Look for the Proof to Find the Program: Decorated-Component-Based Program Synthesis

Adrià Gascón[1⋆], Ashish Tiwari[2⋆⋆], Brent Carmer[3], and Umang Mathur[4]

[1] University of Warwick / Alan Turing Institute
agascon@turing.ac.uk
[2] SRI International
tiwari@csl.sri.com
[3] Oregon State University
carmerb@eecs.oregonstate.edu
[4] University of Illinois at Urbana-Champaign
umathur3@illinois.edu

Abstract. We introduce a technique for component-based program synthesis that relies on searching for a target program and its proof of correctness simultaneously using a purely constraint-based approach, rather than exploring the space of possible programs in an enumerate-and-check loop. Our approach solves a synthesis problem by checking satisfiability of an $\exists\exists$ constraint ϕ, whereas traditional program synthesis approaches are based on solving an $\exists\forall$ constraint. This enables the use of SMT-solving technology to decide ϕ, resulting in a scalable practical approach. Moreover, our technique uniformly handles both functional and nonfunctional criteria for correctness. To illustrate these aspects, we use our technique to automatically synthesize several intricate and non-obvious cryptographic constructions.

1 Introduction

Automated program synthesis has a rich history in computer science. This problem has been studied from several perspectives, and currently lies at the intersection between logic, artificial intelligence, and software engineering. The seminal work by Manna and Waldinger [23], commonly referred to as *deductive synthesis*, is based in the observation that a program with input x and output y, specified by a formula $\phi(x, y)$, can be extracted from a constructive proof of $\forall x \exists y : \phi(x, y)$, as this formula is equivalent to a second-order formula of the form $\exists f\ \forall x : \phi(x, f(x))$.

More recently, program synthesis has taken the form of *inductive synthesis*, where programs are not deduced, but synthesized iteratively by finding candidate programs that work correctly on an ever-increasing input space. In practice, this search is implemented using powerful constraint solvers, typically Boolean

⋆ Part of this work was done while the author was at the University of Edinburgh, supported by the SOCIAM Project under EPSRC grant EP/J017728/2.
⋆⋆ Supported in part by the National Science Foundation under grant CCF 1423296.

Satisfiability (SAT) solvers and Satisfiability Modulo Theory (SMT) solvers. The various choices in the synthesis approach, correctness specification, and restrictions on the program search space, have been explored extensively [2, 16, 19, 28, 29].

Manna and Waldinger [23] foresee the possibility that the user could suggest program segments, i.e. snippets, to the synthesizer, which it could use to construct a full solution. This idea was pursued in the work on *component-based program synthesis* [16, 32], where the target program is constructed from a set of predefined components (library calls). The component-based synthesis problem is also naturally encoded as an $\exists\forall$ problem: there *exists some* placement of components on the different program lines such that *for all* inputs, the function computed by the resulting program satisfies the given property.

Although it comes in many flavors, a synthesis problem is essentially parameterized by a *target language*, i.e. the language of the target program to be synthesized, and a *specification language*, i.e. the formalism in which the functionality of the target program is expressed. Moreover, synthesis may be subject to nonfunctional constraints, such as optimizing for certain metrics like program size or power consumption, or enforcing security properties. Examples of target languages include MapReduce-style programs [27], bit-vector manipulations [16], recursive programs [20], high-level circuit descriptions [13], and domain-specific languages for cryptographic constructions [18, 22]. On the other hand, examples of specification languages include formulas in several modal temporal logics, input-output examples [19, 27], flattened verilog circuits given as SMT/Boolen formulas [13], and assertions in imperative program sketches [29].

Generally speaking, the synthesis problem seeks to find a program P in the target language, such that P satisfies the specification ϕ given in the specification language. As ϕ is often a relation between input-output pairs, this naturally corresponds to an exists-forall check. In fact, inductive synthesis algorithms often consist of two procedures: a procedure to generate candidate programs from the target language, and a procedure for checking ϕ on a given candidate. In that setting, synthesis consists of an enumerate-and-check feedback loop, similar to a Counterexample-Guided Abstraction Refinement (CEGAR) loop, that continues until a valid candidate, i.e. a candidate satisfying ϕ, is found, or no more candidates can be generated. Note that, to achieve scalability, the verification check is often conservative, i.e. ϕ is replaced by a sufficient, but not necessary condition. This is often the case with security properties, as they are costly to check in a sound and complete way.

The paradigm of "enumerate-and-check" for solving synthesis problems is fairly widespread in literature [13, 16, 22, 27]. In this context, the general idea of looking for a program and its proof simultaneously has been also considered in previous work on type-directed synthesis [12, 26], where program candidates that are not type correct are pruned early in the enumerate-and-check. Also, the deductive phase of the Leon synthesizer [20] is also based on proof search.

In this paper, we pursue a framework that enables synthesis using a single search, and avoids quantifier alternation. Inspired by the challenge posed

by Manna and Waldinger [23] pertaining to incorporation of user-defined proof systems in synthesis, and building on the framework of component-based synthesis [16], we present an approach for synthesis that allows users to define simple proof systems, in the form of *constraint-generation rules* for the components of the synthesized program. This framework, which we call *decorated-component-based synthesis*, provides a way for the user to not only easily encode nonfunctional properties, as in [32], but also replace the validity check in the synthesis process by a search for a proof in the provided proof system. This effectively removes a quantifier alternation and hence turns the enumerate-and-check approach into a search problem in which, intuitively, the space of target programs and their corresponding correctness proofs are explored simultaneously, resulting in a much more scalable approach.

Contributions. We make three key contributions in this work. First, we formulate the *decorated-component-based program synthesis* problem, which allows users to encode a bounded search for proof (in a user-picked proof system) in the synthesis problem. Second, we show that decorated-component-based synthesis problem reduces to an exists-forall constraint in general, just as the component-based synthesis problem [16]. However, the additional "decorations" enable users to either augment a synthesis constraint with additional existential parts (similar to the work in [32]), or entirely replace the forall by a existential in the synthesis constraint. This second application of decorations is appealing because solving a purely existential constraint is significantly faster than solving an exists-forall constraint. Third, we demonstrate that security is an ideal domain for application of automated program synthesis technology, thus solidifying preliminary evidence in this regard [3, 7, 18, 22, 32]. Decorated-component-based synthesis eases the task of specifying security requirements. Our synthesizer and all the examples mentioned in this paper, as well as instructions for running them, are available at the SYNUDIC project's public repository [14].

Outline. We start by illustrating our approach (with complete details) on an example synthesis problem (Section 2). We then formally define decorated-component-based synthesis (Section 3), and show that it reduces to an $\exists\forall$ constraint (Section 4). We then present our main result that enables conservative replacement of the $\forall$ by an $\exists$ in the synthesis constraint (Section 5). Finally, we show its application to synthesis of cryptographic schemes (Section 6).

2 An Illustrative Example

Secure Multi-party Computation (MPC) is a subfield of cryptography with the goal of creating protocols for multiple parties to jointly compute a function over their inputs without disclosing the inputs to each other. Here we consider the problem of designing an information secure two-party multiplication protocol, which is a basic component in many privacy-preserving algorithms [5, 10].

Our problem is as follows: find a protocol for Alice and Bob to compute an additive share of the product of Alice's and Bob's private input values. Let Alice's

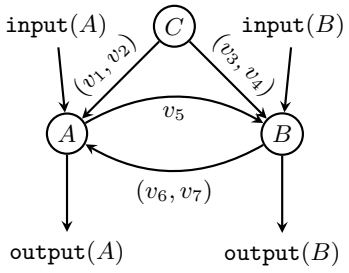
(private input) value be $\texttt{input}(A) \in \mathbb{Z}_q$, and Bob's value be $\texttt{input}(B) \in \mathbb{Z}_q$, for some natural q, say 2^{32}. For *correctness*, our functional requirement is that

$$\texttt{output}(A) + \texttt{output}(B) = \texttt{input}(A) * \texttt{input}(B)$$

In other words, each party computes a *share* of the result. We assume that Alice and Bob can rely on a third *untrusted* party Carol that aids in the computation.

Now that we have the functional correctness requirement, let us consider the nonfunctional security requirement. Informally, the main security requirement is that Alice and Carol should not learn the value $\texttt{input}(B)$, and Bob and Carol should not learn the value $\texttt{input}(A)$. In the static honest-but-curious adversary model, one assumes that the parties – Alice, Bob, and Carol – have an incentive to deduce as much information as possible from the transcripts of the protocol, but they do not deviate from it nor collude. (Formally, the security requirement is formulated in the so-called "simulation paradigm", see [21] for details.)

How do we use our new decorated-component-based synthesis technique to discover a secure multiplication protocol? We first identify the components that we could use in the protocol. These are the three arithmetic operations `plus`, `minus`, and `times`, along with a few calls to a pseudo-random generator, say `genx`, `geny`, `genr`, and `genu` (that generate random numbers x, y, r, u), and the identity function `identity`.



$$\begin{aligned}
(v_1, \ldots, v_4) &= P_c();\\
v_5 &= P_{A,1}(v_1, v_2, \texttt{input}(A));\\
(v_6, v_7) &= P_{B,1}(v_3, v_4, \texttt{input}(B));\\
\texttt{output}(A) &= P_{A,2}(v_1, v_2, v_6, v_7, \texttt{input}(A));\\
\texttt{output}(B) &= P_{B,2}(v_1, v_2, v_5, \texttt{input}(B));
\end{aligned}$$

Fig. 1. High-level synthesis sketch for secure multiplication. The diagram (left) shows the structure of the protocol to be synthesized. Solving the corresponding template (right) involves finding programs $P_C, P_{A,1}, P_{B,1}P_{A,2}, P_{B,2}$, built out of components `genx`, `geny`, `genr`, `genu`, `identity`, `plus`, `minus`, `times` in the library, satisfying (i) the functional requirement, i.e. $\texttt{output}(A) + \texttt{output}(B) = \texttt{input}(A) * \texttt{input}(B)$), and (ii) the security requirement. Moreover, $P_C, P_{A,1}, P_{B,1}P_{A,2}, P_{B,2}$ must have no more than 5, 1, 3, and 2 operations, respectively.

Next, let us fix a communication schedule between the parties. The structure of the protocol, depicted in Figure 1(left) is as follows: (1) C computes some values $(v_1, \ldots, v_4)$ first (5 lines), (2) C sends (v_1, v_2) to A and (v_3, v_4) to B, (3) A computes v_5 (1 line) and sends it to B, (4) B computes some values (3 lines), sends a pair (v_6, v_7) to A and picks one value as its output, (5) A computes its output (2 lines). Note that instances of this template are constant-round protocols, as opposed to approaches to secure multiplication based on Oblivious

Transfer [15]. Our tool [14] takes a description of the library and a template of a straight-line program similar to the one in Figure 1(right) as input.

If we give each of the components its natural interpretation (that is, all variables are integer valued, `plus` is arithmetic addition, and so on), then the correctness requirement is simply the arithmetic equality $\texttt{output}(A)+\texttt{output}(B) = \texttt{input}(A) * \texttt{input}(B)$. Now, synthesis can be performed as in [16] or [32] – the synthesis problem is reduced to an $\exists\forall$ formula over a suitable theory, where the $\exists$ quantifier searches over the space of possible programs, and the $\forall$ quantifier checks correctness over the space of all possible inputs.

Our new decorated-component-based synthesis framework allows us to (a) conservatively turn the validity check above into a satisfiability check over an alternate theory, and (b) provide a natural way of also specifying a nonfunctional security requirement. The key idea behind our approach is associating a constraint with each use of a component in the (yet to be discovered) program, by defining a *decoration*, or a *constraint generation rule*, for every component.

$$\texttt{genx}\quad \frac{}{\{\theta\}\ v := \texttt{genx}\ \{\theta \circ \{v \mapsto pol2vec(\text{“}x\text{”})\}\}} \qquad v_x = 1\ \wedge \bigwedge_{i \neq x} v_i = 0$$

$$\texttt{plus}\quad \frac{}{\{\theta\}\ x := \texttt{plus}(y,z)\ \{\theta \circ \{x \mapsto \theta(y) + \theta(z)\}\}} \bigwedge_{i \in I} x_i = y_i + z_i$$

$$\texttt{times}\quad \frac{[\theta(y) * \theta(z) \text{ is quadratic}]}{\{\theta\}\ x := \texttt{times}(y,z)\ \{\theta \circ \{x \mapsto \theta(y) * \theta(z)\}\}} \bigwedge_{i,j \in L} x_{ij} = y_i z_j + y_j z_i\ \wedge \bigwedge_{i \in NL} y_i = z_i = 0$$

$$\texttt{compose}\quad \frac{\{\theta_0\}\ P_1\ \{\theta_1\} \qquad \{\theta_1\}\ P_2\ \{\theta_2\}}{\{\theta_0\}\ P_1; P_2\ \{\theta_2\}}$$

$$\texttt{check}\quad \frac{\{\theta_0\}\ P\ \{\theta\} \qquad \theta(v) + \theta(w) = \theta(a)\theta(b)}{\{\theta_0\}\ P\ \{\texttt{assert}(\texttt{v} + \texttt{w} = \texttt{a} * \texttt{b})\}} \qquad v_{ab} + w_{ab} = 1\ \wedge \bigwedge_{i \neq ab} v_i + w_i = 0$$

Fig. 2. Selected proof rules and generated constraints for the secure multiplication example. Essentially, the rules perform symbolic execution of the program. The third column shows (some of the) constraints that are generated on the dual variables.

As a first step in defining the decorations, consider an abstract domain $\mathcal{A}$ that consists of symbolic polynomial expressions of degree at most 2 over the six variables: the two inputs a, b and the four random numbers x, y, r, u.

$$\mathcal{A} = \{p(a,b,x,y,r,u) \mid p \text{ is quadratic with no constant term}\}$$

where polynomials in $\mathcal{A}$ are represented in *canonical form* as a sum of ordered monomials.

Let us say we wish our program (that is yet to be discovered) to have a functional correctness proof in this abstract domain. We can design proof rules that essentially perform symbolic execution to check the correctness assertion. Let $\theta : V \mapsto \mathcal{A}$ map a *program variable* $v \in V$ to the symbolic polynomial value

of v. We can compute θ by starting with any substitution, and updating it using the rules in Figure 2. For example, the rule `genx` says that after execution of $v := \texttt{genx}$, we get a new substitution that maps v to the symbolic polynomial x. Similarly, the rule `plus` handles program lines of the form $v := \texttt{plus}(x, y)$ by setting $\theta(v)$ to the polynomial $(\theta(x) + \theta(y)) \in \mathcal{A}$. In Figure 2 we omitted rules for `minus`, `geny`, `genr`, `genu`, and `identity`. Once we have computed the substitution θ at the end of the program, we use the rule `check` to prove an assertion $v + w = a * b$ by checking if $\theta(v) + \theta(w)$ and $\theta(a) * \theta(b)$ are syntactically equal (recall elements of $\mathcal{A}$ are represented in canonical form). The proof rule for `times` in Figure 2 has a condition that allows multiplication to be used only on linear polynomials (so that the result is atmost quadratic).

Note that this proof system for checking correctness turns the validity question over integers into an evaluation over $\mathcal{A}$. Our goal now is to have our synthesizer use this proof system to, instead of searching for a program satisfying the postdondition $\texttt{output}(A) + \texttt{output}(B) = \texttt{input}(A) * \texttt{input}(B)$, search for a program while *simultaneously* searching for its correctness proof. To see how this is done, first note that the elements in the chosen abstract domain can be uniquely identified by 27 parameters – namely, the coefficients of all degree 1 and degree 2 monomials over the six variables ($C(6,1) + C(6,2) + C(6,1) = 6 + 15 + 6 = 27$). Let $L = \{x, y, r, u, a, b\}$ and $NL = \{ij \mid i \neq j, i \in L, j \in L\} \cup \{ii \mid i \in L\}$. Hence, every program variable v is associated with 27 *new variables*, namely, v_i, where index i ranges over the set $I = L \cup NL$ of all indices. If variable v gets a symbolic value $p \in \mathcal{A}$ and p is quadratic, then the new variables v_i's get the value of the coefficient of the monomial i in p. Now, let us say we could prove our functional requirement using quadratic symbolic values. Then, there exists a value of the new variables that witnesses this proof. The converse is even more important for our goal: if we find a consistent valuation for the new variables, then we would establish our functional requirement. Restricting $\mathcal{A}$ to contain *quadratic* polynomials makes the set of new variables, and the proof search, finite.

A sound proof rule application (on the abstract values) induces certain constraints on the new variables. Hence, as mentioned above, we associate to every component a constraint generation rule, also called a *decoration*, that produces the suitable constraint to encode the corresponding proof rule. The generated constraint essentially says what combinations of the 27 parameters for its inputs and outputs are consistent with the proof rule. For example, the Column 3 in Figure 2 shows such constraints for selected components.

Note that the correctness requirement was an equivalence of polynomial expressions, and in our abstract domain $\mathcal{A}$, this maps to equality of coefficients (see right column on last row in Figure 2). Thus, ignoring the security requirement, the synthesis problem is reduced to finding $27 * l$ values, where l is the length of the program, that satisfy some big constraint generated using decorations. This is an $\exists\exists$ problem: we are finding a program (first $\exists$) and a proof of its correctness over the chosen abstract domain (second $\exists$). Decorations have enabled us to replace the $\forall$ check by an $\exists$ check. Note that, although the task of finding a proof system requires human intuition, the process of designing constraint generation

rules, i.e. parameterizing the abstract domain and building the constraint for every component of the library, is systematic (Section 5.1).

We have not yet solved our original problem because we still need to include the security requirement. The formal security requirement, which is based on showing a simulation of the ideal functionality in the actual functionality (see [21] for details on this proof technique), is difficult to capture precisely. We take a very practical approach here: we replace the security requirement by another easily checkable requirement that is sufficient (but not necessary) for security. The new check can itself be described by proof rules, and we can again search for a bounded-size proof to establish security. The sufficiency of the proof rules may itself be proved as a meta-theorem by hand. In our example, we use ideas from [32], which were in turn inspired by [22], to synthesize block cipher modes of operation. Essentially, the decorations rely on a simple type system that propagates a qualifier stating whether a variable always has a "random" value on any program line, in a sound, and possibly incomplete, way.

Using this encoding of the security requirement, our sketch is complete, we run our synthesis tool, and it returns the following protocol:

1. C generates random numbers x, y, r, and computes xy and $r - xy$.
2. C sends (x, r) to A and $(r - xy, y)$ to B.
3. A computes $\texttt{input}(A) - x$ and sends it to B.
4. B computes $\texttt{output}(B) = (\texttt{input}(A) - x)b + (r - xy)$ and sends $y + \texttt{input}(B)$ to A.
5. A computes $\texttt{output}(A) = (y + \texttt{input}(B))x - r$.

We were not aware of this protocol before it was synthesized by our tool. Note that the protocol did not use the fourth random number (u), whereas we were expecting the synthesized protocol to need it.

3 Decorated-Component-Based Program Synthesis

We define the component-based synthesis problem in this section, as introduced in [16]. We then extend it to decorated-component-based synthesis, where components are additionally allowed to be associated with certain constraint-generation rules.

A component library Σ is a set of symbols. Each symbol is associated with an arity, but without loss of generality and for simplicity, we will often implicitly assume that the arity of each symbol in Σ is two. The symbols in Σ should be regarded as functions that can be invoked by a program.

The functions in Σ compute over some values. For simplicity again, let us say these values come from a domain `Domp` of all values. The semantics of the functions in Σ is given over the domain `Domp` by `Semp`.

$$\texttt{Semp} : \Sigma \mapsto 2^{\texttt{Domp}^3} \tag{1}$$

That is, if $f \in \Sigma$, then $\texttt{Semp}(f)$ is a ternary relation on `Domp`. Intuitively, $c = f(a, b)$ iff $(a, b, c) \in \texttt{Semp}(f)$.

We want to synthesize straight-line programs (SLPs) using calls to functions in Σ. A generic template of such a 9-line program is shown in Figure 3. The

$l0 : x_0 := \texttt{input}$
$l1 : x_1 := f_1(a_{11}, a_{12});$
$l2 : x_2 := f_2(a_{21}, a_{22});$
$\vdots$
$l9 : x_9 := f_9(a_{91}, a_{92});$

Variables: Domain
$f_1, \ldots, f_9 : \Sigma$
$a_{11}, \ldots, a_{92} : 0..8$
$vx_0, \ldots, vx_9 : \texttt{Domp}$
$tx_0, \ldots, tx_9 : \texttt{Domd}$

Fig. 3. A template for an arbitrary straight-line program with 9 lines.

semantics Semp of one component can be extended to semantics of a straight-line program P (such as the one shown in Figure 3) that takes one input x_0 and produces one output x_9 so that $\texttt{Semp}(P) \subseteq \texttt{Domp}^2$ contains all pairs (a, b) where $x_9 = b$ is reachable starting with $x_0 = a$.

A specification, $\phi_{\texttt{fspec}}$, of a program P that takes one input and produces one output is given as binary relation on Domp.

Definition 1 (Component-based Synthesis, or CoS [16]). *A CoS problem is a tuple* $(\Sigma, \texttt{Domp}, \texttt{Semp}, \phi_{\texttt{fspec}}, n)$ *consisting of a library* Σ *of functions, a domain* Domp *of values, a semantics function* $\texttt{Semp} : \Sigma \mapsto 2^{\texttt{Domp}^3}$*, a specification relation* $\phi_{\texttt{fspec}} \subseteq \texttt{Domp}^2$*, and an integer* n*. The goal is to find a straight-line program* P *of length* n *that only calls functions in* Σ *to compute a function that refines* $\phi_{\texttt{fspec}}$*; that is,* $\forall x, y : \texttt{Semp}(P)(x, y) \Rightarrow \phi_{\texttt{fspec}}(x, y)$*.*

3.1 Decorated-Component-based Synthesis

We now allow the library components $f \in \Sigma$ to be associated with additional constraint-generation rules, and introduce the problem of synthesizing straight-line programs (SLPs) that use such decorated components.

Let V denote all program variables. The semantics Semp interpreted V as elements in Domp. Now, let Domd be an alternate domain of values, and consider valuations $\sigma : V \mapsto \texttt{Domd}$ that interpret V in this new domain Domd. Each function $f \in \Sigma$ is given an alternate meaning:

$$\texttt{Semd} : \Sigma \mapsto 2^{\texttt{Domd}^3} \tag{2}$$

That is, if $f \in \Sigma$, then $\texttt{Semd}(f)$ is a ternary relation on Domd. Intuitively, if we use the statement $z = f(x, y)$ in a Program P, then we would require the existence of three values in Domd – one value tx associated with x, a value ty associated with y, and a value tz associated with z – such that $(tx, ty, tz) \in \texttt{Semd}(f)$. The alternate meaning of a SLP P is simply the conjunction of the alternate meaning of each statement.

$$\texttt{Semd}(P) = \{\sigma \in \texttt{Domd}^V \mid (\sigma(x), \sigma(y), \sigma(z)) \in \texttt{Semd}(f)\ \forall (z := f(x, y) \in P)\} \tag{3}$$

Definition 2 (Decorated-CoS, or DCoS). *A DCoS problem is an* 8*-tuple* $(\Sigma, \texttt{Domp}, \texttt{Semp}, \phi_{\texttt{fspec}}, n, \texttt{Domd}, \texttt{Semd}, \phi_{\texttt{dspec}})$*, where* $(\Sigma, \texttt{Domp}, \texttt{Semp}, \phi_{\texttt{fspec}}, n)$ *is a CoS problem,* Domd *is an alternate domain of values,* Semd *is a mapping* $\Sigma \mapsto 2^{\texttt{Domd}^3}$*, and* $\phi_{\texttt{dspec}} \subseteq \texttt{Domd}^2$ *is an additional constraint on input* x *and output* y*.*

The goal is to synthesize both a straight-line program P and a valuation $\sigma : V \mapsto$ $\texttt{Domd}$ *such that P solves the component-based synthesis problem* and *σ is a model of* $\texttt{Semd}(P)$ *and* $(\sigma(x), \sigma(y)) \in \phi_{\texttt{dspec}}$.

In a DCoS problem, the `Semp` part could be redundant (if $\phi_{\texttt{fspec}} = \texttt{Domp}^2$), or the `Semd` part could be redundant (if $\texttt{Semd}(f) = \texttt{Domd}^3$ and $\phi_{\texttt{dspec}} = \texttt{Domd}^2$). Hence, DCoS generalizes CoS, and supports `Semd`-only problem formulations too.

Note that decorations are useful to enforce nonfunctional constraints on the target program, such as a bound on the number of a component function to use.

4 Solving the Synthesis Problems

We solve the synthesis problems by converting them to an $\exists\forall$ constraint and using an off-the-shelf $\exists\forall$ SMT solver to solve the constraint. This approach was used in earlier work on component-based synthesis [16]. We note here that the decorated components introduce additional existential constraints, and hence, the overall synthesis constraint continues to be an $\exists\forall$ formula.

4.1 Component-based Program Synthesis as $\exists\forall$

Consider an instance of the CoS problem, depicted in Figure 3, where, for notational convenience, we fixed $n = 9$. Synthesizing the program amounts to finding values for the 9 variables $f_1, \ldots, f_9$ from the set Σ, and values for the 18 variables $a_{11}, a_{12}, \ldots, a_{91}, a_{92}$ from the set $\{0, 1, \ldots, 8\}$. If the value of a_{ij} is k, then it means the j-th argument of the function call on Line i is equal to x_k.

We have the following well-formedness constraint on the a_{ij} variables, which guarantees that the synthesized programs will indeed be a SLP.

$$\phi_1 = \bigwedge_{i \in 1..9} (a_{i1} < i \ \wedge \ a_{i2} < i). \qquad \text{More generally, } \phi_1 = \bigwedge_{i \in 1..9} \bigwedge_{j \in 1..\texttt{arity}(f_i)} a_{ij} < i$$

With each left-hand side variable $x_1, \ldots, x_9$ in the program sketch in Figure 3, we associate one first-order variable vx_i, which denotes the value in `Domp` of x_i. The following constraint imposes consistency of vx_i values with respect to the semantics `Semp`.

$$\phi_2 = \bigwedge_{\substack{i,j,k \in 1..9 \\ f \in \Sigma}} (a_{i1} = j \wedge a_{i2} = k \wedge f_i = f) \ \Rightarrow \ (vx_j, vx_k, vx_i) \in \texttt{Semp}(f)$$

The constraint above says that if the first argument of the functional call on Line i comes from Line j, the second argument comes from Line k, and the function on Line i is $f \in \Sigma$, then the value vx_i should be such that $(vx_j, vx_k, vx_i) \in \texttt{Semp}(f)$.

We are now ready to write our $\exists\forall$ *synthesis constraint* $\Phi_{\exists\forall}$:

$$\exists f_1, \ldots, f_9 \in \Sigma \ \ \exists a_{11}, \ldots, a_{92} \in [0..8] \ \ (\phi_1 \ \wedge$$
$$\forall vx_0, \ldots, vx_9 \in \texttt{Domp} \ \ (\phi_2 \Rightarrow \phi_{\texttt{fspec}}(vx_0, vx_9)))$$

The satisfiability of $\Phi_{\exists\forall}$ is equivalent to the existence of an instance of the sketch in Figure 3 that satisfies the functional requirement $f_{\texttt{spec}}$. Thus, we can solve the CoS problem by generating the above formula and solving it using an $\exists\forall$ solver, as described in [16].

4.2 Decorated-Component-Based Program Synthesis as ∃∀

Let $tx_0, tx_1, \ldots, tx_9$ denote new variables (interperted over $\texttt{Domd}$) corresponding to the 10 lines in the program sketch shown in Figure 3. (We assume program P does not assign twice to the same variable, so there is a 1-1 correspondence between program variables and program lines.) The following constraint imposes consistency of the $\texttt{Domd}$ values (assigned to the new variables) with respect to the semantics $\texttt{Semd}$.

$$\phi_3 = \bigwedge_{\substack{i,j,k \in 1..9 \\ f \in \Sigma}} (a_{i1} = j \wedge a_{i2} = k \wedge f_i = f) \;\; \Rightarrow \;\; (tx_j, tx_k, tx_i) \in \texttt{Semd}(f)$$

Now, the decorated-component-based synthesis problem reduces to satisfiability of the following exists-forall formula $\Psi_{\exists\forall}$:

$$\begin{array}{l} \exists f_1, \ldots, f_9 \in \Sigma \;\; \exists a_{11}, \ldots, a_{92} \in [0..8] \\ \quad \exists tx_0, \ldots, tx_9 \in \texttt{Domd} \;\; (\phi_1 \;\wedge \phi_3 \;\wedge\; \phi_{\texttt{dspec}}(tx_0, tx_9) \wedge \\ \qquad \forall vx_0, \ldots, vx_9 \in \texttt{Domp} \;\; (\phi_2 \Rightarrow \phi_{\texttt{fspec}}(vx_0, vx_9))) \end{array}$$

where $\phi_{\texttt{fspec}}(vx_0, vx_9)$ captures the functional requirement and $\phi_{\texttt{dspec}}(tx_0, tx_9)$ captures the alternate requirement.

The following claim follows from definition of the two synthesis problems and noting that ϕ_2 captures $\texttt{Semp}(P)$ and ϕ_3 captures $\texttt{Semd}(P)$.

Proposition 1. *The CoS problem, respectively DCoS problem, has a solution (a desired program) iff the constraint $\Phi_{\exists\forall}$, respectively $\Psi_{\exists\forall}$, is satisfiable.*

5 Component-based Synthesis Using ∃ Solving

Our main result is that, in some cases, given a CoS problem, one can design a decoration for the components that is an "abstraction" of its primary semantics. Such a decoration allows us to completely ignore the main functional specification while performing synthesis. Since the function specification was the only source of ∀ in the synthesis constraint, the synthesis constraint simplifies to an ∃ constraint, which can be solved using standard SMT solvers [24, 30].

Consider any program P. Let V be the set of program variables in P, and let V be partitioned into $I \uplus O$, where I are the input variables, and O are the variables defined in P. Let $\texttt{PSp}$ denote the set of all program states $\texttt{Domp}^V$, and let $\texttt{PSd}$ denote the set of all alternate states $\texttt{Domd}^V$. A *concretization function* γ is a mapping from the set $\texttt{PSd}$ to the powerset $2^{\texttt{PSp}}$.

Definition 3. *A set $Sd \subseteq \texttt{PSd}$ is an* abstraction *of a set $Sp \subseteq \texttt{PSp}$ with respect to a concretization function γ and a subset $W \subseteq V$ of variables, if*

$$Sp|_W \;\subseteq \bigcap_{\theta \in Sd} \gamma(\theta)|_W$$

where $X|_Y$ denotes the projection of the set X onto the Y components (that is, we consider assignments to the variables in Y and ignore the other variables).

Remark 1. In sharp contrast to Definition 3, recall that in the usual notion of abstraction, we say Sp_2 is an abstraction of Sp_1 if $Sp_1 \subseteq \bigcup_{s \in Sp_2} \gamma(s)$.

Definition 4. *The alternate semantics* $\texttt{Semd}$ *is an* abstraction *of the primary semantics* $\texttt{Semp}$ *in a program* P *if there exists a* concretization *function* $\gamma : \texttt{PSd} \to 2^{\texttt{PSp}}$ *such that for every* $Sp \subseteq \texttt{PSp}$*, for every* $Sd \subseteq \texttt{PSd}$*, if* Sd *is an abstraction of* Sp *w.r.t* γ *and* I*, then* $\texttt{Semd}(P)(Sd)$ *is an abstraction of* $\texttt{Semp}(P)(Sp)$ *w.r.t* γ *and* $I \uplus O$*.*

Remark 2. The definition of abstraction of programs given in Definition 4 is similar to the usual notion of abstraction: if we start with an abstraction of initial states, and apply the abstract transformer, we should get back an abstraction of the concretely transformed initial states. Definition 4 says the same thing, but with the difference that we restrict to the set I when checking abstraction on the initial states, and use the new notion of when a set of alternate program states is said to abstract a set of primary program states (Definition 3).

We note that Definition 4 allows us to compose programs while preserving abstractions if the composed programs modify a disjoint set of variables. More precisely, if the decoration of P_1 is an abstraction, and the decoration of P_2 is an abstraction, then the decoration of $P_1; P_2$ is an abstraction too, under the assumption that P_2 does not change the value of any variable in P_1 (and only treats those values as its inputs).

The main point of having a decoration that is an abstraction is that now, if we can find an interpretation for the program in the alternate semantics, then we know the program is functionally correct in its primary semantics.

Theorem 1. *Let* $\phi_{\texttt{fspec}}$ *and* $\phi_{\texttt{dspec}}$ *be primary and alternate specifications such that* $\gamma(\{\sigma \mid (\sigma(x_0), \sigma(x_9)) \in \phi_{\texttt{dspec}}\}) \subseteq \{\sigma \mid (\sigma(x_0), \sigma(x_9)) \in \phi_{\texttt{fspec}}\}$*. If* $\texttt{Semd}$ *is an abstraction of* $\texttt{Semp}$ *(as in Definition 4) with respect to the concretization function* γ*, then, whenever* $\phi_{\texttt{dspec}}$ *holds in* P*, then* $\phi_{\texttt{fspec}}$ *holds in* P*.*

The main consequence of Theorem 1 is that now we can solve a CoS problem, which is an $\exists\forall$ problem, by checking satisfiability of an existentially quantified constraint (no quantifier alternation). We can do this only if we have a decoration $\texttt{Semd}$ that is an abstraction of $\texttt{Semp}$. Given such an $\texttt{Semd}$, we can solve the CoS problem by checking satisfiability of the following existential formula $\Phi_{\exists\exists}$:

$$\begin{aligned} &\exists f_1, \ldots, f_9 \in \Sigma \;\; \exists a_{11}, \ldots, a_{92} \in [0..8] \\ &\quad \exists tx_0, \ldots, tx_9 \in \texttt{Domd} \;\; (\phi_1 \;\wedge \phi_3 \;\wedge\; \phi_{\texttt{dspec}}(tx_0, tx_9)) \end{aligned}$$

This formula is the same as $\Psi_{\exists\forall}$, but with all references to $vx_0, \ldots, vx_9$ removed. Since these were the only universally quantified variables, we get rid of the $\forall$ and get the above *quantifier-alternation-free synthesis constraint*, which can be solved using existing Satisfiability Modulo Theory (SMT) solvers [24, 30].

Theorem 1 can be viewed a "weak" form of duality because it constructs an $\exists$ formula that implies a $\forall$, but not vice-versa. Also, it must be understood as a template for meta-theorems that argue that a given decoration enables $\exists\exists$ synthesis, such as the one that we used in our example of Section 2.

If we use enumeration over all possible values to check a $\forall$ verification condition, we may find a violation of the $\forall$ formula after some finite search and thus, we may find a bug. If we use enumeration over all possible values to check the

sufficient $\exists$ formula, we may find a suitable valuation of the exists-variables after some finite search, and thus we may find a proof (for the $\forall$ formula). Hence, our notion of abstraction here, and the resulting weak duality in Theorem 1 has an interesting use in program verification: it replaces a "search for bugs" approach (violation of $\forall$) by a "search for proofs" approach (satisfiability of the dual $\exists$).

One may wonder if program analysis community has ever implicitly used Theorem 1 to perform verification. The answer is yes: template-based methods for verification, also called constraint-based verification [17, 31], are an instance of the weak duality principle. We next outline a template-based technique to construct abstract decorations, which can be used to solve CoS problems.

5.1 Constructing Abstract Decorations

We describe a generic approach for constructing abstract decorations. Note that we followed this recipe when constructing the decoration for our secure multiplication example in Section 2. Let us assume we have an abstract domain `PSa`; for example, one over which we could have created an abstract interpreter, or performed predicate abstraction. Let us see how we would generate decorations from `PSa`. Let us say we have proof rules that generate valid Hoare triples $\{\phi_1\}P\{\phi_2\}$ over the abstract states, where P is a program, ϕ_1, ϕ_2 are elements of `PSa`. Now, to define the decoration `Semd`, we first parameterize the elements of `PSa`. Say, we have a template $\Phi(\boldsymbol{u})$ that contains parameters $\boldsymbol{u}$ such that

$$\texttt{PSa} = \{\Phi(\boldsymbol{c}) \mid \boldsymbol{c} \in \texttt{Domd}\}, \text{ for some set } \texttt{Domd}$$

In other words, we can generate all abstract program states by instantiating the parameters $\boldsymbol{u}$ from the set `Domd`. The set `Domd` forms our alternate domain. If $l1, l2, \ldots$ are all the program locations (nodes in the program graph), then $\boldsymbol{u}_{l1}, \boldsymbol{u}_{l2}, \ldots$ are our new program variables that are interpreted over `Domd`. Finally, we need to define the alternate meaning `Semd` for each program statement. This is achieved by considering proof rules comprising of valid Hoare triples $\{\phi_1\}z := f(x, y)\{\phi_2\}$, and trying to generate a constraint $\psi_f(\boldsymbol{u}, \boldsymbol{v})$ such that

$$\forall \boldsymbol{u}, \boldsymbol{v} : \psi_f(\boldsymbol{u}, \boldsymbol{v}) \;\Rightarrow\; \{\Phi(\boldsymbol{u})\}z := f(x, y)\{\Phi(\boldsymbol{v})\}$$

If we can find such a ψ_f (not equivalent to *false*) forall $f \in \Sigma$, then $(\psi_f)_{f\in\Sigma}$ defines `Semd`. By construction, `Semd` is an abstraction of `Semp`. An example of this process of constructing an abstract `Semd` can be found in Figure 2.

We would like to emphasize two points here. First, the task of constructing an abstract decoration `Semd` will not succeed always, because we may not find such ψ_f. Second, while abstract decorations are a powerful concept, decorations that are *not* abstractions of `Semp` also prove to be immensely useful, especially in the application to synthesis, where they can be used to capture nonfunctional properties. This latter use of decorations was explored in [32], and reused here.

6 Cryptographic Schemes

In this section, we present some examples of cryptographic schemes that we synthesized using the DCoS framework. These are summarized in Table 1. Our

synthesis tool takes as input a program sketch, such as the one in Section 2, multiple primal semantics (`Semp`), and multiple decorations (`Semd`) on components, along with requirements specified on these semantics. It solves the synthesis problem by generating and solving either the $\Phi_{\exists\forall}$ formula (in case there are some primal semantics) or the $\Phi_{\exists\exists}$ formula (in case there are only decorations on components). Our tool, along with all the examples and corresponding SMT instances, is available at [14].

Synthesis problem	Search space size	Solution size	Decorations	Synthesis time
BC modes	$\sim 1 \times 10^9$	11 lines	DEC + SEC	$\sim$1s
Secure multiplication	$\sim 2 \times 10^{13}$	21 lines	SPOLY + RAND	$\sim$50s
Oblivious transfer	$\sim 2 \times 10^7$	9 lines	SARITH	$\sim$1s
Du-Atallah multiplication	$\sim 3 \times 10^{19}$	11 lines	SPOLY + RAND	$\sim$1s
Dining Cryptographers	$\sim 2 \times 10^{25}$	12 lines	RAND + SBOOL	$\sim$1s

Table 1. Summary of examples of synthesized cryptographics schemes. Details on the BC modes, secure multiplication, and oblivious transfer examples are given in Sections 6.1, 6.2, and 6.3, respectively. The sketches of all examples are available at [14]. The decorations SPOLY and RAND are the ones introduced informally in Section 2, DEC and SEC are described in Section 6.1 and were inspired by the work of [22], SARITH corresponds to symbolic arithmetic expressions (used to approximate operations in a group), and SBOOL corresponds to symbolic Boolean expressions.

$$\frac{u = f(v,w), \quad \texttt{know}(v), \quad \texttt{know}(w)}{\texttt{know}(u)} \text{ if } f \text{ is known} \quad \Bigg| \quad \frac{u = v \oplus w, \quad \texttt{know}(u), \quad \texttt{know}(w)}{\texttt{know}(v)}$$

Fig. 4. Decryptability check: Assuming that initially only the encrypted message is "known", we can apply the above rules to check if message m can be "known". A k-step (bounded size) proof search can be encoded using decorated components by having *k-length arrays* of alternate values for each program variable.

6.1 Block Cipher Modes

Block ciphers are keyed, invertible functions that map a fixed length bit string (say 128 bits) to a random bit string of the same length. A *block cipher mode* Enc uses a block cipher to encrypt messages longer than this fixed length. We have *two* requirements: *correctness* of Enc, which is captured by the existence of a decryption algorithm Dec such that $\forall k, m : Dec_k(Enc_k(m)) = m$, and *security*,

which is expressed by the fact that no adversary with oracle access to *Enc* is able to learn anything about random ciphertexts.

Malozemoff et al. [22] proposed an algorithm for synthesis of block cipher modes that follows the enumerate-and-check paradigm. The algorithm proceeds by carefully enumerating candidate straight-line programs and checking correctness and security for each of them. The security property is approximated using a labeling system that guarantees that if a candidate straight-line program can be labeled satisfying certain constraints, then it implements a secure block cipher. The search for the *existence* of a correct labeling is then implemented using an SMT solver. Regarding correctness, the authors propose a fix-point algorithm analogous to our encoding of decryptability check as a decoration; see Figure 4.

We used both the $\exists\forall$ and the $\exists\exists$ approach to synthesize block cipher modes, which highlights the flexibility of the DCoS framework. Our formulation of the problem is analogous to the one in [22]; that is, our sketches do not provide additional "hints" to the synthesis tool. In the $\exists\forall$ approach, we specify correctness directly using a primal semantics; that is, we synthesize ($\exists$) both an encryption scheme *Enc and* a decryption scheme *Dec* such that for all ($\forall$) input messages m, $Dec(Enc(m)) = m$. By having a primal semantics for specifying correctness, and a decoration for specifying security, we solved the synthesis problem by generating and solving the $\Psi_{\exists\forall}$ formula shown in Section 4. The $\exists\forall$ approach has two main drawbacks: first, solving $\Psi_{\exists\forall}$ turned out to be expensive because it required us to synthesize two programs at once, *Dec* and *Enc*. Moreover, it requires us to specify primal semantics for the block cipher function itself. This is not ideal, since a bad choice might be a source of unsoundness in the decryptability check.

The new $\exists\exists$ approach, enabled by Theorem 1, addresses both these issues. The crucial observation is that the correctness check used in [22] is in fact an instance of the weak duality of Theorem 1, and hence it can be encoded as a second decoration. Hence, to ensure correctness, it is not necessary to synthesize a decryption scheme, but instead check for "decryptability" (Figure 4). The new $\exists\exists$ approach resulted in a reduction in running time from $\sim$100 seconds (using the $\Psi_{\exists\forall}$ approach) to $\sim$1 second, to synthesize well-known encryption schemes such as CBC, OFB, CFB, OFB, and PCBC. Moreover, another benefit is that we can leverage the incremental solving capabilities of SMT solvers, such as Yices and Z3, to efficiently find hundreds of variants of block cipher modes. Our $\exists\exists$ approach found hundreds of correct modes of operation in less than 5 minutes on a regular laptop, including all the common ones mentioned above.

6.2 Secure Multiplication

In Section 2, we presented an application of our synthesis methodology to synthesize a secure 2-party computation multiplication protocol. The synthesis time for the sketch described in Section 2 is $\sim$ 50 seconds. As explained above, our synthesis tool takes as input a sketch of the solution, i.e. a description of a finite family of protocols $\mathcal{F}$ in this case, and searches for a protocol $P \in \mathcal{F}$ that satisfies the requirements.

Figure 5 reports running times and approximated search space size, i.e. $|\mathcal{F}|$, for 30 variants of our sketch presented in Section 2.

The first 15 variants of our sketch are satisfiable (blue trace in the plot), and were obtained in the following way: we started from a sketch whose only completion is the solution reported in Section 2; that is, $|\mathcal{F}| = 1$, and then increasingly relaxed it until we obtained a most general one.

Hence, the leftmost data point of the satisfiable instances corresponds to simply a verification check. The second one corresponds to a sketch where everything is fixed but the first line of A's program. In subsequent data points (3)-(15), the part of the protocol to be determined is (3) messages from C, (4) messages from C and B, (5) arithmetic operations in A and B, (6) arithmetic operations in C, and messages from C and B, (7) arithmetic operations, (8) arithmetic operations and messages from B, (9) arithmetic operations and messages from C and A, (10) arithmetic operations and messages from C and B, (11) arithmetic operations in A and B, and program for C, (12) arithmetic operations in A, and programs for C and B, (13) everything but first line of A's program, and programs for C and B, (14) programs for A, B, and C, (15) programs for A, B, and C, letting A have a total of 4 lines.

The unsatisfiable instances are obtained from (1)-(15) by adding the additional restriction that C cannot use multiplication. This prevents C from generating appropriately correlated random data, which results in unsatisfiable sketches.

Although it is difficult to make definitive statements about the behaviour of SMT solvers, the plot in Figure 5 confirms a tendency that we have often observed: our approach scales well for satisfiable instances, and hence using general sketches spanning a large $\mathcal{F}$ is fine as long as $\mathcal{F}$ contains a solution. On the other hand, if the synthesis problem is not realizable, proving so for large families of programs may not scale well.

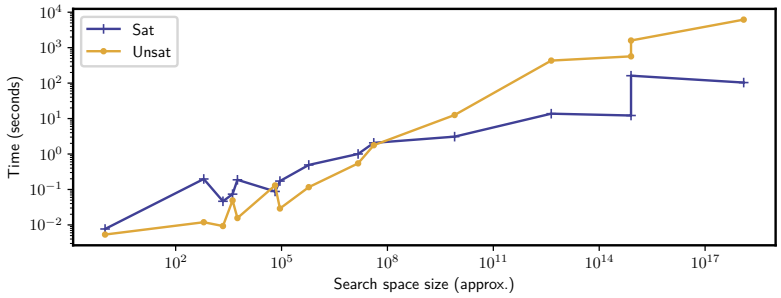


Fig. 5. Running times for satisfiable and unsatisfiable variants of the secure multiplication sketch. The experiments were run using Yices as backend solver on a 2.30GHz machine with 8Gb of memory.

6.3 Oblivious Transfer

In the two-party version of oblivious transfer (OT), one party, the `Sender`, has two messages m_0 and m_1, and the other party, the `Chooser`, can pick which message she wants to receive. The goal of oblivious transfer is to achieve this transfer of message from `Sender` to the `Chooser`, but with the requirements that (a) the `Sender` does not learn the choice made by the `Chooser`, and (b) the `Chooser` does not learn the content of the other message (that was not chosen).

We wish to base the protocol on the decisional Diffie-Hellman (DDH) assumption [6]: given a cyclic group with generator g, the DDH assumption states that (g^a, g^b, g^{ab}) is computationally indistinguishable from (g^a, g^b, g^c) for randomly and independently chosen elements a, b, c from $\mathbb{Z}$. We provide a sketch to the synthesis tool that consists of four blocks of straight-line code (executed by `Sender`, `Chooser`, `Sender`, `Chooser` in turns), where the `Sender` and the `Chooser` are allowed use of upto 3 random numbers each.

While approches based on $\exists\forall$ paradigm timed out due to the complexity of the protocol, we were able to perform $\exists\exists$ synthesis by using only suitably designed decorations. We synthesized two different OT protocols: the first one was also recently reported in [8], and the second one is the well-known Naor-Pinkas protocol [25]. The solutions were obtained in about 1 and 100 seconds, respectively, on a regular laptop using Yices as backend solver.

Due to technical difficulties in formalizing the security requirements, we used approximate requirements that eliminated a large number insecure protocols, but not necessarily all of them. Consequently, there is a need here for *a posteriori* verification of security of the synthesized scheme (using other verification tools; such as, Easycrypt [4]). Program synthesis, however, remains a fast and effective tool to quickly generate plausible schemes.

7 Conclusion

We formulated the decorated-component-based synthesis framework and showed how component decorations can be used to enable a weak duality principle, which allows us to replace a desired $\forall$ check by a stronger $\exists$ check. Besides its applications to speed up program synthesis, it is important to recognize the use of this duality principle in different verification techniques, such as constraint-based verification [17, 31]. Decorations can abstract the concrete meaning, and thus provide sufficient checks for functional properties. They can also be unrelated to the concrete meaning, and encode nonfunctional properties of programs.

It is worth emphasizing that decorations are not abstract interpreters [9]: in abstract interpretation, assertion checking is still a "forall" check (just over abstract values). In contrast, decorations on components behave as constraints, and hence our extension of primal semantics with decorations has flavors of constraint programming [11] and combining inductive and co-inductive constructs [1].

Exploring extension of DCoS to programs with loops, designing decorations to encode more sophisticated proof systems, and studying algebraic properties of decorations remain future challenges.

References

1. A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In *40th ACM Symp. Principles of Prog. Lang., POPL*, 2013.
2. 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 *Formal Methods in Computer-Aided Design, FMCAD*, pages 1–17, 2013.
3. G. Barthe, J. M. Crespo, C. Kunz, B. Schmidt, B. Gregoire, Y. Lakhnech, and S. Zanella-Beguelin. Fully automated analysis of padding-based encryption in the computational model, 2013. `http://www.easycrypt.info/zoocrypt/`.
4. G. Barthe, F. Dupressoir, B. Gregoire, C. Kunz, B. Schmidt, and P. Strub. Easycrypt: a tutorial. In *Foundations of security analysis and design vii*, volume 8604 of *Lecture notes in computer science*, page 146???166. Springer, 2014.
5. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, pages 192–206, 2008.
6. D. Boneh. The decision Diffie-Hellman problem. In *Proc. Third Algorithmic Number Theory Symposium*, volume 1423 of *LNCS*, pages 48–63, 1998.
7. B. Carmer and M. Rosulek. Linicrypt: A model for practical cryptography. In *Proc. 36th Intl. Cryptology Conf., CRYPTO*, 2016.
8. T. Chou and C. Orlandi. The simplest protocol for oblivious transfer. Cryptology ePrint Archive, Report 2015/267, 2015. `http://eprint.iacr.org/`.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. on Principles of Programming Languages, POPL*, pages 238–252, 1977.
10. W. Du and M. J. Atallah. Protocols for secure remote database access with approximate matching. In *E-Commerce Security and Privacy*, pages 87–111. 2001.
11. T. Felgentreff, T. Millstein, A. Borning, and R. Hirschfeld. Checks and balances: Constraint solving without surprises in object-constraint programming languages. In *Proc. Conf. on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2015.
12. J. Frankle, P. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, pages 802–815. ACM, 2016.
13. A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanovic, and S. Malik. Template-based circuit understanding. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 83–90. IEEE, 2014.
14. A. Gascón and A. Tiwari. Synudic: Synthesis using dual interpretation on components. `https://github.com/adriagascon/synudic`, 2016.
15. N. Gilboa. Two party RSA key generation. In *CRYPTO*, pages 116–129. Springer, 1999.
16. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proc. ACM Conf. on Prgm. Lang. Desgn. and Impl. PLDI*, pages 62–73, 2011.
17. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *Proc. ACM Conf. on Prgm. Lang. Desgn. and Impl. PLDI*, pages 281–292, 2008.
18. V. Hoang, J. Katz, and A. Malozemoff. Automated analysis and synthesis of authenticated encryption schemes. In *ACM CCS*, 2015.

19. S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proc. ICSE (1)*, pages 215–224. ACM, 2010.
20. E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, pages 407–426. ACM, 2013.
21. Y. Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. `http://eprint.iacr.org/2016/046`.
22. A. J. Malozemoff, J. Katz, and M. D. Green. Automated analysis and synthesis of block-cipher modes of operation. In *IEEE 27th Computer Security Foundations Symposium, CSF*, pages 140–152. IEEE, 2014.
23. Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.
24. Microsoft Research. *Z3: An efficient SMT solver*. `http://research.microsoft.com/projects/z3/`.
25. M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms, SODA*, pages 448–457, 2001.
26. N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, pages 522–538. ACM, 2016.
27. C. Smith and A. Albarghouthi. Mapreduce program synthesis. In *PLDI*, pages 326–340. ACM, 2016.
28. A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.
29. A. Solar-Lezama, L. Tancau, R. Bodík, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.
30. SRI International. *Yices: An SMT solver*. `http://yices.csl.sri.com/`.
31. S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.
32. A. Tiwari, A. Gascón, and B. Dutertre. Program synthesis using dual interpretation. In *Proc. 25th Intl Conf on Automated Deduction, CADE*. Springer, 2015.