Formal Methods in Computer-Aided Design 2020

Automating Modular Verification of Secure Information Flow

Lauren Pick*, Grigory Fedyukovich†, Aarti Gupta*
*Princeton University, Princeton, NJ, USA
†Florida State University, Tallahassee, FL, USA

***Abstract*—Verifying secure information flow by reducing it to safety verification is a popular approach, based on constructing product programs or self-compositions of given programs. However, most such existing efforts are non-modular, i.e., they do not infer relational specifications for procedures in interprocedural programs. Such relational specifications can help to verify security properties in a modular fashion, e.g., for verifying clients of library APIs. They also provide security contracts at procedure boundaries to aid code understanding and maintenance. There has been recent interest in constructing *modular* product programs, but where users are required to provide procedure summaries and related annotations. In this work, we propose to automatically *infer* relational specifications for procedures in modular product programs. Our approach uses syntax-guided synthesis techniques and grammar templates that target verification of secure information flow properties. This enables automation of modular verification for such properties, thereby reducing the annotation burden. We have implemented our techniques on top of a solver for constrained Horn clauses (CHC). Our evaluation demonstrates that our tool is capable of inferring adequate relational specifications for procedures without requiring annotations. Furthermore, it outperforms an existing state-of-the-art hyperproperty verifier and a modular CHC-based verifier on benchmarks with loops or recursion.**

***Index Terms*—Formal verification, information security, model checking**

I. Introduction

The problem of verifying secure information flow is to check that a program does not leak private inputs to public outputs. To solve this problem, one can verify non-interference [1]: for any two runs of a program with the same public inputs but possibly different private inputs, the public outputs of the program are equal. This property is an instance of a *hyperproperty*, i.e., a relational property involving more than one execution of the same program. In practice, non-interference is often too strong a property to enforce. For example, a password recognizer would have its public output be influenced by whether or not the user-provided private input is the correct password. A common approach is to allow values that need to be leaked to be *declassified* [2].

Barthe *et al.* proposed verifying secure information flow by reducing it to safety verification on a product or self-composed program [3]. Despite advancements in automated program verifiers, the ability to perform successful safety verification in practice can depend critically on how the product program is constructed. Construction of product programs has thus been a focus in subsequent efforts [4]–[14]. These efforts encompass various syntactic and semantic transformations, heuristics, and use of reinforcement learning for constructing suitable product programs. Some relational property verifiers avoid explicitly constructing product programs altogether [15]–[19].

A. Motivation

In this paper, we address a related but distinct limitation of existing efforts based on reduction to safety. Most such techniques are non-modular, i.e., they neither leverage nor infer *relational specifications for procedures* in interprocedural programs. In general, modular verification offers significant benefits over non-modular techniques – it is inherently more scalable, can provide procedure interface contracts (not only verification results), and can improve code understanding and maintenance. For example, relational specifications of procedures can provide security contracts for library APIs, such as in the S2N implementation of the TLS protocol [20].

A few existing approaches do leverage relational specifications of procedures, but they either restrict both copies of the program to always follow the same control flow [6] or are not automated [8], [21]. In particular, the work by Eilers *et al.* [21] proposes a *modular product program (MPP)* construction, which is suitable for performing *modular* relational program verification. Intuitively, this enables reduction to safety on a per-procedure basis without constructing a monolithic product program. In their implementation, Viper back-end verifiers checked secure-information-flow properties on benchmarks, but each procedure required user-provided relational invariants and related annotations rather than relying on tools to derive them automatically. Placing this annotation burden on users becomes a barrier to automated verification.

In general, deriving sufficient relational invariants for procedures is a challenging problem, and existing off-the-shelf safety verifiers [22]–[28] may not be able to infer them. As we will show, for verifying secure information flow, such invariants often have a special form that is unlikely to be produced by standard interpolation and existing heuristics in these verifiers. For example, our experimental results (§VII) show that Spacer often fails to infer invariants needed to verify information flow in programs with recursion.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656466. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was supported in part by the National Science Foundation award FMitF 1837030.

https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_23

This article is licensed under a Creative Commons Attribution 4.0 International License

B. Overview of Proposed Approach

In this work, we propose to use Syntax-Guided Synthesis (SyGuS) [29] to automatically infer useful relational specifications about information flow in procedures. The structure in information-flow specifications makes them suitable targets for grammar-based enumerative search and synthesis. We have chosen to work with MPPs because they enable *modular* relational verification *and* they allow leveraging existing techniques for construction of suitable product programs within each procedure. We represent an MPP as a set of constrained Horn clauses (CHCs), and our approach automatically infers relational specifications that are sufficient for verifying the program with respect to given security properties. If there are no given security properties, our approach can still infer relational specifications for procedures that are useful for code understanding or subsequent verification.

Our SyGuS-based approach is based on an enumerative search using grammars extracted from program syntax. Enumerate-and-check approaches have been shown to be effective for synthesis of quantifier-free invariants [27], [30]–[32] and more recently quantified invariants for CHCs handling arrays [33]. We show that such an approach is also effective for information-flow properties.

We propose three templates to generate grammars for invariant synthesis: one that expresses quantifier-free information-flow properties, and two that express quantified properties, which are often difficult to handle by existing automated verifiers. Of the latter two, one infers quantified information-flow properties over *arrays*, and the other infers specifications involving the *context* in which a procedure is called, making this template well-suited for inferring properties where declassification has occurred prior to the procedure being called, since the declassified values will be low-security in the callee.

We have implemented our approach in a tool called FLOWER. An evaluation on available benchmark examples demonstrates that it is effective in inferring useful relational specifications of procedures, without requiring any user-provided annotations. We also compared FLOWER with other state-of-the-art tools: a hyperproperty verifier (DESCARTES [15]) and a modular CHC-based verifier (SPACER [25]). Our experiments demonstrate that our tool generally outperforms them, especially on benchmark examples that contain loops or recursion.

In summary, this paper makes the following contributions:

- We propose a SyGuS-based approach for inference of quantifier-free relational specifications for procedures for verifying secure information flow (§IV, §V).
- We propose grammar templates for inferring such specifications with *quantifiers*, which are challenging for existing verifiers (§VI).
- We have implemented our approach in a prototype tool FLOWER and present an evaluation that shows its effectiveness on several benchmarks[1] (§VII).

[1]Our tool and translated benchmarks will be made publicly available.

```
main (int[] a, int n) {          main (b1, b2, a1, a2, n1, n2) {
  a := init(a, 0);                 a1, a2 := init(b1, b2, a1,
  outputter(a, 0);                                a2, 0, 0);
  return n;                        assert(outputter(b1, b2, a1,
}                                                   a2, 0, 0));
                                   return n1, n2;
                                 }

init(a, i) {                     init(b1, b2, a1, a2, i1, i2) {
  if (i ≥ 64) return a;            if (¬(b1 ∧ i1 < 64 ∨
  declassify(a[i] = 0);                      b2 ∧ i2 < 64))
  return init(a, i + 1);             return a1, a2;
}                                  l1 := b1 ∧ i1 < 64
                                   l2 := b2 ∧ i2 < 64
                                   assume (l1 ∧ l2 ⇒
                                     (a1[i1] = 0) = (a2[i2] = 0));
                                   return
                                     init(l1, l2, a1,
                                          a2, i1 + 1, i2 + 1);
                                 }

outputter(a, i) {                outputter(b1, b2, a1, a2, i1, i2) {
  if (i ≥ 64) return;              if (¬(b1 ∧ i1 < 64 ∨
  if (a[i] = 0) {                        b2 ∧ i2 < 64))
    assert(low(a[i]));               return true;
    print(a[i]);                   l1 := b1 ∧ i1 < 64;
  }                                l2 := b2 ∧ i2 < 64;
  outputter(a, i + 1);             t1 := l1 ∧ a1[i1] = 0;
}                                  t2 := l2 ∧ a2[i2] = 0;
                                   print(t1, t2, a1[i1], a2[i2]);
                                   ok := t1 ∧ t2 ⇒ a1[i1] = a2[i2];
                                   ok := ok ∧
                                         outputter(b1, b2, a1, a2,
                                                   i1 + 1, i2 + 1);
                                   return ok;
                                 }
```

Fig. 1: Example (**left**: original (P), **right**: modular product program (MP)).

To the best of our knowledge, among methods based on product program construction, our work is the first to *automate modular relational verification* for secure information flow.

II. MOTIVATING EXAMPLE

We demonstrate our approach on an example program P shown in Fig. 1, inspired by a related work [34]. In main, a call to init makes initial assumptions about the array a: for each of the first 64 values in the array, the information about whether or not the value is 0 is declassified recursively. Then, these 64 entries are printed out by the recursive procedure outputter, which contains an assertion that checks that each of the values printed out is public (i.e., low-security) output. Finally, main returns its second argument.

The security primitives used in this example are low, which is a predicate that holds iff its argument is a low-security variable, and declassify, which has the effect of making the value of its argument low-security after the point where declassify is invoked. Without assumptions stating otherwise (i.e., either assume statements that indicate that a value is low-security by using the low primitive or declassify statements), we assume that all inputs are high-security. In the example, after init is called and it declassifies each a[i] = 0 value for i < 64, then the information about whether or not any of the first 64 entries in a is 0 is considered to be public information. The outputter procedure prints out the value of values of a[i] for i < 64 only under the condition that a[i] = 0. This behavior leaks exactly only the declassified information, so the assertion is expected to hold for each call to `outputter`.

The modular product program `MP` for this example is shown in Fig. 1 (right). Note that for *each* variable in `P` (even if irrelevant to verification), `MP` has two copies reflecting the two executions of the program, e.g., `n` is translated to `n1` and `n2`. For each procedure in `P`, two Boolean *activation variables* `b1` and `b2` are added as inputs to the corresponding procedure in `MP`, where they respectively indicate whether the control flow in the corresponding copy of the program has reached the callsite. The idea is that *relational specifications* for procedures hold when both copies of the program have reached the same callsite, i.e., when both activation variables of the callee are true. As a result, all the relational specifications that we infer are implications in which the antecedent contains at least `b1` and `b2` as conjuncts.

The translation to `MP` also shows how the information-flow operation `declassify` is encoded as an assumption, and how the information-flow specification `low(a[i])` is translated into a relational property $\mathtt{t1} \wedge \mathtt{t2} \Rightarrow \mathtt{a1}[\mathtt{i1}] = \mathtt{a2}[\mathtt{i2}]$ in `MP`, where `t1` and `t2` were the activation variables under which the specification `low(a[i])` occurred. Finally, note that the assertion in `outputter` has been hoisted to `main` in `MP`, with the return value of `outputter` being *true* if and only if no assertion failed.

We infer quantifier-free information-flow properties[2] for each procedure. For example, we can infer that for `main` of `MP`, the property $\mathtt{b1} \wedge \mathtt{b2} \wedge \mathtt{n1} = \mathtt{n2} \Rightarrow \mathit{res_1} = \mathit{res_2}$ holds, where $\mathit{res_1}$ and $\mathit{res_2}$ represent the return values of `main`. This property says that the output of `main` depends only on its second argument, and it does not rely on any information about whether the second argument or output of `main` is public or private, nor does it express any such information.

We also infer quantified invariants, e.g., $\phi(\mathtt{i1})$:

$$\forall j_1, j_2 . \mathtt{i1} \leq j_1 \leq 64 \wedge j_1 = j_2 \Rightarrow (\mathtt{a1}[j_1] = 0) = (\mathtt{a2}[j_2] = 0)$$

We can then instantiate this property for the call to `init` in `main` to determine that $\phi(0)$ is true when the call to `outputter` is made. However, we cannot yet verify the program because at this point we have not inferred sufficient properties for `outputter`.

Finally, we use the context in which `outputter` is called to influence the guesses that we make for the antecedent in its relational specification. Then we infer the following property for `outputter`, where *res* is the return value of `outputter`: $\mathtt{b1} \wedge \mathtt{b2} \wedge \phi(0) \wedge \mathtt{i1} = \mathtt{i2} \wedge 0 \leq \mathtt{i1} \Rightarrow \mathit{res}$. Note that this property contains quantifiers because $\phi(0)$ does. This property enables us to verify that the assertion for the program holds, leading to a successful conclusion.

III. Background and Notation

Here we describe the background on modular product programs and their modeling as CHCs, secure information flow, and relational invariants.

[2]We use properties interchangeably with procedure specifications.

```
proc(cond, x) {
  if (cond)
    x = x + 1;
  else
    x = 0;
  return x;
}
```

```
proc(b1, b2, cond1, cond2, x1, x2) {
  t1 = b1 ∧ cond1; f1 = b1 ∧ ¬ cond1;
  t2 = b2 ∧ cond2; f1 = b2 ∧ ¬ cond2;
  if (t1) x1 = x1 + 1; if (f1) x1 = 0;
  if (t2) x2 = x2 + 1; if (f2) x2 = 0;
  return x1, x2;
}
```

Fig. 2: Original (**left**) and modular product (**right**) programs.

A. Modular Product Programs

A k-hyperproperty expresses a property over k runs of the same program. Product programs convert k-hyperproperties into safety properties by creating k renamed versions of all the original variables. In contrast to ordinary product programs, *modular product programs (MPP)* avoid duplicating control structures such as procedure calls by introducing Boolean *activation variables* that indicate whether each program copy has reached a certain execution point [21]. The current activation variable for copy i is *true* if and only if copy i is currently at that location. While the principles of construction of a modular product program are defined in [21], we illustrate it with the following example.

Example 1. Consider the procedure in Fig. 2, for which the activation variables are initially `b1` and `b2`. The activation variables inside the then-branch (resp., else-branch) are `t1` and `t2` (resp., `f1` and `f2`). Each update to variable `x1` (resp., `x2`) is guarded by a condition so that the update is made only when the corresponding current activation variable for the first (resp., second) copy of the program is true. Note that any call to `proc` will also be guarded by a condition that at least one of `b1` or `b2` is *true*. If this doesn't hold, then neither procedure copy has reached the program point at which the procedure is called, so the call should not be made.

For a modular product program with k copies, we define partial functions *idx* and *getIdx* for conveniently handling expressions with renamed copies of variables. For any expression e, $\mathit{getIdx}(e) = i$ iff e represents an expression only over variables from the i^{th} copy; and for any expression e such that $\mathit{getIdx}(e)$ is defined: $\mathit{getIdx}(\mathit{idx}(e, i)) = i$. For example, $\mathit{idx}(\mathtt{b1} \wedge \mathtt{i1} < 64, 2) = \mathtt{b2} \wedge \mathtt{i2} < 64$. We also use *idx* to denote the lifting of *idx* to sets of expressions.

B. Secure Information Flow

We use a standard reduction [3] of a (termination-insensitive) secure-information-flow property to a 2-hyperproperty called *non-interference* [1], which ensures that private inputs do not impact public outputs. For a procedure `f`, this is formalized as follows:

$$\forall \bar{li}, \bar{lo}, \bar{hi}, \bar{ho}, \bar{li}', \bar{lo}', \bar{hi}', \bar{ho}' .$$
$$\bar{lo}, \bar{ho} = \mathtt{f}(\bar{li}, \bar{hi}) \wedge \bar{lo}', \bar{ho}' = \mathtt{f}(\bar{li}', \bar{hi}') \wedge \bar{li} = \bar{li}' \Rightarrow \bar{lo} = \bar{lo}'$$

Variables $\bar{li}$ and $\bar{li}'$ represent public inputs to `f` and $\bar{lo}$ and $\bar{lo}'$ represent public outputs. Variables $\bar{hi}$ and $\bar{hi}'$ represent private input variables to `f` and $\bar{ho}$ and $\bar{ho}'$ represent private outputs.

Non-interference states that for any two runs of `f`, one with inputs $\bar{li}, \bar{hi}$ and one with inputs $\bar{li}', \bar{hi}'$, if their public inputs are equal (i.e., $\bar{li} = \bar{li}'$), then their public outputs should be equal (i.e., $\bar{lo} = \bar{lo}'$) regardless of the private inputs' values.

In a modular product program, relational properties become properties over a single run and take the form of an implication whose antecedent implies the truth of all activation variables, e.g., non-interference takes the following shape:

$$\forall b_1, b_2, \bar{li}, \bar{lo}, \bar{hi}, \bar{ho}, \bar{li}', \bar{lo}', \bar{hi}', \bar{ho}' \ .$$
$$b_1 \wedge b_2 \wedge$$
$$\bar{lo}, \bar{ho}, \bar{lo}', \bar{ho}' = \texttt{f}(b_1, b_2, \bar{li}, \bar{li}', \bar{hi}, \bar{hi}') \wedge \bar{li} = \bar{li}' \Rightarrow \bar{lo} = \bar{lo}'$$

Requiring non-interference can be restrictive since programs may need some amount of leakage to exhibit the desired behavior. *Declassification* can allow secure-information-flow properties to be checked even for programs that leak some information about high-security variables. Declassification is encoded in modular product programs as an assumption that if both programs reach the same `declassify` statement (i.e., if both activation variables are true), then the value being declassified is equal across both copies of the program. Thus `declassify`(e) is encoded as `assume` $b_1 \wedge b_2 \Rightarrow e_1 = e_2$.

C. Constrained Horn Clauses for Modular Verification

The problem of modular program verification can be expressed as a system of CHCs [35].

Definition 1. A CHC is an implicitly universally-quantified implication, which is of the form *body* $\Rightarrow$ *head*. Let $\mathcal{R}$ be a set of uninterpreted predicates. The formula *head* may take either the form $R(\bar{y})$ for $R \in \mathcal{R}$ or else $\bot$. Implications in which *head* $=\bot$ are called *queries*. The formula *body* may take the form $\phi(\bar{x})$ or $\phi(\bar{x}) \wedge R_1(\bar{x}_1) \wedge \ldots \wedge R_n(\bar{x}_n)$, where each R_i is an uninterpreted predicate, and $\phi(\bar{x})$ is a fully interpreted formula over $\bar{x}$, which may contain all variables in each $\bar{x}_i$ and (if the head is of the form $R(\bar{y})$) all variables in $\bar{y}$.

A system of CHCs for a particular program can be generated by introducing an uninterpreted predicate per procedure (or a loop head) and encoding the semantics of each procedure (or a loop body) using these predicates. Fig. 3 gives an example encoding of program `MP` in Fig. 1 (right). Note that *print* is encoded as a nondeterministic procedure with no output.

Definition 2. A *solution* for a system of CHCs is a set of interpretations for predicates in $\mathcal{R}$ that makes all CHC implications valid.

Each interpretation can be viewed as a procedure summary and expresses an invariant for the procedure. In the case of the example program, the following interpretations are sufficient:

$$\begin{array}{ll} main \mapsto \lambda\bar{x}.\top & print \mapsto \lambda\bar{y}.\top \\ init \mapsto \lambda b_1, b_2, a_1, a_2, i_1, i_2.\phi(b_1, b_2, a_1, a_2, i_1, i_2) & \\ outputter \mapsto \lambda b_1, b_2, a_1, a_2, i_1, i_2, res.\ 0 \leq i_1 \Rightarrow res \wedge & \\ \qquad i_1 = i_2 \wedge b_1 \wedge b_2 \wedge \phi(b_1, b_2, a_1, a_2, i_1, i_2) & \end{array}$$

where $\phi(b_1, b_2, a_1, a_2, i_1, i_2) = \forall j_1, j_2. i_1 \leq j_1 < 64 \wedge i_1 = i_2 \wedge j_1 = j_2 \Rightarrow (a_1[j_1] = 0) = (a_2[j_2] = 0)$ and $\bar{x}$ and $\bar{y}$ are vectors of variables of lengths 6 and 4, respectively.

Definition 3. For a mapping M of uninterpreted predicates to interpretations, we say that the interpretations of M are *inductive* iff they satisfy all non-query CHCs.

In particular, an M that maps each n-ary predicate R to $\lambda x_1, \ldots, x_n.\top$ is inductive. For a formula F containing uninterpreted predicates, we let $M(F)$ be the result of replacing each predicate with its interpretation in M. For an inductive M, for each predicate R that represents a program procedure `r`, $M(R)$ is an *overapproximation* of the behavior of procedure `r`. For a given CHC C in the system of CHCs, where C is of the form $R_1(\bar{x}_1) \wedge \ldots \wedge R_n(\bar{x}_n) \wedge \phi(\bar{x}) \Rightarrow head$, an uninterpreted predicate R_i in its body can be *unfolded* in the CHC by replacing the occurrence of $R_i(\bar{x}_i)$ with $\mathit{fresh}(body_i[\bar{y}_i \mapsto \bar{x}_i], \bar{x}_i, \bar{x})$, where $body_i \Rightarrow R_i(\bar{y}_i)$ is another CHC in the system of CHCs, $body_i[\bar{y}_i \mapsto \bar{x}_i]$ is the simultaneous substitution of variables in $\bar{y}_i$ with variables in $\bar{x}_i$ in $body_i$, and $\mathit{fresh}(e, \bar{x}_i, \bar{x})$ is the result of replacing each variable in e that does not occur in $\bar{x}_i$ with a variable not in $\bar{x}$. We call the result of unfolding a predicate in a CHC C (possibly many times) an *unfolding* of C.

Example 2. An unfolding of *init* in the CHC for *main* in Fig. 3 is as follows: $\neg(b_1 \wedge k_1 < 64 \vee b_2 \wedge k_2 < 64) \wedge i_1 = i_2 = k_1 = k_2 = 0 \wedge ok \wedge outputter(b_1, b_2, a_1, a_2, i_1, i_2, ok) \Rightarrow main(b_1, b_2, a_1, a_2, n_1, n_2)$.

For a CHC C of the form $R_1(\bar{x}_1) \wedge \ldots \wedge R_n(\bar{x}_n) \wedge \phi(\bar{x}) \Rightarrow head$, we say that the following formula is the *context* (denoted $ctx(R_i, C)$) for the uninterpreted predicate application $R_i(\bar{x}_i)$: $\bigwedge_{1 \leq j \leq n}^{j \neq i} M(R_j)(\bar{x}_j) \wedge \phi(\bar{x})$. We naturally extend the mappings M from uninterpreted predicates to contexts. That is, for the formula above: $M(ctx(R_i, C)) = \bigwedge_{1 \leq j \leq n}^{j \neq i} M(R_j)(\bar{x}_j) \wedge \phi(\bar{x})$.

IV. SyGuS-based Summary Inference

This section describes our SyGuS-based algorithm for inferring procedure summaries of modular product programs. It takes CHCs as input and maintains a mapping M from uninterpreted predicates in the CHCs to inductive interpretations. The algorithm updates M as it runs and maintains the invariant that M's interpretations are inductive.

Our top-level procedure (Fig. 4) begins with an initial mapping M from each n-ary predicate $R \in \mathcal{R}$ to the coarsest interpretation possible. In pseudocode, we write CheckGuesses(G, M, R) to refer to an iterative procedure over all CHCs, where each application $R(\bar{x})$ of symbol R is replaced by formula $\lambda\bar{x}.M(R)(\bar{x}) \wedge makeGuess(G)(\bar{x})$, where G is a set of guessed interpretations for R based on our grammar templates and $makeGuess(G) = \lambda\bar{x}. \bigwedge\{g(x) \mid g \in G\}$.

The CHCs after the replacement are checked for validity using an SMT solver: if for some CHC C, the corresponding implication does not hold, then the current interpretation for R

$$
\begin{array}{ll}
\mathit{init}(b_1,b_2,a_1,a_2,k_1,k_2) \land k_1 = 0 \land k_2 = 0 \land & \\
\mathit{outputter}(b_1,b_2,a_1,a_2,i_1,i_2,ok) \land i_1 = 0 \land i_2 = 0 \land ok & \Rightarrow \mathit{main}(b_1,b_2,a_1,a_2,n_1,n_2) \\
\neg(b_1 \land i_1 < 64 \lor b_2 \land i_2 < 64) & \Rightarrow \mathit{init}(b_1,b_2,a_1,a_2,i_1,i_2) \\
(b_1 \land i_1 < 64 \lor b_2 \land i_2 < 64) \land l_1 = b_1 \land i_1 < 64 \land l_2 = b_2 \land i_2 < 64 \land & \\
(l_1 \land l_2 \Rightarrow (a_1[i_1] = 0) = (a_2[i_2] = 0)) \land \mathit{init}(l_1,l_2,a_1,a_2,i_1+1,i_2+1) & \Rightarrow \mathit{init}(b_1,b_2,a_1,a_2,i_1,i_2) \\
\neg(b_1 \land i_1 < 64 \lor b_2 \land i_2 < 64) & \Rightarrow \mathit{outputter}(b_1,b_2,a_1,a_2,i_1,i_2,\top) \\
(b_1 \land i_1 < 64 \lor b_2 \land i_2 < 64) \land l_1 = b_1 \land i_1 < 64 \land l_2 = b_2 \land i_2 < 64 \land & \\
t_1 = b_1 \land a_1[i_1] = 0 \land t_2 = b_2 \land a_2[i_2] = 0 \land \mathit{print}(t_1,t_2,a_1[i_1],a_2[i_2]) \land & \\
ok = t_1 \land t_2 \Rightarrow a_1[i_1] = a_2[i_2] \land \mathit{outputter}(l_1,l_2,a_1,a_2,i_1,i_2,\mathit{res}) & \Rightarrow \mathit{outputter}(b_1,b_2,a_1,a_2,i_1,i_2,ok \land \mathit{res}) \\
\top & \Rightarrow \mathit{print}(l_1,l_2,i_1,i_2) \\
\mathit{init}(b_1,b_2,a_1,a_2,k_1,k_2) \land k_1 = 0 \land k_2 = 0 \land & \\
\mathit{outputter}(b_1,b_2,a_1,a_2,i_1,i_2,ok) \land i_1 = 0 \land i_2 = 0 \land \neg ok & \Rightarrow \bot
\end{array}
$$

Fig. 3: CHC encoding of program M from Fig. 1 (right).

```
1: procedure InferSum(CHCs $\mathcal{C}$)
2:     for $R \in \mathcal{R}$ do $M(R) \leftarrow \lambda x_1, \ldots, x_n.\top$
3:     for $C \in \mathcal{C}$ where $C = body \Rightarrow R(\bar{x})$ do
4:         $G \leftarrow$ GetQFGuesses($C$) $\cup$ GetQuantifiedGuesses($C$)
5:         $M \leftarrow$ CheckGuesses($G$, $M$, $R$)
6:     while $M$ is not a solution for $\mathcal{C}$ do
7:         $Q \leftarrow$ GetUnsatisifiedQuery($\mathcal{C}$)
8:         $M \leftarrow$ Solve($Q$, $\mathcal{C}$, $M$)
9:     return $M$
```

Fig. 4: Top-level summary inference procedure.

```
1: procedure Solve($Q$, $\mathcal{C}$, $M$)
2:     unfoldings $\leftarrow \emptyset$
3:     if $M(body_Q)$ is unsatisfiable, then return $M$
4:     for $R$ in $Q$'s body do
5:         for $body \Rightarrow R(\bar{x}) \in \mathcal{C}$ do
6:             $G \leftarrow$ GetPDGuess($Q$, $body \Rightarrow R(\bar{x}), M$)
7:             $M' \leftarrow$ CheckGuesses($G$, $M$, $R$)
8:             if $M' \neq M$ then return $M'$
9:             unfoldings $\leftarrow$ unfoldings $\cup unfold(R, Q)$
10:    for $U \in$ unfoldings do
11:        $M' \leftarrow$ Solve($U$, $\mathcal{C}$, $M$)
12:        if $M' \neq M$ then return Solve($Q$, $\mathcal{C}$, $M'$)
```

Fig. 5: Inference procedure for property-directed guesses.

(which must appear in C) is weakened (using, e.g., the HOUDINI algorithm [22]), and the internal loop in CHECKGUESSES is repeated. Note that a new inductive mapping M' is returned as the result of CHECKGUESSES. Note also that M is already inductive whenever CHECKGUESSES is called, so it would be sufficient to weaken $M(R)(\bar{x}) \land makeGuess(G)(\bar{x})$ based on G, and CHECKGUESSES would return M in the worst case.

a) General quantifier-free and quantified guesses: For each CHC C, the algorithm generates initial guesses for an uninterpreted predicate in the head of C based on the templates specified later in Sec. V and VI-A.

After M has been updated based on these guesses, M's interpretations will have captured information-flow summaries for each procedure. If M is a solution for the system of CHCs, then these summaries may be sufficient for proving that the assertions of the program hold. Otherwise, the current procedure summaries are not strong enough for proving that the assertions hold, and the algorithm aims to learn additional property-directed summaries.

b) Property-directed guesses: Additional summaries are generated by our third template, which is described later in Sec. VI-B. Given a query CHC Q that contains an application of some $R \in \mathcal{R}$ to variables $\bar{y}$ in its body, a CHC of the form $body \Rightarrow R(\bar{x})$, and an inductive mapping M, each property-directed guess in $G = \text{GETPDGUESS}(Q, body \Rightarrow R(\bar{x}), M)$ is such that if it is used as an interpretation for R in the query CHC with all the other predicates using their interpretations in M, then the query CHC will be satisfied (i.e., the body of Q will be unsatisfiable).

For such a G, $makeGuess(G)(\bar{y})$ can be viewed as an interpolant separating $body[\bar{x} \mapsto \bar{y}]$ and $M(ctx(R_i, Q))$; to populate G, GETPDGUESS generates guesses that obey the syntactic requirements for such an interpolant and adds them to G only after checking that they maintain the invariant that $makeGuess(G)(\bar{y})$ is an interpolant. The query CHC should be the result of unfolding a currently-unsatisfied query from the original system of CHCs zero or more times. The way in which the algorithm explores unfoldings is shown in Fig. 5. Our algorithm starts with an unsatisfied query Q and tries to infer property-directed summaries for each predicate in Q's body. If no summary can be inferred, it unfolds each predicate in Q and repeats the process on each of these unfoldings, reconsidering Q with each resulting updated interpretation M'.

Let a query U be an unfolding of the query Q. After each update in an interpretation $M'(R)$ of each $R \in \mathcal{R}$ in U, the query Q is reconsidered with M'.

Lemma 1. *If a query U that leads to an interpretation update was obtained by unfolding $R(\bar{y})$ in Q using CHC $body \Rightarrow R(\bar{x})$, then there exists an interpolant I separating $M'(body[\bar{x} \mapsto \bar{y}])$ and $M'(ctx)$.*

Reconsidering Q with the mapping M' allows us to try to guess this interpolant. This finding of interpolants is similar to prior uses of interpolants [36], [37], but in our case, rather than using an interpolating solver, we rely on SyGuS to obtain quantified interpolants that cannot be generated by usual methods used in interpolating solvers.

Example 3. Consider a modification of the system of CHCs for our motivating example in Fig. 3 such that the CHC for $main$ and the query Q are as follows: $outputter(b_1, b_2, a_1, a_2, i_1, i_2, ok) \wedge i_1 = 0 \wedge i_2 = 0 \Rightarrow main(b_1, b_2, a_1, a_2, n_1, n_2, ok)$, $main(b_1, b_2, a_1, a_2, n_1, n_2, ok) \wedge \neg ok \Rightarrow \bot$.
Let U be the unfolding of $main$ in Q, and let M contain the following interpretations: $main \mapsto \lambda\bar{x}.\top, outputter \mapsto \lambda\bar{z}.\top, print \mapsto \lambda\bar{y}.\top, init \mapsto \lambda b_1, b_2, a_1, a_2, i_1, i_2.\phi(b_1, b_2, a_1, a_2, i_1, i_2)$.
The result of unfolding of $outputter$ in U allows us to update the summary (using the successfully checked guesses) of $outputter$ to the following:

$$\lambda b_1, b_2, a_1, a_2, i_1, i_2, res.$$
$$b_1 \wedge b_2 \wedge \phi(b_1, b_2, a_1, a_2, i_1, i_2) \wedge i_1 = i_2 \wedge 0 \leq i_1 \Rightarrow res$$

Note that the mapping M' containing this updated interpretation for $outputter$ is such that the following implication holds:

$$M'(outputter(b_1, b_2, a_1, a_2, i_1, i_2, ok) \wedge i_1 = 0 \wedge i_2 = 0) \Rightarrow ok$$

The antecedent of this implication is the interpretation of the body of the CHC for $main$, and the consequent is the negation of $main$'s context in Q. We can thus look for an interpolant that separates the body of $main$ and its context in Q.

Different orders in exploring unfoldings may result in learning different summaries. However, regardless of the order of unfoldings, the summaries discovered constitute a solution for the system of CHCs.

Note that if our templates cannot guess the required invariants, our top-level algorithm may not terminate, either because the second top-level loop may never terminate or because the recursive calls in the algorithm in Fig. 5 may never return. Our algorithm can be terminated early by the user and still return the properties discovered so far, which may be useful for code understanding and can provide hints to the user about manual annotations that may be required. In our experiments (Sect. VII), we did not need any manual annotations in the benchmark examples.

The following theorem implies that if INFERSUM returns a solution for a system of CHCs, the assertions in the original program that are captured by the query CHCs hold.

Theorem 1. INFERSUM *always returns an inductive map* M.

Proof. INFERSUM begins with M being the inductive map that maps each n-ary predicate R to $\lambda x_1, \ldots, x_n.\top$. M can be updated only by assigning it to the result of calls to CHECKGUESS, which always returns an inductive map. It follows that M is inductive when returned by INFERSUM. □

Finally, we note that our proposed SyGuS approach is not inherently limited to verifying secure information flow or to two copies of a program ($k = 2$). It can be adapted to verify k-hyperproperties for $k > 2$ by extending the basic grammar (shown later in Fig. 6) to cover target properties. Furthermore, our ideas on property-directed guesses are not specific to information flow and can apply to other properties.

$$\begin{aligned}
guess &::= \lambda\bar{x}.lhs \Rightarrow rhs \\
lhs &::= b_1 \wedge b_2 \mid inEq \wedge lhs \mid inIneq \wedge lhs \\
rhs &::= outEq \mid ok \mid declassify \\
inEq &::= Eq(inArg) \mid EqArr(inArrArg, ctr) \\
outEq &::= Eq(outArg) \mid EqArr(outArrArg, ctr) \\
inIneq &::= c < inIntArg \mid c \leq inIntArg \mid c > inIntArg \mid c \geq inIntArg
\end{aligned}$$

Fig. 6: Grammar for generating quantifier-free guesses for information flow.

V. Grammar Templates without Quantifiers

Fig. 6 lists the grammar used in the INFERSUM algorithm (Fig. 4) to generate quantifier-free guesses that represent information-flow properties. Each guess has the form of an implication and corresponds to a relational property because the activation variables `b1` and `b2` always occur positively in the antecedent. The antecedent (lhs) allows additional conjuncts expressing equalities ($inEq$) and inequalities ($inIneq$) over input arguments of procedures ($inArg$), including arrays ($inArrArg$) indexed by expressions (ctr). The consequent (rhs) allows conjuncts expressing equalities ($outEq$) over output arguments of procedures ($outArg$, $outArrArg$), the results of assertions (ok), or declassify expressions ($declassify$). In the equalities, the expression $Eq(e)$ represents the equality $e = idx(e, 2)$, and $EqArr(e, i)$ represents the equality $e[i] = idx(e[i], 2)$. The inequalities allow comparison of input integer arguments ($inIntArgs$) against constants (c).

The terminals in our grammar are populated from a combination of variable types and a syntactic analysis of the CHC encoding of the body of the target procedure. The candidate variables include input/output parameters of procedure and outputs that store the result of assertions. We extract various expressions, e.g., representing indices in array accesses, or consequents in declassify assertions. The complete set of terminals is listed in Appendix A. Other than activation variables and the results of assertions, all terminals e in our grammar are such that $getIdx(e) = 1$ to reduce redundancy among guesses due to symmetry resulting from indices, e.g., in equality expressions.

VI. Grammar Templates with Quantifiers

In this section, we present two templates for generating guesses with quantifiers – one for arrays and the other for property-directed invariants.

A. Quantified Templates for Arrays

We generate guesses for quantified invariants for a given procedure by adapting a technique from prior work [33] to target *relational* properties. We consider here the task of generating a quantified invariant for a CHC $body(\bar{x}) \Rightarrow R(\bar{x})$. We construct guess for a quantified invariant from four parts:

- a set of quantified variables $qVars$ not in $\bar{x}$,
- a *range* formula over the variables in $inIntArgs \cup qVars$,
- a set of *equalities* over variables in $qVars \cup inIntArgs \cup idx(inIntArgs, 2)$,

• a *cell property* formula over the variables in $\bar{x} \cup qVars$. All these components except *equalities* come directly from prior work [33], which combined them to form a candidate invariant: $\forall qVars.range \Rightarrow cell\ property$. We take a similar approach but use *equalities* to guess invariants over both program copies. We also use activation variables in the antecedent of the implication so that the candidate invariant only applies when both program copies are aligned. Here we only generate *range* formulas over variables for the first program copy and use the equalities to ensure that the corresponding variables in the second copy are equal to those in the first.

Quantified variables and range variables are determined similarly to previous work [33]. For each variable `i` in $inIntArgs \cap ctrs$ used to access an array index, two fresh quantified variables `q1` and `q2` are added to $qVars$, where $idx(\mathtt{q1}, 2) = \mathtt{q2}$. We let $quant(\mathtt{i}) = \mathtt{q1}$. For each such variable, we also generate a range formula that is an inductive invariant for R of the form:

$range ::= \mathtt{i} \leq \mathtt{q1} < boundGt \mid boundLt < \mathtt{q1} \leq \mathtt{i}$

Here, $boundGt$ is the set of expressions e over variables $\bar{x}$ for which $i < e$ or $e > i$ occurs as a subexpression of $body$, the body of a procedure. Similarly, $boundLt$ is the set of expressions e over variables $\bar{x}$ for which $e < i$ or $i > e$ occurs as a subexpression of $body$. Let the set $ranges$ denote the set of such $range$ expressions that are inductive for R (which we first check for each such candidate).

For each variable `i` in $inIntArgs \cap ctrs$, we generate the equality $quant(\mathtt{i}) = idx(quant(\mathtt{i}), 2)$ and the equality $i = idx(\mathtt{i}, 2)$ and add them to the set $equalities$.

Finally, to generate cell properties, we consider the subset of expressions generated by the grammar in Fig. 6 that contain accesses to array cells (also known as *select*-terms and denoted $[\cdot]$) with indices Idx such that for each $i \in Idx$, $ranges$ contains an expression containing $idx(i, 1)$. We take each such expression e and substitute each occurrence of any variable $i \in inIntArg \cap ctr$ with $quant(i)$ and then add the resulting expression to the set $cellProps$.

For each $cellProp \in cellProps$, we generate the following candidate invariant:

$$\lambda\bar{x}.\forall qVars. \bigwedge ranges \wedge \bigwedge equalities \wedge b_1 \wedge b_2 \Rightarrow cellProp$$

B. Property-Directed Templates

The final template allows us to generate property-directed guesses for a particular procedure `r` given a mapping M to inductive interpretations. This template consists of two parts: a *context guess* and a *quantifier-free guess*. As mentioned previously, we aim to find interpolants using SyGuS rather than an interpolating solver. The context guess is used to incorporate relevant properties from the context into the guess, and the quantifier-free guess is used to strengthen it.

We first describe how to generate the context guess given a CHC C that is an unfolding of a *query* Q, a predicate application $R(\bar{y})$ for procedure `r` that occurs in the body of the unfolding, and a CHC $body \Rightarrow R(\bar{x})$. Let ctx be the context for $R(\bar{y})$ in the unfolding of Q.

```
1: procedure FILTER($Ands$, $R(\bar{y})$, $\mathcal{C}$, $M$)
2:     $M' \leftarrow M[R \mapsto \lambda\bar{y}.\bigwedge Ands]$
3:     for $body \Rightarrow R(\bar{x}) \in \mathcal{C}$ do
4:         for application $R(\bar{x}')$ in $body$, context $ctx$ do
5:             $query \leftarrow M'(R)(\bar{x}) \wedge M'(ctx) \wedge \neg M'(R)(\bar{x}')$
6:             if $query$ satisfiable then
7:                 $m \leftarrow$ GETMODEL($query$)
8:                 $FC \leftarrow$ FALSECONJS($m$, $M'(R)(\bar{x}')$, $Ands$)
9:                 return FILTER($Ands \setminus FC$, $R$, $\mathcal{C}$, $M$)
10:    return $Ands$
```

Fig. 7: Procedure to find largest useful element in $\mathcal{P}(Ands)$.

Let $Ands$ be the set of conjuncts in $M(ctx)$. Each element of the powerset $\mathcal{P}(Ands)$ can become a context guess. We are interested only in elements p in $\mathcal{P}(Ands)$ that represent properties that, while initially not guaranteed to be true whenever `r` is called, are guaranteed to hold for any subsequent recursive calls to `r` provided that they held at the initial invocation of `r`. We discover the largest set $conseqAnds \subseteq \mathcal{P}(Ands)$ that represents such properties through a procedure based on the Houdini algorithm [22] (as shown in the algorithm in Fig. 7).

The procedure in Fig. 7 examines each CHC in $\mathcal{C}$ with an application of R to variables $\bar{x}$ in its head. The mapping M' maps R to the interpretation $\lambda\bar{y}.\bigwedge Ands$ but is otherwise the same as the current mapping M. For each such CHC, it checks if $M(R)$ is inductive (line 5) and uses a model (called a counterexample-to-induction) to weaken $Ands$. We can now use $\mathcal{P}(conseqAnds)$ as the set of context guesses.

We generate quantifier-free guesses $QFGuesses$ for $body \Rightarrow R(\bar{x})$ as shown in Sec. V, except now the set c of integer constants also includes all integer constants in ctx.

The algorithm in Fig. 8 describes how the context and quantifier-free guesses are combined to make a guess for R with context ctx and the current set of interpretations M. For each $\lambda\bar{x}.lhs \Rightarrow rhs \in QFGuess$ and $p \in \mathcal{P}(conseqAnds)$, we consider the mapping $M' = M[R \mapsto \lambda\bar{x}.M(R)(\bar{x}) \wedge rhs]$, which is the same as the mapping M except the interpretation for R is updated to $\lambda\bar{x}.M(R)(\bar{x}) \wedge rhs$. If $M'(ctx)$ is unsatisfiable and $lhs \wedge p$ is satisfiable (line 5), we generate the following guess: $\lambda\bar{x}.lhs \wedge p \Rightarrow rhs$. We only consider guesses such that $M'(ctx)$ is unsatisfiable because these guesses are such that if they are treated as an interpretation for R in C, they make $M'(C)$ satisfiable. This requirement ensures that the guesses considered help make progress toward proving the assertion in the original program corresponding to query Q. The checks on line 5 guarantee that each element added to $Guesses$, when applied to $\bar{y}$, is an interpolant separating $body[\bar{x} \mapsto \bar{y}]$ and $M'(ctx)$. If all guesses in $Guesses$ are interpolants separating these formulas, then it follows that $makeGuess(Guesses)(\bar{y})$ is also such an interpolant. Note that these guesses may contain quantifiers if the interpretations in M contain quantifiers.

VII. Implementation and Evaluation

We have implemented our technique in a prototype tool called FLOWER, developed on top of the CHC solver FREQ-

```
1: procedure CombineGuess($QFGuess$, $conseqAnds$, $M$, $R$, $ctx$)
2:     for $\lambda\bar{x}.lhs \Rightarrow rhs \in QFGuess$ do
3:         for $p \in \mathcal{P}(conseqAnds)$ do
4:             $M' \leftarrow M[R \mapsto \lambda\bar{x}.M(R)(\bar{x}) \wedge rhs]$
5:             if $M'(ctx)$ unsat, $lhs \wedge p$ sat then
6:                 $Guesses \leftarrow Guesses \cup \{\lambda\bar{x}.lhs \wedge p \Rightarrow rhs\}$
```

Fig. 8: Inference procedure for property-directed guesses.

Horn [27], [38]. We evaluated it on a suite of benchmarks[3] from the literature and real-world examples.

In our implementation, all candidate guesses allowed by our grammars are enumerated and checked, i.e., there is no further heuristic selection (currently) in our tool. Although this can be problematic if there are too many guesses, we did not encounter this issue in practice. For property-directed guesses, the unfoldings are explored in a breadth-first like manner.

a) Benchmarks: Of our 29 benchmarks, 15 are based on a subset[4] of the evaluation set for MPPs [4], [6], [19], [21], [34], [39]–[42]. While small in size, with the original programs ranging from 24-70 lines of Viper [43] code, these programs include non-trivial features such as arrays and declassification that are challenging for automated verifiers. We added two benchmarks based on code from Amazon Web Service's s2n [20], about 160 lines of SMT-LIB2 code that involve reading/writing from buffers. We also translated six benchmarks based on Blazer's "Literature" and "STAC" benchmarks [19], which ranged from 41-208 lines of Java.

The Viper benchmarks contained many manual annotations of information-flow specifications for both procedures and loops. We treated the specification for the apparent top-level procedure as an assertion, and eliminated the remaining annotations. Loops were encoded as recursion, as is typical in CHC encodings. Memory locations and memory-related annotations in the benchmarks were not encoded in CHCs; structures were either flattened or encoded as arrays.

The Blazer benchmarks considered were written in Java and originally checked for timing side channels. This can be reduced to checking for noninterference with appropriate instrumentation [44]. We manually instrumented and encoded these benchmarks into CHCs.

b) Evaluation: We also compared our tool against a state-of-the-art relational verifier Descartes [15] and a modular CHC-based verifier Spacer[5] [25]. For Descartes, we translated CHC benchmarks to intraprocedural Java programs.

Results from experiments on our suite of 29 benchmarks with a timeout of 10 minutes are shown in Table I. Blazer benchmarks are prefixed with "B" and s2n benchmarks are prefixed with "s2n." A timeout is indicated with **TO** and an unknown result with **U**. N/A indicates that Descartes was unable to handle the benchmark because of the presence of arrays or declassification. Benchmarks were run on a MacBook Pro, with a 2.7GHz Intel Core i5 processor and 8GB RAM.

[3]Available at https://github.com/lmpick/flower-benchmarks

[4]We left out termination-related properties; automation would require synthesis of ranking functions, which we do not currently support.

[5]Spacer outperformed all tools in CHC-Comp'19 in all LIA categories [45].

TABLE I: Results for 29 benchmarks. Times shown in seconds.

Example	Recursive	Flower Time	Spacer Time	Descartes Time
Banerjee		8.00	0.04	N/A
B GPT14	✓	73.91	**TO**	**U**
B K96	✓	12.60	**TO**	**U**
B Login	✓	18.20	**TO**	N/A
B ModPow1	✓	60.86	**TO**	**U**
B ModPow2	✓	104.59	**TO**	**U**
B PWCheck	✓	18.04	**TO**	N/A
Costanzo (2)	✓	3.94	0.65	N/A
Costanzo (4)	✓	3.85	7.10	N/A
Costanzo (8)	✓	3.85	62.50	N/A
Costanzo (16)	✓	4.08	**TO**	N/A
Costanzo (32)	✓	3.88	**TO**	N/A
Costanzo (64)	✓	3.93	**TO**	N/A
Costanzo (unbounded)	✓	8.17	**TO**	N/A
Darvas		2.04	0.03	N/A
Declassification	✓	4.91	0.03	N/A
Joana Fig. 1 top left		0.96	0.03	N/A
Joana Fig. 2 bottom left		0.90	0.02	0.06
Joana Fig. 2 top		0.58	0.02	0.08
Joana Fig. 13 left		0.25	0.03	0.07
Kusters		8.07	0.03	0.09
Main Example	✓	135.90	**U**	N/A
Main Example (det.)	✓	13.98	**TO**	N/A
s2n Ex. 1	✓	352.70	0.06	N/A
s2n Ex. 2	✓	30.95	**TO**	N/A
Smith	✓	23.26	**TO**	N/A
Terauchi Fig. 1		0.40	0.03	0.08
Terauchi Fig. 2		0.84	0.03	N/A
Terauchi Fig. 3	✓	3.55	**TO**	**U**

Our tool Flower is able to solve all 29 benchmarks, including all 15 benchmarks originally used to assess the usefulness of MPPs. Note that our tool successfully solved all these examples *without the annotations required by* Viper [43]. This demonstrates the effectiveness of our approach in reducing the annotation burden for verifying secure information flow.

Spacer is able to solve 14 of the 29 benchmarks, timing out for 14, and reporting **U** for one. Descartes cannot handle the majority of the benchmarks; of the 10 benchmarks it can take as input, Descartes solves 5. Out of the 20 examples with recursion (marked in Column 2), Spacer can only solve 5, whereas our tool can handle all 20. Spacer finds invariants via interpolation, which is unlikely to directly capture relational properties, so it is unable to find suitable invariants for these recursive procedures. For recursion-free examples, relational invariants are less crucial; invariants capturing precise behaviors are easier to find and are often sufficient for verification.

Descartes is similarly unable to find appropriate invariants. For each of the 5 recursive benchmarks that it can take as input, it is unable to find the required loop invariant to verify the program. Although Descartes also uses a template-based approach for generating candidate invariants, the templates are insufficient for these benchmarks.

To evaluate scalability, we considered versions of the Costanzo benchmark with different array bounds (shown in parentheses in Table I). Fig. 9 shows the performance comparison against Spacer as the array bound increases. Spacer's behavior indicates its inability to find relational properties; it

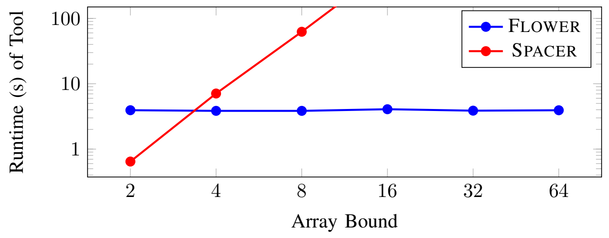

Fig. 9: Timing results for Costanzo benchmark with different array bounds.

learns properties for each array index individually, rendering it unable to solve the Costanzo benchmark within 10 minutes after the array bound reaches 16 (note that the original Costanzo benchmark has bound 64). Although it was run in a mode that allows it to learn quantified properties, Spacer is unable to find the desired relational property. In contrast, our approach solves all the bounded Costanzo benchmarks in about the same time because the quantified guesses are the same except for the constant bound. Our approach is also able to solve the Costanzo benchmark in which the array is *unbounded*, which Spacer is unable to do.

VIII. Related Work

There are many related efforts in relational program verification, information-flow checkers, and syntax-guided synthesis.

a) Relational Program Verification: While this work focuses on modular product programs [21], many other approaches also reduce relational program verification to safety verification [3]–[9], including those that employ a reduction to systems of CHCs [10], [46]. *However, most do not perform modular reasoning over procedures but inline them, and do not generate relational specifications for procedures.* One modular approach restricts both copies of the program to always follow the same control flow [6]. Another uses *mutual* summaries but does not provide an automatic procedure for inferring summaries as we do [8].

Other relational verifiers avoid explicitly constructing a product program. Some use program logics to work with Hoare triples [15]–[17], construct product programs implicitly [18], use decomposition instead of composition [19], or employ reinforcement learning [12]. These approaches also do not modularly reason about nor infer relational specifications for procedures, though they may modularly handle loops [15].

b) Information-Flow Checkers: Most automatic hyperproperty verifiers can handle information-flow properties by constructing product programs either implicitly [15], [18], or by lazily performing self-composition [13], [14] or synchronization [10], [46]. However, most of these techniques do not perform modular reasoning over the product programs or results of self-composition. One synchronization approach uses property-directed reachability and use modular reasoning for inference of relational procedure summaries [25], [46], [47], but our experiments show that the property-directed reachability tool Spacer upon which this tool was built often fails to infer the needed invariants in programs with recursion.

Other efforts focus on verifying *resource* leakage, such as the presence of timing side channels [11], [19], [44]. With appropriate instrumentation for resource leakage [11], [44], checking for timing leakage can be reduced to hyperproperty verification. As seen in our evaluation, our tool can be used to check the absence of timing side channels after appropriate resource usage instrumentation.

Approaches based on types and abstract interpretation can modularly infer information-flow properties of procedures. There are many type-inference-based approaches for checking secure information flow [48]–[51]. Such approaches employ a security type system such that terms only type check if they do not have any illegal information flows (e.g., from low-security to high-security variables). There are also approaches based on dynamic taint analysis [52]–[57], which involves instrumenting code with taint variables and code to track taint. However, type-inference-based and taint analysis approaches suffer from imprecision (e.g., due to path-insensitivity or an inability to infer invariants over arrays) that may lead to failure in type inference even for leakage-free programs. In contrast, our approach is path-sensitive and requires only the annotations that specify the property to be verified. One abstract-interpretation-based approach can infer possible information-flow dependencies, indicating which variables' values may depend on others' [58]. This approach, like ours, does not require annotations indicating which inputs and outputs are public or private. However, unlike our approach, it does not handle programs with procedures, arrays, or declassification.

c) Syntax-Guided Synthesis: Our approach is also related to a wide range of guess-and-check SyGuS techniques [27], [29]–[33], [38]. Especially relevant are enumerate-and-check approaches to solve CHCs [27], [33], [38]. Our template for guessing quantified invariants for arrays adapts a previous technique [33] to the setting of reasoning about secure information flow. As far as we know, such techniques have not been applied to inferring or verifying information-flow properties. The structure of information-flow properties makes them ideal targets for grammar-based enumerative search and synthesis.

IX. Conclusions

We have introduced a SyGuS-based technique for automatic inference of modular relational specifications that are useful for verifying secure information flow in interprocedural programs. Our technique relies on three grammar templates to infer procedure summaries in modular product programs, where these procedure summaries are of a particular form. The first template guesses quantifier-free summaries for information flow, the second guesses quantified summaries for expressing properties over arrays, and the third template guesses summaries that depend on the calling context of a procedure. An implementation of our techniques on top of a CHC solver and an experimental evaluation on benchmarks demonstrates that our approach finds useful procedure summaries to verify secure information flow, thereby reducing the annotation burden in prior work. Our tool outperforms a state-of-the-art hyperproperty verifier and a modular CHC-based verifier on several benchmark examples.

References

[1] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[2] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *CSFW*. IEEE Computer Society, 2005, pp. 255–269.

[3] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *CSFW*. IEEE Computer Society, 2004, pp. 100–114.

[4] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *SAS*, ser. Lecture Notes in Computer Science, vol. 3672. Springer, 2005, pp. 352–367.

[5] G. Barthe, J. M. Crespo, and C. Kunz, "Product programs and relational program logics," *J. Log. Algebraic Methods Program.*, vol. 85, no. 5, pp. 847–859, 2016.

[6] ——, "Relational verification using product programs," in *FM*, ser. Lecture Notes in Computer Science, vol. 6664. Springer, 2011, pp. 200–214.

[7] ——, "Beyond 2-safety: Asymmetric product programs for relational program verification," in *LFCS*, ser. Lecture Notes in Computer Science, vol. 7734. Springer, 2013, pp. 29–43.

[8] C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo, "Towards Modularly Comparing Programs Using Automated Theorem Provers," in *CADE*, ser. LNCS, vol. 7898. Springer, 2013, pp. 282–299.

[9] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[10] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, "Relational verification through horn clause transformation," in *SAS*, ser. Lecture Notes in Computer Science, vol. 9837. Springer, 2016, pp. 147–169.

[11] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *ACM Conference on Computer and Communications Security*. ACM, 2017, pp. 875–890.

[12] J. Chen, J. Wei, Y. Feng, O. Bastani, and I. Dillig, "Relational verification using reinforcement learning," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 141:1–141:30, 2019.

[13] W. Yang, Y. Vizel, P. Subramanyan, A. Gupta, and S. Malik, "Lazy self-composition for security verification," in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 10982. Springer, 2018, pp. 136–156.

[14] R. Shemer, A. Gurfinkel, S. Shoham, and Y. Vizel, "Property directed self composition," in *CAV (1)*, ser. Lecture Notes in Computer Science, vol. 11561. Springer, 2019, pp. 161–179.

[15] M. Sousa and I. Dillig, "Cartesian hoare logic for verifying k-safety properties," in *PLDI*. ACM, 2016, pp. 57–69.

[16] G. Barthe, B. Köpf, F. Olmedo, and S. Z. Béguelin, "Probabilistic relational reasoning for differential privacy," in *POPL*. ACM, 2012, pp. 97–110.

[17] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *POPL*. ACM, 2004, pp. 14–25.

[18] A. Farzan and A. Vandikas, "Automated hypersafety verification," in *CAV (1)*, ser. Lecture Notes in Computer Science, vol. 11561. Springer, 2019, pp. 200–218.

[19] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *PLDI*. ACM, 2017, pp. 362–375.

[20] Amazon Web Services, "https://github.com/awslabs/s2n," 2019.

[21] M. Eilers, P. Müller, and S. Hitz, "Modular product programs," in *ESOP*, ser. Lecture Notes in Computer Science, vol. 10801. Springer, 2018, pp. 502–529.

[22] C. Flanagan, R. Joshi, and K. R. M. Leino, "Annotation inference for modular checkers," *Inf. Process. Lett.*, vol. 77, no. 2-4, pp. 97–108, 2001.

[23] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "From under-approximations to over-approximations and back," in *TACAS*, ser. LNCS, vol. 7214. Springer, 2012, pp. 157–172.

[24] P. Rümmer, H. Hojjat, and V. Kuncak, "Disjunctive interpolants for Horn-Clause verification," in *CAV*, ser. LNCS, vol. 8044. Springer, 2013, pp. 347–363.

[25] A. Komuravelli, A. Gurfinkel, and S. Chaki, "Smt-based model checking for recursive programs," *Formal Methods in System Design*, vol. 48, no. 3, pp. 175–205, 2016.

[26] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: safety verification by interactive generalization," in *PLDI*. ACM, 2016, pp. 614–630.

[27] G. Fedyukovich, S. J. Kaufman, and R. Bodík, "Sampling invariants from frequency distributions," in *FMCAD*. IEEE, 2017, pp. 100–107.

[28] H. Zhu, S. Magill, and S. Jagannathan, "A data-driven CHC solver," in *PLDI*. ACM, 2018, pp. 707–721.

[29] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD*. IEEE, 2013, pp. 1–8.

[30] S. Padhi, R. Sharma, and T. D. Millstein, "Data-driven precondition inference with learned features," in *PLDI*. ACM, 2016, pp. 42–56.

[31] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling Enumerative Program Synthesis via Divide and Conquer," in *TACAS, Part I*, ser. LNCS, vol. 10205, 2017, pp. 319–336.

[32] A. Reynolds, H. Barbosa, A. Nötzli, C. W. Barrett, and C. Tinelli, "cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis," in *CAV, Part II*, ser. LNCS, vol. 11562. Springer, 2019, pp. 74–83.

[33] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Quantified invariants via syntax-guided synthesis," in *CAV (1)*, ser. Lecture Notes in Computer Science, vol. 11561. Springer, 2019, pp. 259–277.

[34] D. Costanzo and Z. Shao, "A separation logic for enforcing declarative information flow control policies," in *POST*, ser. Lecture Notes in Computer Science, vol. 8414. Springer, 2014, pp. 179–198.

[35] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*. ACM, 2012, pp. 405–416.

[36] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV*, ser. Lecture Notes in Computer Science, vol. 4144. Springer, 2006, pp. 123–136.

[37] ——, "Lazy annotation revisited," in *CAV*, ser. Lecture Notes in Computer Science, vol. 8559. Springer, 2014, pp. 243–259.

[38] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Solving Constrained Horn Clauses Using Syntax and Data," in *FMCAD*. ACM, 2018, pp. 170–178.

[39] A. Banerjee and D. A. Naumann, "Secure information flow and pointer confinement in a java-like language," in *CSFW*. IEEE Computer Society, 2002, p. 253.

[40] Á. Darvas, R. Hähnle, and D. Sands, "A theorem proving approach to analysis of secure information flow," in *SPC*, ser. Lecture Notes in Computer Science, vol. 3450. Springer, 2005, pp. 193–209.

[41] D. Giffhorn and G. Snelting, "A new algorithm for low-deterministic security," *Int. J. Inf. Sec.*, vol. 14, no. 3, pp. 263–287, 2015.

[42] G. Smith, "Principles of secure information flow analysis," in *Malware Detection*, ser. Advances in Information Security. Springer, 2007, vol. 27, pp. 291–307.

[43] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 9583. Springer, 2016, pp. 41–62.

[44] K. Athanasiou, B. Cook, M. Emmi, C. MacCárthaigh, D. Schwartz-Narbonne, and S. Tasiran, "Sidetrail: Verifying time-balancing of cryptosystems," in *VSTTE*, ser. Lecture Notes in Computer Science, vol. 11294. Springer, 2018, pp. 215–228.

[45] CHC-Comp, "https://chc-comp.github.io," 2019.

[46] D. Mordvinov and G. Fedyukovich, "Property directed inference of relational invariants," in *FMCAD*. IEEE, 2019, pp. 152–160.

[47] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, ser. Lecture Notes in Computer Science, vol. 7317. Springer, 2012, pp. 157–171.

[48] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[49] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996.

[50] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *POPL*. ACM, 1999, pp. 228–241.

[51] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, "Secure compilation to protected module architectures," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 2, pp. 6:1–6:50, 2015.

[52] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kâafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices," in *SECRYPT*. SciTePress, 2013, pp. 461–468.

[53] M. Costa, J. Crowcroft, M. Castro, A. I. T. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: end-to-end containment of internet worms," in *SOSP*. ACM, 2005, pp. 133–147.

[54] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *MICRO*. IEEE Computer Society, 2004, pp. 221–232.

[55] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: dynamic taint analysis with targeted control-flow propagation," in *NDSS*. The Internet Society, 2011.

[56] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010, pp. 317–331.

[57] D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *ICISS*, ser. Lecture Notes in Computer Science, vol. 5352. Springer, 2008, pp. 1–25.

[58] M. Zanioli and A. Cortesi, "Information leakage analysis by abstract interpretation," in *SOFSEM*, ser. Lecture Notes in Computer Science, vol. 6543. Springer, 2011, pp. 545–557.

Appendix

A. Terminals in SyGuS Grammar for Secure Information Flow

The terminals in our grammar to generate quantifier-free guesses (Fig. 6) are populated by a combination of tagging types of variables and a syntactic analysis of the CHC encoding of the body of the target procedure under consideration.

a) Tagging the types: For a CHC with head $R(\bar{x})$ that encode a modular product procedure `r`, each $x \in \bar{x}$ is tagged as follows:

- `in`: if x corresponds to a non-activation input argument `x` in `r` with $\mathit{getIdx}(\mathtt{x}) = 1$;
- `out`: if x corresponds to an output ret in `r` with $\mathit{getIdx}(\mathit{ret}) = 1$;
- `arr`: if x is an array;
- `int`: if x is an integer;
- `ok`: if x is an output value storing the result of assertions.

The following metavariables specify what the terminals based on tags range over:

- inArg: the set inArgs of variables tagged `in`;
- $\mathit{inArrArg}$: the set $\mathit{inArrArgs}$ of variables tagged both `in` and `arr`;
- outArg: the set of variables $\mathit{outArgs}$ tagged `out`;
- $\mathit{outArrArg}$: the set of variables $\mathit{outArrArgs}$ of variables tagged both `out` and `arr`;
- $\mathit{inIntArg}$: the set of variables $\mathit{inIntArg}$ tagged `in` and `int`;
- ok: the set of variables tagged `ok`.

The activation variables in $\bar{x}$ are denoted b_1 and b_2.

b) Syntactic Analysis: The terminal ctr is based on a syntactic analysis of the body of the CHC. It ranges over a set ctrs comprising the following:

- all expressions e with $\mathit{getIdx}(e) = 1$ that occur in the procedure body within subexpressions of the form $a[e]$ for some a;
- terminals that c ranges over, consisting of all integer constants that occur as the right- or left-hand side of equalities or inequalities in the body of the procedure;
- terminals that $\mathit{declassify}$ ranges over, which consists of the consequents $e_1 = e_2$ of any implications of the form $b_1 \wedge b_2 \Rightarrow e_1 = e_2$, where b_1 and b_2 are Boolean variables, $\mathit{getIdx}(e_1) = 1$, and $\mathit{getIdx}(e_2) = 2$.