We are using a state-of-the-art off-the-shelf predicate abstraction tool called CPAChecker. Sequentially composing V_1 and V_2 as V_1 ; V_2 is a valid input to CPAChecker, but it might have trouble verifying the assertions. Constructing a product program by synchronizing the statements of V_1 and V_2 gives the predicate abstraction tool more power to reason about the composition. This synchronized approach is semantically equivalent to the sequential composition of the two programs. Although it is easier for CPAChecker to prove the assertions, the synchronized approach can also lead to an exponential blowup in the product program size in the presence of control flow branching or loops. Thus we outline three possible approaches:

1. Sequential composition
2. Synchronized composition
3. Heuristic-optimized synchronized composition

We will work with a simple set of standard semantics:

$$\begin{array}{lll l}
\textit{Statement} & S & := & A \\
 & & & \mid S_1 \ ; \ S_2 \\
 & & & \mid \textit{if } p \textit{ then } S_1 \textit{ else } S_2 \\
 & & & \mid \textit{while } p \ S \\
\textit{Predicate} & p & := & \top \mid \bot \mid A \mid \neg p \mid p \odot p \\
\textit{Operator} & \odot & := & \wedge \mid \vee
\end{array}$$

0.1 Sequential Composition

Consider the following code snippet and :

```
if complicatedFunction() then
    $x \leftarrow legitRandomness()$
else
    $x \leftarrow getPID()$
end if
```

and it's self-composiiton

```
if complicatedFunction() then
    $x \leftarrow legitRandomness()$
else
    $x \leftarrow getPID()$
end if
if complicatedFunction() then
    $x' \leftarrow legitRandomness()$
else
    $x' \leftarrow getPID()$
end if
$assert(x == x')$
```

```
if complicatedFunction() then
    $x \leftarrow legitRandomness()$
    if complicatedFunction() then
        $x' \leftarrow legitRandomness()$
    else
        $x' \leftarrow getPID()$
    end if
    $assert(x == x')$
else
    $x \leftarrow getPID()$
    if complicatedFunction() then
        $x' \leftarrow legitRandomness()$
    else
        $x' \leftarrow getPID()$
    end if
    $assert(x == x')$
end if
```

0.2 Synchronized Composiiton

Here are our naive inference rules. These represent our approach for synchronized composition

$$\frac{}{A_1 \otimes A_2 \rightsquigarrow A_1 \ ; \ A_2}$$

$$\frac{S_2 \otimes S_1 \rightsquigarrow P}{S_1 \otimes S_2 \rightsquigarrow P}$$

$$\frac{\begin{array}{l} S_1 \otimes S \rightsquigarrow S_1' \\ S_2 \otimes S \rightsquigarrow S_2' \\ P = if(p) \ then \ S_1' \ else \ S_2' \end{array}}{if(p) \ then \ S_1 \ else \ S_2 \otimes S \rightsquigarrow P}$$

$$\frac{\begin{array}{l} P_0 = while(p_1 \wedge p_2) \ S_1 \ ; \ S_2 \\ P_1 = while(p_1) \ S_1 \\ P_2 = while(p_2) \ S_2 \end{array}}{while(p_1) \ S_1 \otimes while(p_2) \ S_2 \rightsquigarrow P_0 \ ; \ P_1 \ ; \ P_2}$$

0.3 Heuristic-optimized synchronized composition

Now we can add an environment Γ such that $\Gamma \vdash S$ if there is some atomic statement A in S such that A references a tainted variable. Conversely $\Gamma \nvdash S$ if for every variable v in S, v is not tainted by a source of genuine randomness.

Our key insight is that the following inference rule can be used to optimize the synchronization

$$\frac{\begin{array}{c} \Gamma \nvdash S_1 \\ \Gamma \nvdash S_2 \end{array}}{\Gamma \vdash S_1 \otimes S_2 \rightsquigarrow S_1 \ ; \ S_2}$$

Our previous inference rules can also be trivially modified with Γ. This would give us the following set of inference rules:

$$\frac{\begin{array}{c}\Gamma \nvdash S_1 \\ \Gamma \nvdash S_2\end{array}}{\Gamma \vdash S_1 \otimes S_2 \rightsquigarrow S_1 \ ; \ S_2}$$

$$\frac{}{\Gamma \vdash A_1 \otimes A_2 \rightsquigarrow A_1 \ ; \ A_2}$$

$$\frac{\Gamma \vdash S_2 \otimes S_1 \rightsquigarrow P}{\Gamma \vdash S_1 \otimes S_2 \rightsquigarrow P}$$

$$\frac{\begin{array}{l}\Gamma \vdash S_1 \otimes S \rightsquigarrow S_1' \\ \Gamma \vdash S_2 \otimes S \rightsquigarrow S_2' \\ P = if(p) \ then \ S_1' \ else \ S_2'\end{array}}{\Gamma \vdash if(p) \ then \ S_1 \ else \ S_2 \otimes S \rightsquigarrow P}$$

$$\frac{\begin{array}{l}P_0 = while(p_1 \wedge p_2) \ S_1 \ ; \ S_2 \\ P_1 = while(p_1) \ S_1 \\ P_2 = while(p_2) \ S_2\end{array}}{\Gamma \vdash while(p_1) \ S_1 \otimes while(p_2) \ S_2 \rightsquigarrow P_0 \ ; \ P_1 \ ; \ P_2}$$

This is only a heuristic because it is not clear that the heuristic-optimized synchronized composition invariably gives CPAChecker as much power as the synchronized version. For example, consider the following counterexample and its self-composition:

```
**if** flag **then**
    $x \leftarrow complicatedFunction1()$
**else**
    $x \leftarrow complicatedFunction2()$
**end if**
**if** $x == 7$ **then**
    $y \leftarrow taint()$
**end if**
$sink(y)$
```

The theorem prover with our optimized rules would sequentially compose the initial if-statements because none of their program variables in the initial if statement are tainted. This would make it difficult or impossible for CPAChecker to prove the assertion (it would involve reasoning about the complicated functions). In contrast, the synchronized version might be able to relationally verify that if y gets taint in V_1 then y will get taint in V_2 while still treating the complicated functions as uninterpreted (it would still reason about the flag in version one and the flag in version two).