# 《Memory-Safe_Elimination_of_Side_Channels》

# 论文复现报告[1]

张序

## 1. 特别说明：

本论文的实验复现主要是基于Soares等人于2021年[1]在IEEE-CGO(Code Generation and Optimization)上发表的文章。

在实际执行输出的图片里，将使用 Meng 替换 Wu。

## 2. 背景：

一个系统的行为如果除了输入$x$和输出$y$外还有额外的输出信息，比如不定的运行时间等，我们将这些额外的信息称为侧向通道。我们称一个程序是同步传输的，如果它能在任何输入下都执行相同的指令并且访问同一内存。因此我们需要一种转换机制使得给定的程序再经过转换后实现同步传输，以尽可能地消除侧向通道。这种转换机制在这里也称为修复程序。

在过去的研究中，Wu 等人[2]的发明 SC-Eliminator 通过 gem5[3]进行模拟，无论是从理论还是实践上，均够相对较好地解决消除侧向通道。然而它有一个缺陷：转换后的程序可能将越界内存访问插入到原始安全的代码中。因为通常情况下，对于一种转换，想要同时满足时间不变性和内存安全是困难的。本文将在这样一个背景下进行进一步的研究。

## 3. 研究问题：

本文将在保证消除侧向通道的前提下，研究如何保证内存安全。

## 4. 方法：

这里具体开发优化了一种代码转换技术。转换后的程序能够保证数据一致性（无论输入如何，在 cache 上读或写同一集合下的地址，尽管访问顺序可能发生变化）。它能够保证如下三个特性：

（1）转换机制是内存安全；

（2） 操作不变性（无论输入如何，一个转换后的程序总是读取同一串指令 cache 下的地址）；

（3）数据不变性（无论输入如何，一个程序转换后后的程序总是读或写同一串数据 cache 下的地址）。

以下将详细介绍具体实现原理，具体分为四个部分。第一个部分介绍一种玩具语言，其包括了描述我们的方法所需要的最少指令集。第二部分定义了一套转换规则，它能够作用于玩具语言，从而生成同步传输的程序。第三部分解释了在面对普遍不可能的情况

下达成妥协的基于协议的方法（内存协议：令$f(\ldots,a,n,\ldots)$为一个包括至少两个声明的函数，其中$a$是一个数组，$n$是一个整数。通过$(f,a,n)$元组形成的内存协议是一个前提条件。它表明每次当$f$被调用时，数组$a$至少有$n$个有效单元。）。第四部分则展示了如何将这个想法扩展到过程间的环境中。文章使用了 Fig 1 的程序来说明同步传输化一个程序所需要的步骤。

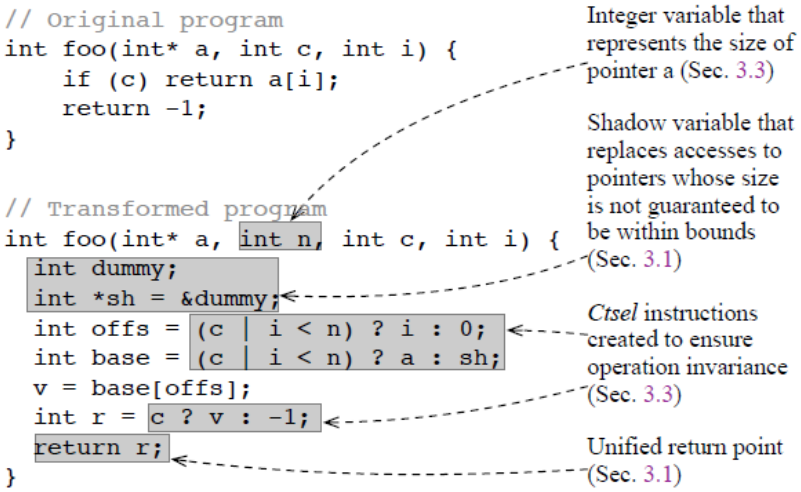


Fig 1. 文章讨论的代码转换技术所进行的处理

## 4.1. 语言基础

在 Fig 2 中，有显示这个玩具语言，它将用于解释我们的方法。这个语言包括了一个小的但是图灵完备的三地址代码指令。其中{}表示零或多次出现，[]表示可选术语，$id$表示变量名称，$n$表示数字，$\ell$表示基本块标签的范围。

Program ::= { BasicBlock }
BasicBlock ::= [ $\ell$: ] { Instruction } Terminator
Instruction ::= alloc (*id*, Expr)
| mov (*id*, Expr)
| load (*id*, *id*, Value)
| store (Value, *id*, Value)
| phi (*id*, Value : $\ell$ { , Value : $\ell$ })
| ctsel (id, Value, Value, Value)
Terminator ::= jmp ($\ell$)
| br (Value, $\ell$, $\ell$)
| ret (Expr)
Expr ::= Value | Unop Value | Value Binop Value
Value ::= $n$ | *id*
Unop ::= – | ! | ~
Binop ::= + | – | * | & | "|" | = | < | ...

Fig 2. 用于实现加密函数的语言基础

我们假设其具有四个特点：

（1） 程序均以静态单任务形式存在[4] 。

（2） 它们包括返回点。

（3） 它们包括虚拟变量，称之为$shadow$，这里表示为$sh$。

（4） 程序是无循环的。（有循环必须展开，在编译时即可得知迭代次数。否则会

出现“”）

## 4.2. 代码转换机制

1) 恒定时间选择器：

本文使用一种特殊的指令$ctsel$。$ctsel(x, c, v_t, v_f)$将$v_t$赋予$c$，如果$c$是真的，否则将$v_f$赋予$c$。

采集条件$ctsel$是通过条件$c$参数化的，其控制了对于$x$的赋值。此处需要将基本块映射到控制它们的谓词上。

路径条件：一个路径条件$c_{o\to n}$是由一个$p_i \& \ldots \& p_k, 0 \le i \le k \le n$的为此构成的。一旦它是真的，将强制执行由$l_n$标记的基本块。每个$p_i$是一个控制分支的谓词。这个分支是从程序开始到$l_n$的路径上的分支之一。

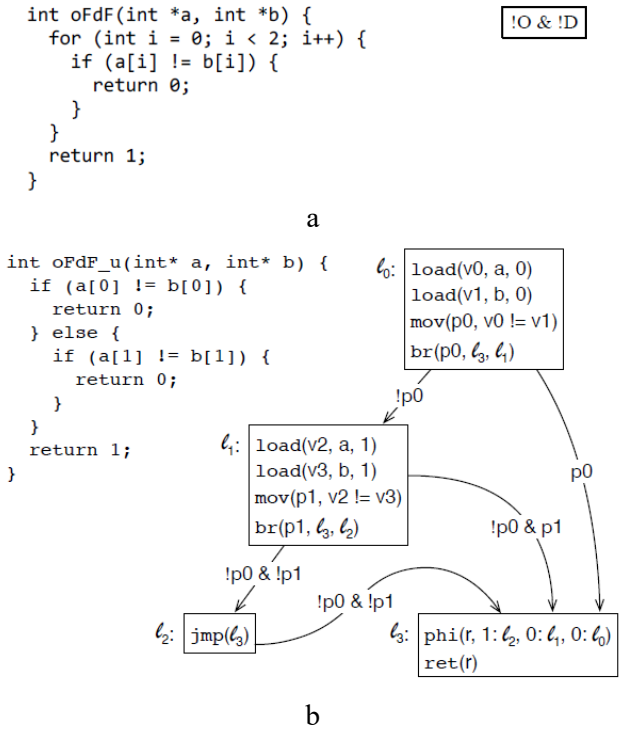


Fig 3. 与程序每个执行块相对应的路径条件

在 Fig 3 中显示了将 oFdF 循环展开后的结果。当$p0$和$p1$都是假的时，$\ell_2$处的$jmp(\ell_3)$执行。因此$!p0 \& !p1$是一个控制此操作执行的路径条件。

[Incoming]

$\mathtt{In}[\ell] = \{\, \mathtt{inc}(\ell, \mathtt{terminator}(\ell_p)) : \ell_p \in \mathtt{predecessors}(\ell) \,\}$

where $\mathtt{inc}(\ell, \mathtt{br}(p, \ell, \_)) = \mathtt{Out}[\ell] \,\&\, p$

$\mathtt{inc}(\ell, \mathtt{br}(p, \_, \ell)) = \mathtt{Out}[\ell] \,\&\, !p$

$\mathtt{inc}(\ell, \_) = \mathtt{Out}[\ell]$

[Outgoing]

$\mathtt{Out}[\ell] = c_0 \mid \ldots \mid c_n,\ c_i \in \mathtt{In}[\ell]$

Fig 4. 数据流分析，查找控制流图中控制基本块执行的路径条件

Fig 4 中的数据流分析收集了程序中控制每个基本块的条件。此分析将路径条件分为传入或传出的。一个基本块具有可变数量的传入条件：每个前置条件一个。相比之下，它只有一个传出条件，即传入条件的合取。只要传出条件为真，块就在执

行路径中。

首先生成一个从标签到新创建的变量的映射，该变量将存储每个块的传出条件。使用 Fig 4 中的符号，将映射命名为$Out$。输出条件是独特的，其在执行后将保持不变。一旦得到了$In[l]$，其将保持不变。所以，控制好流图的先序遍历足以收集格式良好的 SSA 形式的程序所产生的所有传入和传出条件。[1]

2) 转换规则：

$$[\mathtt{phi}_1] \quad (\mathtt{phi}(x, v_0{:}\ \ell_0), \_) \xrightarrow{i} (\{\mathtt{mov}(x, v_0)\}, \emptyset)$$

$$[\mathtt{phi}_2] \quad \frac{\mathtt{In}[\ell] = \{c_0, c_1\}}{\mathtt{In} \vdash (\mathtt{phi}(x, v_0{:}\ \ell_0, v_1{:}\ \ell_1), \ell) \xrightarrow{i} (\{\mathtt{ctsel}(x, c_0, v_0, v_1)\}, \emptyset)}$$

$$[\mathtt{phi}_n] \quad \frac{\begin{array}{c}\mathtt{In}[\ell] = \{c_0, c_1, ..., c_k\}, \quad \mathtt{In}[\ell \mapsto \{c_1, ..., c_k\}] = \mathtt{In}', \\ \mathtt{In}' \vdash (\mathtt{phi}(z, v_1{:}\ \ell_1, ..., v_k{:}\ \ell_k), \ell) \xrightarrow{i} (I, V), \\ I \cup \{\mathtt{ctsel}(x, c_0, v_0, z)\} = I'\end{array}}{\mathtt{In} \vdash (\mathtt{phi}(x, v_0{:}\ \ell_0, v_1{:}\ \ell_1, ..., v_k{:}\ \ell_k), \ell) \xrightarrow{i} (I', V \cup \{z\})}$$

$$[\mathtt{load}] \quad \frac{\begin{array}{c}\mathtt{Out}[\ell] = c, \quad \mathcal{L}[m] = n, \\ \{\mathtt{mov}(z_0, \mathtt{idx} < n), \mathtt{mov}(z_1, c \mid z_0), \mathtt{ctsel}(z_2, z_1, \mathtt{idx}, 0) \\ \mathtt{ctsel}(z_3, z_1, m, \mathtt{sh}), \mathtt{load}(x, z_3, z_2)\} = I\end{array}}{\mathtt{Out}, \mathcal{L} \vdash (\mathtt{load}(x, m, \mathtt{idx}), \ell) \xrightarrow{i} (I, \{z_0, ..., z_3\})}$$

$$[\mathtt{store}] \quad \frac{\begin{array}{c}\mathtt{Out}, \mathcal{L} \vdash (\mathtt{load}(z_4, m, \mathtt{idx}), \ell) \xrightarrow{i} (I, \{z_0, ..., z_3\}), \\ \mathtt{Out}[\ell] = c, \quad \{\mathtt{ctsel}(z_5, c, v, z_4), \mathtt{store}(z_5, z_3, z_2)\} = I'\end{array}}{\mathtt{Out}, \mathcal{L} \vdash (\mathtt{store}(v, m, \mathtt{idx}), \ell) \xrightarrow{i} (I \cup I', \{z_0, ..., z_5\})}$$

$$[\mathtt{br}] \quad (\mathtt{br}(p, \ell_t, \ell_f, \ell') \xrightarrow{f} \mathtt{jmp}(\ell')$$

Fig 5. 用于程序修复的一些转换规则

此处我们使用 Fig 5 中的重写规则来执行程序修复。它是基于以下两个关系进行建模的。

$$(i, \ell) \xrightarrow{i} (I, V) \tag{1}$$

$$(t, \ell) \xrightarrow{f} t' \tag{2}$$

关系(1)将$i \in l$中的指令集映射到一个新的集合操作$I$中，同时伴随有一个集合$V$，其有新的变量名，它是由$I$中的指令定义的。这里使用记号$In[l \mapsto \{c_1, \dots, c_k\}]$来指明在表$In$中与$\ell$相关的项已经被更新为$\{c_1, \dots, c_k\}$了。$store$与$load$规则需要$Out$映射，以指明是否控制去$\ell$的流。

此处我们使用位于阵列与它们大小之间的映射$\mathcal{L}$以保证数据不变性。这些也规则使用$sh$以保证内存安全性。$load$使用$sh$是显式的。隐变量中的地址可能流向变量$z_3$,它表示$load$间接引用的位置。另一方面，$store$规则间接通过$z_3$引用了$sh$。

关系(2)则将$t \in Terminator$映射到了另一个$t' \in Terminator$。规则$br$将条件替换为了一个无条件的声明。在这种情况下，$l'$按拓扑顺序标记了紧跟在包含$br$操

[1] 如果变量的定义主导了其所有用途，则这个 SSA 形式的程序是格式良好的。

作后的基本块。

我们在 Fig 6 中说明了利用提及到的规则实现后的程序的效果。而且能够说明，Fig 7 中的转换满足正确性（语法是得到保证的）、操作不变性、数据不变性、满足同步传输条件的。

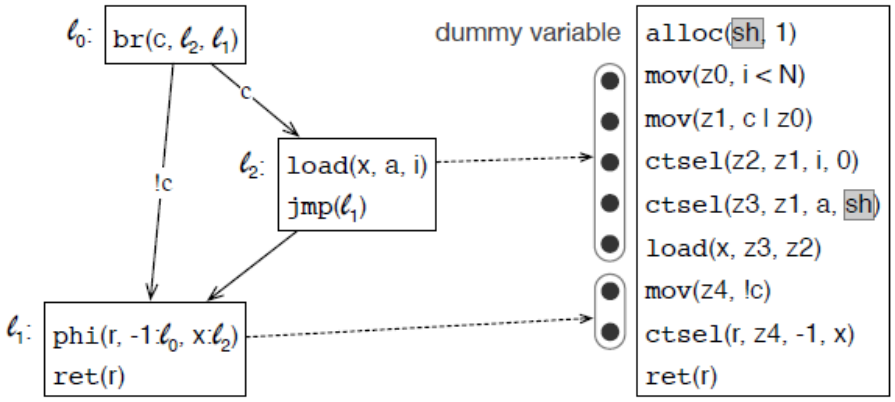


Fig 6. 左侧的程序是 Fig 1 的低层级表示。右侧程序是使用 Fig 5 中的重写规则实现的同步传输程序。

## 4.3. 约束协议

如果约束协议的前提得到了满足，那么转换代码是具有操作与数据不变性的，同时能够保证内存的安全性。否则数据不变性将不能够得到保证。此处协议的创建分为下面三个操作：

（1） 创建接口：由于协议的创建，函数签名发生了改变。这里我们为函数接口的每个指针均添加一个整数。这个整数是阵列 a 大小的下界。

（2） 调用点的改变：当应用于整个程序时，这种转换会更改调用修改后的函数的调用点。这一修改涉及添加表示数组大熊的表达式作为函数调用的额外参数。文章使用了 Pasiante 等人的[5]静态分析以推测阵列的长度。Paisante 等人提出了一种前向必要分析，它将指针绑定到表示其大小的符号表达式。要获得程序间分析（从而扩展程序内分析 Paisante 等人的工作），文章依赖于这样的知识：每个指针后面的函数参数表示指针的最大偏移量。文章利用这一观察结果来传播跨功能的信息。

（3） 隐藏内存：隐变量充当内存位置的占位符，这些内存位置的访问不能保证在约定的范围内。每个函数存在一个隐变量。它的大小等于体系结构中最大可寻址字的大小。Fig 7 显示了将协议作为先决条件添加到内存访问的重写过程。它可以保证内存访问的安全。

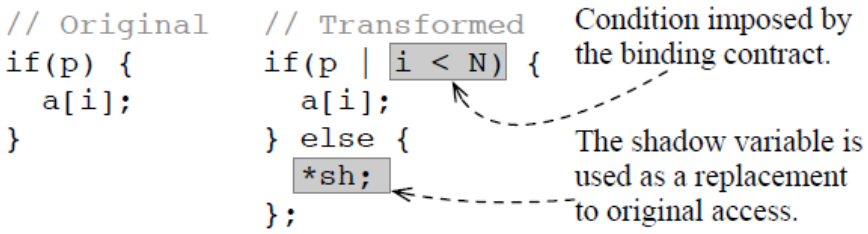


Fig 7. 将潜在不安全内存访问替代为访问隐变量的重写过程

令$f$是一个函数，只要能够满足内存协议的前提，Fig 5 的转换是内存安全的。

## 4.4. 过程间分析

在同步传输时，为了避免内联，这里进行了原理重写。Fig 8 展示了具体过程。转换后的程序调用的每个函数以三种方式修改。第一，被调用者的签名被增加了新的条件(Fig 8-i)。第二，被调用者的主体被条件测试包围，且收到新条件的保护(Fig 8-ii)。第三，修改调用者，将调用点的路径条件传递给被调用者(Fig 8-iii)。条件作为转换本身的一部分进行计算。

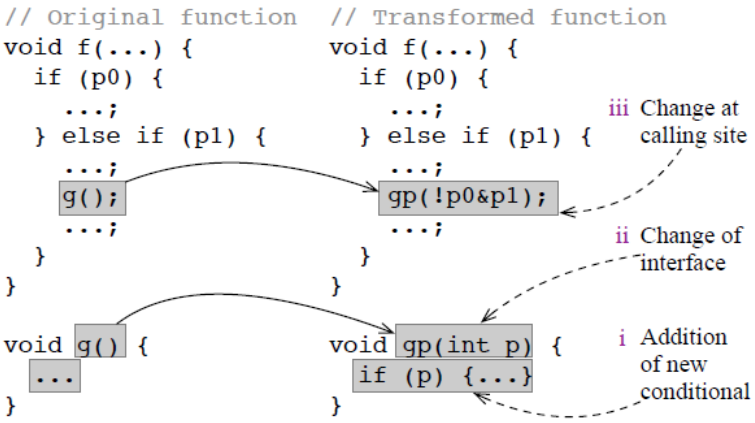


Fig 8. 过程间转换

# 5. 本地复现实验结果：

## 5.1. 环境搭建

### 5.1.1. 软硬件基础配置

参照[1]的硬件要求，本次实验使用了Ubuntu20.04的linux系统，且保证了8GB的内存Fig 9。

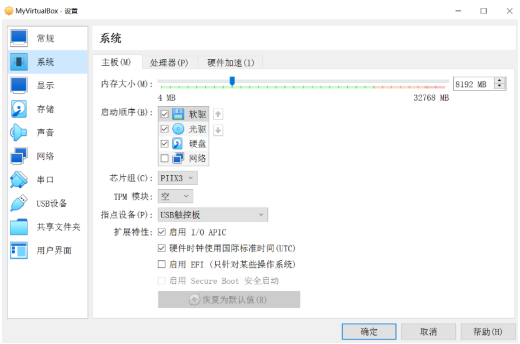


Fig 9. 硬件内存分配

而后，经过较长时间的配置，使得软件包配置成功后符合[6]里Readme文件下软件的配置需求后，所有必要配置显示如下见Fig 10。

```
root@ElyonZ-VirtualBox:/home/elyonz2002# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.6 LTS
Release:        20.04
Codename:       focal
root@ElyonZ-VirtualBox:/home/elyonz2002# cmake --version
cmake version 3.28.0-rc5

CMake suite maintained and supported by Kitware (kitware.com/cmake).
root@ElyonZ-VirtualBox:/home/elyonz2002# llvm-as --version
Ubuntu LLVM version 13.0.1

  Optimized build.
  Default target: x86_64-pc-linux-gnu
  Host CPU: znver3
root@ElyonZ-VirtualBox:/home/elyonz2002# clang --version
Ubuntu clang version 13.0.1-++20220120110924+75e33f71c2da-1~exp1~20220120231001.58
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/lib/llvm-13/bin
root@ElyonZ-VirtualBox:/home/elyonz2002# stack --version
Version 2.3.1, Git revision de2a7b694f07de7e6cf17f8c92338c16286b2878 (8103 commits) x86_64 hpack-0.33.0
```

a.

```
root@ElyonZ-VirtualBox:/home/elyonz2002# python --version
Python 3.8.10
root@ElyonZ-VirtualBox:/home/elyonz2002# docker --version
Docker version 20.10.24, build 297e128
```

b.

Fig 10. 软件配置

### 5.1.2. 仓库选择

而后，我们将[6]中的仓库从实体机上下载，并放入主页面下Fig 11。

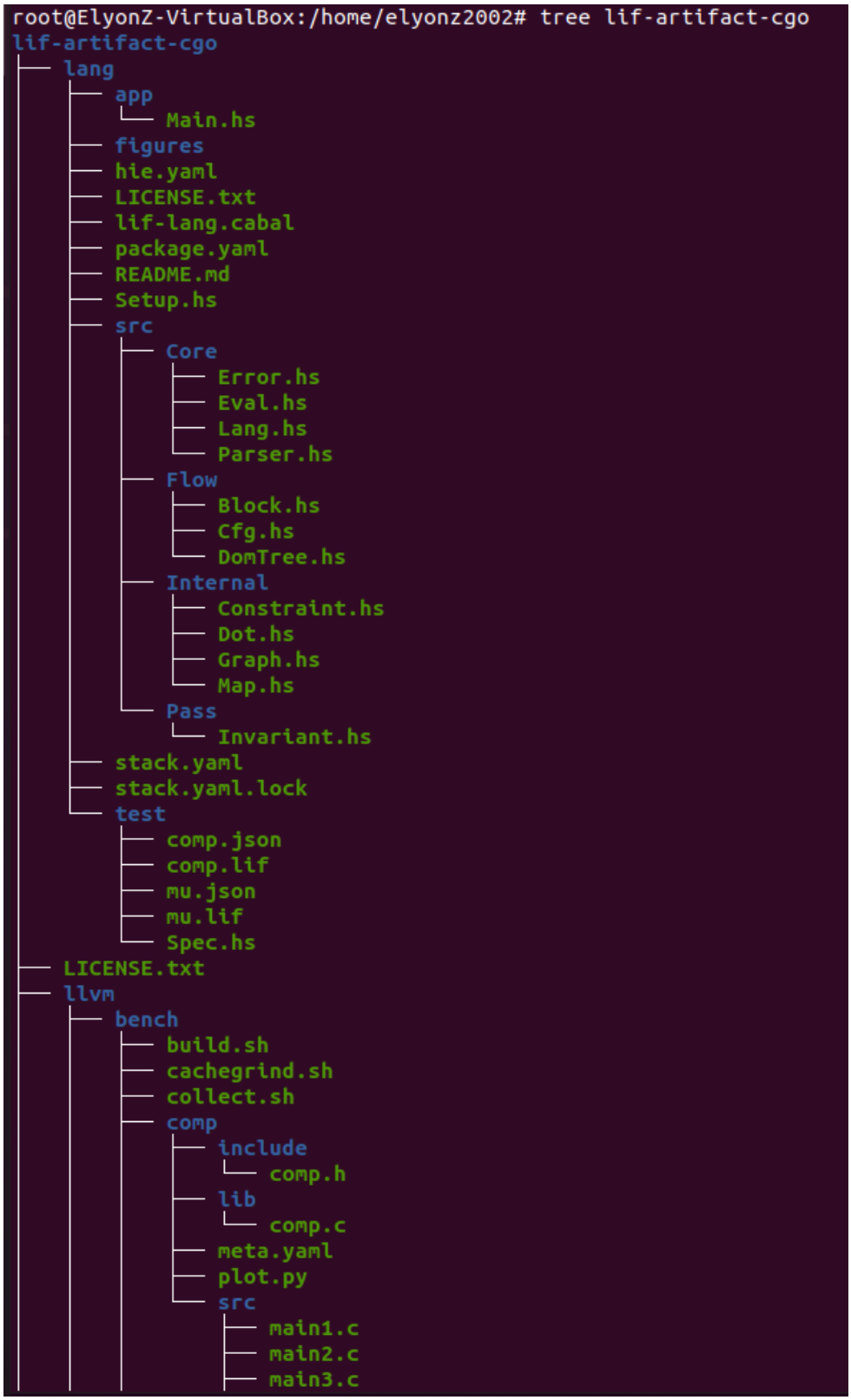


Fig 11. 软件lif-artifact-cgo仓库包含内容的完整性

（此处仅能够部分显示，但仓库与github上原始仓库所包含的文件基本上是一样的。）

### 5.1.3. Artifact 文件

下载 Artifact，以复现代码 Fig 12。

进入 lif-artifact-cgo/lang 目录下执行 docker pull luigidcsoares/lif:cgo 指令。

```
root@ElyonZ-VirtualBox:/home/elyonz2002/lif-artifact-cgo/lang# docker pull luigidcsoares/lif:cgo
cgo: Pulling from luigidcsoares/lif
4d6a3daaa4e1: Pull complete
92b65ac2377d: Pull complete
5d5a7d9d2712: Pull complete
05aae39893a7: Pull complete
74d01483340c: Pull complete
832db2d57aaa: Pull complete
83d9772198e2: Pull complete
df5bba184033: Pull complete
0e987a9e35e3: Pull complete
754e376df8ed: Pull complete
5ffb53e019bc: Pull complete
a0b94ae01231: Pull complete
685e81be0800: Pull complete
e972e4ec65c2: Pull complete
Digest: sha256:413eb2cfe0e6d843bb8ac881e7ebbfafbd3ddbdc54aa611304ce619ac8b8f4b5
Status: Downloaded newer image for luigidcsoares/lif:cgo
docker.io/luigidcsoares/lif:cgo
```

Fig 12. Artifact 下载记录

## 5.2. 实验问题与其它准备：

### 5.2.1. 问题

1) 修复程序所花费的时间？
2) 程序修复技术对程序添加了多少时间开销？
3) 程序修复技术对程序添加了多少空间开销？

### 5.2.2. 其它准备

1) 衡量指标：为了解决这些实验问题，作者使用了综合指标和实际加密例程。后者由CTBench[7]和$SC-Eliminator$[3]一起分发的基准子集组成。当给出$CTBench$的三个指标时，$SC-Eliminator$未成功终止。此外，应用于$loki91$和$oFdF$时，它会产生错误代码。所以在讨论与时间相关的绝对和平均数字时，会忽略这五个基准。但在图表中会包括它们，因为它们能被有效处理。
2) 在文章中，作者假设加密例程中使用的每个输入都是敏感的。
3) 在报告修复程序所占用的时间时，每个实验运行 50 次；而对于修复的程序，则运行 1000 次。在两种情况下，作者都使用阈值为 3 的 z 分数来消除异常值。此外此处还展示了利用两种版本获得的结果，二者分别是未经优化和经由$opt-01$优化后的结果。
4) 验证：这里使用了$cachegrind$来确保所有修复过的程序符合 3 中规定的条件。Valgrind 验证了所有测试输入下所有程序的操作不变性和内存安全性。这个验证过程应用在转换后的程序及其优化版本上。对于 12 个基准测试，数据不变性得到了保证。剩下的 12 个无法实现数据不变性：其中 11 个本质上是数据不一致的，因为它们使用输入来索引内存。对于另一个程序，作者的静态分析无法找到数组的符号表达式。通过手动提供这些边界，可以确保数据不变性。

## 5.3. 实验复现：

### 5.3.1. 实际操作：

在终端执行 Fig 13 所示指令。

```
root@ElyonZ-VirtualBox:/home/elyonz2002/lif-artifact-cgo/lang# docker run \
> -v $(pwd)/figures:/lif/llvm/bench/figures \
> -it luigidcsoares/lif:cgo /bin/bash
```

Fig 13. 进入容器指令

进入容器后，执行 Fig 14 所示指令。

```
[root@4833b00fc948 bench]# ./run.sh -b
```

a.

```
[root@4833b00fc948 bench]# ./run.sh -c && ./run.sh -p
```

b.

```
[root@4833b00fc948 bench]# cp results/pass_time.pdf figures/11.pdf \
> && cp results/exec_time.pdf figures/13.pdf \
> && cp results/size.pdf figures/15.pdf \
> && cp comp/results/pass_time.pdf figures/12.pdf \
> && cp comp/results/exec_time.pdf figures/14.pdf \
> && cp comp/results/size.pdf figures/16.pdf
```

c.

Fig 14. 运行实验

新建终端，保持容器开启状态并并 cachegrind 的报告，见 Fig 15。

```
[root@4833b00fc948 bench]# vim -r meng/chronos/aes/results/exec_time.csv
[root@4833b00fc948 bench]# vim -r meng/chronos/aes/results/pass_time.csv
[root@4833b00fc948 bench]# vim -r meng/chronos/aes/results/size.csv
[root@4833b00fc948 bench]# vim -r  meng/chronos/aes/results/cachegrind.csv
```

Fig 15. cachegrind 报告。

与此同时，也可以访问 Fig 16 里实验生成的具体图片。

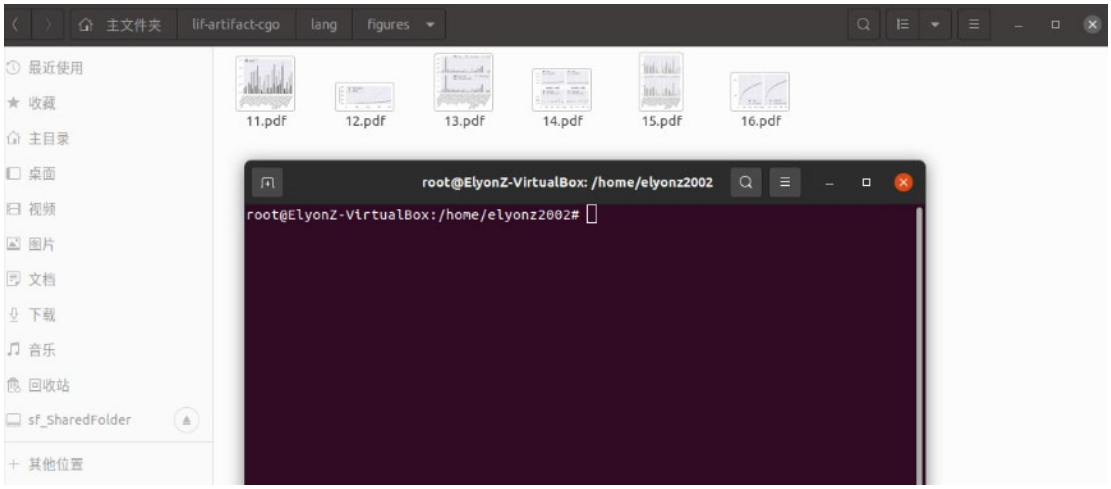


Fig 16. 实验报告图片

至此实验复现部分完成。

以下将结合论文里实验的具体执行过程，详细阐述解释。

### 5.3.2. Q1-修复时间

这一部分记录了作者与$SC-Eliminator$修复程序所消耗的时间。仅仅记录了修复的时间，而不包括其余 LLVM 的处理时间。

加密例程：在 Fig 17 中，有绘制复现实验关于文章方法与$SC-Eliminator$修复加密例程的实现。文章消耗时间显然要快于 Wu 等人的方法。

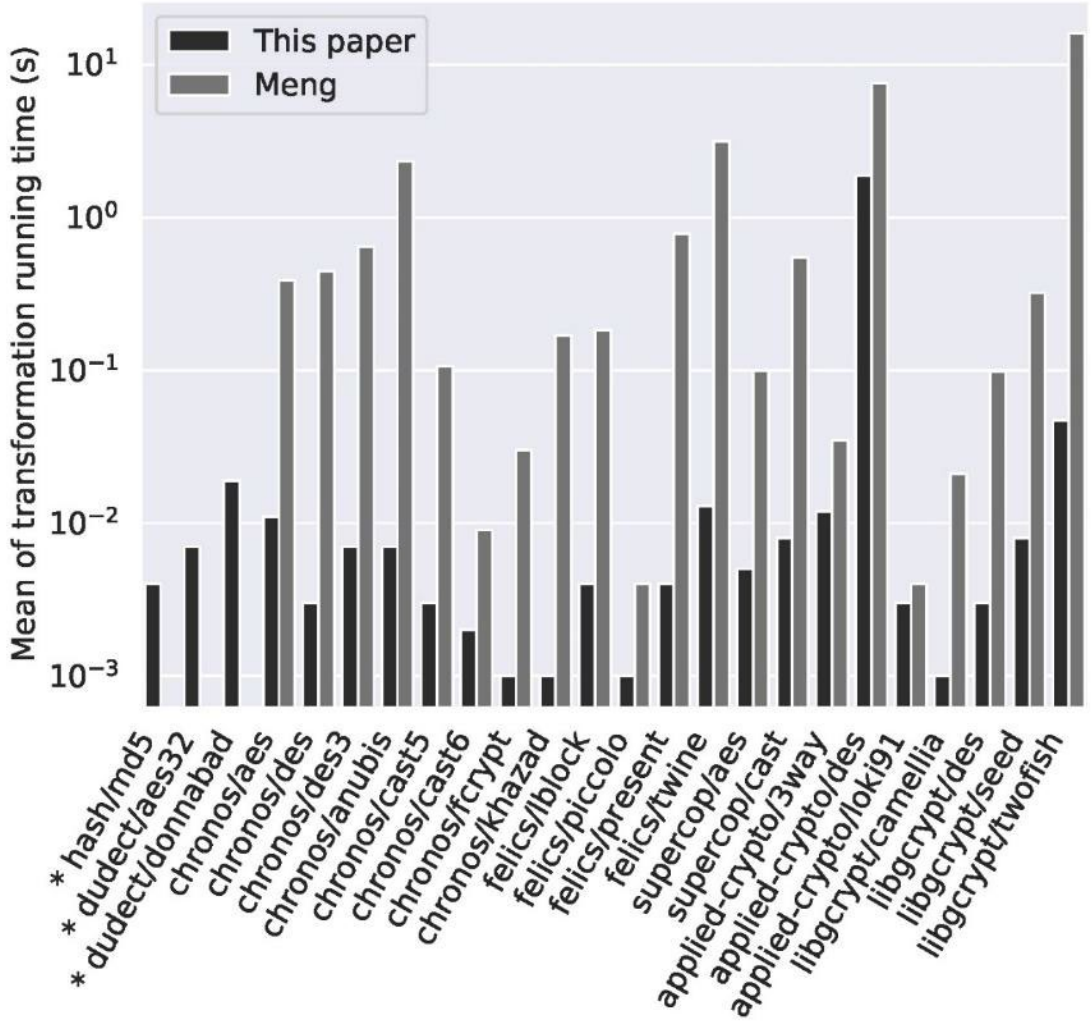


Fig 17. 修复加密例程消耗时间。带有星号前缀的基准测试是那些在 Wu 等人的工件中无法运行的。

经验渐进行为：在 Figure 18 中，本文对比了文章方法与$SC - Eliminator$于$OFDF$例程上平均传输时间的对比。对于大小为$N$的阵列，主循环被静态修改为进行$N$次迭代。其实现似乎都是线性的，而且文章的方法的斜率显然要更小，从而扩展性更好。

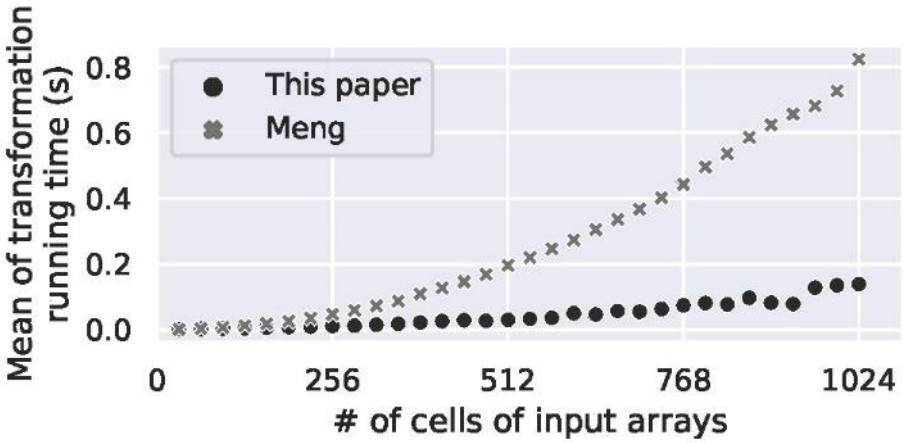


Fig 18. 修复函数$oFdF$的时间，考虑数组 a 和 b 的不同大小。这个大小决定了循环中的最大迭代次数。

### 5.3.3. Q2-增加的时间开销

修复程序会因为插入额外指令以保持时间不变性而导致程序减速。本节分析了这种影响。

加密例程：Fig 19 显示了程序修复对密码学基准测试的影响，考虑了没有优化和有优化的代码。

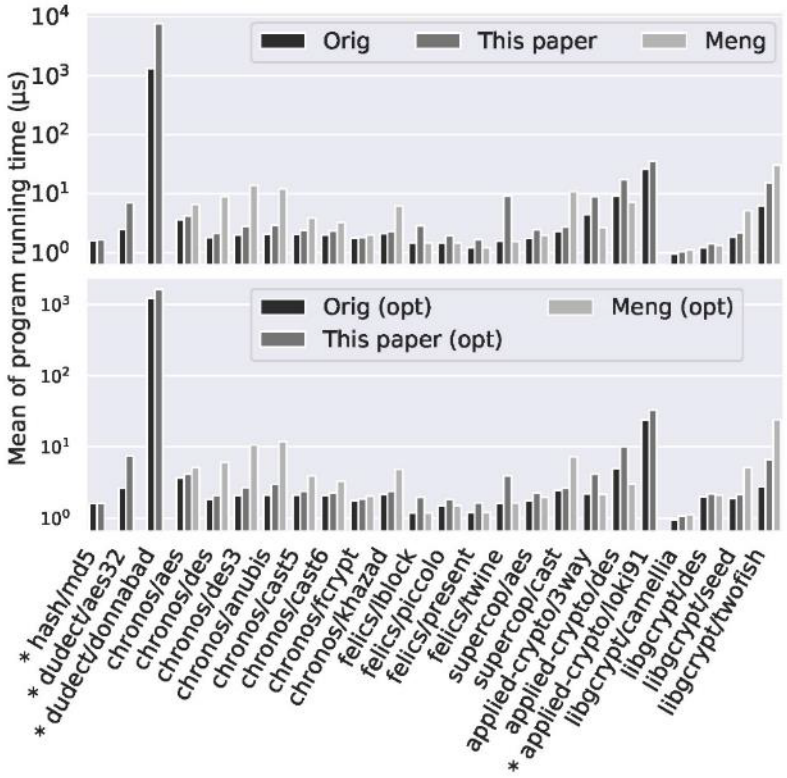


Fig 19. 程序修复对密码学例程运行时间的影响。吴等人的程序修复方法对于带有星号前缀的基准测试无法转换或产生不正确结果。

显然作者等人的方法增加的时间开销要小于 Wu 等人。

经验渐进行为：Fig 20 比较了不同大小输入阵列下函数$oFdF$的原始版本和转换后的版本。无论阵列的内容如何，转换后的函数将执行相同的操作。Fig 20(a)和(b)的视觉比较暗示了这一点。相反，原始函数仅在两个输入数组存储相同值时才运行主循环的所有迭代。两个程序都有线性近似，但此论文的转换程序在没有优化的情况下，导致了近似 4 倍的额外时间运行。两个程序在输入数组中的单元数$N$上都显示出线性行为。编译器的优化对程序有着较大的影响。在 Fig 20 下方的图片说明了原始程序与优化后的程序处理总的运行时间是相近的

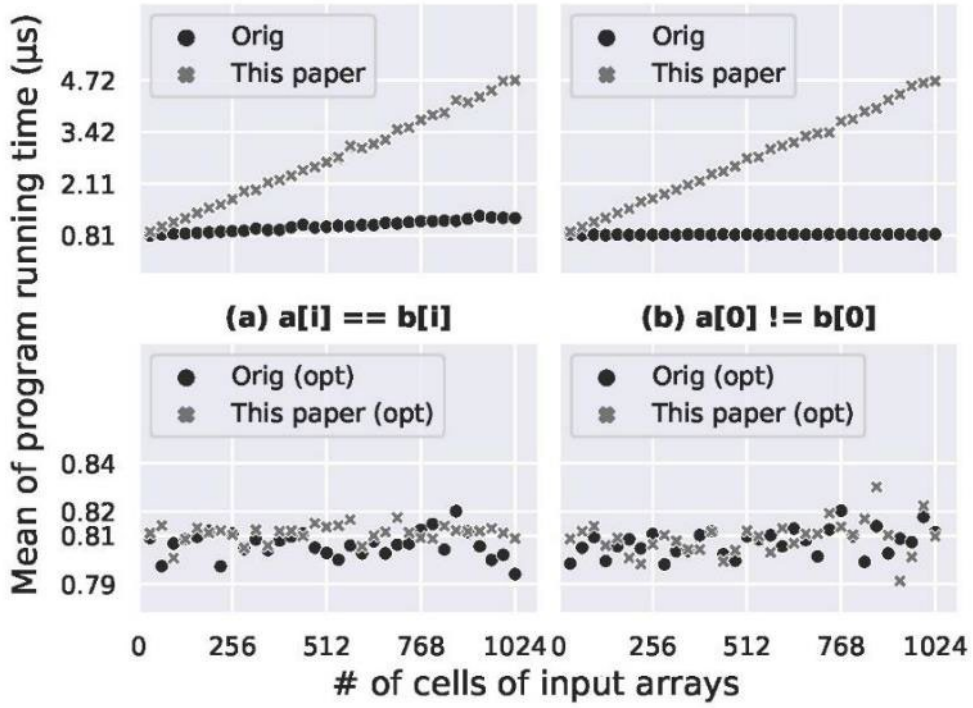


Fig 20. 对函数$oFdF$不同实现的运行时间进行比较。这个大小决定了循环中的最大迭代次数。

### 5.3.4. Q3-空间开销

修复会增加程序的大小，因为它通过增加额外的指令来确保时间和数据的不变性。本节分析了这种增长。

加密例程：Fig 21 比较了原始和修复的密码库的 LLVM 指令的数量。修复代码有本文的版本和$SC-Eliminator$的版本。在分析优化程序时，无论是本文还是$SC-Eliminator$都会增加代码的大小，但$SC-Eliminator$增加的量要显著多于本文。

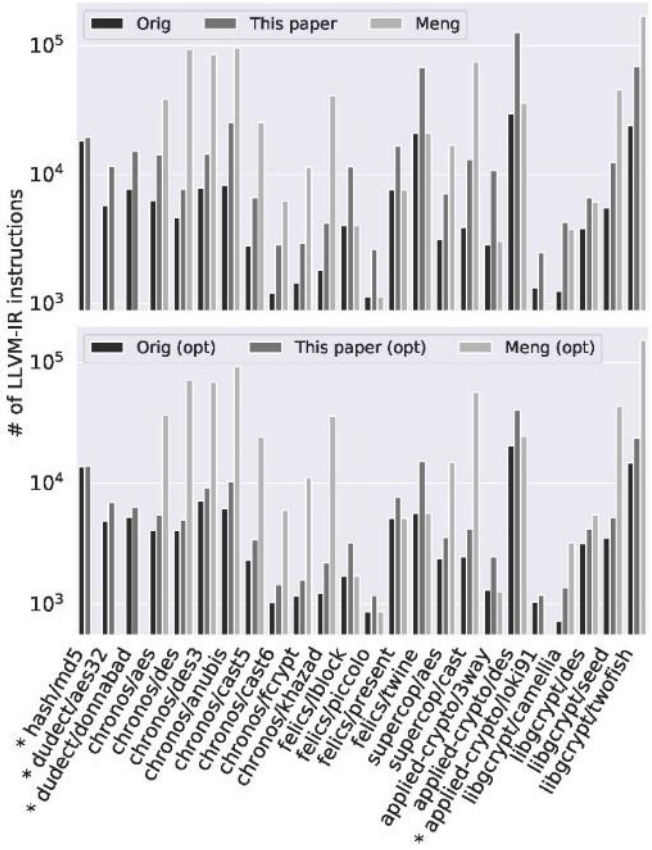


Fig 21. 程序修复对实际密码例程大小的影响。吴等人的程序修复方法对于带有星号前缀的基准测试无法转换或产生不正确结果。

经验渐进行为：Fig 22 则显示无论是优化与否，转换后的程序大小都会比原始程序大。转换后的程序大小在未优化时占用空间增加量要大于未优化时占用空间增加量。

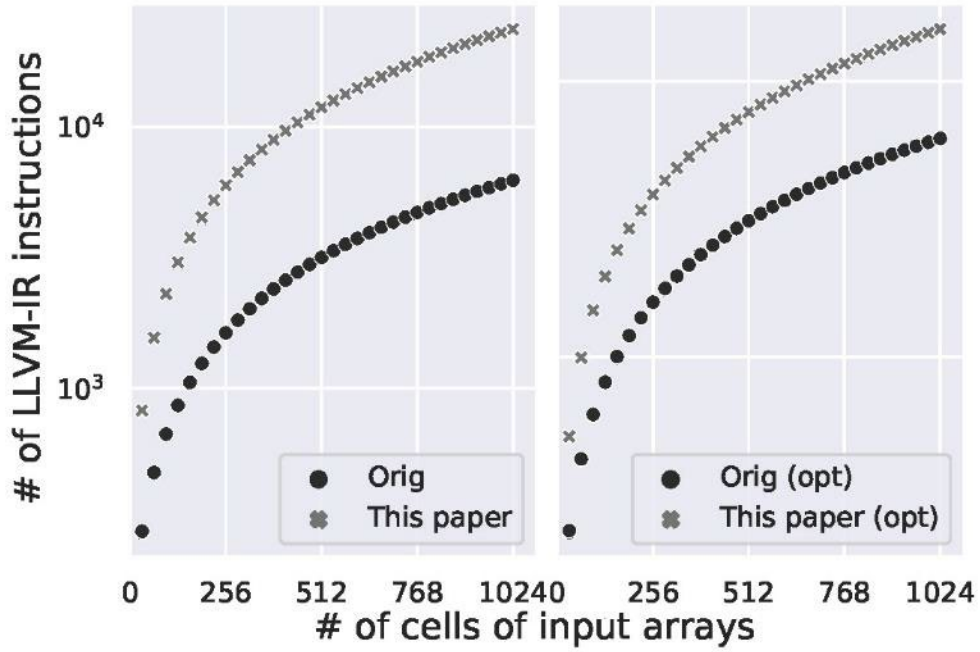


Fig 22 对函数 oFdF 不同实现的 LLVM 指令数量进行比较

# 6. 总结与收获

## 6.1. 总结

此论文提出了对吴等人的转换方法的改进，称之为$lif$。这种方法有效防止了越界访问问题，实现了更短的修复时间，同时生成了更小效率更高的代码。此转换方法比原始方法更灵敏且高效。

## 6.2. 收获

通过本次实验，学习到了较先进的安全代码转换方法，提升了自身的视野；同时阅读论文也提升了自身的阅读论文的能力；而配置环境的过程虽然消耗了自身非常长的时间，但也提升了自己配置系统的能力；而复现代码的过程则提升了自身的实操能力。总的来说，收获颇丰。

# 参考资料：

**[1]** Soares L, Pereira F M Q. Memory-safe elimination of side channels[C]//2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2021: 200-210.

**[2]** M. Wu, S. Guo, P. Schaumont, and C. Wang, “Eliminating timing side-channel leaks using program repair,” in ISSTA. New York, NY, USA: Association for Computing Machinery, 2018, p. 15–26.

**[3]** N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” SIGARCH Comput. Archit. News, vol. 39, no. 2, p. 1–7, 2011.

**[4]** R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “An efficient method of computing static single assignment form,” in POPL. New York, NY, USA:

Association for Computing Machinery, 1989, p. 25–35.

**[5]** V. Paisante, M. Maalej, L. Barbosa, L. Gonnord, and F. M. Quint˜ao Pereira, “Symbolic range analysis of pointers,” in CGO. New York, NY, USA: Association for Computing Machinery, 2016, p. 171–181.

**[6]** https://github.com/lac-dcc/lif/tree/artifact/cgo

**[7]** A. C. Lopes and D. F. Aranha, “Benchmarking tools for verification of constant-time execution,” in SBSEG. Bento Goncalves, Brazil: SBC, 2017, pp. 716–726, https://github.com/arthurlopes/ctbench.

**[8]** P. C. Kocher, “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems,” in Advances in Cryptology — CRYPTO’ 96, N. Koblitz, Ed. Berlin, Heidelberg: Springer, 1996, pp. 104–113.