# SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses

https://softsec.kaist.ac.kr/~jschoi/data/ase2021.pdf

https://github.com/SoftSec-KAIST/Smartian

## Overview

在这部分，我们首先提出了一个样例来描述一个在智能合约模糊测试的独特挑战。然后我们简洁的描述一下Smartian是如何通过使用动态和静态分析来应对这个挑战。

```
1  contract C {
2    // State variables in the storage.
3    address owner = 0;
4    uint private stateA = 0;
5    uint private stateB = 0;
6    uint CONST = 32;
7
8    function C() { // Constructor
9      owner = msg.sender;
10   }
11   function f(uint x) {
12     if (msg.sender == owner) { stateA = x; }
13   }
14   function g(uint y) {
15     if (stateA % CONST == 1) {
16       stateB = y - 10;
17     }
18   }
19   function h() {
20     if (stateB == 62) { bug(); }
21   }
22 }
```

Fig. 1. Example smart contract.

### *A. Motivationg Example*

智能合约对模糊测试提出了独特的挑战，因为它们的内在结构是多个交易通过持久状态变量相互关联。

在图一，有constructor C，以及三个函数f，g和h。当我们的系统在EVM 字节码上操作时，为了便于解释，示例代码是用 Solidity 编写的。

当部署者实例化合约时运行一次，constructor C简单的存储了在存储器里部署者的地址。这确实是一种常见的模式，因为它提供了将部署者与普通用户区分开来的方法。请注意， msg.sender 是一个表达式，在运行时计算为当前发送者的地址。

在h中有个bug，当*state B*为62时，会被触发。例如，一个可以触发bug的交易序列[f(33), g[72], h()]。请注意三个交易应当按照确切的顺序来触发bug。并且f应当被部署者发送。

乍一看，每个函数的条件并不难满足，例如，在15行，随机改变stateA，有1/32的几率可以触发此行的条件。第20行的条件更难解决，但在灰盒模糊测试的最新进展中提供了可实践的解决方案。在我们的实施中，我们采用了来自Eclipser的灰盒concolic测试技术。

但是发现这个bug仍然具有难度，因为我们需要以正确的顺序生成交易。例如，让我们假设，我们有个序列$[f(*), h(), g(*)]$作为一个种子，*可以作为任意的值。函数参数的任何突变尝试都不会触发bug，因为h不能观察到stateB的任何区别。因此我们的fuzzer需要有一个类似$[f(), h(), g()]$的交易序列在种子池中发现bug。

可能有人会说，灰盒模糊器能够通过随机改变交易顺序来发现这样一个关键的交易序列。然而它并不像看起来那么不重要。即使我们设法通过尝试任意的不同交易顺序来生成一个序列[f(), h(), g()]，我们没有意识到这的确是一个有意义的测试样例因为传统代码覆盖率不够敏感。例如，考虑两个能够实现相同分支覆盖的交易序列$S_A = [f(33), g(0), h()]$和$S_B = [f(33), h(), g(0)]$。如果$S_B$已经在我们的测试用例池中，我们的模糊器没有机会添加$S_A$到种子池中，即使它是关键的一个。

初步实验：尽管图 1 中的示例合约很简单，但我们测试的现有 fuzzer 都无法找到错误。 具体来说，我们运行了三个开源智能合约 fuzzer，Echidna、ILF 和 sFuzz，每个 1 小时。 另一方面，SMARTIAN 能够在几秒钟内找到漏洞。

## *B. Our Approach*

为了应对上述挑战，SMARTIAN 同时利用静态和动态分析。 图 2 概述了 SMARTIAN 的整体架构。在高层次上，我们的系统以三个主要步骤运行：（1）INFOGATHER，（2）SEEDPOOLINIT，和（3）DATAFLOWFUZZ

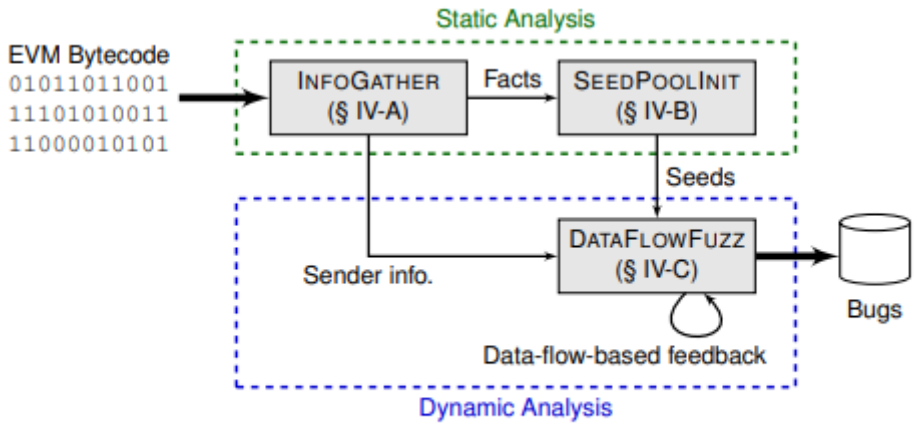


Fig. 2. SMARTIAN architecture.

1. INFOGATHER：首先，SMARTIAN 将被测 EVM 字节码作为输入，并运行静态分析以收集有用的数据流事件，以指导 SEEDPOOLINIT 和 DATAFLOWFUZZ。 具体来说，对于合约中的每个函数，SMARTIAN 会计算出该函数定义和使用了哪些状态变量，以及该函数是否将交易发送者地址与部署者地址进行比较。 从我们的示例合约中，我们的静态分析将收集以下事例。由于我们的分析直接在低级 EVM 字节码上运行，因此收集此类信息并非易事（参见 §IV-A）
    - stateA被f定义，被g使用
    - stateB被g定义，被h使用
    - owner被C定义，作为部署者的地址
    - owner被f使用，来检查交易发送者
2. SEEDPOOLINIT：根据收集到的信息，SMARTIAN 预测哪些交易序列可能导致有意义的探索，并将它们视为初始种子。 具体来说，SMARTIAN 意识到 f 必须先于 g 才能探索受 stateA 影响的执行路径。 类似地，它推断出必须在 h 之前调用 g 才能改变 stateB 的值并探索 h 中的更多路径。 最后，它发现 f 必须由部署者执行才能通过我们在第一步中确定的发件人检查。 因

此，SMARTIAN 创建 S0 = [f(0),g(0),h()] 作为模糊测试的初始种子，同时确保 f 的发送者是部署者。 transaction 参数默认设置为 0。 有关种子初始化的详细算法，请参见§IV-B。

3. DATAFLOWFUZZ：虽然用有意义的事务序列初始化种子池可以增加发现错误的概率，但这并不能立即解决整个挑战。 理想情况下，我们将首先随机变异给定的种子 S0 并获得一个新的种子 S1 = [f(33),g(0),h()]，可以到达第 16 行。如果我们可以将 S1 识别为关键测试用例，我们将把它添加到测试用例池中，然后应用前面提到的灰盒 concolic 测试来找出触发 bug 的 g 的正确参数值（参见 §IV-C）。

   不幸的是，正如我们在 §III-A 中强调的那样，如果我们仅仅使用现有的代码覆盖率反馈，我们可能无法识别这些关键的中间种子。 假设我们在 S1 之前意外生成了 S2 = [f(33),g(0)]，并将 S2 添加到测试用例池中。 这是可能的，因为我们的 fuzzer 可以从给定的种子中随机添加、删除或重新排序事务。 一旦将 S2 添加到测试用例池中，S1 将不再被认为是有趣的，因为它没有提供对现有种子 {S0，S2} 的覆盖增益。

   为了缓解这个问题，我们采用动态数据流分析来收集基于数据流的反馈。 在高层次上，我们的方法将数据流覆盖率与 Eclipser [20] 使用的分支覆盖率一起视为模糊反馈，也就是说，我们采用动态检测来观察运行时给定事务序列中发生的数据流，并使用 他们也作为反馈。 对于 S1，从第 16 行定义 stateB 到第 20 行使用 stateB 有一个数据流。但是，S0 和 S2 都没有这个数据流。 基于此，我们可以得出结论，S1 发现了一个在 S0 和 S2 中都没有观察到的有趣的程序行为。 我们在§IV-C 中详细介绍了我们的方法。

4. 数据流分析的影响：在静态和动态分析的帮助下，SMARTIAN 可以在我们的机器上在五秒钟内找到上述示例中的错误。 同时，当我们禁用我们的分析时（参见§V-B），SMARTIAN 在一小时内未能找到错误。

## C. Our Contribution over Previous Work

我们的技术贡献有两个方面：（1）我们首先提出了系统地生成智能合约模糊测试的种子；（2）我们使用基于数据流覆盖来有效的指导智能合约模糊测试。

1）种子生成：以前的模糊器会系统地生成适当的事务序列。 例如，Echidna 和 sFuzz 随机生成序列。

Harvey [70] 通过运行时启发式方法部分解决了这一挑战。 首先，Harvey 强行改变状态变量以找出受它们影响的函数。 然后它随机地将其他事务添加到这些函数中。 但是，这种方法无法扩展到具有大量功能的复杂合约。 此外，Harvey 可能无法将常量（例如我们的激励示例中的 CONST）与变量区分开来，并花费其模糊预算来更改这些常量。

ILF [35] 基于通过符号执行智能合约获得的机器学习模型生成交易序列。 虽然 ILF 可以通过学习模型潜在地找到有意义的事务序列，但结果不是确定性的，因为它基于统计推理。 此外，我们的方法是对 ILF 的补充，因为我们直接分析合约的语义来生成种子。

2）基于数据流的反馈：数据流图覆盖率的使用先前已经在数据流测试中进行了研究。 然而，除了 ContraMaster 之外，没有一个现有的模糊器使用数据流覆盖作为模糊反馈。 虽然基于数据流的反馈与 ContraMaster 具有相同的关键直觉，但 ContraMaster 使用数据流覆盖来决定是否对交易顺序执行突变。 同时，我们使用反馈来评估生成的种子，并决定是否将它们放入种子池。

# Design

## A. Information Gathering

INFOGATHER分析了EVM字节码给的，以及返回四元组<funcs，defs，uses，senderchecks>：

• Funcs 是一组已识别的函数。

• Defs 是从每个已识别函数到函数定义的状态变量的映射。

• Uses 是从每个已识别函数到函数使用的状态变量的映射。

• SenderChecks 是一组包含发件人检查例程的函数。

它首先从给定的 EVM 字节码构建控制流图 (CFG)。 它在内部反汇编 EVM 指令，并将它们提升为中间表示 (IR)。 然后，它对 IR 进行持续传播分析，以找出控制流传输指令的目的地，例如 JUMP，并识别包括构造函数在内的函数。 我们注意到，此步骤还包括解决合同中的调用边缘，以实现过程间分析。 最后，它基于抽象解释运行主要分析。

我们的主要静态分析以流和上下文敏感的方式计算存储在堆栈和内存中的抽象值 。 跟踪堆栈值很重要，因为 EVM 是一个基于堆栈的机器，它将指令操作数推入堆栈。 跟随内存值也很关键，因为这些操作数通常也是从内存中加载的。 我们的最终目标是找出每个函数定义和使用的状态变量。 为了区分使用（或定义）哪个状态变量，我们检查哪个值用作 SLOAD（或 SSTORE）指令的键。

为了近似值，我们使用包含三个不同域的产品域 。 首先，我们使用提升的整数域 来跟踪常量。 这是因为智能合约使用硬编码的唯一常量作为访问原始数据类型（例如 uint）状态变量的键。 其次，我们使用提升整数域的变体来抽象哈希指令 SHA3 的输出。 这是因为智能合约通过计算的哈希值作为键来访问复合数据类型的状态变量，例如映射。 使用这两个域，我们可以跟踪每个程序点访问的特定状态变量，因此可以相应地更新 Defs 和 Uses。

产品域的最后一个组成部分是污染域，用于跟踪部署者地址的流动（回想一下 §III-B）。 使用这个域，我们通过分析以下两个条件来计算 SenderChecks。 首先，我们检查智能合约的构造函数是否将部署者的地址保存到存储中。 其次，我们查看由 CALLER 指令返回的发送者地址是否流入条件分支，并与部署者的地址进行比较。 请注意，使用传统的静态污点分析可以轻松跟踪这两个流。 如果两个条件都成立，那么我们将包含条件的函数放入 SenderChecks。

产品域的最后一个组成部分是污染域，用于跟踪部署者地址的流动（回想一下 §III-B）。 使用这个域，我们通过分析以下两个条件来计算 SenderChecks。 首先，我们检查智能合约的构造函数是否将部署者的地址保存到存储中。 其次，我们查看由 CALLER 指令返回的发送者地址是否流入条件分支，并与部署者的地址进行比较。 请注意，使用传统的静态污点分析可以轻松跟踪这两个流。 如果两个条件都成立，那么我们将包含条件的函数放入 SenderChecks。

## B. Seed Pool Initialization (SEEDPOOLINIT)

为了生成初始种子，SMARTIAN首先根据INFOGATHER收集的信息推导出有用的函数调用顺序，然后根据顺序生成具体的交易序列。

*1*）导出有用的调用顺序：算法 1 说明了函数调用顺序的决定。 它以从 INFOGATHER 获得的 Funcs、Defs 和 Uses 作为输入，并输出一组函数序列。

**Algorithm 1:** Deriving Function Call Orders.

```
1  function GenSequences(Funcs, Defs, Uses)
2      seqs ← ∅
3      works ← InitWorks({[f] | f ∈ Funcs, Defs(f) ≠ ∅})
4      while works ≠ ∅ do
5          s ← works.pop()
6          nogain ← true
7          for f in Funcs do
8              if DataFlowGain(s || [f], Defs, Uses) then
9                  works.push(s || [f])
10                 nogain ← false
11         if nogain then
12             seqs ← seqs ∪ {s}
13     return seqs
```

在第 3 行中，我们使用包含 Funcs 中的每个函数的单例序列来初始化工作列表（works）。 我们忽略未定义任何状态变量的函数，因为它们不会影响持久状态。 接下来，我们从工作列表中拉出一个序列 s（第 5 行），并通过将 Funcs 中的每个函数附加到 s 来创建新序列。 然后，我们使用 DataFlowGain 检查每个生成的序列，以确定哪个序列覆盖了以前看不见的数据流（第 7-8 行）。 第 9 行将此类序列推送到工作列表，并在内部删除冗余条目以提高效率。

具体来说，DataFlowGain 通过从给定序列中收集三元组 $< f_1、v、f_2 >$来静态近似函数级数据流，其中 (1) f1 和 f2 是出现在序列中的函数，(2) f1 定义 v，并且 (3) f2 使用该 v。如果从序列中找到以前未见过的三元组，则返回 true。

只要观察到新的数据流，我们就会重复扩展工作列表中的序列。 如果一个序列通过扩展它没有产生增益，我们通过将它添加到输出集（第 11-12 行）来最终确定该序列。

*2)Generating Seeds*: 我们现在将生成的函数序列具体化为交易序列，主要分两步。

首先，对于每个事务中的每个函数，我们决定每个函数是否属于 SenderChecks。 如果是这样，我们将事务的发送者设置为部署者。 否则，我们随机选择发送者（部署者或用户）。

其次，我们需要初始化每个事务的函数参数。 在这里，我们也认为交易的以太量作为附加参数。 SMARTIAN 在内部将每个参数表示为字节流。 当目标合约附带其 ABI 规范时，我们利用它来设置参数类型以及相应的字节流长度。 当 ABI 规范不存在时，SMARTIAN 将简单地将每个字节流的长度设置为预定义的最大值。 在没有 ABI 的情况下正确推断函数参数的数据类型超出了本文的范围，我们将其留作未来的工作。

## C. Data-Flow-Based Fuzzing (DATAFLOWFUZZ)

使用生成的初始种子池，SMARTIAN 迭代地选择一个并对其进行变异以生成新的测试用例（§IV-C1）。 SMARTIAN 然后通过对每个测试用例运行被测智能合约来评估新生成的测试用例的有用性（§IV-C2）。 在每次执行期间，我们的错误预言机都会检查它是否存在错误（§IV-C3）。

1）突变方法：SMARTIAN 采用两种互补的策略来突变种子。 一种是随机突变，另一种是来自 [20] 的灰盒 concolic 测试。 SMARTIAN 在它们之间交替以实现协同作用。

首先，我们的随机变异策略在序列级别和事务级别都运行。 序列级变异由以下操作组成：（1）为随机函数插入新事务；（2) 删除随机交易；（3) 交换两个随机交易。 插入交易时，我们参考从静态分析中收集的 SenderChecks 并使用它来决定发送者，如 §IV-B 中所示。 事务级突变主要修改每个事务的参数。 我们利用广泛用于灰盒模糊器的经典变异算子，例如位翻转变异。 此外，我们也随机改变交易的发送者。

然而，众所周知，随机突变很容易卡在条件分支上，例如magic value检查 。 Eclipser 通过引入灰盒 concolic 测试技术解决了这一挑战，该技术的操作类似于传统的 concolic 测试，但没有 SMT 解决方案 或昂贵的额外仪器。

2）基于数据流的反馈：回想一下§III-A，以前的代码覆盖率反馈不足以在 fuzzing 活动中识别有趣的种子。 为了克服这个问题，SMARTIAN 除了传统的代码覆盖率反馈外，还引入了基于数据流的反馈。也就是说，当种子展示出以前未见过的数据流或覆盖以前未访问过的代码时，SMARTIAN 认为它很有趣。

为了收集数据流，我们通过修改 EVM 模拟器来动态检测 EVM 字节码，以便在执行期间监控存储访问。 特别是，我们使用 def-use 链捕获动态数据流。 令 $p_1\stackrel{v}{\rightarrow} p_2 $是在程序点 p1 中定义的状态变量 v 上的定义使用链，并在程序点 p2 中使用。 然后，我们可以从图 1 中的示例中表示 def-use 链，如下所示。 SA 产生了 $12 \overset{stateA}{\rightarrow} 15$和 $16 \overset{stateB}{\rightarrow} 20$，而 SB 只产生$12 \overset{stateA}{\rightarrow} 15$。因此，SMARTIAN 可以识别出 SA 表现出 SB 没有呈现的有趣的程序行为。 在实际实现中，我们使用指令地址作为程序点 pi ，使用存储的键作为状态变量 v。

回想一下，我们还在 §IV-B 中的种子池初始化期间检查数据流。 请注意，在算法 1 中，我们在函数级别静态近似数据流。 同时，在模糊测试中，我们跟踪实际发生的数据流，并以指令级粒度计算它们。 也就是说，我们首先静态分析数据流，以确定有希望的交易序列，这些交易序列可能会在模糊测试期间揭示更多动态数据流。 然后，我们在模糊测试阶段采用具体和更细粒度的数据流作为反馈。

3)Bug Oracles：在这里，我们总结了SMARTIAN 支持的13 类bug 的bug oracle 实现。 同样，我们修改了 EVM 模拟器来实现这些错误预言。 因此，我们的运行时工具既负责收集基于数据流的反馈，又负责在执行期间检测错误。

**AF** 在字节码级别，断言失败对应于 INVALID 指令的执行。 因此，我们可以通过检查是否执行了 INVALID 指令来精确检测 AF。 我们注意到编译器还会自动插入断言语句以防止诸如被零除之类的错误。 我们也将这些编译器插入的断言的失败视为 AF

**BD** 我们利用动态污点分析来检查块状态是否会影响以太传输。 为此，我们跟踪直接和间接的污染流。 我们首先污染获取块状态的指令的返回（例如 TIMESTAMP、NUMBER）。 然后，我们监控受污染的值是否流入 CALL 或 JUMPI 的操作数。

**CH** 首先，如果普通用户可以将 DELEGATECALL 的目标合约设置为任意用户合约，我们会发出警报。 其次，如果 JUMP 指令的目标是可操作的，我们也会报告警报。

**EL** 我们使用类似于 Mythril 的预言机，它检查普通用户是否可以通过向合约发送交易来获得以太币。但是，这很容易出现误报，因为某些合约允许部署者将合约的所有权移交给另一个用户。在此类合约中，用户可以在部署者允许的情况下提取合约余额是一种预期行为。为了避免这种误报，我们仅在事务序列没有来自部署者的任何先前事务时才报告警报。

**IB** 我们监控 ADD、SUB、MUL 指令以检查它们是否导致整数上溢/下溢。如果是这样，我们会污染结果值，并执行动态污点分析以检查污染值是用于确定要转移的以太量，还是用于更新状态变量。这是为了避免在良性整数上溢/下溢时发出警报。例如，如果没有这种污点分析，我们将对安全代码片段 $'if(x+y<x)revert();'$发出警报。

**ME** 我们运行污点分析以确保 CALL 指令的返回值流入 JUMP I 指令的谓词中。 如果有 JUMP I 未使用的返回值，我们会报告警报。

**MS** 我们检测到单个交易中发生了多个以太币转移。

**RE** 我们首先监控以太传输期间是否存在循环调用链，就像 ContractFuzzer 或 sFuzz 所做的那样。此外，我们使用污点分析来识别影响这种以太转移的状态变量，类似于我们对 BD 所做的。 然后，如果这些变量在转移发生后更新，我们会报告 RE。

**RV** 我们检查 REVERT 指令的执行，这对应于违反要求。

**SC** 我们检查普通用户是否可以执行 SELFDESTRUCT 指令并销毁合约。 与 EL 类似，我们通过过滤掉序列中具有来自部署者的任何先前事务的测试用例来减少误报。

**TO** 我们污染了 ORIGIN 指令的返回值，并检查它是否流入了 JUMP I 指令的谓词中。

对于剩下的bug种类，我们实现了与 ContractFuzzer (FE) 和 Harvey (AW) 相同的错误预言机。

# D. Implementation

为了实现我们的静态分析器，我们使用 B2R2 [37] 作为前端来解析和反汇编 EVM 字节码。 静态分析的主要逻辑 (§IV-A) 是用 1,053 行 F# 代码编写的。 SMARTIAN (§IV-C1) 的模糊测试组件是通过扩展 Eclipser [19] 以使用 EVM 和事务序列来实现的，它由 3,112 行 F# 代码组成。 我们使用 Nethermind EVM [7] 来部署合约并使用动态工具模拟交易。 具体来说，我们向 Nethermind 添加了 979 行 C# 代码来实现基于数据流的反馈 (§IV-C2) 以及我们的错误预言 (§IV-C3)。 我们将所有源代码和基准测试公开在：https://github.com/SoftSec-KAIST/Smartian。