H J, et al. Shieldgen: automatic data patch generation for unknown vulnerabilities with informed probing[C]//2007 IEEE Symposium on Security and Privacy (SP'07). 2007: 252-266.

[34] DANNEN C. Introducing Ethereum and solidity[M]. Berlin: Springer, 2017.

[35] ZHUANG X, SERRANO M J, CAIN H W, et al. Accurate, efficient, and adaptive calling context profiling[J]. ACM Sigplan Notices, 2006(41): 263-271.

[36] LI X, JIANG P, CHEN T,et al. A survey on the security of blockchain systems[C]//Future Generation Computer Systems. 2017.

[37] MANNING D A. Solidity security: comprehensive list of known attack vectors and common antipatterns[EB].

[38] DERIVATIVES D N | T. A survey of solidity security vulnerability[EB].

[39] BEC spiked 4000% on first trading day, another pump-and-dump scheme[EB].

[40] WANG X, CHEUNG S C, CHAN W K, et al. Taming coincidental

correctness: coverage refinement with context patterns[C]//ICSE. 2009: 45-55.
[41] Truffle Suite. Sweet Contracts[EB].
[42] BALL T, LARUS J R. Optimally profiling and tracing programs[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1994, 16(4): 1319-1360.
[43] TJHAI G C, PAPADAKI M, FURNELL S M, et al. Clarke, investigating the problem of IDS false alarms: an experimental study using snort[C]//IFIP International Information Security Conference. 2008: 253-267.
[44] Understanding the DAO attack[EB].
[45] An in-depth look at the parity multisig bug[EB].
[46] Post-mortem[EB].
[47] Detecting integer arithmetic bugs in Ethereum smart contracts[EB].
[48] Smart contract wallets created in frontier are vulnerable to phishing attacks[EB].
[49] MerdeToken[EB].
[50] CryptoKitty[EB].
[51] StatusNetwork[EB].
[52] DecentralizedVoting[EB].
[53] PoolShark[EB].
[54] Etherscan[EB].
[55] SafeMath OpenZeppelin[EB].

[作者简介]

赵淦森（1977− ），男，广东东莞人，华南师范大学教授、博士生导师，主要研究方向为信息安全、区块链。

谢智健（1995− ），男，广东云浮人，华南师范大学硕士生，主要研究方向为软件测试、区块链应用。

王欣明（1980− ），男，香港科技大学博士生，主要研究方向为金融科技、软件测试与软件分析、区块链。

何嘉浩（1995− ），男，广东广州人，华南师范大学硕士生，主要研究方向为软件分析、区块链应用。

张成志（1963− ），男，中国香港人，香港科技大学教授、博士生导师，主要研究方向为软件工程。

林成创（1990− ），男，广东韶关人，华南师范大学博士生，主要研究方向为计算机视觉、医疗人工智能。

Ziheng Zhou（1979− ），男，新加坡人，博士，主要研究方向为区块链、共识算法、计算机视觉和机器学习。

陈冰川（1975− ），男，广东财经大学讲师，主要研究方向为人工智能、推荐系统和软件工程。

Chunming Rong（1969− ），男，挪威人，挪威工程院院士，挪威斯塔万格大学终身教授，主要研究方向为区块链。