Modular Product Programs

MARCO EILERS, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland
SAMUEL HITZ, ETH Zurich, Switzerland

Many interesting program properties like determinism or information flow security are hyperproperties, that is, they relate multiple executions of the same program. Hyperproperties can be verified using relational logics, but these logics require dedicated tool support and are difficult to automate. Alternatively, constructions such as self-composition represent multiple executions of a program by one product program, thereby reducing hyperproperties of the original program to trace properties of the product. However, existing constructions do not fully support procedure specifications, for instance, to derive the determinism of a caller from the determinism of a callee, making verification non-modular.

We present modular product programs, a novel kind of product program that permits hyperproperties in procedure specifications and, thus, can reason about calls modularly. We provide a general formalization of our product construction and prove it sound and complete. We demonstrate its expressiveness by applying it to information flow security with advanced features such as declassification and termination-sensitivity. Modular product programs can be verified using off-the-shelf verifiers; we have implemented our approach for both secure information flow and general hyperproperties using the Viper verification infrastructure. Our evaluation demonstrates that modular product programs can be used to prove hyperproperties for challenging examples in reasonable time.

CCS Concepts: • **Software and its engineering → Correctness**; **Software verification**; **Formal software verification**; • **Security and privacy →** *Information flow control*;

Additional Key Words and Phrases: Hyperproperties, product programs

ACM Reference Format:
Marco Eilers, Peter Müller, and Samuel Hitz. 2010. Modular Product Programs. *ACM Trans. Program. Lang. Syst.* 9, 4, Article 39 (March 2010), 38 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

The past decades have seen significant progress in automated reasoning about program behavior. In the most common scenario, the goal is to prove trace properties of programs such as functional correctness or termination. However, important program properties such as information flow security, injectivity, and determinism cannot be expressed as properties of individual traces; these so-called *hyperproperties* relate different executions of the same program. For example, proving determinism of a program requires showing that any two executions from identical initial states will result in identical final states.

An important attribute of reasoning techniques about programs is *modularity*. A technique is modular if it allows reasoning about parts of a program in isolation, e.g., verifying each procedure

Authors' addresses: Marco Eilers, Department of Computer Science, ETH Zurich, Zurich, Switzerland, marco.eilers@inf.ethz.ch; Peter Müller, ETH Zurich, Zurich, Switzerland, peter.mueller@inf.ethz.ch; Samuel Hitz, ETH Zurich, Zurich, Switzerland, samuel.hitz@inf.ethz.ch.

0164-0925/2010/3-ART39 $15.00
https://doi.org/0000001.0000001

separately and using only the *specifications* of other procedures. Modularity is vital for scalability and to verify libraries without knowing all of their clients. Fully modular reasoning about hyperproperties thus requires the ability to formulate *relational* specifications, which relate different executions of a procedure, and to apply those specifications where the procedure is called. As an example, the statement

$$\texttt{if (x) then \{y:=x\} else \{y:= call f(x)\}}$$

can be proved to be deterministic if `f`'s relational specification guarantees that its result deterministically depends on its input.

Relational program logics [10, 39, 41] allow directly proving general hyperproperties. However, automating relational logics is difficult and requires building dedicated tools. Alternatively, self-composition [8] and product programs [6, 7] reduce a hyperproperty to an ordinary trace property, thus making it possible to use off-the-shelf program verifiers for proving hyperproperties. Both approaches construct a new program that combines the behaviors of multiple runs of the original program. However, by the nature of their construction, neither approach supports modular verification based on relational specifications: Procedure calls in the original program will be duplicated, which means that there is no single program point at which a relational specification can be applied. For the aforementioned example, self-composition yields the following program:

$$\texttt{if (x) then \{y:=x\} else \{y:= call f(x)\};}$$
$$\texttt{if (x') then \{y':=x'\} else \{y':= call f(x')\}}$$

Determinism can now be verified by proving the trace property that identical values for `x` and `x'` in the initial state imply identical values for `y` and `y'` in the final state. However, such a proof cannot make use of a relational specification for procedure `f` (expressing that `f` is deterministic). Such a specification relates several executions of `f`, whereas each call in the self-composition belongs to a single execution. Instead, verification requires a *precise functional specification* of `f`, which *exactly* determines its result value in terms of the input. Verifying such precise functional specifications increases the verification effort and is at odds with data abstraction (for instance, a collection might not want to promise the exact iteration order); inferring them is beyond the state of the art for most procedures [40]. Existing product programs allow aligning or combining some statements and can thereby lift this requirement in some cases, but this requires manual effort during the construction, depends on the used specifications, and does not solve the problem in general. Some techniques for proving equivalence between different programs use product constructions that enable using relational specifications between different functions. However, they cannot exploit the inherent advantages resulting from combining a program with itself as opposed to an arbitrary other program [22, 25].

In this paper, we present modular product programs, a novel kind of product programs that allows modular reasoning about hyperproperties. Modular product programs enable proving k-safety hyperproperties, i.e., hyperproperties that relate finite prefixes of k execution traces, for arbitrary values of k [12]. We achieve this via a transformation that, unlike existing products, does not duplicate loops or procedure calls, meaning that for any loop or call in the original program, there is exactly one statement in the k-product at which a relational specification can be applied. Like existing product programs, modular products can be reasoned about using off-the-shelf program verifiers.

We demonstrate the expressiveness of modular product programs by applying them to prove secure information flow, which can be expressed as a 2-safety hyperproperty. We show how modular products enable proving traditional non-interference using natural and concise information flow

```
procedure main(people)
          returns (count)
{
  i := 0;
  count := 0;
  while (i < |people|) {
    current := people[i];
    f := is_female(current);
    count := count + f;
    i := i + 1;
  }
}
```

```
procedure is_female(person)
          returns (res)
{
  // gender encoded in first bit
  gender = person mod 2;
  if (gender == 0) {
    res := 1;
  }else{
    res := 0;
  }
}
```

Fig. 1. Example program. The parameter `people` contains a sequence of integers that each encode attributes of a person; the `main` procedure counts the number of females in this sequence.

specifications, and how to extend our approach for proving the absence of timing or termination channels, and supporting declassification in an intuitive way.

To summarize, we make the following contributions:

- We introduce modular k-product programs, which enable modular proofs of arbitrary k-safety hyperproperties for sequential programs using off-the-shelf verifiers.
- We prove soundness and completeness of the modular product program transformation.
- We demonstrate the usefulness of modular product programs by applying them to secure information flow, with support for declassification and preventing different kinds of side channels.
- We implement our product-based approach for verification of both secure information flow and arbitrary other safety hyperproperties in the Viper verification infrastructure [29].
- We show that our tool can automatically prove information flow security and other hyperproperties of challenging examples.

The current paper is an extended and revised version of a paper with the same title presented at ESOP 2018 [18]. Compared to the earlier version, the current paper provides a full proof of soundness and completeness of our approach, an extended evaluation covering hyperproperties other than secure information flow, and various smaller changes like an extended language and an extended overview of the related work.

The structure of this paper is as follows: After giving an informal overview of our approach in Section 2 and introducing our programming and assertion language in Section 3, we formally define modular product programs in Section 4. We prove the soundness and completeness of the approach in Section 5. Section 6 demonstrates how to apply modular products for proving secure information flow. We describe and evaluate our implementation in Section 7, discuss related work in Section 8, and conclude in Section 9.

2 OVERVIEW

In this section, we will illustrate the core concepts behind modular k-products on an example program. We will first show how modular products are constructed, and subsequently demonstrate how they allow using relational specifications to modularly prove hyperproperties.

2.1 Relational Specifications

Consider the example program in Figure 1, which counts the number of female entries in a sequence of people. Now assume we want to prove that the program is deterministic, i.e., that its output state is completely determined by its input arguments. This can be expressed as a 2-safety hyperproperty

which states that, for two terminating executions of the program with identical inputs, the outputs will be the same. This hyperproperty can be expressed by stating that when `main` terminates from two states that fulfill the *relational* (as opposed to *unary*) precondition $\texttt{people}^{(1)} = \texttt{people}^{(2)}$, the resulting two states fulfill the relational postcondition $\texttt{count}^{(1)} = \texttt{count}^{(2)}$, where $x^{(i)}$ refers to the value of the variable x in the ith execution. We write this relational specification of `main` as $\texttt{main} : \texttt{people}^{(1)} = \texttt{people}^{(2)} \twoheadrightarrow \texttt{count}^{(1)} = \texttt{count}^{(2)}$.

Intuitively, it is possible to prove this specification by giving `is_female` a precise functional specification like $\texttt{is\_female} : \mathit{true} \rightsquigarrow \texttt{res} = 1 - \texttt{person} \bmod 2$, meaning that `is_female` can be invoked in any state fulfilling the precondition *true* and that $\texttt{res} = 1 - \texttt{person} \bmod 2$ will hold if it returns. From this specification and an appropriate loop invariant, `main` can be shown to be deterministic. However, this specification is unnecessarily strong. For proving determinism, it is irrelevant what exactly the final value of `count` is; it is only important that it is uniquely determined by the procedure's inputs. Proving hyperproperties using only unary specifications, however, critically depends on having exact specifications for every value returned by a called procedure, as well as all heap locations modified by it. Not only are such specifications difficult to infer and cumbersome to provide manually; this requirement also fundamentally removes the option of underspecifying program behavior, which is often desirable in practice. Because of these limitations, verification techniques that require precise functional specifications for proving hyperproperties often do not work well in practice, as observed by Terauchi and Aiken for the case of self-composition [40].

Proving determinism of the example program becomes much simpler if we are able to reason about two program executions at once. If both runs start with identical values for `people` then they will have identical values for `people`, `i`, and `count` when they reach the loop. Since the loop guard only depends on `i` and `people`, it will either be true for both executions or false for both. Assuming that `is_female` behaves deterministically, all three variables will again be equal in both executions at the end of the loop body. This means that the program establishes and preserves the relational loop invariant that `people`, `i`, and `count` have identical values in both executions, from which we can deduce the desired relational postcondition. Our modular product programs enable this modular and intuitive reasoning, as we explain next.

2.2 Modular Product Programs

Like other product programs, our modular k-product programs multiply the state space of the original program by creating k renamed versions of all original variables. However, unlike other product programs, they do *not* duplicate control structures like loops or procedure calls, while still allowing different executions to take different paths through the program.

Modular product programs achieve this as follows: The set of transitions made by the execution of a product is the union of the transitions made by the executions of the original program it represents. This means that if two executions of an if-then-else statement execute different branches, an execution of the product will execute the corresponding versions of *both* branches; however, it will be aware of the fact that each branch is taken by only one of the original executions, and the transformation of the statements *inside* each branch will ensure that the state of the other execution is not modified by executing it.

For this purpose, modular product programs use boolean *activation variables* that store, for each execution, whether it is currently active. All activation variables are initially true. For every statement that directly changes the program state, the product performs the state change for all active executions. Control structures update which executions are active (for instance based on the loop condition) and pass this information down (into the branches of a conditional, the body of a

```
procedure main(p1, p2, people1, people2)
          returns (count1, count2)
{
  if (p1) { i1 := 0; }
  if (p2) { i2 := 0; }
  if (p1) { count1 := 0; }
  if (p2) { count2 := 0; }
  while ((p1 && i1 < |people1|) ||
         (p2 && i2 < |people2|)) {
    l1 := p1 && i1 < |people1|;
    l2 := p2 && i2 < |people2|;
    if (l1) { current1 := people1[i1]; }
    if (l2) { current2 := people2[i2]; }
    if (l1 || l2) {
      t1, t2 := is_female(l1, l2,
                current1, current2);
    }
    if (l1) { f1 := t1; }
    if (l2) { f2 := t2; }
    if (l1) { count1 := count1 + f1; }
    if (l2) { count2 := count2 + f2; }
    if (l1) { i1 := i1 + 1; }
    if (l2) { i2 := i2 + 1; }
  }
}
```

```
procedure is_female(p1, p2,
                    person1,
                    person2)
          returns (res1, res2)
{
  if (p1) {
    gender1 := person1 mod 2;
  }
  if (p2) {
    gender2 := person2 mod 2;
  }
  t1 := p1 && gender1 == 0;
  t2 := p2 && gender2 == 0;
  f1 := p1 && !(gender1 == 0);
  f2 := p2 && !(gender2 == 0);
  if (t1) { res1 := 1; }
  if (t2) { res2 := 1; }
  if (f1) { res1 := 0; }
  if (f2) { res2 := 0; }
}
```

Fig. 2. Modular 2-product of the program in Fig. 1 (slightly simplified). Parameters, local variables, and simple statements like assignments have been duplicated, but control flow statements have not. Instead, they are expressed by creating new sets of activation variables, and simple statements are conditional on their values. The initial activation variables p1 and p2 of the main procedure can be thought of as *true*.

loop, or the callee of a procedure call) to the level of atomic statements; in this way, the activation variables are similar to path conditions used in symbolic execution. This representation avoids duplicating these control structures.

Figure 2 shows the modular 2-product of the program in Figure 1. Consider first the main procedure. Its parameters have been duplicated, there are now two copies of all variables, one for each execution. This is analogous to self-composition or existing product programs. In addition, the transformed procedure has two boolean parameters p1 and p2; these variables are the initial activation variables of the procedure. Since main is the entry point of the program, the initial activation variables can be assumed to be true.

The product will first initialize i1 and i2 to zero, like it does with i in the original program, and analogously for count1 and count2. The loop in the original program has been transformed to a single loop in the product. Its condition is true if the original loop condition is true for any active execution. This means that the loop will iterate as long as at least one execution of the original program would. Inside the loop body, the fresh activation variables l1 and l2 represent whether the corresponding executions would execute the loop body. That is, for each execution, the respective activation variable will be true if the previous activation variable (p1 or p2, respectively) is true, meaning that this execution actually reaches the loop, and the loop guard is true for that execution. All statements in the loop body are then transformed using these new activation variables. Consequently, the loop will keep iterating while at least one execution executes the loop, but as soon as the loop guard is false for any execution, its activation variable will be false and the loop body will have no effect on the variables of this execution.

Conceptually, procedure calls are handled very similarly to loops. For the call to is_female in the original program, only a single call is created in the product. This call is executed if at

least one activation variable is true, i.e., if at least one execution would perform the call in the original program (without the conditional, the product would perform calls that do not correspond to anything in the original executions, which could lead to non-termination). In addition to the (duplicated) arguments of the original call, the current activation variables are passed to the called procedure. In the transformed version of `is_female`, all statements are then made conditional on those activation variables. Therefore, like with loops, a call in the product will be performed if at least one execution would perform it in the original program, but it will have no effect on the state of the executions that are not active when the call is made.

The transformed version of `is_female` shows how conditionals are handled. We introduce four fresh activation variables `t1`, `t2`, `f1`, and `f2`, two for each execution. The first pair encodes whether the then-branch should be executed by either of the two executions; the second encodes the same for the else-branch. These activation variables are then used to transform the branches. Consequently, neither branch will have an effect for inactive executions, and exactly one branch has an effect for each active execution.

To summarize, our activation variables ensure that the sequence of state-changing statements executed by each execution is the same in the product and the original program. We achieve this without duplicating control structures or imposing restrictions on the control flow.

2.3 Interpretation of Relational Specifications

Since modular product programs do not duplicate calls, they provide a simple way of interpreting relational procedure specifications: A precondition that requires some relation between the states of different executions is required to hold before the call, and a relational postcondition can be assumed to hold afterwards. However, a relational procedure specification should only apply if all the executions it refers to actually perform the procedure call. If a call is performed by some of those executions but not all, the relational specification are not meaningful, and thus cannot be required to hold. To encode this intuition, we transform every relational pre- or postcondition $\hat{Q}$ of the original program that refers to some subset E of all executions into an implication $(\bigwedge_{i \in E} \mathtt{p}_i) \Rightarrow \hat{Q}$. As a result, relational specifications will be trivially true if at least one execution it refers to is not active at the call site.

In our example, we give `is_female` the relational specification $\mathtt{is\_female} : \mathtt{person}^{(1)} = \mathtt{person}^{(2)} \twoheadrightarrow \mathtt{res}^{(1)} = \mathtt{res}^{(2)}$, which expresses determinism. This specification will be transformed into a unary specification of the product program: $\mathtt{is\_female} : \mathtt{p1} \wedge \mathtt{p2} \Rightarrow \mathtt{person1} = \mathtt{person2} \rightsquigarrow \mathtt{p1} \wedge \mathtt{p2} \Rightarrow \mathtt{res1} = \mathtt{res2}$.

Assume for the moment that `is_female` also has a unary precondition $\mathtt{person} \geq 0$. Such a specification should hold for *every* call, and therefore for every active execution, even if other executions are inactive. Therefore, its interpretation in the product program is $(\mathtt{p1} \Rightarrow \mathtt{person1} \geq 0) \wedge (\mathtt{p2} \Rightarrow \mathtt{person2} \geq 0)$. The translation of other unary assertions is analogous.

Note that it is possible (and useful) to give a procedure both a relational and a unary specification; in the product this is encoded by simply conjoining the transformed versions of the unary and the relational assertions.

2.4 Product Program Verification

We can now prove determinism of our example using the product program. Verifying `is_female` is simple. For `main`, we want to prove the transformed specification $\mathtt{main} : (\mathtt{p1} \wedge \mathtt{p2} \Rightarrow \mathtt{people1} = \mathtt{people2}) \rightsquigarrow (\mathtt{p1} \wedge \mathtt{p2} \Rightarrow \mathtt{count1} = \mathtt{count2})$. We use the relational loop invariant $\mathtt{i}^{(1)} = \mathtt{i}^{(2)} \wedge \mathtt{count}^{(1)} = \mathtt{count}^{(2)} \wedge \mathtt{people}^{(1)} = \mathtt{people}^{(2)}$, encoded as $\mathtt{p1} \wedge \mathtt{p2} \Rightarrow \mathtt{i1} = \mathtt{i2} \wedge \mathtt{count1} = \mathtt{count2} \wedge \mathtt{people1} = \mathtt{people2}$. The loop invariant holds trivially if `p1` or `p2` is false. Otherwise,

$$
\begin{array}{llll}
(\textit{Procedures}) & \mathit{Proc} & ::= & \texttt{procedure}\ m(\overline{x})\ \texttt{returns}\ (\overline{y})\{s\} \\
(\textit{FStatements}) & \dot{s} & ::= & \texttt{skip} \mid \texttt{error} \mid \texttt{magic} \\
(\textit{RTStatements}) & \hat{s} & ::= & x{:=}e \mid \hat{s};\hat{s} \mid \texttt{if}\ (e)\ \texttt{then}\ \{\hat{s}\}\ \texttt{else}\ \{\hat{s}\} \\
 & & & \mid \texttt{while}\ (e)\ \texttt{do}\ \{\hat{s}\} \mid \overline{x}{:=}\ \texttt{call}\ m(\overline{e}) \\
 & & & \mid \texttt{assert}\ e \mid \texttt{assume}\ e \mid \texttt{havoc}\ x \\
 & & & \mid \overline{x}{:=}\texttt{frame}_{\overline{y}}(\hat{s},\sigma) \mid \dot{s} \\
(\textit{Expressions}) & e & ::= & c \mid x \mid e \oplus e \text{ where } c \in \mathbb{Z} \text{ and } \oplus \in \{+,-,\times,=,<\} \\
 & & & \mid e \wedge e \mid e \vee e \mid \neg e \\
(\textit{Assertions}) & P & ::= & P \wedge P \mid P \Rightarrow P \mid \forall x.\, P \mid e \\
(\textit{RelExpressions}) & \hat{e} & ::= & c \mid x^{(i)} \mid \hat{e} \oplus \hat{e} \mid \hat{e} \wedge \hat{e} \mid \hat{e} \vee \hat{e} \mid \neg \hat{e} \\
(\textit{RelAssertions}) & \hat{P} & ::= & \hat{P} \wedge \hat{P} \mid \hat{P} \Rightarrow \hat{P} \mid \forall x^{(1)}, \ldots, x^{(k)}.\, \hat{P} \mid \hat{e} \\
(\textit{MixAssertions}) & \check{P} & ::= & P \wedge \hat{P}
\end{array}
$$

Fig. 3. Supported programming language. Relational expressions and assertions generalize unary ones to refer to the state of multiple different executions. We distinguish three kinds of statements: Runtime statements (RTStatements) $\hat{s}$ denote all statements that can occur in a trace, whereas final statements (FStatements) $\dot{s}$ denote finished computations. Programs initially consist only of program statements s, which are all runtime statements that do not contain `magic`, `error`, or $\overline{x}{:=}\texttt{frame}_{\overline{y}}(\hat{s},\sigma)$. We write $\overline{x}$ to denote a vector of variables.

it ensures `l1 = l2` and `current1 = current2`. Using the specification of `is_female`, we obtain `t1 = t2`, which implies that the loop invariant is preserved. The loop invariant implies the postcondition.

3 PRELIMINARIES

Figure 3 shows the language we use to define modular product programs. x ranges over the set of local integer variable names Var.

Program configurations have the form $\langle \hat{s}, \sigma \rangle$, where a store $\sigma \in \Sigma$ is a partial function mapping variable names to integer values. We use $\dot{s}$ to range over the *final* statements `skip`, `error`, and `magic`. A configuration is final if it has the form $\langle \dot{s}, \sigma \rangle$. Executions ending in `skip` are regular executions, whereas those ending with `error` represent failing executions caused by failing `assert` statements. Similarly, an execution halts in a `magic` state if an `assume` statement assumes something that does not hold in the current state (and the execution is therefore assumed, magically, not to exist).

Our language supports non-determinism via `havoc` statements that assign an arbitrary value to a variable; all other statements are deterministic. We distinguish between runtime statements $\hat{s}$ and program statements s: Program statements are runtime statements that do not syntactically contain `error`, `magic`, or `frame` statements. Programs contain only program statements; `error`, `magic`, and `frame` can occur only in the process of executing a program statement. `frame` statements are used to model stack-like behaviors for procedure calls: $\overline{x}{:=}\texttt{frame}_{\overline{y}}(\hat{s},\sigma)$ executes the statement $\hat{s}$ in the framed store σ and, if it reaches a final configuration, returns the results to the original store by assigning the value of y_i in the framed store to the target variable x_i in the original store.

The small-step transition relation for program configurations has the form $\langle \hat{s}, \sigma \rangle \rightarrow \langle \hat{s}', \sigma' \rangle$ and is defined in Fig. 4. We assume as given a procedure context Φ that maps procedure names to triples $(\overline{q}, \overline{r}, s_m)$, where $\overline{q}$ is a vector of input parameter variables, $\overline{r}$ is a vector of return parameter variables, and s_m is the body of the procedure.

We denote that expression e evaluates to value v in store σ by $e \Downarrow_\sigma v$. The inference rules for expression evaluation are standard and therefore omitted. Expression evaluation is deterministic

$$\frac{e \Downarrow_\sigma v}{\langle x{:=}e, \sigma\rangle \to \langle \mathsf{skip}, \sigma[x \mapsto v]\rangle}\ (\textsc{Assign})$$

$$\frac{}{\langle \mathsf{havoc}\ x, \sigma\rangle \to \langle \mathsf{skip}, \sigma[x \mapsto v]\rangle}\ (\textsc{Havoc})$$

$$\frac{e \Downarrow_\sigma \top}{\langle \mathsf{assert}\ e, \sigma\rangle \to \langle \mathsf{skip}, \sigma\rangle}\ (\textsc{Assert1}) \qquad \frac{e \Downarrow_\sigma \bot}{\langle \mathsf{assert}\ e, \sigma\rangle \to \langle \mathsf{error}, \sigma\rangle}\ (\textsc{Assert2})$$

$$\frac{e \Downarrow_\sigma \top}{\langle \mathsf{assume}\ e, \sigma\rangle \to \langle \mathsf{skip}, \sigma\rangle}\ (\textsc{Assume1}) \qquad \frac{e \Downarrow_\sigma \bot}{\langle \mathsf{assume}\ e, \sigma\rangle \to \langle \mathsf{magic}, \sigma\rangle}\ (\textsc{Assume2})$$

$$\frac{e \Downarrow_\sigma \top}{\langle \mathsf{if}\ (e)\ \mathsf{then}\ \{\hat{s}_1\}\ \mathsf{else}\ \{\hat{s}_2\}, \sigma\rangle \to \langle \hat{s}_1, \sigma\rangle}\ (\textsc{Cond1})$$

$$\frac{e \Downarrow_\sigma \bot}{\langle \mathsf{if}\ (e)\ \mathsf{then}\ \{\hat{s}_1\}\ \mathsf{else}\ \{\hat{s}_2\}, \sigma\rangle \to \langle \hat{s}_2, \sigma\rangle}\ (\textsc{Cond2})$$

$$\frac{\langle \hat{s}_1, \sigma\rangle \to \langle \hat{s}_1', \sigma'\rangle}{\langle \hat{s}_1; \hat{s}_2, \sigma\rangle \to \langle \hat{s}_1'; \hat{s}_2, \sigma'\rangle}\ (\textsc{Seq1}) \qquad \frac{}{\langle \mathsf{skip}; \hat{s}_2, \sigma\rangle \to \langle \hat{s}_2, \sigma\rangle}\ (\textsc{Seq2})$$

$$\frac{}{\langle \mathsf{error}; \hat{s}_2, \sigma\rangle \to \langle \mathsf{error}, \sigma\rangle}\ (\textsc{Seq3}) \qquad \frac{}{\langle \mathsf{magic}; \hat{s}_2, \sigma\rangle \to \langle \mathsf{magic}, \sigma\rangle}\ (\textsc{Seq4})$$

$$\frac{e \Downarrow_\sigma \top}{\langle \mathsf{while}\ (e)\ \mathsf{do}\ \{\hat{s}_1\}, \sigma\rangle \to \langle \hat{s}_1; \mathsf{while}\ (e)\ \mathsf{do}\ \{\hat{s}_1\}, \sigma\rangle}\ (\textsc{Whl1})$$

$$\frac{e \Downarrow_\sigma \bot}{\langle \mathsf{while}\ (e)\ \mathsf{do}\ \{\hat{s}_1\}, \sigma\rangle \to \langle \mathsf{skip}, \sigma\rangle}\ (\textsc{Whl2})$$

$$\frac{\Phi(f) = ([p_1, \ldots, p_n], [r_1, \ldots, r_m], s_f) \quad e_1 \Downarrow_\sigma v_1 \quad \ldots \quad e_n \Downarrow_\sigma v_n}{\langle \overline{x}{:=}\ \mathsf{call}\ f(e_1, \ldots, e_n), \sigma\rangle \to \langle \overline{x}{:=}\mathsf{frame}_{\overline{r}}(s_f, [][p_1 \mapsto v_1]\ldots[p_n \mapsto v_n]), \sigma\rangle}\ (\textsc{Call})$$

$$\frac{\langle \hat{s}_f, \sigma_f\rangle \to \langle \hat{s}_f', \sigma_f'\rangle}{\langle \overline{x}{:=}\mathsf{frame}_{\overline{r}}(\hat{s}_f, \sigma_f), \sigma\rangle \to \langle \overline{x}{:=}\mathsf{frame}_{\overline{r}}(\hat{s}_f', \sigma_f'), \sigma\rangle}\ (\textsc{Frame1})$$

$$\frac{r_1 \Downarrow_{\sigma_f} v_1 \quad \ldots \quad r_m \Downarrow_{\sigma_f} v_m}{\langle x_1, \ldots, x_m{:=}\mathsf{frame}_{r_1,\ldots,r_m}(\mathsf{skip}, \sigma_f), \sigma\rangle \to \langle \mathsf{skip}, \sigma[x_1 \mapsto v_1]\ldots[x_m \mapsto v_m]\rangle}\ (\textsc{Frame2})$$

$$\frac{}{\langle \overline{x}{:=}\mathsf{frame}_{\overline{r}}(\mathsf{error}, \sigma_f), \sigma\rangle \to \langle \mathsf{error}, \sigma\rangle}\ (\textsc{Frame3})$$

$$\frac{}{\langle \overline{x}{:=}\mathsf{frame}_{\overline{r}}(\mathsf{magic}, \sigma_f), \sigma\rangle \to \langle \mathsf{magic}, \sigma\rangle}\ (\textsc{Frame4})$$

Fig. 4. Operational semantics. We denote the empty store as [].

and total, and variable names not present in the store evaluate to some default value. Programs cannot get stuck in non-final configurations, but they can diverge. All variables and expression in our language have type integer. When evaluating conditions, any non-zero value is interpreted as true. For convenience, we write $\top$ and $\bot$ to mean any arbitrary non-zero value and zero, respectively; that is, $\top$ is not one fixed non-zero value but is used as a stand-in for any number of potentially different values. As an example, we write $e_1 \Downarrow_\sigma \top \wedge e_2 \Downarrow_\sigma \top$ as a shorthand for $\exists v_1, v_2.\, e_1 \Downarrow_\sigma v_1 \wedge e_2 \Downarrow_\sigma v_2 \wedge v_1 \neq 0 \wedge v_2 \neq 0$.

The judgment $\vDash s : P \rightsquigarrow Q$ denotes that statement s, when executed in a store fulfilling the unary assertion P, will not fail, and if the execution terminates regularly, the resulting store will fulfill the unary assertion Q. Formally, $\vDash s : P \rightsquigarrow Q$ if and only if for all $\sigma, \sigma', \dot{s}$ s.t. $\sigma \vDash P$ and $\langle s, \sigma\rangle \to^* \langle \dot{s}, \sigma'\rangle$, we have either have $\dot{s} = \texttt{magic}$ or $(\dot{s} = \texttt{skip} \wedge \sigma' \vDash Q)$.

In addition to standard unary expressions and assertions, we define relational expressions and assertions. They differ from normal expressions and assertions in that they contain parameterized variable references of the form $x^{(i)}$ and are evaluated over a tuple of stores instead of a single one. A relational expression is k-relational if for all contained variable references $x^{(i)}$, we have that $1 \leq i \leq k$. The definition of k-relational assertions is analogous. The value of a variable reference $x^{(i)}$ with $1 \leq i \leq k$ in a tuple of stores $(\sigma_1, \ldots, \sigma_k)$ is $\sigma_i(x)$; the evaluation of arbitrary relational expressions and the validity of relational assertions $(\sigma_1, \ldots, \sigma_k) \vDash \hat{P}$ are defined accordingly.

Definition 3.1. A k-relational specification $s : \hat{P} \twoheadrightarrow_k \hat{Q}$ holds iff $\hat{P}$ and $\hat{Q}$ are k-relational assertions, and for all $\sigma_1, \ldots, \sigma_k, \sigma'_1, \ldots, \sigma'_k$, if $(\sigma_1, \ldots, \sigma_k) \vDash \hat{P}$ and $\forall i \in \{1, \ldots, k\}.\, \langle s, \sigma_i\rangle \to^* \langle \texttt{skip}, \sigma'_i\rangle$, then $(\sigma'_1, \ldots, \sigma'_k) \vDash \hat{Q}$. ∎

We write $s : \hat{P} \twoheadrightarrow \hat{Q}$ for the most common case $s : \hat{P} \twoheadrightarrow_2 \hat{Q}$.

Mixed assertions combine unary and relational assertions and hold if the unary parts hold for every single store, and the relational parts hold for the tuple of stores as a whole. Formally, we define the validity of a mixed assertion $\check{P} = P \wedge \hat{P}$, where $\hat{P}$ is a k-relational assertion, as $(\sigma_1, \ldots, \sigma_k) \vDash \check{P} \Leftrightarrow (\forall i \in \{1, \ldots, k\}.\, \sigma_i \vDash P) \wedge (\sigma_1, \ldots, \sigma_k) \vDash \hat{P}$.

4 MODULAR k-PRODUCT PROGRAMS

In this section, we define the construction of modular products for arbitrary k. We will subsequently define the transformation of both relational and unary specifications to modular products.

4.1 Product Construction

Assume as given an injective function $\phi : (\textsc{Var}, \mathbb{N}) \to \textsc{Var}$ that renames variables for different executions. We write $e^{(i)}$ for the renaming of expression e for execution i. We write $\mathit{fresh}(x_1, x_2, \ldots)$ to denote that the variable names $x_1, x_2, \ldots$ are fresh names that do not occur in the program and have not yet been used during the transformation. $\mathring{e}$ is used to abbreviate $e^{(1)}, \ldots, e^{(k)}$.

We denote the modular k-product of a program statement s that is parameterized by the activation variables $p^{(1)}, \ldots, p^{(k)}$ as $[\![s]\!]^{\mathring{p}}_k$. The product construction for procedures is defined as

$$[\![\texttt{procedure } m(x_1, \ldots, x_m) \texttt{ returns } (y_1, \ldots, y_n)\{s\}]\!]_k$$
$$= \texttt{procedure } m(\mathring{p}, \mathring{x_1}, \ldots, \mathring{x_m}) \texttt{ returns } (\mathring{y_1}, \ldots, \mathring{y_n})\{[\![s]\!]^{\mathring{p}}_k\}$$

for fresh variable names $\mathring{p}$.

Figure 5 shows the product construction rules for statements, which generalize the transformation explained in Section 2. We abbreviate $\texttt{if } (e) \texttt{ then } \{s\} \texttt{ else } \{\texttt{skip}\}$ as $\texttt{if } (e) \texttt{ then } \{s\}$, and use $\bigodot_{i=1}^{k} s_i$ as a shorthand for the sequential composition of k statements $s_1; \ldots; s_k$.

$$
\begin{array}{lcl}
[\![s_1; s_2]\!]^{\mathring{p}}_k & = & [\![s_1]\!]^{\mathring{p}}_k; [\![s_2]\!]^{\mathring{p}}_k \\
[\![\texttt{skip}]\!]^{\mathring{p}}_k & = & \texttt{skip} \\
[\![x{:=}e]\!]^{\mathring{p}}_k & = & \bigodot_{i=1}^{k} \texttt{if } (p^{(i)}) \texttt{ then } \{x^{(i)}{:=}e^{(i)}\} \\
[\![\texttt{assert } e]\!]^{\mathring{p}}_k & = & \bigodot_{i=1}^{k} \texttt{if } (p^{(i)}) \texttt{ then } \{\texttt{assert } e^{(i)}\} \\
[\![\texttt{assume } e]\!]^{\mathring{p}}_k & = & \bigodot_{i=1}^{k} \texttt{if } (p^{(i)}) \texttt{ then } \{\texttt{assume } e^{(i)}\} \\
[\![\texttt{havoc } x]\!]^{\mathring{p}}_k & = & \bigodot_{i=1}^{k} \texttt{if } (p^{(i)}) \texttt{ then } \{\texttt{havoc } x^{(i)}\} \\
[\![\texttt{if } (e) \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}]\!]^{\mathring{p}}_k & = & \bigodot_{i=1}^{k} (p_1{}^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \\
 & & \bigodot_{i=1}^{k} (p_2{}^{(i)}{:=}p^{(i)} \wedge \neg e^{(i)}); \\
 & & [\![s_1]\!]^{\mathring{p}_1}_k; [\![s_2]\!]^{\mathring{p}_2}_k \\
 & & \text{where} \\
 & & \mathit{fresh}(\mathring{p}_1) \wedge \mathit{fresh}(\mathring{p}_2) \\
[\![\texttt{while } (e) \texttt{ do } \{s\}]\!]^{\mathring{p}}_k & = & \texttt{while } (\bigvee_{i=1}^{k} (p^{(i)} \wedge e^{(i)})) \texttt{ do } \{ \\
 & & \quad \bigodot_{i=1}^{k} (p_1{}^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \\
 & & \quad [\![s]\!]^{\mathring{p}_1}_k \\
 & & \} \\
 & & \text{where} \\
 & & \mathit{fresh}(\mathring{p}_1) \\
[\![x_1, \ldots, x_n{:=} \texttt{ call } f(e_1, \ldots, e_m)]\!]^{\mathring{p}}_k & = & \texttt{if } (\bigvee_{i=1}^{k} p^{(i)}) \texttt{ then } \{ \\
 & & \quad \bigodot_{i=1}^{k} \texttt{if } (p^{(i)}) \texttt{ then } \{\bigodot_{j=1}^{m} (a_j{}^{(i)}{:=}e_j{}^{(i)})\}; \\
 & & \quad \overline{t}{:=} \texttt{ call } f(p^{(1)}, \ldots, p^{(k)}, \overline{a}); \\
 & & \quad \bigodot_{i=1}^{k} \texttt{if } (p^{(i)}) \texttt{ then } \{\bigodot_{j=1}^{n} (x_j{}^{(i)}{:=}t_j{}^{(i)})\} \\
 & & \} \\
 & & \text{where} \\
 & & \mathit{fresh}(\mathring{a}_1, \ldots, \mathring{a}_m) \wedge \mathit{fresh}(\mathring{t}_1, \ldots, \mathring{t}_n) \\
 & & \overline{a} = [\mathring{a}_1, \ldots, \mathring{a}_m] \\
 & & \overline{t} = [\mathring{t}_1, \ldots, \mathring{t}_n]
\end{array}
$$

Fig. 5. Construction rules for statement products.

The core principle behind our encoding is that statements that directly change the state are duplicated for each execution and made conditional under the respective activation variables, whereas control statements are not duplicated and instead manipulate the activation variables to pass activation information to their sub-statements. This enables us to assert or assume relational assertions before and after any statement from the original program. The only state-changing statements in our language, variable assignments and havocs, are therefore transformed to a sequence of conditional assignments or havocs, one for each execution. Each one is executed only if the respective execution is currently active. Similarly, `assert` and `assume` statements should

only check and assume that an assertion is true for active executions, and are therefore treated in the same way.

Duplicating conditionals would also duplicate the calls and loops in their branches. To avoid that, modular products eliminate top-level conditionals; instead, new activation variables are created and assigned the values of the current activation variables conjoined with the guard for each branch. The branches are then sequentially executed based on their respective activation variables.

A while loop is transformed to a single while loop in the product program that iterates as long as the loop guard is true for *any* active execution. Inside the loop, fresh activation variables indicate whether an execution reaches the loop and its loop condition is true. In particular, if the number of iterations of a loop differs between executions, there will be iterations where the fresh activation variables will be false for the executions that no longer iterate, but true for the others. The loop body will then modify the state of an execution only if its activation variable is true. The resulting construct affects the program state in the same way as a self-composition of the original loop would, but the fact that our product contains only a single loop enables us to use relational loop invariants instead of full functional specifications. In particular, we do not require that loops iterate the same number of times for all executions in order to use relational loop invariants; it is sufficient that the iterations that are performed by only some of the executions preserve the relational invariant.

Once the loop condition is false for all active executions, the loop in the product will no longer be executed. As a result, the loop in the product is guaranteed to terminate if the loop execution terminates in all executions of the original program.

For procedure calls, it is crucial that the product contains a single call for every call in the original program, in order to be able to apply relational specifications at the call site. As explained before, initial activation parameters are added to every procedure declaration, and all parameters are duplicated k times. Procedure calls are therefore transformed such that the values of the current activation variables are passed, and all arguments are passed once for each execution. The return values are stored in temporary variables and subsequently assigned to the actual target variables only for those executions that actually execute the call, so that for all other executions, the target variables are not affected.

The transformation wraps the call in a conditional so that it is performed only if at least one execution is active. This prevents the transformation from introducing infinite recursion that is not present in the original program.

Note that for an inactive execution i, arbitrary argument values are passed in procedure calls, since the passed variables ${a_j}^{(i)}$ are not initialized. This is unproblematic because these values will not be used by the procedure. It is important to not evaluate ${e_j}^{(i)}$ for inactive executions, since this could lead to false alarms for languages where expression evaluation can fail. Note that expression evaluation in our language is side-effect free; for a language with effectful expression evaluation, the product construction would have to be adapted slightly. In particular, the condition expressions in loops and conditionals would have to be evaluated exactly once per execution of the loop or conditional, which could easily be achieved by assigning them to additional auxiliary variables before using them in the product.

4.2 Transformation of Assertions

We now define how to transform unary and relational assertions for use in a modular product.

Unary assertions such as ordinary procedure preconditions describe state properties that should hold for every single execution. When checking or assuming that a unary assertion holds at a specific point in the program, we need to take into account that it only makes sense to do so for executions that actually reach that program point. We can express this by making the assertion

conditional on the activation variable of the respective execution; as a result, any unary assertion holds trivially for all inactive executions.

A k-relational assertion, on the other hand, describes the relation between the states of multiple executions executions. Typically, to prove a k-relational property, assertions will relate all k executions; however, some intermediate assertions (e.g., loop invariants or postconditions of auxiliary procedures) may need to relate some subset of the executions to summarize their relative behavior in the case where other executions do not reach a program point. We say that an execution is *relevant* w.r.t. a relational assertion if a variable from this execution is syntactically referenced in the assertion. Checking or assuming a relational assertion at some point is meaningful only if *all* relevant executions actually reach that point. This can be expressed by making relational assertions conditional on the conjunction of the current activation variables of all relevant executions. If at least one relevant execution does not reach the assertion, it holds trivially.

Assertions can be transformed for use in a k-product as follows:

- The transformation $\lfloor P \rfloor_k^{\mathring{p}}$ of a unary assertion P is $\bigwedge_{i=1}^{k}(p^{(i)} \Rightarrow P^{(i)})$.
- The transformation $\lfloor \hat{P} \rfloor_k^{\mathring{p}}$ of a k-relational assertion $\hat{P}$ with the activation variables $p^{(1)}, \ldots, p^{(k)}$ and relevant executions $R \subseteq \{1, \ldots, k\}$ is $(\bigwedge_{i \in R} p^{(i)}) \Rightarrow \hat{P}$.

Importantly, our approach allows using *mixed* assertions and specifications, which represent conjunctions of unary and relational assertions. For example, it is common to combine a unary precondition that ensures that a procedure will not raise an error with a relational postcondition that states that it is deterministic.

A mixed assertion $\check{R}$ of the form $P \wedge \hat{Q}$ means that the unary assertion P holds for every single execution, and if all relevant executions are currently active, the relational assertion $\hat{Q}$ holds as well. The transformation of mixed assertions is straightforward: $\lfloor \check{R} \rfloor_k^{\mathring{p}} = \lfloor P \rfloor_k^{\mathring{p}} \wedge \lfloor \hat{Q} \rfloor_k^{\mathring{p}}$.

4.3 Heap-Manipulating Programs

The approach outlined so far can easily be extended to programs that work on a mutable heap, assuming that object references are opaque, i.e., they cannot be inspected or used in arithmetic. In order to create a distinct state space for each execution represented in the product, allocation statements are duplicated and made conditional like assignments, and therefore create a different object for each active execution. The renaming of a field dereference $e.f$ is then defined as $e^{(i)}.f$. As a result, the heap of a k-product will consist of k partitions that do not contain references to each other, and execution i will only ever interact with objects from its partition of the heap.

The verification of modular products of heap-manipulating programs does not depend on any specific way of achieving framing. Our implementation is based on implicit dynamic frames [37], but other approaches, in particular flavors of separation logic, are feasible as well, provided that procedures can be specified in such a way that the caller knows the heap stays unmodified for all executions whose activation variables are false.

Since the handling of the heap is largely orthogonal to our main technique, we will not go into further detail here, but we do support heap-manipulating programs in our implementation.

4.4 Discussion

One obvious property of the product construction is that it introduces a high number of branching statements into the transformed programs, which can make it expensive to reason about them using some analysis and verification techniques. However, since the product construction enables modular reasoning about loops and procedure calls, verifiers have to reason only about small pieces of code at a time and can check different procedures independently from one another. Moreover,

the ability to use relational specifications tends to make specifications simpler (and therefore easier to prove) in many cases. Additionally, the specific structure of the generated program (e.g., many branches on the same condition, many identical statements on mirrored states) makes it very efficient to reason about the generated program using some standard techniques, as we will show in Section 7.

A limitation of our approach is that, while the product construction is complete in principle, relational specifications can only be used to relate the same loop or procedure call in multiple executions, but not different loops or calls. As an example, for the program `if` (e) `then` $\{m()\}$ `else` $\{m()\}$, relational specifications for m cannot be used to relate the behavior of the call in the then-branch to the one in the else-branch. Note, however, that in this case, a full functional specification of m will still be sufficient to prove any relational property of the program; modular product programs are therefore never weaker or require more specification than self-composition. Some approaches from the related field of program equivalence are able to relate different calls and loops [22, 25]. However, this ability comes at a cost (e.g., losing the ability to reason about the resulting program using separation logic, or to perform standard static analyses on it), which we discuss in more detail in Section 8.

5 SOUNDNESS AND COMPLETENESS

A product construction is sound if an execution of a k-product mirrors k separate executions of the original program such that a property of a product execution reflects a hyperproperty of the original program. Similarly, the construction is complete if for any hyperproperty of the original program, the corresponding property holds for the product.

In this section, we first prove, based on the operational semantics of our language, that every execution of a k-product is simulated by k executions of the original program. In particular, if the product of a statement is executed from a store that mirrors k original stores and terminates normally then executing the original statement from any of the k original stores will also terminate normally, and the final store of the product will mirror the final original stores. Subsequently, we show the reverse, namely that any set of k executions can be simulated by a product execution. Additionally, we show that the termination behavior of modular product programs mirrors that of the underlying original executions: If a set of executions terminates, the product execution that mirrors it is guaranteed to terminate as well; conversely, normally-terminating product executions correspond to sets of terminating executions. We discuss the remaining case (the product terminates abnormally) below. Finally, we show how properties of a product program relate to hyperproperties of the original program. In particular, we prove that if a transformed relational specification holds for a product program, the original relational specification holds for the original program (*soundness*), and, conversely, that if a relational specification holds for a program, the transformed specification holds for its product program (*completeness*).

5.1 Preliminaries

Throughout this section, we will denote statements, stores and procedure mappings of product programs by $\underline{s}$, $\underline{\sigma}$, and $\underline{\Phi}$, respectively, and statements, stores and mappings of the original (non-product) executions by s, σ, and Φ.

We first define what it means for the store of a product program to mirror the stores of multiple original executions. Stores in product executions contain renamed versions of the stores of the original program executions. By $\sigma \in_i \underline{\sigma}$ we denote that $\underline{\sigma}$ contains an i-renamed version of all variables in σ and no other i-renamed program variables, and the values of those variables agree in both stores.

Definition 5.1. $\sigma \in_i \underline{\sigma}$ if and only if $(\forall x \in \text{PVar}.\, (x^{(i)} \in dom(\underline{\sigma}) \Leftrightarrow x \in dom(\sigma)) \wedge (x^{(i)} \in dom(\underline{\sigma}) \Rightarrow \sigma(x) = \underline{\sigma}(x^{(i)})))$. where PVar is the set of variable names used in the original program. We call RPVar the set of all renamed versions of all variables in PVar; i.e., $x \in \text{PVar} \Leftrightarrow (\forall i.\, x^{(i)} \in \text{RPVar}.)$ ∎

We assume that all fresh variable names picked during the product construction are not in RPVar, so that this definition allows $\underline{\sigma}$ to contain i-renamed auxiliary or activation variables that have no counterpart in σ.

Similarly, we use the notion $\underline{\sigma} \preccurlyeq_i^V \underline{\sigma'}$, where V is a set of variable names, to say that $\underline{\sigma'}$ contains all variables in $\underline{\sigma}$ that belong to the i-th execution and maps them to the same values, except for the variables in V:

Definition 5.2. $\underline{\sigma} \preccurlyeq_i^V \underline{\sigma'}$ if and only if $\forall x.\, x^{(i)} \notin V \Rightarrow (x^{(i)} \in dom(\underline{\sigma}) \Rightarrow x^{(i)} \in dom(\underline{\sigma'}) \wedge \underline{\sigma}(x^{(i)}) = \underline{\sigma'}(x^{(i)}))$. By $\underline{\sigma} \preccurlyeq^V \underline{\sigma'}$ we denote that this is true for all executions, i.e., $\underline{\sigma} \preccurlyeq^V \underline{\sigma'} \Leftrightarrow \forall i \in \{1, \ldots, k\}.\, \underline{\sigma} \preccurlyeq_i^V \underline{\sigma'}$. ∎

Note that both of these notions are reflexive.

We call $\mathit{freshvars}(\underline{s})$, where $\underline{s} = [\![s]\!]_k^{\mathring{p}}$ for some $k, \mathring{p}$, and s, the set of variables used as fresh variables in the construction of $\underline{s}$. For convenience, we abbreviate $\underline{\sigma} \preccurlyeq^{\mathit{freshvars}(\underline{s})} \underline{\sigma'}$ as $\underline{\sigma} \preccurlyeq^{\underline{s}} \underline{\sigma'}$ and $\underline{\sigma} \preccurlyeq_i^{\mathit{freshvars}(\underline{s})} \underline{\sigma'}$ as $\underline{\sigma} \preccurlyeq_i^{\underline{s}} \underline{\sigma'}$. Additionally, we denote $\underline{\sigma} \preccurlyeq^{\text{RPVar} \cup \mathit{freshvars}(\underline{s})} \underline{\sigma'}$ as $\underline{\sigma} \precsim^{\underline{s}} \underline{\sigma'}$ and $\underline{\sigma} \preccurlyeq_i^{\text{RPVar} \cup \mathit{freshvars}(\underline{s})} \underline{\sigma'}$ as $\underline{\sigma} \precsim_i^{\underline{s}} \underline{\sigma'}$.

Finally, we denote the free variables in a mixed assertion $\check{P}$ as $fv(\check{P})$.

We prove soundness and completeness under the assumption that all procedures referenced by our product program have been transformed to their modular products. This means that for an original program with procedures Φ, the product has procedures $\underline{\Phi}$ such that

$$\forall f.\, \Phi(f) = ([q_1, \ldots, q_n], [r_1, \ldots, r_m], s) \Leftrightarrow \underline{\Phi}(f) = (\mathit{args}, \mathit{rets}, [\![s]\!]_k^{\mathring{p}})$$

where $\mathit{args} = \mathring{p}, \mathring{q_1}, \ldots, \mathring{q_n}$ and $\mathit{rets} = \mathring{r_1}, \ldots, \mathring{r_m}$, and the parameters $\mathring{p}$ corresponding to activation variables are not in RPVar. If this is the case, we say that $\mathit{match}(\Phi, \underline{\Phi})$.

We start by showing that our notion of mirrored stores has the intended effect, namely, that evaluating an expression in a store results in the same value as evaluating a renamed expression in a store that mirrors the original store:

Lemma 5.3. *If $\sigma \in_i \underline{\sigma}$ and $fv(e) \subseteq$ PVar then $e^{(i)} \Downarrow_{\underline{\sigma}} v \Leftrightarrow e \Downarrow_\sigma v$.*

Proof. By induction on the structure of e. For the case $e = x$, the proof follows trivially from the definition of $\in_i$; for all others, it is either immediate or follows from applying the induction hypothesis to the subexpressions. □

5.2 Properties of Product Executions

We can now establish properties of modular product programs on the level of the operational semantics.

5.2.1 Normal executions simulate product executions. The first result we prove is that each execution of a product that terminates normally is simulated by a normally-terminating execution of the original statement for the executions whose activation variables are true, and the parts of the product's store belonging to executions whose activation variables are false do not change.

Theorem 5.4. *Assume that $\mathit{match}(\Phi, \underline{\Phi})$ and that for some sets $A \subseteq \{1, \ldots, k\}$ and $I = \{1, \ldots, k\} \setminus A$ we have that $\forall i \in A.\, p^{(i)} \Downarrow_{\underline{\sigma}} \top \wedge \sigma_i \in_i \underline{\sigma}$ and $\forall i \in I.\, p^{(i)} \Downarrow_{\underline{\sigma}} \bot$. Let $\{p^{(1)}, \ldots, p^{(k)}\} \cap (\mathit{RPVar} \cup \mathit{freshvars}(\underline{s})) = \emptyset$ and $\underline{s} = [\![s]\!]_k^{\mathring{p}}$ and $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^l \langle \mathtt{skip}, \underline{\sigma'} \rangle$ under $\underline{\Phi}$.*

Then
(a) for all $i \in A$, $\langle s, \sigma_i \rangle \to^* \langle \texttt{skip}, \sigma_i' \rangle$ *under* Φ *for some* σ_i' *s.t.* $\sigma_i' \in_i \underline{\sigma}'$, *and* $\underline{\sigma} \precsim_i^s \underline{\sigma}'$,
(b) for all $i \in I$, $\underline{\sigma} \preccurlyeq_i^s \underline{\sigma}'$.

Proof. The proof goes by strong induction on the length l of the derivation $\langle \mathring{[\![s]\!]}_k^p, \underline{\sigma} \rangle \to^l \langle \texttt{skip}, \underline{\sigma}' \rangle$. We perform a case split on the structure of s, which determines the structure of the product $\underline{s}$. The cases for assignments, havocs, assumes, and assertions are simple using Lemma 5.3. For other statements, we generally perform as many steps of $\underline{s}$ as necessary to get to a configuration where the statement is again a product program. We show that the product state in these configuration mirrors the states of (some of) the original executions, so that we can apply the induction hypothesis. Subsequently, we construct complete derivations of the original executions and show that the final states mirror the final state of the product. □

We also prove that products always terminate and do not modify existing state (except for fresh variables) if all activation variables are false:

Lemma 5.5. *If, for all* $i \in \{1, ..., k\}$, $p^{(i)} \Downarrow_{\underline{\sigma}} \bot$, *then for some* $\underline{\sigma}'$, $\langle \underline{s} = \mathring{[\![s]\!]}_k^p, \underline{\sigma} \rangle \to^* \langle \texttt{skip}, \underline{\sigma}' \rangle$ *and* $\underline{\sigma} \preccurlyeq^s \underline{\sigma}'$.

Proof. By induction on the structure of s. For simple statements the proof is immediate. For loops, the condition of the loop in the product must be false if all activation variables are false, and the product immediately steps to skip. The same is true for procedure calls and the condition of the conditional that wraps the call in the product. The cases for sequential composition and conditionals follow immediately from applying the induction hypothesis to the substatements. □

We now show that a product execution terminating abnormally is simulated by a set of executions of which at least one terminates abnormally.

Lemma 5.6. *Assume that* $match(\Phi, \underline{\Phi})$ *and that for some sets* $A \subseteq \{1, \ldots, k\}$ *and* $I = \{1, \ldots, k\} \setminus A$ *we have that* $\forall i \in A.\, p^{(i)} \Downarrow_{\underline{\sigma}} \top \wedge \sigma_i \in_i \underline{\sigma}$ *and* $\forall i \in I.\, p^{(i)} \Downarrow_{\underline{\sigma}} \bot$. *If* $\langle \underline{s}, \underline{\sigma} \rangle \to^l \langle \dot{s}, \underline{\sigma}' \rangle$, *where* $\underline{s} = \mathring{[\![s]\!]}_k^p$ *and* $\dot{s} \neq \texttt{skip}$, *under* $\underline{\Phi}$, *then for at least one* $i_e \in A$, $\langle s, \sigma_{i_e} \rangle \to^* \langle \dot{s}, \sigma_{i_e}' \rangle$ *under* Φ *for some* σ_{i_e}'.

Proof. The proof goes by strong induction on l and follows roughly the same outline as the proof for Thm. 5.4. We again perform a case split on the structure of s. We show that the cases for assignments and havocs always terminate normally, and that product assertions and assumes terminate abnormally if at least one corresponding assert or assume terminates abnormally in an original execution (using Lemma 5.3). For all other statements, we perform case splits on which part of the product leads to abnormal termination. We use Thm. 5.4 to reason about the effects of all parts that are products and terminate normally, and apply the induction hypothesis to those that do not. □

As a corollary of the previous theorem and lemma, we observe that terminating product executions are simulated by sets of original executions that either all terminate, or of which at least one fails. An example of the latter is a program $\texttt{assert}\ e_1; \texttt{while}\ (e_2)\ \texttt{do}\ \{\texttt{skip}\}$: If the assertion fails for one of the original executions but succeeds for all others, the other executions will diverge, whereas the product execution fails while executing the product of the assertion.

5.2.2 Product executions simulate normal executions. Now we prove that any set of terminating executions is simulated by a terminating execution of a modular product. The active executions of the product each mirror one of the original executions, and its inactive executions' state does not change (modulo assignments to fresh variables). We assume that we can map each of the executions to a distinct index in the range $1, \ldots, k$.

Theorem 5.7. *Assume that match$(\Phi, \underline{\Phi})$ and that for a set of indices $A \subseteq \{1, \dots, k\}$ there is a derivation $d_i = \langle s, \sigma_i \rangle \rightarrow^{l_i} \langle \texttt{skip}, \sigma_i' \rangle$ under Φ for each $i \in A$. Assume also that $\sigma_i \in_i \underline{\sigma}$ and $p^{(i)} \Downarrow_{\underline{\sigma}} \top$ for all $i \in A$, and $p^{(i)} \Downarrow_{\underline{\sigma}} \bot$ for any $i \in I$, where $I = \{1, \dots, k\} \setminus A$.*

Then $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^* \langle \texttt{skip}, \underline{\sigma}' \rangle$, where $\underline{s} = \mathring{[\![s]\!]}^{\mathring{p}}_k$, under $\underline{\Phi}$ for some $\underline{\sigma}'$ s.t. for all $i \in A$, $\sigma_i' \in_i \underline{\sigma}'$ and $\underline{\sigma} \precsim_i^{\mathring{s}} \underline{\sigma}'$, and for all $i \in I$, $\underline{\sigma} \preccurlyeq_i^{\mathring{s}} \underline{\sigma}'$.

Proof. The proof goes by strong induction on the sum of the lengths of derivations $l = \sum_{i \in A} l_i$. For $l = 0$ and $|A| = 0$, the conclusion follows from Lemma 5.5. If $l = 0$ and $|A| \neq 0$ then $s = \texttt{skip}$ and the conclusion follows trivially. For $l > 0$ we perform a case split on the structure of s. The cases for assignments, asserts, assumes, and havocs are essentially the reverse of the corresponding cases in the proof of Thm. 5.4. For all other statements, we identify sets of traces that proceed in the same way and show using the induction hypothesis that the product executes subproducts that mirror the executions for those sets. □

Similarly to before, we now show that any set of terminating traces, at least one of which ends abnormally, is mirrored by a product execution that ends abnormally.

Lemma 5.8. *Assume that for a set of indices A s.t. $|A| = j$ and $1 \leq j \leq k$ and $A \subseteq \{1, \dots, k\}$ there is a derivation $d_i = \langle s, \sigma_i \rangle \rightarrow^{l_i} \langle \dot{s}_i, \sigma_i' \rangle$, where $\dot{s}_i \in \{\texttt{skip}, \texttt{error}, \texttt{magic}\}$, under Φ for each $i \in A$. Assume given $A_s \subset A$ and $A_e = A \setminus A_s$ s.t. for all $i \in A_e$, $\dot{s}_i \neq \texttt{skip}$, and for all $i \in A_s$, $\dot{s}_i = \texttt{skip}$. Assume also that $\sigma_i \in_i \underline{\sigma}$ and $\underline{\sigma}(p^{(i)}) = \top$ for all $i \in A$, and $\underline{\sigma}(p^{(i)}) = \bot$ for any $i \in I$, where $I = \{1, \dots, k\} \setminus A$.*

Then $\langle \underline{s}, \underline{\sigma} \rangle \rightarrow^* \langle \dot{s}, \underline{\sigma}' \rangle$ under $\underline{\Phi}$, where $\underline{s} = \mathring{[\![s]\!]}^{\mathring{p}}_k$, for some $\underline{\sigma}'$ and $\dot{s}$ s.t. $\exists i \in A_e . \dot{s}_i = \dot{s}$.

Proof. The proof follows roughly the same outline as the proof for Thm. 5.7. We again perform a case split on the structure of s, and a further case split on which part of the original executions lead to errors. We use Thm. 5.7 to reason about successful executions of substatements that are products, and apply the induction hypothesis to the executions of failing product substatements. □

As a corollary, we observe that if a set of executions terminates, the corresponding product execution always terminates as well.

5.3 Provable Properties

We now show that since product executions mirror sets of executions of the original program, it follows that properties that hold for product programs correspond to hyperproperties of the original programs.

In particular, we prove soundness, i.e., if a transformed relational specification holds for a product program then the relational specification holds for sets of executions of the original program. Conversely, we show completeness, i.e., for any hyperproperty which holds for multiple executions of a program, the transformed property holds for its product.

First, we lift the correspondence of expression evaluation in product stores and original stores to unary assertions, and show that a product store that mirrors the original stores of all active executions fulfills a transformed assertion if and only if the original stores fulfill the original assertion.

Lemma 5.9. *If $A \subseteq \{1, \dots, k\}$ and $\forall i \in A . \sigma_i \in_i \underline{\sigma} \wedge p^{(i)} \Downarrow_{\underline{\sigma}} \top$ and $\forall i \in \{1, \dots, k\} \setminus A . p^{(i)} \Downarrow_{\underline{\sigma}} \bot$ and $fv(P) \subseteq PVar$ then $(\bigwedge_{i \in A} \sigma_i \vDash P) \Leftrightarrow \underline{\sigma} \vDash \mathring{\lfloor P \rfloor}^{\mathring{p}}_k$.*

Proof. Both directions of the proof go by induction on the structure of P. Cases relating to expressions follow from Lemma 5.3; all others follow from using the induction hypothesis on subassertions. □

Second, we show that the same property holds for tuples of stores and mixed assertions (and therefore relational assertions as well).

Lemma 5.10. *If* $\forall i \in \{1,\dots,k\}.\ \sigma_i \in_i \underline{\sigma} \wedge p^{(i)} \Downarrow_{\underline{\sigma}} \top$ *and* $fv(\check{P}) \subseteq PVar$ *then* $(\sigma_1,\dots,\sigma_k) \vDash \check{P} \Leftrightarrow \underline{\sigma} \vDash \lfloor \check{P} \rfloor_k^{\mathring{p}}$.

Proof. Both directions of the proof go by induction on the structure of $\check{P}$. Cases relating to unary assertions are covered by Lemma 5.9; cases relating to expressions follow from Lemma 5.3; all others follow from the induction hypothesis on subassertions. □

We can now prove the soundness theorem: If a transformed relational specification holds for a modular k-product program, then the relational specification holds for any k executions of the original program.

Theorem 5.11. *If* $\vDash [\![s]\!]_k^{\mathring{p}} : \lfloor \check{P} \rfloor_k^{\mathring{p}} \rightsquigarrow \lfloor \check{Q} \rfloor_k^{\mathring{p}}$ *then* $\vDash s : \check{P} \twoheadrightarrow_k \check{Q}$

Proof. We have to show that for all $(\sigma_1,\dots,\sigma_k)$ s.t. $(\sigma_1,\dots,\sigma_k) \vDash \check{P}$, if for all $i \in \{1,\dots,k\}$ $\langle s, \sigma_i\rangle \to^* \langle \dot{s}_i, \sigma_i'\rangle$ for some σ_i' and $\dot{s}_i \in \{\texttt{skip}, \texttt{error}\}$, then all $\dot{s}_i = \texttt{skip}$ and $(\sigma_1',\dots,\sigma_k') \vDash \check{Q}$.

Since we have $\vDash [\![s]\!]_k^{\mathring{p}} : \lfloor \check{P} \rfloor_k^{\mathring{p}} \rightsquigarrow \lfloor \check{Q} \rfloor_k^{\mathring{p}}$, we know that if $\underline{\sigma} \vDash \lfloor \check{P} \rfloor_k^{\mathring{p}}$ and $\langle [\![s]\!]_k^{\mathring{p}}, \underline{\sigma}\rangle \to^* \langle \dot{s}, \underline{\sigma}'\rangle$ where $\dot{s} \in \{\texttt{skip}, \texttt{error}, \texttt{magic}\}$, then either $\dot{s} = \texttt{magic}$ or $\dot{s} = \texttt{skip} \wedge \underline{\sigma}' \vDash \lfloor \check{Q} \rfloor_k^{\mathring{p}}$.

We choose $\underline{\sigma}$ s.t. $p^{(i)} \Downarrow_{\underline{\sigma}} \top$ and $\sigma_i \in_i \underline{\sigma}$ for all $i \in \{1,\dots,k\}$. By Lemma 5.10 we have that $\underline{\sigma} \vDash \lfloor \check{P} \rfloor_k^{\mathring{p}}$.

Assume that for all i we have $\langle s, \sigma_i\rangle \to^* \langle \dot{s}_i, \sigma_i'\rangle$ (otherwise the proof goal is trivially true).

- If all $\dot{s}_i = \texttt{skip}$, then by Thm. 5.7 this implies that there is a product execution $\langle [\![s]\!]_k^{\mathring{p}}, \underline{\sigma}\rangle \to^* \langle \texttt{skip}, \underline{\sigma}'\rangle$ for some $\underline{\sigma}'$. By Thm. 5.4 we then have that that $\underline{\sigma}' \vDash \lfloor \check{Q} \rfloor_k^{\mathring{p}}$ and for all $i \in \{1,\dots,k\}$, $\underline{\sigma}'(p^{(i)}) = \top$ and $\sigma_i' \in_i \underline{\sigma}'$. By Lemma 5.10 we have $(\sigma_1',\dots,\sigma_k') \vDash \check{Q}$.
- Otherwise: If all $\dot{s}_i \neq \texttt{error}$, we must have that some $\dot{s}_i = \texttt{magic}$, in which case we are done. If some $\dot{s}_i = \texttt{error}$ we case split on the structure of the precondition $\check{P}$:
  - If $\check{P} = \hat{P}'$ for some $\hat{P}'$: Then we trivially have that $\vDash s : \check{P} \twoheadrightarrow_k \check{Q}$.
  - If $\check{P} = P'$ for some P': Then we can choose some $\underline{\sigma}''$ s.t. $\underline{\sigma}(p^{(i)}) = \top$ and $\sigma_i \in_i \underline{\sigma}''$ and $\underline{\sigma}(p^{(j)}) = \bot$ for all $j \neq i$. By Lemma 5.9 we have that $\underline{\sigma}'' \vDash \lfloor \check{P} \rfloor_k^{\mathring{p}}$. Then by Thm. 5.8 this implies that there is a product execution $\langle [\![s]\!]_k^{\mathring{p}}, \underline{\sigma}''\rangle \to^* \langle \texttt{error}, \underline{\sigma}'''\rangle$ for some $\underline{\sigma}'''$, which is impossible because $\vDash [\![s]\!]_k^{\mathring{p}} : \lfloor \check{P} \rfloor_k^{\mathring{p}} \rightsquigarrow \lfloor \check{Q} \rfloor_k^{\mathring{p}}$.
  - If $\check{P} = \hat{P}' \wedge P''$ for some $\hat{P}', P''$: Then this case is impossible for the same reason as in the previous case.

□

Similarly, we prove completeness: If some unary precondition P guarantees that a program executes without errors, and some relational specification expressing a hyperproperty holds for the program, then the transformed version of the resulting mixed specification holds for the modular product program.

The unary precondition P is necessary to guarantee that the product program does not fail in executions where some activation variables are false and a k-relational precondition is therefore not required to hold. As an example, the program $s = \texttt{assert}\ x > 0$ fulfills the relational specification $s : x^{(1)} > 0 \wedge x^{(2)} > 0 \twoheadrightarrow_2 \texttt{true}$; however, $\lfloor x^{(1)} > 0 \wedge x^{(2)} > 0 \rfloor_2^{\mathring{p}} = (p^{(1)} \wedge p^{(2)} \Rightarrow x^{(1)} > 0 \wedge x^{(2)} > 0)$, and a store $[p^{(1)} \mapsto \bot, p^{(2)} \mapsto \top, x^{(1)} \mapsto 0, x^{(2)} \mapsto 0]$ therefore fulfills the transformed precondition

but raises an error when the product is executed. This is not a limitation in practice, since the unary precondition that guarantees error-free execution can always be extracted from a relational specification; in the example, we can replace the relational precondition $x^{(1)} > 0 \wedge x^{(2)} > 0$ by a unary precondition $x > 0$.

Theorem 5.12. *If $\hat{P}$ and $\hat{Q}$ are k-relational assertions, and $\vDash s : P \rightsquigarrow \texttt{true}$ and $\vDash s : \hat{P} \twoheadrightarrow_k \hat{Q}$ then $\vDash [\![s]\!]_k^{\dot{p}} : \lfloor P \wedge \hat{P} \rfloor_k^{\dot{p}} \rightsquigarrow \lfloor \hat{Q} \rfloor_k^{\dot{p}}$.*

Proof. We first show that for all $\underline{\sigma}$ s.t. $\underline{\sigma} \vDash \lfloor P \wedge \hat{P} \rfloor_k^{\dot{p}}$, if $\langle [\![s]\!]_k^{\dot{p}}, \underline{\sigma} \rangle \rightarrow^* \langle \dot{s}, \underline{\sigma}' \rangle$ for some $\dot{s}, \underline{\sigma}'$, then $\dot{s} = \texttt{magic}$ or $\dot{s} = \texttt{skip} \wedge \underline{\sigma}' \vDash \lfloor \hat{Q} \rfloor_k^{\dot{p}}$.

Assume that $\underline{\sigma} \vDash \lfloor P \wedge \hat{P} \rfloor_k^{\dot{p}}$ and $\langle [\![s]\!]_k^{\dot{p}}, \underline{\sigma} \rangle \rightarrow^* \langle \dot{s}, \underline{\sigma}' \rangle$ (otherwise the goal is trivially true).

- If $\dot{s} \neq \texttt{skip}$ then we get by Lemma 5.6 that for some $i \in \{1, \dots, k\}$, $\underline{\sigma}(p^{(i)}) = \top$, and for any σ_i s.t. $\sigma_i \in_i \underline{\sigma}$ we have $\langle s, \sigma_i \rangle \rightarrow^* \langle \dot{s}, \sigma_i' \rangle$ for some σ_i'.
  - If $\dot{s} = \texttt{magic}$ then the specification holds trivially.
  - If $\dot{s} = \texttt{error}$ then we choose some such σ_i and by Lemma 5.9 we get that $\sigma_i \vDash P$. But since $\vDash s : P \rightsquigarrow \texttt{true}$ we have that if $\langle s, \sigma_i \rangle \rightarrow^* \langle \dot{s}_i', \sigma_i' \rangle$ then $\dot{s}_i' \neq \texttt{error}$. Therefore we cannot have that $\langle s, \sigma_i \rangle \rightarrow^* \langle \texttt{error}, \sigma_i' \rangle$, and this case is impossible.
- If $\dot{s} = \texttt{skip}$, then we have by Thm. 5.4 that for all i s.t. $\underline{\sigma}(p^{(i)}) = \top$, $\langle s, \sigma_i \rangle \rightarrow^* \langle \texttt{skip}, \sigma_i' \rangle$ for some σ_i' s.t. $\sigma_i' \in_i \underline{\sigma}'$ and $\underline{\sigma}'(p^{(i)}) = \top$.
  - If $\forall i \in \{1, \dots, k\}. \underline{\sigma}(p^{(i)}) = \top$, then we have that also $\forall i \in \{1, \dots, k\}. \underline{\sigma}'(p^{(i)}) = \top$. Then by Lemma 5.10, we have that $\underline{\sigma}' \vDash \lfloor \hat{Q} \rfloor_k^{\dot{p}}$.
  - If $\exists i \in \{1, \dots, k\}. \underline{\sigma}(p^{(i)}) = \bot$, then we have that for said i, $\underline{\sigma}'(p^{(i)}) = \bot$. Since $\lfloor \hat{Q} \rfloor_k^{\dot{p}} = (\bigwedge_{i=1}^k p^{(i)}) \Rightarrow \hat{Q}$, we trivially have that $\underline{\sigma}' \vDash \lfloor \hat{Q} \rfloor_k^{\dot{p}}$.

Therefore we have $\vDash [\![s]\!]_k^{\dot{p}} : \lfloor P \wedge \hat{P} \rfloor_k^{\dot{p}} \rightsquigarrow \lfloor \hat{Q} \rfloor_k^{\dot{p}}$. □

6 MODULAR VERIFICATION OF SECURE INFORMATION FLOW

In this section, we demonstrate the expressiveness of modular product programs by showing how they can be used to verify non-interference, an important hyperproperty which can be used to express information flow security. We first concentrate on secure information flow in the classical sense [8], and later demonstrate how the ability to check relational assertions at any point in the program can be exploited to prove advanced properties like the absence of timing and termination channels, and to encode declassification.

6.1 Non-Interference

Secure information flow, i.e., the property that secret information is not leaked to the public outputs of a program, can be expressed as a 2-safety hyperproperty of a program called *non-interference*. Non-interference states that, if a program is run twice, with the public (often called *low*) inputs being equal in both runs but the secret (or *high*) inputs possibly being different, the public outputs of the program must be equal in both runs [8]. This property guarantees that the high inputs do not influence the low outputs.

We can formalize non-interference as follows:

Definition 6.1. A statement s that operates on a set of variables $X = \{x_1, \dots, x_n\}$, of which some subset $X_l \subseteq X$ is low, satisfies non-interference iff for all σ_1, σ_2 and σ_1', σ_2', if $\forall x \in X_l. \sigma_1(x) = \sigma_2(x)$ and $\langle s, \sigma_1 \rangle \rightarrow^* \langle \texttt{skip}, \sigma_1' \rangle$ and $\langle s, \sigma_2 \rangle \rightarrow^* \langle \texttt{skip}, \sigma_2' \rangle$ then $\forall x \in X_l. \sigma_1'(x) = \sigma_2'(x)$. ■

Since our definition of non-interference describes a hyperproperty, we can verify it using modular product programs:

Theorem 6.2. *A statement s that operates on a set of variables $X = \{x_1, \ldots, x_n\}$, of which some subset $X_l \subseteq X$ is low, satisfies non-interference under a unary precondition P if* $\vDash [\![s]\!]_2^{\mathring{P}} : \lfloor P \rfloor_2^{\mathring{P}} \wedge (\bigwedge_{x \in X_l} x^{(1)} = x^{(2)}) \rightsquigarrow \forall x \in X_l.\, x^{(1)} = x^{(2)}$

Proof. Since non-interference can be expressed using a 2-relational specification, the theorem follows directly from Theorem 5.11. □

An expanded notion of secure information flow considers observable *events* in addition to regular program outputs [21]. An event is a statement that has an effect that is visible to an outside observer, but may not necessarily affect the program state. The most important examples of events are output operations like printing a string to the console or sending a message over a network. Programs that cause events can be considered information flow secure only if the sequence of produced events is not influenced by high data. One way to verify this using our approach is to track the sequence of produced events in a ghost variable and verify that its value never depends on high data. This approach requires substantial amounts of additional specifications.

Modular product programs offer an alternative approach for preventing leaks via events, since they allow formulating assertions about the relation between the activation variables of different executions. In particular, if a given event has the precondition that all activation variables are equal when the event statement is reached then this event will either be executed by both executions or be skipped by both executions. As a result, the sequence of events produced by a program will be equal in all executions.

6.2 Information Flow Specifications

The relational specifications required for modularly proving non-interference with the previously described approach have a specific pattern: they can contain functional specifications meant to be valid for both executions (e.g., to make sure both executions run without errors), they may require that some information is low, which is equivalent to the two renamings of the same expression being equal, and, in addition, they may assert that the control flow at a specific program point is low.

We therefore introduce modular *information flow specifications*, which can express properties required for proving secure information flow but are transparent w.r.t. the encoding or the verification methodology, i.e., they allow expressing that a given event or value must not be secret without knowledge of the encoding of this fact into an assertion about two different program executions. We define information flow specifications as follows:

$$\begin{aligned}(\mathit{SIFAssertions}) \quad \tilde{P} \quad ::= \quad & \tilde{P} \wedge \tilde{P} \mid e \mid \mathit{low}(e) \mid \mathit{lowEvent} \mid \forall x.\, \tilde{P} \\ & \mid e \Rightarrow e \mid e \Rightarrow \mathit{low}(e) \mid \mathit{low}(e) \Rightarrow \mathit{low}(e) \mid e \Rightarrow \mathit{lowEvent}\end{aligned}$$

$\mathit{low}(e)$ specifies that the value of the expression e is not influenced by high data. Note that e can be any expression and is not limited to variable references; this reflects the fact that our approach can label secrecy in a more fine-grained way than, e.g., a type system; one can, for example, declare to be public whether a number is odd while keeping its value secret.

lowEvent specifies that high data must not influence if and how often the current program point is reached by an execution, which is a sufficient precondition of any statement that causes an observable event. In particular, if a procedure outputs an expression e, the precondition $\mathit{lowEvent} \wedge \mathit{low}(e)$ guarantees that no high information will be leaked via this procedure.

$$
\begin{array}{lcl}
\lceil e \rceil^{\dot{p}} & = & (p^{(1)} \Rightarrow e^{(1)}) \wedge (p^{(2)} \Rightarrow e^{(2)}) \\
\lceil low(e) \rceil^{\dot{p}} & = & p^{(1)} \wedge p^{(2)} \Rightarrow e^{(1)} = e^{(2)} \\
\lceil lowEvent \rceil^{\dot{p}} & = & p^{(1)} = p^{(2)} \\
\lceil \tilde{P}_1 \wedge \tilde{P}_2 \rceil^{\dot{p}} & = & \lceil \tilde{P}_1 \rceil^{\dot{p}} \wedge \lceil \tilde{P}_2 \rceil^{\dot{p}} \\
\lceil e_1 \Rightarrow e_2 \rceil^{\dot{p}} & = & (p^{(1)} \wedge {e_1}^{(1)} \Rightarrow {e_2}^{(1)}) \wedge (p^{(2)} \wedge {e_1}^{(2)} \Rightarrow {e_2}^{(2)}) \\
\lceil e_1 \Rightarrow low(e_2) \rceil^{\dot{p}} & = & p^{(1)} \wedge {e_1}^{(1)} \wedge p^{(2)} \wedge {e_1}^{(2)} \Rightarrow {e_2}^{(1)} = {e_2}^{(2)} \\
\lceil e_1 \Rightarrow lowEvent \rceil^{\dot{p}} & = & (p^{(1)} \wedge e^{(1)} \vee p^{(2)} \wedge e^{(2)}) \Rightarrow p^{(1)} = {p_2}^{(2)} \\
\lceil low(e_1) \Rightarrow low(e_2) \rceil^{\dot{p}} & = & p^{(1)} \wedge p^{(2)} \wedge {e_1}^{(1)} = {e_1}^{(2)} \Rightarrow {e_2}^{(1)} = {e_2}^{(2)} \\
\lceil \forall x.\, \tilde{P} \rceil^{\dot{p}} & = & \forall x^{(1)}, x^{(2)}.\, x^{(1)} = x^{(2)} \Rightarrow \lceil \tilde{P} \rceil^{\dot{p}}
\end{array}
$$

Fig. 6. Translation of information flow specifications. The intuition behind the translation of universal quantifiers is that we quantify over a single variable x but give it two different names $x^{(1)}$ and $x^{(2)}$ so that we can transform the body of the quantifier like any other assertion.

Information flow specifications can express complex properties. $e_1 \Rightarrow low(e_2)$, for example, expresses that if e_1 is true, e_2 must not depend on high data; $e_1 \Rightarrow lowEvent$ says the same about the current control flow. A possible use case of these assertions is the precondition of a library function that prints e_2 to a low-observable channel if e_1 is true, and to a secure channel otherwise.

In addition to the two primitives *low*() and *lowEvent*, information flow specifications can also contain ordinary functional specifications for single executions. These can be used as a fallback if said primitives are not expressive enough. As an example, for proving that the statement `if (e) then {x.m()} else {x.n()}` ends with x being low, it may be necessary to give a description of the functional behavior of m and n. Similarly, if both m and n perform the same output, requiring the calls to be a *lowEvent* (which is sufficient but generally not necessary) is too strong a requirement, and their functional contracts must model the output explicitly instead.

Note that high-ness of some expression is not modelled by its renamings being definitely unequal, but by leaving underspecified whether they are equal or not, meaning that high-ness is simply the absence of the knowledge of low-ness. As a result, it is never necessary (or possible) to specify explicitly that an expression is high. This approach (which is also used in self-composition) is analogous to the way type systems encode security levels, where low is typically a subtype of high.

The encoding $\lceil \tilde{P} \rceil^{\dot{p}}$ of an information flow assertion $\tilde{P}$ under the activation variables $p^{(1)}$ and $p^{(2)}$ is defined in Figure 6. For the example in Figure 1, a possible, very precise information flow specification could say that the results of `main` are low if the first bit of all entries in `people` is low. We can write this as $\texttt{main} : low(|\texttt{people}|) \wedge \forall i \in \{0, \ldots, |\texttt{people}| - 1\}.\, low(\texttt{people[i] mod 2}) \rightsquigarrow low(\texttt{count})$. In the product, this will be translated to

$$
\begin{array}{lll}
\texttt{main}: & \texttt{p1} \wedge \texttt{p2} \Rightarrow & (|\texttt{people1}| = |\texttt{people2}| \wedge \\
 & & \forall i \in \{0, \ldots, |\texttt{people1}| - 1\}.\, (\texttt{people1[i] mod 2}) = (\texttt{people2[i] mod 2})) \\
\rightsquigarrow & \texttt{p1} \wedge \texttt{p2} \Rightarrow & \texttt{count1} = \texttt{count2}
\end{array}
$$

In this scenario, the loop in `main` could have the simple invariant $low(\texttt{i}) \wedge low(\texttt{count})$, and the procedure `is_female` could have the contract $\texttt{is\_female} : true \rightsquigarrow (low(\texttt{person mod 2}) \Rightarrow low(\texttt{res}))$. This contract follows a useful pattern where, instead of requiring an input to be low and promising that an output will be low for all calls, the output is decribed as *conditionally* low based on the level of the input, which is more permissive for callers.

The example shows that the information required for proving secure information flow can in many cases be expressed concisely, without requiring any knowledge about the methodology used

```
procedure check(password, input)
          returns (result)
{
  result := |password| == |input|;
  i := 0;
  while (i < |password| && result) {
     result := result && password[i] == input[i];
     i := i + 1;
  }
}
```

Fig. 7. Password check example: Leaking secret data is desired.

for verification. Modular product programs therefore enable the verification of the information flow security of `main` based solely on modular, relational specifications, and without depending on functional specifications.

6.3 Secure Information Flow with Arbitrary Security Lattices

The definition of secure information flow used in Definition 6.1 is a special case in which there are exactly two possible classifications of data, high and low. In the more general case, classifications come from an arbitrary lattice $\langle \mathcal{L}, \sqsubseteq \rangle$ of security levels s.t. for some $l_1, l_2 \in \mathcal{L}$, information from an input with level l_1 may influence an output with level l_2 only if $l_1 \sqsubseteq l_2$. Instead of the specification $low(e)$, information flow assertions can therefore have the form $levelBelow(e, l)$, meaning that the security level of expression e is at most l.

It is well-known that techniques for verifying information flow security with two levels can conceptually be used to verify programs with arbitrary finite security lattices [30] by splitting the verification task into $|\mathcal{L}|$ different verification tasks, one for each element of $\mathcal{L}$. Instead, we propose to combine all these verification tasks into a single task by using a symbolic value for l, i.e., declaring an unconstrained global constant representing l. Specifications can then be translated as follows:

$$levelBelow(e, l') \triangleq l' \sqsubseteq l \Rightarrow e^{(1)} = e^{(2)}$$

Since no information about l is known, verification will only succeed if all assertions can be proven for all possible values of l, which is equivalent to proving them separately for each possible value of l.

6.4 Declassification

In practice, non-interference is too strong a property for many use cases. Often, some leakage of secret data is required for a program to work correctly. Consider the case of a password check (see Figure 7): A secret internal password is compared to a non-secret user input. While the password itself must not be leaked, the information whether the user input matches the password should influence the public outcome of the program, which is forbidden by non-interference.

To incorporate this intention, the part of the secret information that should be leaked by the program can be *declassified* [35], e.g., via a declassification statement `declassify` e in the style of delimited information release [34] that declares an arbitrary expression e to be low. With modular products (as with self-composition [40]), declassification can be encoded via a simple assumption stating that, if the declassification is executed in both executions, the expression is equal in both executions:

$$[\![\texttt{declassify}\ e]\!]_2^{\hat{p}} = \texttt{assume}\ p^{(1)} \wedge p^{(2)} \Rightarrow e^{(1)} = e^{(2)}$$

```
procedure main(h: Int)
{
  while (h != 0) {
    h := h - 1;
  }
}
```

```
procedure main(h: Int)
{
  i := 0;
  while (i < h) {
    i := i + 1
  }
  print(0)
}
```

Fig. 8. Programs with a termination channel (left), and a timing channel (right). In both cases, h is high.

Importantly, an assumptions of this form can never contradict current knowledge (i.e., assume *false*) if the information flow specifications from Section 6.2 are used to specify the program. Recall that high-ness is encoded as the absence of the knowledge that an expression is equal in both executions, *not* by the knowledge that they are different. In fact, the format of information flow specifications makes it impossible to specify that an expression is definitely not equal in both executions, since $low(e)$ can only occur in positive positions, i.e., not on the left side of an implication; the only exception, $low(e_1) \Rightarrow low(e_2)$, implies that ${e_1}^{(1)} \neq {e_1}^{(2)}$ only under the condition that one can already prove ${e_2}^{(1)} \neq {e_2}^{(2)}$, which as just argued can never happen. As a result, there is no danger that assuming equality will contradict current knowledge, since any such assumption (and any combination of such assumptions) will always be fulfilled if the state of both executions represented in the product program is completely identical. Nevertheless, assumptions outside of procedure contracts can obfuscate what property was actually proved. In order to get better formal guarantees, one could for example generate declassification statements based on some declassification policy. The exact process of doing this, however, is orthogonal our approach. As in the information flow specifications, the declassified expression can be arbitrarily complex, so that it is for example possible to declassify the parity of a number while keeping all other information about it secret.

The example in Fig. 7 becomes valid if we add `declassify result` at the end of the procedure, or `declassify` *equal*(`password`, `input`) at some earlier point. The latter would arguably be safer because it specifies exactly the information that is intended to be leaked, and would therefore prevent accidentally leaking more if the implementation of the checking loop was faulty.

This kind of declassification has the following interesting properties: First, it is *imperative*, meaning that the declassified information may be leaked (e.g., via a `print` statement) after the execution of the declassification statement, but not before. Second, it is *semantic*, meaning that the declassification affects the value of the declassified expression as opposed to, e.g., syntactically the declassified variable. As a result, it will be allowed to leak any expression whose value contains the same (or a part of the) secret information which was declassified, e.g., the expression $f(e)$ if f is a deterministic function and e has been declassified.

6.5 Preventing Termination Channels

In Definition 6.1, we have considered only terminating program executions. In practice, however, termination is a possible side-channel that can leak secret information to an outside observer. Figure 8 (left) shows an example of a program that verifies under the methodology presented so far, but leaks information about the secret input h to an observer: If h is initially negative, the program will enter an endless loop. Anyone who can observe the termination behavior of the program can therefore conclude if h was negative or not.

To prevent leaking information via a termination side channel, it is necessary to verify that the termination of a program depends only on public data. We will show that modular product

programs are expressive enough to encode and check this property. We will focus on preventing non-termination caused by infinite loops here; preventing infinite recursion works analogously. In particular, we want to prove that if a loop iterates forever in one execution, any other execution with the same low inputs will also reach this loop and iterate forever. More precisely, this means that

(A) if a loop does not terminate, then whether or not an execution reaches that loop must not depend on high data.
(B) whether a loop that is reached by both executions terminates must not depend on high data.

We propose to verify these properties by requiring additional specifications that state, for every loop, an *exact* condition under which it terminates. This condition may neither over- nor underapproximate the termination behavior; the loop must terminate if and only if the condition is true. For Figure 8 (left) the condition is $h \geq 0$. We also require a ranking function for the cases when the termination condition is true. We can then prove the following:

(a) If the termination condition of a loop evaluates to false, then any two executions with identical low inputs either both reach the loop or both do not reach the loop (i.e., reaching the loop is a low event). This guarantees property (A) above.
(b) For loops executed by both executions, the loop's termination condition is low. This guarantees property (B) under the assumption that the termination condition is exact.
(c) The termination condition is sound, i.e., every loop terminates if its termination condition is true. We prove this by showing that if the termination condition is true, we can prove the termination of the loop using the supplied ranking function.
(d) The termination condition is complete, i.e., every loop terminates only if its termination condition is true. We prove this by showing that if the condition is false, the loop condition will always remain true. This check, along with the previous proof obligation, ensures that the termination condition is exact.
(e) Every statement in a loop body terminates if the loop's termination condition is true, i.e., the loop's termination condition implies the termination conditions of all statements in its body.

We introduce an annotated while loop `while` (e) `terminates`(e_c, e_r) `do` $\{s\}$, where e_c is the exact termination condition and e_r is the ranking function, i.e., an integer expression whose value decreases with every loop iteration but never becomes negative if the termination condition is true. Based on these annotations, we present a program instrumentation $term(s, c)$ that inserts the checks outlined above for every while loop in s. c is the termination condition of the outside scope, i.e., for the instrumentation of a nested loop, it is the termination condition e_c of the outer loop. The instrumentation is defined for annotated while loops in Figure 9; for all other statements, it does not make any changes except instrumenting all subnodes. The instrumentation guarantees that if the product of the instrumented program verifies, the original program does not leak secret information via a termination channel. We slightly abuse the notation and use `assert` statements with information flow assertions (as defined in Section 6.2) instead of expressions; the intented meaning is that the product of such a statement is a statement that asserts the encoded version of said assertion (which can be written as an expression). Again, we make use of the fact that modular products allow checking relational assertions at arbitrary program points and formulating assertions about the control flow.

We now prove that if an instrumented statement verifies under some 2-relational precondition then any two runs from a pair of states fulfilling that precondition will either both terminate or both loop forever.

$$
\begin{array}{lcl}
term(w, c) & = & cond{:=}e_c; \\
 & & \texttt{assert}\ \neg cond \Rightarrow lowEvent; \qquad // \textit{ checks (a)} \\
 & & \texttt{assert}\ low(cond); \qquad // \textit{ checks (b)} \\
 & & \texttt{assert}\ cond \Rightarrow 0 \leq e_r; \qquad // \textit{ checks (c)} \\
 & & \texttt{assert}\ c \Rightarrow cond; \qquad // \textit{ checks (e)} \\
 & & \texttt{assert}\ \neg cond \Rightarrow e; \qquad // \textit{ checks (d)} \\
 & & \texttt{while}\ (e)\ \texttt{do}\ \{ \\
 & & \ \texttt{if}\ (cond)\ \texttt{then}\ \{rank{:=}e_r\}; \\
 & & \ term(s, cond); \\
 & & \ \texttt{assert}\ cond \Rightarrow 0 \leq e_r \wedge e_r < rank \qquad // \textit{ checks (c)} \\
 & & \ \texttt{assert}\ \neg cond \Rightarrow e; \qquad // \textit{ checks (d)} \\
 & & \}
\end{array}
$$

Fig. 9. Program instrumentation for termination leak prevention. We abbreviate $\texttt{while}\ (e)\ \texttt{terminates}(e_c, e_r)\ \texttt{do}\ \{s\}$ as w. If $[\![term(s, false)]\!]_2^{\mathring{P}}$ verifies under some precondition, s does not have a termination side channel under this precondition. Statements of the form $\texttt{assert}\ \check{P}$ are to be interpreted as asserting the encoding $\texttt{assert}\ \lceil \check{P} \rceil^{\mathring{P}}$ in the product program.

Theorem 6.3. *Assume that* $\underline{s} = [\![term(s, false)]\!]_2^{\mathring{P}}$ *and* $\vDash \underline{s} : \lfloor \check{P} \rfloor_2^{\mathring{P}} \rightsquigarrow$ *true and* $(\sigma_1, \sigma_2) \vDash \check{P}$ *and* s *does not terminate with* `magic` *from* σ_1 *and* σ_2 *and* s *does not contain calls to (mutually) recursive procedures.*

Then either s *always terminates from both* σ_1 *and* σ_2 *or* s *never terminates from both* σ_1 *and* σ_2.

Proof. We first prove lemmas that state that if the product of an instrumented loop does not fail from a state, single executions of this loop from states that mirror the product state terminate (1) if and (2) only if their termination condition is true. We show (1) by a standard termination proof using the provided ranking function, and (2) by showing that the loop condition never becomes false. We then extend these lemmas to general instrumented statements. The main proof then goes by induction on the structure of s; we show that if the product of an instrumented statement does not fail from a state, pairs of executions from states that mirror the product state either both terminate or both diverge. □

Note that, while this approach is sound, it is incomplete, since it requires that the termination of every single loop by itself does not leak information. It will therefore reject, for example, a program consisting two consecutive loops, if the first terminates depending on secret data, and the second never terminates at all. Assuming that an attacker can only observe the termination of the program as a whole, no information is leaked since the program never terminates, and the program should be accepted. This incompleteness is similar to the one that occurs when specifying method invocations to be low events (discussed in Section 6.2).

6.6 Preventing Timing Channels

A program has a *timing channel* if high input data influences the program's execution time, meaning that an attacker who can observe the time the program executes can gain information about those secrets. Timing channels can occur in combination with observable events; the time at which an event occurs may depend on a secret even if the overall execution time of a program does not. Consider the example in Figure 8 (right). Assuming `main` receives a positive secret `h`, both the `print` statement and the end of the program execution will be reached later for larger values of `h`.

Using modular product programs, we can verify the absence of timing side channels by adding ghost state to the program that tracks the time passed since the program has started; this could, for example, be achieved via a simple step counting mechanism, or by tracking the sequence of previously executed bytecode statements. This ghost state is updated separately for both executions. We can then assert anywhere in the program that the passed time does not depend on high data in the same way we do it for program variables. In particular, we can enforce that the passed time is equal whenever an observable event occurs, and we can enable users to write relational specifications that compare the time passed in both executions of a loop or a procedure.

7 IMPLEMENTATION AND EVALUATION

We have implemented both the general modular product program transformation and our approach for secure information flow in the Viper verification infrastructure [29] and applied it to a number of example programs from the literature. Both the implementation and examples are available at http://viper.ethz.ch/modularproducts/.

7.1 Viper Overview

We first give a brief overview of the Viper verification infrastructure. Viper provides an intermediate verification language which is essentially a simple imperative programming language with a mutable heap and built-in support for specifications, such as pre- and postconditions and loop invariants.

Viper uses implicit dynamic frames [37], a flavor of separation logic [33], to locally reason about heap-manipulating programs. This means that every heap location is associated with a *permission*, and reading or modifying a heap location requires having the respective permission. Procedure specifications can declare which permissions are required via a predicate of the form $\texttt{acc}(x.f)$ representing the permission to field f of the object referenced by x (similar to a points-to predicate in separation logic). A procedure call then transfers the permission required in the callee's precondition to the callee, and subsequently transfers the permissions specified in the postcondition back to the caller. Procedures therefore know that heap locations whose permissions they do not give away cannot be modified by said callee, and must therefore be unchanged after the call.

For supporting unbounded heap data structures like lists or trees, Viper offers *quantified* permission assertions of the form $\texttt{forall}\ x \in s.\ \texttt{acc}(x.f)$, which represent the permissions to the f fields for all references in set s [28].

Viper provides two different backend verifiers that can verify programs written in the Viper language (both ultimately using the SMT solver Z3 [16]). One is based on symbolic execution (SE), and the other is based on verification condition generation (VGC) and encodes a Viper program into a Boogie [5] program.

7.2 Implementation in Viper

Our implementation supports a version of the Viper language that adds the following features:

(1) Expressions of the form `rel(x, i)` that represent a reference $x^{(i)}$ to variable x from execution i and can be used to express general relational specifications
(2) The assertions *low*(e) and *lowEvent* for information flow specifications
(3) A `declassify` statement
(4) Variations of the existing method declarations and while loops that include the termination annotations shown in Section 6.5

The implementation transforms a program in this extended language into a modular k-product for arbitrary k in the original language, which can then be verified by the (unmodified) Viper back-end verifiers. Error messages are automatically translated back to the original program.

In the resulting language, users can use unary, relational, and mixed assertions in method pre- and postconditions as well as loop invariants, and can therefore specify any hyperproperty that is expressible in the framework we presented so far.

Alternatively, if $k = 2$, specifications can be provided as information flow specifications (see Section 6.2) such that users need no knowledge about the transformation or the methodology behind information flow verification.

Declassification is implemented as described in Section 6.4. Our implementation optionally verifies the absence of timing channels; the metric chosen for tracking execution time is simple step-counting.

For languages with opaque object references, secure information flow can require proving that pointers are low, i.e., equal up to a consistent renaming of addresses. To avoid the complexity of reasoning about such a renaming, our approach to duplicating the heap state space in the implementation differs from that described in Section 4.3: Instead of duplicating objects, our implementation creates a single `new` statement for every `new` in the original program, but duplicates the fields each object has. As a result, if both executions execute the same `new` statement, the newly created object will be considered low afterwards (but the values of its fields might still be high).

7.3 Evaluation: Secure Information Flow

We have evaluated our approach for verifying secure information flow by verifying a number of examples in the extended Viper language using our implementation. The examples are listed in Table 1 and include all code snippets shown in this paper as well as a number of examples from the literature [2, 3, 6, 14, 15, 21, 24, 30, 38, 40]. They combine complex language features like mutable state on the heap, arrays and procedure calls, as well as timing and termination channels, declassification, and non-trivial information flows (e.g., flows whose legality depends on semantic information not available in a standard information flow type system). We manually added pre- and postconditions as well as loop invariants; for those examples that have forbidden flows and therefore should not verify, we also added a legal version that declassifies the leaked information.

7.3.1 Qualitative Evaluation. Our implementation returns the correct result for all examples. In all cases but one, our approach allows us to express all information flow related assertions, i.e., procedure specifications and loop invariants, purely as relational specifications in terms of *low* and *lowEvent* assertions (see Table 1). For all these examples, we completely avoid the need to specify the functional behavior of the program.

The only exception is an example that, depending on a high input, executes different loops with identical behavior, and for which we need to prove that the execution time is low. In this case we have to provide invariants for both loops that exactly specify their execution time in order to prove that the overall execution time after the conditional is low. Nevertheless, the specification of the procedure containing the loop is again expressed with a relational specification using only *low*. For all other examples, unary specifications were only needed to verify the absence of runtime errors (e.g., out-of-bounds array accesses), which Viper verifies by default. Consequently, a verified program cannot leak low data through such errors, which is typically not guaranteed by type systems or static analyses.

7.3.2 Performance. For all but one example, the runtime (averaged over 10 runs on a Lenovo ThinkPad T450s with an Intel Core i7-5600U CPU running at 3.2 GHz with 12GB RAM on Ubuntu) with both the Symbolic Execution (SE) and the Verification Condition Generation (VCG) verifiers is under or around one second (see Table 1). The one exception, which makes extensive use of unbounded heap data structures, takes ca. five seconds when verified using VCG, and 15 in the SE verifier. This is likely a result of inefficiencies in our encoding: As noted before, the created product

File	Ev.	Hp.	Arr.	Decl.	Call	Ter.	Tm.	LOC	Ann/SF/NI/TM/F	T_{VCG}	T_{SE}
antopolous1 [2]							x	25	7/3/3/0/2	0.78	1.10
antopolous2 [2]				x			x	61	14/0/14/0/0	0.72	0.91
banerjee [3]		x		x	x			76	17/11/6/0/0	1.02	0.61
constanzo [14]	x		x					22	7/2/5/0/0	0.67	0.28
darvas [15]		x		x				33	12/8/4/0/0	0.67	0.35
example			x		x			31	7/1/6/0/0	0.73	0.59
example_decl			x	x				19	5/2/3/0/0	0.72	0.77
example_term				x		x		31	8/4/2/2/0	0.77	0.43
example_time	x			x	x		x	32	9/0/9/0/0	0.70	0.38
joana_1_tl [21]	x			x	x			28	1/0/1/0/0	0.62	0.23
joana_2_bl [21]	x				x			18	2/0/2/0/0	0.63	0.25
joana_2_t [21]	x							15	1/0/1/0/0	0.62	0.20
joana_3_bl [21]	x			x	x	x		47	5/1/2/2/0	0.77	0.47
joana_3_br [21]	x			x	x	x		43	8/0/2/6/0	0.83	0.60
joana_3_tl [21]	x				x	x		33	8/2/2/4/0	0.75	0.53
joana_3_tr [21]	x			x	x	x		35	8/4/2/2/0	0.76	0.51
joana_13_l [21]					x			12	1/0/1/0/0	0.62	0.24
kusters [24]		x			x			29	9/6/3/0/0	0.64	0.44
naumann [30]		x	x					20	6/3/6/0/0	0.81	0.88
product [6]		x	x		x			65	30/21/21/0/0	5.47	15.73
smith [38]			x	x				43	12/6/8/0/0	0.87	0.89
terauchi1 [40]								14	2/0/2/0/0	0.62	0.26
terauchi2 [40]				x	x			21	4/0/4/0/0	0.63	0.30
terauchi3 [40]								24	5/1/4/0/0	0.66	0.40

Table 1. Evaluated examples for secure information flow. We show the used language features (Ev. = observable events, Hp. = mutable heap, Arr. = arrays, Decl. = declassification, Call = procedure calls) and proved properties (Ter. = absence of termination channels, Tm. = absence of timing channels), lines of code including specifications (LOC), overall lines used for specifications (Ann), unary specifications for safety (SF), relational specifications for non-interference (NI), specifications for termination (TM), and functional specifications required for non-interference (F). Note that some lines contain specifications belonging to multiple categories. Columns T_{SE} and T_{VCG} show the running times of the verifiers for the SE backend and the VCG backend, respectively, in seconds.

has a high number of branching instructions, and some properties have to be proved more than once, two issues which have a much larger performance impact for SE than for VCG. We believe that it is feasible to remove much of this overhead by optimizing the encoding; we leave this as future work.

7.4 Evaluation: Other Hyperproperties

For the second part of our evaluation, we evaluated our approach and implementation for other hyperproperties and compared them to DESCARTES[1], a specialized tool by Sousa and Dillig that automates a relational logic and handles loops by guessing and checking possible invariants [39], as well as SYNONYM[2], a version of DESCARTES by Pick et al. with additional optimizations to speed up the verification process [31].

We considered 34 Java implementations of comparators, taken from the evaluation of DESCARTES. The examples were originally taken from posts on websites like *Stackoverflow*, where developers asked questions about buggy comparators and other users responded with proposed fixes. Additionally, we considered all eight correct *expanded* examples from the evaluation of SYNONYM; these are variations of correct comparators from DESCARTES that were artificially made more complex by Pick et al. Instead of comparing two objects, they take three objects and decide which one is the greatest according to the defined comparison metric. These implementations are generally both

[1]https://github.com/marcelosousa/descartes
[2]https://github.com/lmpick/synonym

```
public class AInt
      implements Comparator<AInt>{
  int length;
  int get(int pos) { ... }

  public int compare(AInt o1, AInt o2){
    int index, aentry, bentry;
    index = 0;
    while ((index < o1.length) &&
           (index < o2.length)) {
      aentry = o1.get(index);
      bentry = o2.get(index);
      if (aentry < bentry) {
        return -1;
      }
      if (aentry > bentry) {
        return 1;
      }
      index++;
    }
    return 0;
  }
}
```

```
public class AInt
      implements Comparator<AInt>{
  int length;
  int get(int pos) { ... }

  public int compare(AInt o1, AInt o2){
    int index, aentry, bentry;
    index = 0;
    while ((index < o1.length) &&
           (index < o2.length)) {
      aentry = o1.get(index);
      bentry = o2.get(index);
      if (aentry < bentry) {
        return -1;
      }
      if (aentry > bentry) {
        return 1;
      }
      index++;
    }
    if (o1.length < o2.length) {
      return -1;
    }
    if (o1.length > o2.length) {
      return 1;
    }
    return 0;
  }
}
```

Fig. 10. Example of a faulty comparator implementation (left) and a fixed version (right) [39]. The left implementation fulfills properties P1 and P2 but violates property P3, the right version fulfills all three properties.

longer than the original versions (the longest, `PokerHand`, is 297 LOC long in the original Java version) and contain more branching, and can thus be used to test the scalability of our verification approach. Since the examples are derived from real-world code, they have a number of challenging properties: Many of them call the `compare` methods of other classes, many have deeply nested conditional structures, and many loop over some data structure on both objects to compare them.

Comparators have a single method `compare` that has to fulfill, among others, the following hyperproperties:

- P1: $\forall x, y.\ sgn(\texttt{compare}(x, y)) = -sgn(\texttt{compare}(y, x))$
- P2: $\forall x, y, z.\ \texttt{compare}(x, y) > 0 \land \texttt{compare}(y, z) > 0 \implies \texttt{compare}(x, z) > 0$
- P3: $\forall x, y, z.\ \texttt{compare}(x, y) = 0 \implies sgn(\texttt{compare}(x, z)) = sgn(\texttt{compare}(y, z))$

For the modified `pick` methods, we want to check the following hyperproperties (retaining the naming by Pick et al.):

- P13: $\forall x, y, z.\ pick(x, y, z) = pick(y, x, z)$
- P14: $\forall x, y, z.\ pick(x, y, z) = pick(y, x, z) \land pick(x, y, z) = pick(z, y, x)$

Since all implementations have the signatures `int compare(T x, T y)` and `int pick(T x, T y, T z)`, respectively, where `T` is the class whose objects are being compared, these properties can be expressed as follows in our specification language:

- P1: $\texttt{compare} : x^{(1)} = y^{(2)} \land y^{(1)} = x^{(2)} \nrightarrow_2 sgn(res^{(1)}) = -sgn(res^{(2)})$
- P2: $\texttt{compare} : y^{(1)} = x^{(2)} \land x^{(3)} = x^{(1)} \land y^{(3)} = y^{(2)} \land res^{(1)} > 0 \land res^{(2)} > 0) \nrightarrow_3 res^{(3)} > 0$
- P3: $\texttt{compare} : x^{(1)} = x^{(2)} \land y^{(1)} = x^{(3)} \land y^{(3)} = y^{(2)} \land res^{(1)} = 0 \nrightarrow_3 sgn(res^{(2)}) = sgn(res^{(3)})$

- P13: pick : $true \twoheadrightarrow_s (x^{(1)} = y^{(2)} \wedge x^{(2)} = y^{(1)} \wedge z^{(1)} = z^{(2)}) \Rightarrow res^{(1)} = res^{(2)}$
- P14: pick : $true \twoheadrightarrow_s ((x^{(1)} = y^{(2)} \wedge x^{(2)} = y^{(1)} \wedge z^{(1)} = z^{(2)}) \vee (x^{(1)} = z^{(2)} \wedge y^{(1)} = y^{(2)} \wedge z^{(1)} = x^{(2)})) \Rightarrow res^{(1)} = res^{(2)}$

For each comparator example, Sousa and Dillig provide a version of the comparator that was initially reported as broken, as well as (both successful and incorrect) attempts at fixed versions. Like them, we check all three hyperproperties for all versions of the examples. Fig. 10 shows an example of one of the simpler faulty comparator implementations from the data set as well as a fixed version. For the modified examples by Pick et al., we verify both properties only for the correct implementations.

Our encoding is based on the examples as encoded by Sousa and Dillig and Pick et al., since the original versions of some of the examples are no longer available. In particular, this means that we adopt some simplifications performed by them; however, where we could find the original versions, we tried to stay as close to them as possible. As an example, in several cases, Sousa and Dillig exchanged loops that iterate over a list by loops with a statically fixed number of iterations, and with the current element in each iteration being the result of calling an unspecified function; we reverted this change.

Since the examples were originally written in Java, they contain `return` statements, which do not exist in the Viper language. We manually emulate them in the standard way by declaring an output parameter `res` as well as a boolean flag `returned`, which is initially not set. `return` statements are then encoded by assigning the returned expression to the result variable, setting the `returned` flag, and making subsequent statements conditional on `returned` not being set. Additionally, we add the conjunct ¬`returned` to the guards of loops whose bodies contain `return` statements.

As before, we manually added pre- and postconditions as well as loop invariants to all examples.

7.4.1 Qualitative Evaluation. Our implementation returns the correct results (i.e., verifies correct implementations and shows errors for incorrect ones) for all of the original comparator examples (see Table 2). In that, it behaves identically to Descartes. For the modified examples, our implementation can correctly verify all examples with the VCG backend, but fails to verify four cases with the SE backend due to timeouts. Synonym times out on one example and is unable to infer sufficient invariants in two other cases, and Descartes times out seven times and does not find sufficient invariants in one case. Note, however, that the feature set of the different tools is quite different and comparing the performance can therefore only give an indication of their respective scalability; we discuss the differences in detail in the next section.

In all cases, we were able to use purely relational specifications for other `compare` methods called by the comparators to be verified. This is vital, since doing otherwise would require information about the precise behavior and therefore the contents and internal data structures of referenced classes (some of which are presumably quite complex), which clearly violates information hiding. By using relational specifications, we were able to prove the desired hyperproperties of each comparator only assuming that the same hyperproperties hold for called comparators of other classes.

For loop invariants, we followed a mixed approach for specifications. For proving P1, we had to use only simple, relational loop invariants. For P2 and P3, where $k = 3$, we opted to use unary loop invariants (i.e., full functional specifications of the loop behavior) for some loops with simple functional behavior. The main reason for this choice was that the proof would otherwise have required invariants that describe both the relative behavior of all three executions if they all execute the loop, *and* the relative behavior of any pair of executions if two executions reach the loop and one does not. This is therefore an example where some of the used specifications are relational but not all executions are relevant for them.

	P1				P2				P3			
	V	T_{VCG}	T_{SE}	T_{DC}	V	T_{VCG}	T_{SE}	T_{DC}	V	T_{VCG}	T_{SE}	T_{DC}
ArrayInt	✓	1.22	1.23	0.06	✓	1.65	21.11	0.06	✗	1.79	11.87	0.11
ArrayInt †	✓	1.25	1.52	0.11	✓	1.91	23.21	0.16	✓	3.43	23.43	0.11
CatBPos	✗	1.25	1.09	0.26	✗	1.50	15.39	5.85	✗	1.52	7.90	1.71
Chromosome	✓	1.18	0.91	0.11	✗	1.22	2.48	0.15	✗	1.31	1.08	0.22
Chromosome †	✓	1.25	1.20	0.06	✓	1.61	22.41	2.02	✓	1.39	9.48	0.28
CollItem	✗	1.19	0.43	0.06	✗	1.27	2.21	0.11	✗	1.27	1.85	0.16
CollItem †	✓	1.15	1.94	0.06	✓	1.22	26.89	0.16	✓	1.25	27.13	0.13
Contact	✓	1.38	5.91	0.11	✗	2.30	15.37	0.78	✗	1.92	2.41	1.34
ContainerV1	✗	1.15	0.30	0.06	✗	1.17	1.08	0.03	✗	1.25	0.88	0.06
ContainerV2	✗	1.15	0.31	0.06	✗	1.21	1.34	0.03	✗	1.22	1.02	0.06
Container †	✓	1.19	0.94	0.16	✓	1.31	4.17	2.32	✓	1.25	2.99	0.94
DataPoint	✗	1.30	0.78	0.21	✗	1.63	16.20	0.67	✗	1.61	38.80	0.42
FileItem	✓	1.14	0.63	0.03	✓	1.37	4.58	0.03	✗	1.23	0.72	0.11
FileItem †	✓	1.20	1.52	0.06	✓	1.34	11.29	0.06	✓	1.30	9.44	0.06
IsoSpriteV1	✗	1.16	0.33	0.06	✗	1.19	1.13	0.03	✗	1.21	1.01	0.06
IsoSpriteV2	✗	1.35	0.58	0.31	✗	2.13	32.94	1.22	✓	2.13	168.62	0.12
Match	✗	1.16	0.47	0.04	✓	1.14	5.33	0.03	✗	1.19	1.72	0.06
Match †	✓	1.17	0.96	0.03	✓	1.20	7.16	0.06	✓	1.22	7.29	0.06
NameComparator	✗	1.24	0.74	0.07	✓	1.77	17.17	0.11	✓	1.70	16.44	0.10
NameComparator †	✓	1.31	1.63	0.10	✓	1.66	24.10	0.11	✓	1.66	24.31	0.16
Node	✓	1.13	0.77	0.03	✓	1.16	5.13	0.03	✗	1.20	1.37	0.07
Node †	✓	1.13	0.77	0.03	✓	1.19	4.90	0.06	✓	1.21	5.01	0.06
NzbFile	✗	1.39	0.56	0.11	✓	1.96	121.60	0.21	✓	1.76	108.31	0.11
NzbFile †	✓	1.35	3.05	0.16	✓	2.39	71.57	0.21	✓	2.38	65.09	0.42
PokerHand	✓	1.56	5.83	0.26	✗	9.19	118.18	0.37	✗	8.67	120.81	0.99
PokerHand †	✓	1.62	6.94	0.27	✓	9.15	237.42	0.37	✓	9.09	233.72	0.66
Solution	✓	1.17	0.86	0.16	✓	1.22	4.29	0.37	✗	1.24	1.25	0.52
Solution †	✓	1.16	0.96	0.31	✓	1.29	4.84	0.97	✓	1.23	4.53	0.92
TextPosition	✓	1.24	1.84	0.01	✗	1.37	7.77	0.01	✗	1.43	6.87	0.01
TextPosition †	✓	1.28	1.99	0.11	✓	1.33	26.29	0.37	✓	1.42	24.98	0.16
Time	✗	1.15	0.43	0.11	✓	1.34	6.23	0.32	✓	1.21	3.80	0.02
Time †	✓	1.11	0.45	0.06	✓	1.28	3.92	0.26	✓	1.19	1.90	0.16
Word	✗	1.30	0.61	0.23	✗	1.60	3.73	4.43	✓	1.65	34.68	0.03
Word †	✓	1.26	1.54	0.11	✓	1.65	23.93	0.17	✓	1.77	24.41	0.11

Table 2. Evaluated examples of Java comparators. We show the verification outcomes for all three hyperproperties for the original version(s) and the fixed versions (marked with †) of each example. Columns T_{VCG}, T_{SE}, and T_{DC} show the running times of the verifiers for the VCG backend, the SE backend, and Descartes, respectively, in seconds.

As a result, relational loop invariants would in these cases have been more verbose and arguably more complex. Additionally, we subjectively found it non-trivial to find the required relational loop invariants in some cases: Unlike in the case of secure information flow, where relational invariants translate straightforwardly to high-level concepts like some data being low, finding relational invariants for properties 2 and 3 actually required thinking in terms of multiple executions and was thus more complex. We proved properties P13 and P14 on the modified examples also using a combination of relational of unary specifications for similar reasons.

Nevertheless, we benefited from the ability to use relational loop invariants in several cases. In particular, in examples where other comparators are called inside loop bodies, using functional specifications in the invariant is not possible without also requiring them for the called comparator, which, as pointed out before, is undesirable and not modular. In one example, a loop called another comparator under a condition and performed some simple computation by itself under a different condition; in this case we found it easiest to use a mixed specification that describes the behavior of the comparator in relational terms and the alternative functional behavior in unary terms.

7.4.2 Performance. Table 2 shows the timings measured to verify all original comparator examples using Viper's VCG backend, its SE backend, and the Descartes tool by Sousa and Dillig. The times for the modified examples are shown in Table 3. All times were measured under the same

	P13				P14			
	T_{VCG}	T_{SE}	T_{DC}	T_{SY}	T_{VCG}	T_{SE}	T_{DC}	T_{SY}
ArrayInt-pick3-simple †	3.49	18.08	2.16	0.82	3.48	22.60	TO	292.90
ArrayInt-pick3 †	3.53	51.95	2.03	1.01	3.64	58.95	TO	210.71
Chromosome-pick3-simple †	3.52	55.65	1.12	0.68	3.47	78.28	TO	177.69
Chromosome-pick3 †	3.66	28.12	3.15	1.97	3.63	34.02	TO	980.77
PokerHand-pick3-part1 †	4.61	202.41	6.96	2.86	5.19	188.74	TO	1548.27
PokerHand-pick3-part2 †	7.14	1082.14	11.90	5.18	9.73	1125.56	TO	-
PokerHand-pick3 †	12.05	TO	20.63	8.63	18.07	TO	-	-
Solution-pick3 †	4.79	TO	84.92	19.01	5.28	TO	TO	TO

Table 3. Evaluated modified examples. All examples fulfill both properties and verify successfully in every configuration for which a time is shown. Columns T_{VCG}, T_{SE}, T_{DC}, and T_{SY} show the running times of the verifiers for the VCG backend, the SE backend, Descartes, and Synonym, respectively, in seconds. "TO" indicates that the test times out after one hour, "-" means that the tool reports that it could not infer sufficient invariants.

conditions as the timings for secure information flow. In particular, our timings for Descartes and Synonym were measured using the version of the tools currently on Github; in virtually all cases, they are similar to the timings reported by Sousa and Dillig [39] and Pick et al. [31], respectively.

We first observe that our implementation generally achieves good performance when using the VCG backend. The average verification time for the original comparators is under two seconds for all three properties. With maximal verification times of under ten seconds even for the most complex comparator examples and 18 seconds for the most complex modified example, we conclude that our technique can be used to verify hyperproperties of realistic code in very reasonable time.

When using our tool with Viper's SE backend, we observe considerably worse performance than with VCG. While the average verification times for the original comparators are comparable for the first property, the SE backend performs much worse for the two 3-safety hyperproperties, particularly for the examples that have deeply nested conditionals, where the worst case is an example taking almost four minutes to verify. For the larger modified examples, performance becomes considerably worse; two of the examples do not verify within one hour. This confirms our earlier suspicion that the performance with SE (which considers each branch separately) is impaired by the large amount of branching in modular product programs: Since our product transformation introduces many branches into the program (more for $k = 3$ than for $k = 2$), the SE backend has to consider a large number of possible paths through the program. While the SE backend struggles with this, the VCG backend is able to exploit the fact that many of those branches are on the same or related conditions, and the actual number of feasible paths through the program is therefore much smaller than it seems.

We can now compare the performance of our implementation to that of Descartes and Synonym. While this leads to some interesting conclusions, we stress that the feature sets supported by the tools are quite different. Viper is a mature, general purpose verification tool with built-in support for mutable heap datastructures and several basic datatypes; it performs several additional tasks like type checking, well-definedness checking of specifications, and proving memory safety, all of which are not performed by Descartes and Synonym. The latter, on the other hand, are specialized prototype implementations that do not support heap mutation or data types beyond integers, but, unlike Viper, are able to automatically infer loop invariants via a guess-and-check-approach.

Compared to Descartes and Synonym, our implementation using the VCG backend is generally slower for small examples (the original comparators) but faster for the more complex modified examples; in fact, all verifiers except for VCG time out on at least one example. This is likely at least partly due to the aforementioned differences between the tools; while Viper has a higher constant

overhead for each program due to startup time and additional checks, the additional work needed by Descartes and Synonym for guessing and checking loop invariants becomes more of a factor for more complex examples. SE has a lower constant overhead than VCG[3] and is more competitive than VCG on small examples (though still slower than Descartes and Synonym) but scales much worse in most cases and is thus the slowest tool for most (though not all) of the more complex examples.

We conclude that while our technique creates programs that lead to bad performance with some verification techniques (SE), it can be combined with other standard verification techniques (in particular, VCG) to verify arbitrary hyperproperties of real-world code in reasonable time. Compared to existing more automated tools, our technique delivers better performance on very complex examples when used with VCG. In particular, since our approach allows modular (and therefore independent) verification of different procedures, we expect it to scale to large programs.

One advantage of using a program transformation approach for verifying hyperproperties is that one can freely choose which tool to use to reason about the resulting program, based on the desired level of automation and required language features. In contrast, adding mutable heap support to Descartes would require changes to the underlying logic itself.

8 RELATED WORK

The notion of k-safety hyperproperties was originally introduced by Clarkson and Schneider [12]. Here, we focus on statically proving hyperproperties for imperative and object-oriented programs; much more work exists for testing or monitoring hyperproperties like secure information flow at runtime, or for reasoning about hyperproperties in different programming paradigms.

Relational logics such as Relational Hoare Logic [10], Relational Separation Logic [41] and others [1, 9] allow reasoning directly about relational properties of two different program executions. Unlike our approach, they usually allow reasoning about the executions of two *different* programs; as a result, they do not give special support for two executions of the same program calling the same procedure with a relational specification. Recently, Banerjee et al. [4] introduced biprograms, which allow explicitly expressing alignment between executions and using relational specifications to reason about aligned calls; however, this approach requires that procedures with relational specifications are always called by both executions, which is for instance not the case if a call occurs under a high guard in secure information flow verification. We handle such cases by interpreting relational specifications as trivially true; one can then still resort to functional specifications to complete the proof. Their work also does not allow mixed specifications, which are easily supported in our product programs. Relational program logics are generally difficult to automate. Recent work by Sousa and Dillig [39] presents a logic that can be applied automatically by an algorithm that implicitly constructs different product programs that align *some* identical statements, but does not fully support relational specifications. Pick et al. [31] use similar ideas but exploit symmetries in the verification problem and further align the execution of loops for improved performance. Both approaches require dedicated tool support, whereas our modular product programs can be verified using off-the-shelf tools.

The approach of reducing hyperproperties to ordinary trace properties was introduced by self-composition [8]. While self-composition is theoretically complete, it does not allow modular reasoning with relational specifications. The resulting problem of having to fully specify program behavior was pointed out by Terauchi and Aiken [40]; since then, there have been a number of different attempts to solve this problem by allowing (parts of) programs to execute in lock-step.

[3]In the VCG backend, each verification task requires starting a .NET runtime for running the Boogie [5] verifier in the background, which leads to some constant amount of overhead.

Terauchi and Aiken [40] did this for secure information flow by relying on information from a type system; other similar approaches exist [30].

Product programs [6, 7] allow different interleavings of program executions. The initial product program approach [6] would in principle allow the use of relational specifications for procedure calls, but only under the restriction that both program executions always follow the same control flow. The generalized approach [7] allows combining different programs and arbitrary numbers of executions. This product construction is non-deterministic and usually interactive; in some (but not all) cases, it allows avoiding duplicated calls and loops and thereby using relational specifications. However, the construction requires manual work by the programmer, and whether the product can be constructed in such a way that call is not duplicated depends on the used specification, meaning that the product construction and verification are intertwined and a new product has to be constructed when specifications change. In contrast, our product construction is fully deterministic and automatic, allows arbitrary control flows while still being able to use relational specifications for all loops and calls, and therefore avoids the issue of requiring full functional specifications.

Techniques for proving program equivalence, whose goal is to prove relational properties about pairs of programs, can be used to prove hyperproperties of a single program as well. In particular, Hawblitzel et al. [22] and Lahiri et al. [25] propose two different techniques for modularly proving and using *mutual summaries* that relate the behavior of two different procedures. Compared to our work, both techniques have the advantage of being able to relate arbitrary pairs of procedure calls. In the former approach [22], this is achieved by axiomatically assuming mutual summaries for any pair of calls and subsequently generating verification conditions that check the mutual summary of one pair of procedures assuming these axioms for all calls. In contrast to modular product programs, this approach does not allow checking relational *preconditions*, and it does not work with standard static analysis tools, since those typically cannot make use of the axioms that are used to assume relational postconditions. In the latter approach [25], mutual summaries are used via a product construction that stores the local and global state before and after each procedure call in auxiliary variables, sequentially composes the bodies of both different procedures, and subsequently assumes or asserts relational properties about the stored information for each pair of related calls. As a result, the technique is limited in the presence of non-termination (since no relational properties are checked if one of the programs does not terminate) and, for example, cannot support reasoning about termination channels. Importantly, both of the aforementioned approaches require that dependencies on global variables are statically known and finite (since their values need to be mentioned in axioms or stored for each call, respectively) and therefore do not easily extend to programs with dynamic heap allocation and to separation logics, unlike modular product programs. Additionally, both approaches rely on eliminating loops by transforming them to tail recursion. This is always possible but can require additional specifications from the user to carry information about local variables into procedure calls inserted by the transformation. Felsing et al. [20] present a weakest precondition calculus for showing (conditional) program equivalence as well as an encoding into Horn clauses for automation. Their approach to loop verification is based on similar ideas as our loop encoding and they also support mutual function summaries. In contrast to our approach, Felsing et al. consider only terminating programs in a language without arrays or a mutable heap, and their approach requires dedicated tool support. Similar ideas for loop fusion have been used for other purposes, e.g., to simplify loop invariants in a single program and for program optimization [23].

The equivalence checking approach by Hawblitzel et al. [22] also supports checking *relative termination* under some relational precondition, which is similar but not identical to the absence of termination channels; it asserts that if one execution terminates, the other *can* terminate as well. The presented approach requires fewer annotations than ours, in particular, it does not require

ranking functions, but as a result it is less precise and does not allow any loops or procedure calls under high conditions. Similarly, Elenbogen et al. [19] provide an algorithm for proving *mutual termination*, which can be used for proving the absence of termination channels, via a product construction. It, too, is less precise than our approach but requires less user input.

Reducing control flow to straight-line code with conditional statements, also called *predicated execution*, is sometimes performed as a performance optimization to avoid branching. The principle is also employed in modern GPUs, which execute multiple threads at once in a SIMD fashion, and implement diverging control flow by deactivating some threads while a branch they do not take is executed. Betts et al. [11] exploit this and present an encoding similar to our product construction to prove trace properties of (concurrently executed) GPU kernels, but do not explore its application to hyperproperties of sequential programs. Collingbourne et al. [13] build on this work to present an encoding for GPU kernels with arbitrary reducible control flow graphs; the main ideas behind this extended encoding could likely also be used to create modular product programs based on control flow graphs.

Considerable work has been invested into proving specific hyperproperties like secure information flow. One popular approach is the use of type systems [38]; while those are modular and offer good performance, they overapproximate possible program behaviors and are therefore less precise than approaches using logics. In particular, they require labeling any single value as either high or low, and do not allow distinctions like the one we made for the example in Fig. 1, where only the first bits of a sequence of integers were low. In addition, type systems typically struggle to prevent information leaks via side channels like termination or program aborts. There have been attempts to create type systems that handle some of these limitations (e.g. [17]).

Static analyses [2, 21] enable fully-automatic reasoning. They are typically not modular and, similarly to type systems, need to abstract semantic information, which can lead to false positives. They strike a trade-off different from our solution, which requires specifications, but enables precise, modular reasoning.

A number of logic-based approaches to proving specific hyperproperties exist. As an example, Darvas et al. use dynamic logic for proving non-interference [15]; this approach offers some automation, but requires user interaction for most realistic programs. Leino et al. [26] verify determinism up to equivalence using self-composition, which suffers from the drawbacks explained above. Prabawa et al. [32] present a logic for proving secure information flow. While their approach, like ours, enables modular verification, it can only track secrecy on a more coarse-grained level, does not take into account semantic information, and will therefore produce more false alarms.

The approach of declassifying data via special statements or expressions has been introduced by Sabelfeld and Myers [34]. An extended version of JML by Scheben and Schmitt [36] supports declassification expressions as parts of method contracts that can be verified using self-composition. Other kinds of declassification have been studied extensively; Sabelfeld and Sands [35] provide a good overview. Li and Zdancewic [27] introduce downgrading policies that describe which information can be declassified and, similar to our approach, can do so for arbitrary expressions. Costanzo and Shao [14] define a similar system that allows users to define program preconditions to describe which parts of the secret data may be released. Our approach allows for similar specifications that describe that some aspects of some data are low (and everything else is high by default).

9 CONCLUSION AND FUTURE WORK

We have presented modular product programs, a novel form of product programs that enable modular reasoning about k-safety hyperproperties using relational specifications with off-the-shelf verifiers. We have proved our approach sound and complete for arbitrary values of k. We showed that modular products are expressive enough to handle advanced aspects of secure information

flow verification. They can prove the absence of termination and timing side channels and encode declassification. Our implementation shows that our technique works in practice on a number of challenging examples from both the literature and real user code, and exhibits good performance even without optimizations.

For future work, we plan to infer hyperproperties by using standard program analysis techniques on the products. We also plan to generalize our technique to prove probabilistic secure information flow for concurrent program by combining our encoding with ideas from concurrent separation logic. Finally, we plan to optimize our encoding to further improve performance.

ACKNOWLEDGMENTS

We would like to thank Toby Murray and David Naumann for helpful discussions. We are grateful to the TOPLAS and ESOP reviewers for their valuable comments. We also gratefully acknowledge support from the Zurich Information Security and Privacy Center (ZISC).

REFERENCES

[1] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A relational logic for higher-order programs. *PACMPL* 1, ICFP (2017), 21:1–21:29.

[2] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* 362–375.

[3] Anindya Banerjee and David A. Naumann. 2002. Secure Information Flow and Pointer Confinement in a Java-like Language. In *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24-26 June 2002, Cape Breton, Nova Scotia, Canada.* 253.

[4] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. 2016. Relational Logic with Framing and Hypotheses. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India.* 11:1–11:16.

[5] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (Lecture Notes in Computer Science)*, Vol. 4111. Springer, 364–387.

[6] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings.* 200–214.

[7] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification. In *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings.* 29–43.

[8] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6 (2011), 1207–1252. https://doi.org/10.1017/S0960129511000193

[9] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009.* 90–101.

[10] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004.* 14–25.

[11] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012.* 113–132. https://doi.org/10.1145/2384616.2384625

[12] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[13] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. 2013. Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings.* 270–289. https://doi.org/10.1007/978-3-642-37036-6_16

[14] David Costanzo and Zhong Shao. 2014. A Separation Logic for Enforcing Declarative Information Flow Control Policies. In *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 179–198.

[15] Ádám Darvas, Reiner Hähnle, and David Sands. 2005. A Theorem Proving Approach to Analysis of Secure Information Flow. In *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*. 193–209.

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[17] Zhenyue Deng and Geoffrey Smith. 2004. Lenient Array Operations for Practical Secure Information Flow. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. 115.

[18] Marco Eilers, Peter Müller, and Samuel Hitz. 2018. Modular Product Programs. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 502–529.

[19] Dima Elenbogen, Shmuel Katz, and Ofer Strichman. 2015. Proving mutual termination. *Formal Methods in System Design* 47, 2 (2015), 204–229. https://doi.org/10.1007/s10703-015-0234-3

[20] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 349–360. https://doi.org/10.1145/2642937.2642987

[21] Dennis Giffhorn and Gregor Snelting. 2015. A new algorithm for low-deterministic security. *Int. J. Inf. Sec.* 14, 3 (2015), 263–287.

[22] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 2013. Towards Modularly Comparing Programs Using Automated Theorem Provers. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*. 282–299. https://doi.org/10.1007/978-3-642-38574-2_20

[23] Akifumi Imanishi, Kohei Suenaga, and Atsushi Igarashi. 2018. A guess-and-assume approach to loop fusion for program verification. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Los Angeles, CA, USA, January 8-9, 2018*. 2–14.

[24] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. 2015. A Hybrid Approach for Proving Noninterference of Java Programs. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. 305–319.

[25] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 345–355. https://doi.org/10.1145/2491411.2491452

[26] K. Rustan M. Leino and Peter Müller. 2008. Verification of Equivalent-Results Methods. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 307–321.

[27] Peng Li and Steve Zdancewic. 2005. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. 158–170.

[28] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. 405–425.

[29] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. 41–62.

[30] David A. Naumann. 2006. From Coupling Relations to Mated Invariants for Checking Information Flow. In *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings*. 279–296.

[31] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. 2018. Exploiting Synchrony and Symmetry in Relational Verification. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. 164–182.

[32] Adi Prabawa, Mahmudul Faisal Al Ameen, Benedict Lee, and Wei-Ngan Chin. 2018. A Logical System for Modular Information Flow Verification. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*. 430–451.

[33] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74.

[34] Andrei Sabelfeld and Andrew C. Myers. 2003. A Model for Delimited Information Release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*. 174–191.

[35] Andrei Sabelfeld and David Sands. 2005. Dimensions and Principles of Declassification. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. 255–269.

[36] Christoph Scheben and Peter H. Schmitt. 2011. Verification of Information Flow Properties of Java Programs without Approximations. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*. 232–249. https://doi.org/10.1007/978-3-642-31762-0_15

[37] Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 2:1–2:58.

[38] Geoffrey Smith. 2007. Principles of Secure Information Flow Analysis. In *Malware Detection*. 291–307.

[39] Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 57–69.

[40] Tachio Terauchi and Alexander Aiken. 2005. Secure Information Flow as a Safety Problem. In *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. 352–367.

[41] Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 308–334.

Received February 2007; revised March 2009; accepted June 2009