现代密码学项目报告

MP-SPDZ项目部分代码分析

汇报人：钟毅文

2024年6月12日

主要内容

本次讲解的内容

编译执行

这是理解项目代码行为的基本工作

意义：通过分析编译行为和执行流，可以对深入研究项目提供帮助

指令系统

该项目有一整套完整的指令系统（类似于CISC指令集）

更准确来说应该是建立在虚拟机上的字节码指令

意义：理解这套指令系统，可以帮助我们更加轻松地添加自定义协议和指令

编译执行的命令行操作

—— 浮在表面的工作

以编译 `Programs/Source/tutorial.mpc` 并使用Rep3协议执行为例

如果不需要额外的参数：

```
./compile.py tutorial
Scripts/ring.sh tutorial
```

- 第一行会分别在 `Programs/Schedules` 和 `Programs/Bytecode` 下生成schedule文件和二进制的字节码
- 第二行调用自动脚本通过编译好的对应协议的二进制文件来执行字节码

  这里对应的协议二进制文件就是 `replicated-ring-party.x`

程序底层的操作

—— 编译阶段的剖析

`compile.py` 代码如下：

```
2
3   def compilation(compiler):
4       prog = compiler.compile_file()
5
6       if prog.public_input_file is not None:
7           print(
8               "WARNING: %s is required to run the pr
9           )
10
11  def main(compiler):
12      compiler.prep_compile()
13      if compiler.options.profile:
14          import cProfile
15          p = cProfile.Profile().runctx("compilation
16          p.dump_stats(compiler.args[0] + ".prof")
17          p.print_stats(2)
18      else:
19          compilation(compiler)
20
```

1. `prep_compile` 会调用 `Compiler` 类中的 `build` 函数，来进行必要的初始化和创建 `Program` 类的实例

2. `build` 会调用 `build_program()` 将程序实体给到 `self.prog`，再调用 `build_vars()` 将所有**需要的**指令加入到 `self.VARS` 中去

3. `compilation()` 调用 `compiler.compile_file()`，后者会使用python的 `compile` 函数编译源文件，并将结果信息给到 `self.VARS`

4. 编译完成后还会调用 `finalize_compile()` 打印信息，其中调用函数 `prog.finalize()` 来生成schedule文件和二进制字节码文件（**真正的有效输出**）

程序底层的操作

—— 执行阶段的剖析

还是以Rep3协议执行程序为例，`Scripts/ring.sh`内容如下：

```
#!/usr/bin/env bash

HERE=$(cd `dirname $0`; pwd)
SPDZROOT=$HERE/..

export PLAYERS=3

. $HERE/run-common.sh

run_player replicated-ring-party.x $* || exit 1
```

这个脚本主要做了两件事：

1. 设置环境变量，其中比较重要的是`PLAYERS`，它决定了有多少个参与方（如果有多个参与方就需要使用`Scripts/setup-ssl.sh`来生成各方的SSL密钥）

2. 载入`Scripts/run-common.sh`脚本，使用其中定义的`run-player`方法来执行程序

程序底层的操作

—— 执行阶段的剖析

在 `run-common.sh` 中的 `run-player` 主要代码：

```
1   for i in $(seq 0 $[players-1]); do
2     if test "$GDB_PLAYER" -a $i = "$GDB_PLAYER"; then
3     my_prefix=gdb_front
4     else
5     my_prefix=$prefix
6     fi
7     front_player=${GDB_PLAYER:-0}
8     >&2 echo Running $my_prefix $SPDZROOT/$bin $i $pa
9     log=logs/$log_prefix$i
10    $my_prefix $SPDZROOT/$bin $i $params 2>&1 |
11    {
12      if test "$BENCH"; then
13      if test $i = $front_player; then tee -a $log; e
14      else
15      if test $i = $front_player; then tee $log; else
16      fi
17    } &
```

- 这个代码块就是 `run-player` 函数的主要逻辑，其按照参与方个数来定义执行的进程数，并指定执行的协议
- 最核心的代码是第10行 `$my_prefix $SPDZROOT/$bin $i $params`
  - 其中 `$SPDZROOT/$bin` 指的就是例子中的 `replicated-ring-party.x`
  - `$i` 代表参与者的编号
  - `$params` 表示其他参数，包括程序名和各种执行参数
- 脚本除了完成代码块中的工作，还会将命令行输出重定向到 `logs/` 文件目录中，最后等待所有参与方进程完成

指令系统

—— 程序的基本单元

与指令集相关的代码：

- `Compiler/instructions.py`

  以类的形式定义了所有指令的名称和参数形式，也被用来当作生成指令集的文档的基础

- `Compiler/instuctions_base.py`

  定义了所有指令的二进制形式，也就是字节码，这和 `Processor/Instruction.h` 中的定义完全相同

- `Processor/instructions.h`

  用宏的方式定义了所有指令的行为，即需要执行的操作

- `Processor/Instruction.hpp`

  定义并实现了很多和指令相关的方法，并且针对不同的指令定义了不同的 `X` 宏

指令类型

- 存取指令：Load and store
- 硬件指令：Machine
- 基本算术：Basic arithmetic
- 位运算：Bitwise operations
- 带立即数的运算：Arithmetic with immediate
- 移位指令：Shift instructions
- 数据访问：Data access instructions
- 输入输出：I/O
- 整型操作：Integer operations
- 比较清除：Clear comparison
- 跳转指令：Jumps
- 类型转换：Conversion
- 其他：Other instructions

```
opcodes = dict(
    # Emulation
    CISC = 0,
    # Load/store
    LDI = 0x1,
    LDSI = 0x2,
    LDMC = 0x3,
    LDMS = 0x4,
    STMC = 0x5,
    STMS = 0x6,
    LDMCI = 0x7,
    LDMSI = 0x8,
    STMCI = 0x9,
    STMSI = 0xA,
    MOVC = 0xB,
    MOVS = 0xC,
    PROTECTMEMS = 0xD,
    PROTECTMEMC = 0xE,
    PROTECTMEMINT = 0xF,
    LDMINT = 0xCA,
    STMINT = 0xCB,
    LDMINTI = 0xCC,
    STMINTI = 0xCD,
    PUSHINT = 0xCE,
    POPINT = 0xCF,
    ...
```

指令行为

—— 案例驱动的指令分析

这里以一个样例程序的执行流为例，来完成指令的分析

```
a = sint(1)
b = sint(2)

print_ln('%s', (a + b).reveal())
```

通过编译之后的字节码反编译结果如下：

```
ldsi s1, 1 # 0
ldsi s2, 2 # 1
adds s0, s1, s2 # 2
asm_open 3, 1, c0, s0 # 3
print_reg_plain c0 # 4
print_char 10 # 5
...
```

下面还有`use 0, 7, 1`等指令，那些是用来指示虚拟机进行资源分配等预处理操作的信息

对程序的主体逻辑没有太大影响，暂不列出

程序的基本语法

—— mpc的编程语法

程序中的`sint()`类型和`reveal()`函数都不是python中的常规语法，在这里需要补充一下这方面的语法知识

1. 数据类型

   在`Compiler/types.py`中定义了程序的基本数据类型和容器数据类型

   其中基本类型有：sint, cint, regint, sfix, cfix, sfloat 等

   容器类型有：MemValue, Array, Matrix, MultiArray

2. `reveal()`

   对于`s`类型（secret text）的数据，不能直接进行打印，需要使用`reveal`函数解除加密，对应于指令就是`asm_open`；与之对应的，`c`类型（clear text）数据可以直接打印

3. 寄存器

   在`Compiler/instructions_base.py`中定义了5种寄存器类型，其中对于`ClearModp`是`c`寄存器，而`SecretModp`是`s`寄存器，本例中的`sint`就使用`s`寄存器

指令行为

—— 案例驱动的指令分析

在 `Processor/instructions.h`