Automation in Cryptology

Final Report

Ashish Tiwari

Report Date: Oct 15, 2015
Subcontract no. 60106452-107484-C
Prime award no. N00014-12-1-0914

1 Executive Summary

The goal of the project was to develop new verification and synthesis techniques for programs that model cryptographic protocols. In this document, we report our main accomplishments.

We developed a new approach for synthesis that we used to successfully synthesize a variety of security schemes. The most significant accomplishments that led to the development of the new synthesis approach are as follows:

Dual interpretations in programs: We used *dual* interpretations to define type-based constraints on programs. *Primal* interpretations correspond to the usual semantics of programs, and specify how values are computed and propagated through a program. *Dual* interpretations map to constraints and can be used to specify user-defined typing rules. Dual interpretations are especially useful for cryptographic schemes since often it is possible to design a special type system that can capture properties related to security.

Program synthesis using *multiple* interpretations: We introduced the idea of assigning multiple, potentially mutually incomparable, interpretations to a single program. The different interpretations can be used to capture different correctness criteria for a program. For example, functional correctness can be specified in one interpretation, whereas resource constraints and security requirements could be specified in a second and third interpretation.

The Synudic tool: We built a program synthesis tool that support multiple interpretations, each of which can be a primal or a dual interpretation. We used to tool to synthesize several known algorithms for padding-based encryption, block cipher modes of operation, and oblivious transfer. The tool is available online (`www.csl.sri.com/~tiwari/softwares/auto-crypto/`).

The Yices2 tool: We introduced several new features and optimizations in our widely-used Satisfiability Modulo Theory (SMT) solver, Yices2, with the goal of better supporting our synthesis tool, Synudic. For example, one significant new feature in the latest release of Yices2 is the ability to check satisfiability of $\exists\forall$ formulas.

Program analysis-based optimizations for multiparty computation: We developed optimizations that can be used to improve efficiency of oblivious RAM-based multiparty computation.

We describe these key contributions in the subsequent sections. Further details about the technical work, including software, published papers and forthcoming papers, can be found on the website:

`http://www.csl.sri.com/~tiwari/softwares/auto-crypto/`

2 Dual interpretations in programs

Programs work with two kinds of entities: values and types. Values are manipulated during program executions, and users can program these manipulations in a flexible way. For example, users can define new functions and compose these functions to define more complex value manipulation routines. This is not the case for the other entity, namely types. Programming languages usually have a predefined set of base types, and even though a user can build more complex types, such as records, arrays, and lists, using the base types, there is still less flexibility available than for values; for example, a user typically cannot define rules (programs) for the manipulation of types.

We have developed a small domain-specific language, called Synudic, where users can define and manipulate type-like entities in conjunction with regular values. What is the behavior of Synudic programs? It has been pointed out by Cousot [Cou97] that there is a certain *duality* between values and types: specifically, there is a Galois connection between properties (on values) and types where, for example, set union in one lattice corresponds to set intersection in the other. More intuitively and less formally, a variable that has *more* types can take on *fewer* values and vice versa. For example, a variable that has types `even` and `pos_int` (union of types) can only be a positive even integer (intersection on values). In order to perform computation on type-like entities, we would need to perform *dual* operations: rather than union at control flow join points (that we do for values), we would perform intersection, and rather than least fixpoint computation in loops (that we do for values), we would perform greatest fixpoit computation. Our programs, semantically, perform unions at control flow join points for certain values, and simultaneously, semantically, perform intersection at control flow join points for certain other "type" values.

Types and values in traditional programming languages are strongly related to each other: a type represents a set of concrete values. SYNUDIC goes beyond this standard notion and allows user to specify arbitrary user-defined entities that behave like types do, but that *need not* be related in any way to the concrete semantics of the program. Let us call this more general notion *attributes*. In SYNUDIC, a program variable is mapped to not only a value, as in the usual semantics of programs, but also simultaneously to an attribute (or a set of attributes). A user can not only program how values are updated in a program, but also program the rules that govern the update of attributes. Thus, SYNUDIC enables programming with primal and dual semantics.

The obvious question to ask here is why should one be interested in programmable attributes and the dual semantics that govern their behavior. There are several reasons.

Using types in program synthesis. The primary motivation comes from the field of program synthesis. Suppose we are interested in synthesizing a 10 line program that meets some specification. Even for a relatively small expression language, the number of 10 line programs that can be constructed using that langauge will be huge. However, the number of well-typed programs could be significantly smaller. This point was also made in recent work on synthesis of functional programs [OZ15]. So, how to impose type constraints on a yet-to-be synthesized program? If $x := f(y)$ is the form of assignment statement on, say, line 3, where f is some *unknown* function symbol that needs to be chosen from the expression language, then *a priori* we do not know the types of x and y. However, if every value variable x can carry with it a type attribute (say, $x.0$ is the value and $x.1$ is the type), then we can encode typing constraints on the program by constraining the attributes $x.1$ and $y.1$ (the constraint will involve f). As a result, when solving the synthesis problem, we now synthesize both the program and a type correct annotation for it. Thus, we can use attributes and dual semantics to impose typing constraints on "program sketches". The dual semantics on the "type" attributes help prune the search space of possible programs.

Nonfunctional program properties. Since programs are assigned one concrete semantics that specifies the functional behavior of the program, we can easily state properties that relate to *functional correctness* of programs. Moreover, by using abstractions of the concrete semantics [CC77a], we can reason about the concrete semantics and perhaps prove functional correctness. However, we are interested in *nonfunctional properties* of programs – properties of programs that are not directly related to its concrete semantics. One example class of nonfunctional properties consists of properties related to security, such as, secure information flow [DD77]. Since these are nonfunctional properties, it is then no surprise that dedicated "type systems" that are unrelated to the concrete program are developed for reasoning about them [Smi01, MKG14]. The development of attributes and dual semantics enables programming of such nonstandard type systems in a general-purpose language, and potentially increase utility of programming languages in general.

Theoretical interest. We note an asymmetry in the complexity of assertion checking for imperative programs: for a polynomial time (PTime) program, the problem of checking if an assertion is valid is in co-NP: if the assertion is *violated*, then there is some input that causes assertion failure, and hence one could (nondeterministically) guess that input and check that the assertion is violated in PTime. For the same program, under reasonable assumptions, checking if an assertion on the dual semantics is valid happens to be in NP: we just need to guess the attribute for each variable and check that the guesses satisfy the rules defining the dual semantics.

2.1 Primal and Dual Semantics of Straight-line Programs

We consider straight-line programs, where each *statement* is of the form $x := f(\vec{x})$, where f is a symbol (representing a function or a constant) taken from a fixed signature Σ with arity n, x is a program variable, and $\vec{x}$ is a vector of n program variables. Note that f can be a constant with arity $n = 0$. Let us use $\texttt{Terms}(\Sigma, \texttt{Vars})$ to denote all terms constructed using the signature Σ and free variables $\texttt{Vars}$. A straight-line program is just a (sequentially composed) sequence of statements of the form above.

2.1.1 Primal Semantics of Straight-line Programs

In the *primal* semantics, the meaning of a term is given by a *structure* $(\texttt{Dom}, \texttt{Int})$ where the domain $\texttt{Dom}$ is a nonempty set, and the interpretation $\texttt{Int}$ maps
(a) every constant $c \in \Sigma$ to an element $c^{\texttt{Int}} \in \texttt{Dom}$, and
(b) every function symbol $f \in \Sigma$ with arity n to a concrete function $f^{\texttt{Int}} : \texttt{Dom}^n \mapsto \texttt{Dom}$.

For example, $\texttt{Dom}$ could be the set of integers, or machine integers, or length n bitvectors. If $\texttt{Dom}$ consists of bitvectors, $\texttt{Int}$ could map a symbol $\oplus$ in Σ to the bitwise xor operator.

The mapping $\texttt{Int}$ is extended to a mapping over all ground terms $\texttt{Terms}(\Sigma, \emptyset)$ in the usual way: specifically, if f is a arity n function symbol, then $f(t_1, \ldots, t_n)^{\texttt{Int}}$ is defined as $f^{\texttt{Int}}(t_1^{\texttt{Int}}, \ldots, t_n^{\texttt{Int}})$ recursively. Intuitively, the primal meaning of any term in $\texttt{Terms}(\Sigma, \emptyset)$ is just its evaluation (to a value in $\texttt{Dom}$) in the underlying structure $(\texttt{Dom}, \texttt{Int})$. We will treat applications of function $f \in \Sigma$ as "library" calls.

We now fix $\texttt{Vars}$ to be the set of all program variables. A (primal) *state* of a program is simply a mapping from the set $\texttt{Vars}$ of program variables to values in $\texttt{Dom}$.

$$\textit{Program states} \qquad \sigma : \texttt{Vars} \mapsto \texttt{Dom}$$

We use the symbol σ, possibly with subscripts or superscripts, to denote a program state. We note that a program state is just a substitution.

Note that $\texttt{Int}$ defines the meaning of all ground expressions in $\texttt{Terms}(\Sigma, \emptyset)$. A program state gives meaning to variables in $\texttt{Vars}$, and thus, a program state, together with the structure $(\texttt{Dom}, \texttt{Int})$, provides meaning to any expression in $\texttt{Terms}(\Sigma, \texttt{Vars})$.

Let P denote the single assignment statement $x := f(x_1, \ldots, x_k)$. The meaning of P in the primal semantics is defined by the strongest postcondition operation as expected:

$$\begin{aligned}\texttt{Sem}(P)(S) \;=\; & \{\sigma \mid \exists \sigma_1 \in S : \\ & \sigma(x) = f^{\texttt{Int}}(\sigma_1(x_1), \ldots, \sigma_1(x_k)), \\ & \sigma(y) = \sigma_1(y) \text{ forall } y \neq x\}\end{aligned}$$

$x := 0;$	$\Sigma = \{+, *, 0, 1.0, \ldots\}$	$\Sigma = \{+, *, 0, 1.0, \ldots\}$
$y := 1.0;$	$\texttt{Dom} = \mathbb{Z} \cup \mathbb{Q}$	$\texttt{DomB} = \{\texttt{zero}, \texttt{int}, \texttt{float}\}$
$z := y * x;$	$\texttt{Int} \sim$ arithmetic	$\texttt{IntB} =$ see text
$x := 1 + z;$	$\texttt{Sem} \sim$ finally x=1	$\texttt{SemB} \sim$ finally $x : \texttt{int}$
(a) Program	(b) Primal Semantics	(c) Dual Semantics

Figure 1: Illustrating primal and dual semantics on the same straight-line program. Assertion $x = 1$ holds at the end of the program in the primal semantics, whereas assertion $x : \texttt{int}$ (and the assertion $x : \texttt{float}$) holds at the end of the program in the dual semantics.

The meaning of a sequential composition $P; Q$ of two programs is also as expected:

$$\texttt{Sem}(P; Q)(S) \quad = \quad \texttt{Sem}(Q)(\texttt{Sem}(P)(S))$$

Thus, we get the (usual) primal semantics of straight-line programs.

2.1.2 Dual Semantics of Straight-line Programs

For defining the primal semantics, we started with a structure $(\texttt{Dom}, \texttt{Int})$ that provides meaning to the underlying expression language Σ. Just as in the primal case, the basic elements for the dual semantics is a pair $(\texttt{DomB}, \texttt{IntB})$.

Let $(\texttt{DomB}, \texttt{IntB})$ be such that

- $\texttt{IntB}$ maps a constant $c \in \Sigma$ to a *subset* $c^{\texttt{IntB}} \in 2^{\texttt{DomB}}$, and
- $\texttt{IntB}$ maps a function $f \in \Sigma$ of arity n to a function $f^{\texttt{IntB}} : \texttt{DomB}^n \mapsto 2^{\texttt{DomB}}$

Note that $c^{\texttt{IntB}}$ is not an element of $\texttt{DomB}$, but a subset of it. Similarly, $f^{\texttt{IntB}}$ is not a function over $\texttt{DomB}$, but really a relation. Once we have such a $\texttt{IntB}$, we can use it to give meaning to an assignment $x := f(\vec{x})$ as follows:

- the semantics $(x := f(x_1, \ldots, x_k))^{\texttt{IntB}}$ of an assignment statement $x := f(x_1, \ldots, x_k)$ is a set of program states, $\theta : \texttt{Vars} \mapsto \texttt{DomB}$, "consistent" with that assignment; that is,

$$\begin{aligned} &(x := f(x_1, \ldots, x_k))^{\texttt{IntB}} \\ &\quad = \quad \{\theta \mid \theta(x) \in f^{\texttt{IntB}}(\theta(x_1), \ldots, \theta(x_k))\} \end{aligned} \tag{1}$$

Intuitively, an assignment statement is a (constraint on the) set of (allowable dual) states.

Let P denote the single assignment statement $x := f(x_1, \ldots, x_k)$. The meaning of P in the dual semantics is given as follows:

$$\texttt{SemB}(P)(S) \quad = \quad S \cap (x := f(x_1, \ldots, x_k))^{\texttt{IntB}}$$

where the dual meaning $(x := f(x_1, \ldots, x_k))^{\texttt{IntB}}$ of the assignment statement is given in Equation 1.

In the dual semantics, the sequential composition operator is given meaning as in the primal case. As a result, note that the sequential composition operator behaves as a logical conjunction in the dual semantics.

2.2 Example: Straight-Line Program

Consider the simple four line program in Figure 1. The expressions used in the program are constructed using symbols from the signature $\Sigma = \{+, *, 0, 1.0, \ldots\}$. It is possible to give this program both a primal and a dual semantics. The primal semantics is the natural one: the symbols in Σ are interpreted over the domain `Dom` of integers ($\mathbb{Z}$) and floats ($\mathbb{Q}$) in the natural way. An assertion that is true (in the primal semantics) at the end of the program is $x = 1$.

We now give a second dual semantics to this same program. In particular, in the dual semantics, the symbols in Σ are interpreted over the new domain `DomB` consisting of just three elements, $\texttt{zero}, \texttt{int}$, and `float`. We have to now interpret the symbols in Σ over this new domain. One possible interpretation `IntB` is as follows:

$$\begin{array}{rcl}
\texttt{IntB}(0) & = & \{\texttt{zero}, \texttt{int}, \texttt{float}\} \\
\texttt{IntB}(1) & = & \{\texttt{int}, \texttt{float}\} \\
\texttt{IntB}(1.0) & = & \{\texttt{float}\} \\
\texttt{IntB}(+) & = & \{(e, \texttt{zero}) \mapsto \{e\}, \cdots\} \\
\texttt{IntB}(*) & = & \{(e, \texttt{zero}) \mapsto \{\texttt{zero}, \texttt{int}, \texttt{float}\}, \cdots\}
\end{array}$$

where e denotes an arbitrary element of `DomB`. Note that the constant 0 is not interpreted as a constant from `DomB`, but as a subset of `DomB`. Similarly, the function $+$ is interpreted as a function from $\texttt{DomB} \times \texttt{DomB}$ to $2^{\texttt{DomB}}$. In this example, the interpretations are just abstractions of the primal semantics.

So, in the dual semantics,
(1) x is constrained to take a (nondeterministically chosen) value in `DomB` in

Line 1,
(2) y is constrained to take value `float` in Line 2,
(3) z is constrained to take a (nondeterministically chosen) value in `DomB` in Line 3, and finally,
(4) x is *additionally* constrained to take a value in the set $\{\texttt{int}, \texttt{float}\}$ in Line 4.
Apart from the nondeterministic interpretation of constants and functions in the dual semantics, a second key difference is in the interpretation of nondeterminism. It is treated in an angelic manner. Hence, at the end of the program, the assertion $x : \texttt{int}$ is true in the dual semantic – because there is (at least) one assignment of variables to dual values consistent with all the four constraints and in which x is assigned `int`. Due to the angelic nature of nondeterminism, the assertion $x : \texttt{float}$ also holds at the end of the program. This is the reason for using the notation $x : e$ when a variable takes a value e in the dual semantics.

2.3 Duality in the Assertion Checking Problem

We can define the assertion checking problem on the two semantics.

Definition 1 (Assertion checking problem) *Given a program P and a primal assertion $x = c$, where x is a program variable and $c \in \texttt{Dom}$ is a constant, the* primal assertion checking problem *seeks to determine if $\sigma(x) = c$ for all program states σ in $\texttt{Sem}(P)(U)$.*

Given a program P and a dual assertion $x : c$, where x is a program variable and $c \in \texttt{DomB}$ is a constant, the dual assertion checking problem *seeks to determine if $\sigma(x) = c$ for some program state σ in $\texttt{Sem}(P)(U)$.*

The duality between the primal and the dual interpretations is exhibited in the following result.

Proposition 1 *If the primal interpretation structure $(\texttt{Dom}, \texttt{Int})$ is polynomial-time computable, then the primal assertion checking problem for straight-line programs is co-NP complete.*

If the dual interpretation structure $(\texttt{DomB}, \texttt{IntB})$ is polynomial-time computable, then the dual assertion checking problem for straight-line programs is NP complete.

For details and proofs, the reader is referred to the article on the webpage mentioned above.

3 Program synthesis using multiple interpretations

One of the key insights underlying SYNUDIC is that we can constrain the synthesis search space using *multiple*, possibly *unrelated*, interpretations. Each interpretation can be used to specify some requirement. Each interpretation can be primal or dual.

As we describe in Section 4, synthesis in presence of a primal interpretation results in an $\exists\forall$ formula, whereas synthesis in presence of a dual interpretation results in an $\exists$ formula. If a SYNUDIC sketch has requirements that are specified using both primal and dual interpretations, then we need to solve an $\exists\forall$ formula to synthesize the desired program.

However, in many cases – and certainly in the security schemes discussed in this report – it is possible to trade a primal requirement with a dual requirement. This is a form of *duality*, akin to Farkas lemma in arithmetic [Ber01], but for assertion checking in programs. This is very beneficial, since it enables us to use a solver for $\exists$ formulas, rather than one for an $\exists\forall$ formulas, to perform program synthesis. SYNUDIC supports the use of this duality principle for performing synthesis.

4 The Synudic tool

We have developed a tool, also called SYNUDIC, for synthesizing straight-line programs using multiple interpretations, where each interpretation can be a primal interpreation or a dual interpretation.

Synudic ($\underline{\text{Syn}}$thesis $\underline{\text{u}}$sing $\underline{\text{d}}$ual $\underline{\text{i}}$nterpretation on $\underline{\text{c}}$omponents) consists of (a) a language for specifying program synthesis problems with multiple requirements, (b) a compiler that converts the synthesis problem into an $\exists\forall$ constraint, and (c) invocation to the solver YICES2 to get a solution of the $\exists\forall$ problem. We briefly describe the step (b) in this section.

Consider the problem of synthesizing a straight-line program P with N lines. We assume we have two requirements: one functional and one nonfunctional requirement. The functional correctness requirement states that on input x_0, the program P computes $f_{\mathtt{spec}}(x_0)$ for some given function $f_{\mathtt{spec}}$. We assume that we are given a signature Σ with a primal interpretation $\mathtt{Int}$ over some domain $\mathtt{Dom}$, and $f_{\mathtt{spec}}$ is a unary function on $\mathtt{Dom}$.

The nonfunctional requirement states that the output variable can be assigned a value $e \in \mathtt{DomB}$, given a dual interpretation $(\mathtt{DomB}, \mathtt{IntB})$ for Σ.

We assume that the length N of the program is a given constant. For notational convenience, fix $N = 9$. The form of the program we wish to synthesize is shown in Figure 2. Without loss of generality, we have assumed that all function symbols in Σ have arity 2. 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 $\{0, 1, \ldots, 9\}$. The meaning of variables a_{ij} is given as follows: if $a_{ij} = k$, then 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.

$$\phi_1 \quad = \quad \bigwedge_{i \in 1..9} (a_{i1} < i \ \wedge \ a_{i2} < i)$$

The constraint above says that a value should be defined (on Line a_{i1} and on Line a_{i2}) *before* it is used (on Line i). With each left-hand side variable $x_1, \ldots, x_9$ in the program sketch in Figure 2, we associate two first-order variables:
(a) vx_i is the value in `Dom` of x_i in the primal semantics, and
(b) tx_i is the value in `DomB` of x_i in the dual semantics.

The following constraint imposes consistency of vx_i values with respect to the primary semantics.

$$\phi_2 \quad = \quad \bigwedge_{\substack{i \in 1..9 \\ j,k \in 1..9 \\ f \in \Sigma}} \begin{array}{l}(a_{i1} = j \wedge a_{i2} = k \wedge f_i = f \ \Rightarrow \\ \qquad vx_i = f^{\texttt{Int}}(vx_j, vx_k))\end{array}$$

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 is $f^{\texttt{Int}}(vx_j, vx_k)$.

The following constraint imposes consistency of attributes with respect to the dual semantics.

$$\phi_3 \quad = \quad \bigwedge_{\substack{i \in 1..9 \\ j,k \in 1..9 \\ f \in \Sigma}} \begin{array}{l}(a_{i1} = j \wedge a_{i2} = k \wedge f_i = f \ \Rightarrow \\ \qquad tx_i \in f^{\texttt{IntB}}(tx_j, tx_k))\end{array}$$

If we ignore the nonfunctional requirement, then satisfiability of the following $\exists\forall$ formula will indicate existence of a program of the form in Figure 2 that satisfies the functional requirement.

$$\begin{array}{l}\exists f_1, \ldots, f_9 \in \Sigma \ \ \exists a_{11}, \ldots, a_{92} \in [0..8] \ \ (\phi_1 \ \wedge \\ \qquad \forall vx_0, \ldots, vx_9 \in \texttt{Dom} \ \ (\phi_2 \Rightarrow vx_9 = f_{\texttt{spec}}(vx_0)))\end{array}$$

The Synudic tool generates the above $\exists\forall$ formula and solves it to synthesize straight-line programs. However, it can do more. Let us now consider the nonfunctional requirements. We need to include the constraint ϕ_3

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

Variables: Domain
$f_1, \ldots, f_9 : \Sigma$
$a_{11}, \ldots, a_{92} : 0..9$
$vx_0, \ldots, vx_9 : \texttt{Dom}$
$tx_0, \ldots, tx_9 : \texttt{DomB}$

Figure 2: An arbitrary straight-line program with 9 lines. To synthesize such a program, we need to find values for the 9 variables $f_1, \ldots, f_9$ from the set Σ, and find values for the 18 variables $a_{11}, a_{12}, \ldots, a_{91}, a_{92}$ from the set $L = \{0, \ldots, 9\}$. The meaning of a_{41} is as follows: if a_{41} is 2, then the first argument of f_4 is x_2.

induced by the dual semantics on the tx_i variables.

$$
\begin{array}{l}
\exists f_1, \ldots, f_9 \in \Sigma \;\; \exists a_{11}, \ldots, a_{92} \in [0..8] \\
\quad \exists tx_0, \ldots, tx_9 \in \texttt{DomB} \;\; (\phi_1 \;\wedge \phi_3 \;\wedge\; tx_9 = e \;\wedge \\
\qquad \forall vx_0, \ldots, vx_9 \in \texttt{Dom} \;\; (\phi_2 \Rightarrow vx_9 = f_{\texttt{spec}}(vx_0)))
\end{array}
$$

The main point to note here is that the variables $tx_0, tx_1, \ldots$ are all existentially quantified, whereas $vx_0, vx_1, \ldots$ are universally quantified. The constraint ϕ_3 and the nonfunctional requirement $tx_9 = e$ are outside the scope of the $\forall$ quantifier, and hence, they prune the search space of valid programs, just like constraint ϕ_1.

The tool SYNUDIC starts with a sketch, generates an $\exists\forall$ formula and uses the solver YICES2 to solve that formula. The input language for specifying the sketch allows the user to provide multiple primal, and multiple dual, interpretations for the program, and add requirements (postconditions) for each interpretation.

4.1 Synthesizing padding-based schemes.

In public key cryptography, padding is the process of preparing a message for encryption. A modern form of padding is OAEP, which is often paired with RSA public key encryption. Padding schemes, and in particular OAEP, satisfy the goals of (1) converting a deterministic encryption scheme, e.g. RSA, into a probabilistic one, and (2) ensuring that a portion of the encrypted message cannot be decrypted without being able to invert the full encryption.

$$f(G(r)||(G(r) \oplus m))$$
$$f(r||(G(r) \oplus m))$$
$$f(G(r \oplus H(m))||(G(r \oplus H(m)) \oplus m))$$
$$f((r \oplus H(m))||(G(r \oplus H(m)) \oplus m))$$
$$f((G(r) \oplus m)||(H(G(r) \oplus m) \oplus r))$$

Figure 3: Some automatically synthesized padding-based encryption schemes: Here H, G are two unary Hash functions, $\oplus$ is the xor operator, $||$ denotes pairing operator, m denotes the input message, r denotes a random input, and finally f is the one-way encryption function that is applied to the "padded" message m.

We used SYNUDIC to synthesize several padding-based encryption schemes, which are shown in Figure 3. Our results are reminiscent of the results obtained by the synthesis tool Zoocrypt [BCK$^+$13]. For the SYNUDIC sketch used to synthesize the six schemes shown in Figure 3, the reader should consult the project webpage.

The key observation here is that we use a primal interpretation to ensure that any synthesized encryption scheme is decryptable: that is, we synthesize both an encryption scheme and a decryption scheme, and the functional requirement is that the result of applying encryption, followed by decryption, to an input message returns the same message. We use a second, dual, interpretation to encode the requirement that the result of applying encryption to a message m is something that is essentially "random". For this, we design a special-purpose type system that carries information on whether a value is essentially "random" or not.

4.2 Synthesizing Block Ciphers Modes of Operation

A *mode of operation* is a pair of algorithms that features the use of a symmetric block cipher algorithm (F, F^{-}), e.g. AES, to encrypt/decrypt amounts of data larger than a block. A secure mode of operation must provide the same level of security as its associated block cipher. For example, the encryption algorithm of the popular Cipher Block Chaining (CBC) mode is depicted in Figure 4. CBC, when equipped with a secure block cipher, provides IND\$-CPA security, i.e. an attacker cannot distinguish its output from an uniformly random string with significant probability (under certain constraints on the computational power of the attacker). Note that CBC

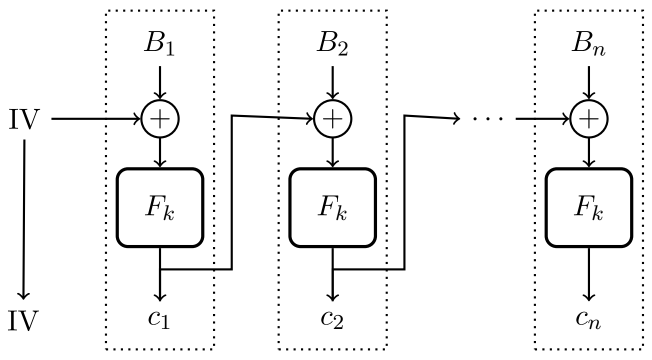

Figure 4: The CBC mode of operation for the encryption of an n-block message. The dotted boxes correspond to the multiple copies of the block processing procedure.

encryption consists of an initialization algorithm, where a random initialization vector IV is produced, followed by n copies of a block processing algorithm, while exactly one value is fed from one copy to the next one. This structure is common to many of the popular modes of operation.

Using our tool we could synthesize the well-known modes ECB, OFB, CFB, CBC, and PCBC, also automatically found in [MKG14], as well as some variants of those. When using our tool, we need to specify the number of lines in the protocol to synthesize. These values are presented by giving values to certain parameters that are used in Synudic sketch. The tables of Figure 5 show the size parameters needed to obtain each of the modes listed above. The reported times corresponds to a complete exploration of the search space that correspond to the parameters. For example, the second row of the first table means that, with parameters $na = 2$, $nb = 6$, $nc = 3$, it took our tool 6.07 seconds to conclude that *exactly* two instances of the sketch are secure and decryptable modes of operation. The parameter na is the number of lines in the initialization step, nb is the number of lines used in the block step, and nc is the number of lines used in the decryption routine. The modes marked with an asterisk (*) correspond to redundant variants of the corresponding mode.

Again, as in the case of padding-based schemes, the key observation behind using Synudic to synthesize modes of operation is as follows: we use a primal interpretation to ensure that any synthesized modes of operation are decryptable: that is, we synthesize both an encryption function and a decryption function, and the functional requirement is that the result of

Parameters			Modes	Time
na	nb	nc		(s)
2	4	3	CBC	3.3
2	6	3	CBC* OFB*	6.1
2	6	4	CBC*	22.9
2	6	5	CBC OFB* CFB	5.8

Parameters			Modes	Time
na	nb	nc		(s)
2	7	6	OFB variant CBC variant	6.3
2	8	5	CBC* OFB* CFB*	39.5
2	9	5	PCBC OFB variant	109.8

Figure 5: Results of the synthesis of block cipher modes of operation using Synudic.

applying encryption, followed by decryption, to an input message returns the same message. We use a second, dual, interpretation to encode the requirement that the result of applying encryption to a message m is something that is essentially "random". We used the type system developed in [MKG14] for this purpose: The main contribution in [MKG14] is a type system T that guarantees that type correct modes of operation are *secure*. Instead of separately filtering modes of operation that are not decryptable as done in [MKG14], we encoded the existence of a decoding algorithm as a *functional requirement*. This has the advantage that encryption algorithms are synthesized together with their corresponding decryption procedure.

4.3 Synthesizing Oblivious Transfer Protocols

We used SYNUDIC to synthesize two different ways to perform oblivious transfer. Both these protocols are known. We are currently using our tool to explore more of the space of protocols to discover new protocols.

The security of both protocols is based on the DDH assumption. In oblivious transfer, there are two parties, a Sender and a Chooser. The Sender has two messages m_0 and m_1, and the Chooser can chooose to get *one* of those two messages, but the Chooser does not wish to reveal his choice to the Sender, and the Sender does not want the Chooser to learn contents of *both* messages.

The primitives given to the synthesis procedure are: (1) each party can pick some random numbers; (2) each party can compute g^x, x^y, $x*y$ and x/y if they have access to x and y, where g is the generator of the cyclic group; (3) the Chooser is allowed access to an *if-then-else* statement, to enable him to make a choice.

The first protocol we synthesize, which was also recently reported in [?], is as follows:

- Sender picks random r and sends g^r to the Chooser.
- Chooser picks random a. Chooser makes a choice $i \in \{0, 1\}$.
  If $i = 0$, then the Chooser sends $g^r * g^a$ to the Sender.
  If $i = 1$, then the Chooser sends g^a to the Sender.
- Let x be the value received by Sender. Sender encrypts m_0 with key $k_0 = (\frac{x}{g^r})^r$, and encrypts m_1 with key $k_1 = x^r$.
- Let m_0^e, m_1^e denote the encrypted values received by Chooser. Chooser generates key $k = (g^r)^a$ and retrieves message m_i using that key. Note that k will be equal to k_i.

The second protocol we synthesized is the Naor-Pinkas oblivious transfer protocol [NP01].

- Chooser picks random a, b, c.
  Chooser makes a choice $i \in \{0, 1\}$.
  If $i = 0$, Chooser sends $\langle g^a, g^b, g^{ab}, g^c \rangle$.
  If $i = 1$, Chooser sends $\langle g^a, g^b, g^c, g^{ab} \rangle$.
- Let $\langle x, y, z, w \rangle$ be the tuple received by the Sender.
  Sender picks random r, s.
  Sender encrypts m_0 with key $k_0 = z^r * (g^b)^s$
  Sender encrypts m_1 with key $k_1 = w^r * (g^b)^s$
  Sender computes helper key $k = x^r * g^s$.
  Sender sends $\langle k, m_0^e, m_1^e \rangle$
- Chooser retrieves message m_i by decrypting m_i^e using the key k^b (which will be equal to k_i).

One of the challenges in synthesizing oblivious transfer is capturing the requirements of the protocol. It is easy to capture the requirement that the Chooser is able to decrypt the message m_i, depending on the choice i; but it is difficult to capture the requirement that the Chooser be *unable* to decrypt the other message m_{1-i}; and that the Sender does not *learn* the choice i made by the Chooser. For the requirements that were difficult to capture accurately in SYNUDIC, we used "approximations" that were enough to invalidate several "obviously wrong" protocols, but were not guaranteed to pick only the "truly secure" protocols.

Because of the approximations involved in capturing the security property in some cases, in our applications of SYNUDIC to security, there is often an *a posteriori* need for establishing security of the synthesized scheme using other dedicated verification tools; such as, Easycrypt [BDG+14].

5 The Yices2 tool

The SYNUDIC synthesis tool uses the YICES2 Satisfiability Modulo Theory (SMT) solver as its backend engine. YICES2 is an SMT solver that decides the satisfiability of formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, bitvectors, scalar types, and tuples.

One of the key features in YICES2 developed to support SYNUDIC is the $\exists\forall$ solver, which decides satisfiability of formulas of the form $\exists\vec{x} : \forall\vec{y} : \phi(\vec{x}, \vec{y})$. As we observed before, synthesis problems are naturally mapped into $\exists\forall$ formulas.

YICES2 is a widely-used SMT solver, and its current version 2.4.1 was released in August 2015. More information can be found at `http://yices.csl.sri.com/`.

6 Optimizations for ORAM-based secure Multiparty Computation

Oblivious Random Access Memories (ORAMs), introduced by Goldreich and Ostrovsky [GO96], allow a client to outsource its data to a server while preserving privacy. A crucial observation is that encrypting the outsourced data is not enough to maintain privacy, since access patterns may reveal information and hence must be kept hidden from the server and an eavesdropping adversary.

Another use of ORAMs is to speed up generic approaches to Secure Multiparty Computation (MPC). Frameworks for compiling general programs into an ORAM-based MPC computation framework are already available (see, for example [LHS+14]). Our work focuses in this application area. More concretely, we use static analysis and compiler optimization techniques to (i) learn memory access patterns of a given program to obtain a more efficient ORAM-based secure multiparty implementation, and (ii) apply program transformation that are suited for MPC.

6.1 A compiler for Multiparty secure computation in the RAM model

In secure Two-party Computation, two parties A and B jointly compute the value of a function f over their inputs while keeping the inputs private. A common restriction on f is that it must have *finite domain*, and hence A and B must fix a bound on the size of their inputs. Moreover, A and B must of course agree on a particular formalism to describe the function f.

In our work, f is described as a program in the C programming language, augmented with a type system to qualify every variable of a given program as *public* or *secret*. To avoid information leakage between the parties, our type system also enforces that loop conditions do not depend on secret variables (neither directly nor indirectly). Moreover, procedure calls are not allowed.

Our system can be seen as a *compiler* for Two-party computation that, given a C program P, produces an *efficient implementation of P* into a protocol for secure two-party computation (in the semi-honest adversary model). Our focus is in efficiency in terms of running time.

Many generic protocols for secure MPC rely on a program representation whose execution is data-agnostic (or data-oblivious), in the sense that the control flow of the program does not depend on the parties' private data. This is the case for both Boolean and arithmetic circuits.

We are interested in ORAM-based secure computation: protocols for secure computation that leverage Oblivious Random Access Memories (ORAMs) to emulate the RAM model of computation in a secure (and distributed) way. The main advantage of this approach is that the execution does not need to be data-agnostic, since it involves a random access memory. This has two implications, one from the control flow perspective and another for the data access perspective:

- In a given execution, not every instruction of the program needs to be evaluated, as opposed to protocols that represent programs exclusively as circuits (such as garbled circuits or protocols based on fully homomorphic encryption), where partial evaluation of the circuit is not possible. In contrast, it has been already shown that ORAMs can be used to construct protocols with sublinear (amortized) running time [GKK+12].
- An access into an array $T[i]$, where i is qualified as secret, must be implemented in linear time in a completely data-agnostic manner, whereas efficient ORAMs implement dynamic access with polylogarithmic cost (in an amortized notion).

6.2 Program representation

Our compiler implements several program transformations and analysis that operate on LLVM bitcode which, roughly speaking, corresponds to a Single Static Assignment (SSA) representation of the input program. The SSA form is a well-understood standard intermediate representation in compiler design. Hence, our compiler for Secure MPC can benefit from existing general static analysis techniques such as abstract interpretation [CC77b], symbolic Execution [CGK$^+$11], and data-flow analysis (see [NNH99]), as well as optimizing program transformations [BGS94] to produce an efficient implementation of ORAM-based MPC protocol.

The main interesting property of ORAMs is that they provide dynamic access with polylogarithmic cost, as mentioned above, while not revealing any information of the access pattern across several accesses. Hence, intuitively, for any sequence of accesses into an oblivious RAM M, M hides from the parties involved in the computation (1) what position is being accessed each time and (2) how many times is a position accessed. However, if we could infer from the program description that every position in M is never accessed twice, then the condition (2) above can be ignored, which enables a more efficient implementation.

6.3 Monotone accesses

```
secret int T[n][n];
int i = n;
int j = n;
for(int x = 0; n ; ++i){
    v = T[i][j];
    if(v == 1) { --i; --j; ...
    } else if(v == 2){ --i; ...
    } else { --j; ... }
  }
```

Note that the instruction $v = T[i][j]$ in the example program above will never be executed with the same values of i and j across all executions of the loop, and hence it satisfies the property mentioned above. Hence, having the parties involved in the computation shuffle T suffices to guarantee that the access $T[i][j]$ is oblivious. In other words, we do not need the full power of ORAMs.

It is not difficult to see that checking this property is undecidable in general, but in many practical cases that arise, for example, in dynamic programming algorithms, it can be done practically. This fact is partially due to the enormous recent advance in constraint solving technology, and in particular Satisfiability Modulo Theories (SMT). To implement this analysis, we leverage *symbolic execution techniques* enabled by efficient SMT solvers. More concretely, we encode the semantics of the program as a logic formula so that the property that the value of the pair of indexes (i, j) strictly increases (decreases) at every iteration of the loop (in absence of integer overflows) is equivalent to validity of a formula ϕ. Whether ϕ is valid is then checked using an SMT solver such as Yices [Dut14]. Thanks to the SSA representation this process, including the extraction of loops in the original program can be fully automated in our compiler.

6.4 Data compaction

A simple program transformation that leads to a running time optimization in Secure MPC is *data compaction*. Let us introduce this transformation by means of an example. Consider the following piece of code. By "some secret condition" we mean an expression over variables qualified as secret whose evaluation must be kept hidden to all the parties involved in the computation.

```
j=0;
for(i=0 to n-1)
{  // ...compute x...
   if(some secret condition) result[j++] = x;
   // more code
}
```

In essence, the code above corresponds to a push-only stack. It is a common construct in dynamic programming algorithms, specially in the result extraction phase (consider, for example, the classic dynamic programming approach to the shortest common subsequence problem). The goal of the data compaction transformation is to rewrite the above code into something like the following.

```
j=0;
for(i=0 to n-1)
{ // ...
  temp[j]=x;
  condition[j]=(some secret condition);
  // ...
  ++j;
}
result = obliviousDataCompaction(temp, condition)
}
```

The oblivious `obliviousDataCompaction` function returns the array

$$[temp[j] \mid condition[j],\ 0 \leq j < n]$$

and hence the transformation is correct. Moreover, the call

$$\texttt{obliviousDataCompaction}(temp, condition)$$

can be implemented obliviously in $O(nlogn)$ time using oblivious sorting implemented using a sorting network. This gives a speed up with respect to the cost of n oblivious memory lookups of index j, which would have a polylogarithmic cost.

6.5 Value analysis and instruction alignment

As mentioned above, state of the art ORAM implementations add a polylogarithmic overhead, with respect to the size of the ORAM, to every memory lookup. Recall that in our system we use ORAMs to implement the RAM model of computation (also known as Von Neumann architecture). In that model, both data and instructions reside in memory, and hence we incur in such overhead not only every time an array is accessed in a program, but also when the next instruction, indicated by a global variable “program counter”, is retrieved from memory.

6.5.1 Instruction alignment

Note that the reason for which memory accesses must remain oblivious to all parties involved in the computation even for the instructions memory is that, when executing a program that branches on secret conditions, *exposing which instruction is being executed may leak information about secret inputs.* However, using an easy structural analysis on the SSA graph representing the computation we can easily compute the set of instructions that might be executed at a given execution step, *for some input.* That analysis leads to a partition of the set of instruction that represents *which program instruction must be indistinguishable from each other.* Hence, instead of having all instructions reside in a single ORAM, me can have multiple ORAMs “banks” containing such instructions. This transformation may result in a running time speed up for some programs, since the polylogarithmic cost of retrieving an ORAM bank depends on its size. For small enough banks a single scan of the whole bank might be more efficient than an ORAM lookup and hence it can be implemented as regular memory. Additionaly, we have to take into account also the cost of retrieving what bank must be accessed at each execution step, but this information is public.

Tipically, instruction scheduling in compiler optimization is used to improve instruction-level parallelism, which improves performance on machines with instruction pipelines. We are implementing instruction reordering (and dummy instruction insertion) in our compiler as well, but with the goal of obtaining a "good" partition of the instruction ORAM into banks that results in a speed-up in the task of retrieving the next instruction of the execution.

Obtaining a good partition of the instruction memory of the program has further implications than the one stated above. Note that, once an instruction has been retrieved from memory, it still has to be executed. In our setting, such execution usually boilds down to running an instance of Yao's garbled circuit protocol, which involves representing the program to be executed as a Boolean circuit. Note that, to maintain obliviousness with respect to which instruction in a given bank is executed, such circuit must describe *all instructions in the bank*. For example, if instructions $v3 \;=\; add\ v1\ v2$ and $v6 \;=\; sub\ v4\ v5$ form a bank, for some variables of the program v_i, whenever $v3 \;=\; add\ v1\ v2$ has to be executed, the protocol will have to run Yao's protocol on a circuit implementing both addition and subtraction. This fact is specially relevant for memory access instructions, since they are significantly more costly than, for example, logic operations. Hence, a primary goal of our instruction alignment heuristics is to enforce that memory access intructions are assign to the same memory bank whenever that is possible.

6.5.2 Value analysis of index variables

Using a similar argument than the one used for instruction memory, data memory can be safely partitioned if we can statically determine that certain memory locations, for every possible execution, will never be accessed at the same execution step. For example, in the ith execution step of a binary search on an array v of length n, only indexes in teh set $\{j \mid j \equiv 0\ mod\ \lceil \frac{n}{2^i} \rceil\}$ might be accessed, inducing a partition v into several memory "banks" as in the previous section.

This kind of analysis boilds down to value analysis of index variables on, for example, arrays. A lot of effort has been devoted to this task in program analysis, with the different goal of identifying buffer overflows and null pointers. A prominent technique in that setting is abstract interpretation [CC77b]. Specific constraints of secure MPC, such as the fact that the execution time is known for every execution, may allow us to use variants of these kind of techniques that are easier to automate in a general setting.

7 Conclusion

Our key technical contribution consisted of the idea of performing program synthesis using multiple interpretations, where each interpretation can be a primal or a dual interpretation. The main outcomes include the software SYNUDIC and the new version of the SMT solver YICES2, along with the SYNUDIC sketches for the cryptographic schemes that were used to synthesize encryption schemes, modes of operation, and oblivious transfer protocols.

In the future, we plan to continue to build upon the work above, and explore synthesis techniques on more examples, as well as continue work on optimizing secure multiparty computation using program analysis techniques.

References

[BCK$^+$13] 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/`.

[BDG$^+$14] 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 146166. Springer, 2014.

[Ber01] Leonard D. Berkovitz. *Convexity and Optimization in* $\mathbb{R}^n$. John Wiley & Sons, 2001.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[CC77a] 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.

[CC77b] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.

[CGK+11] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1066–1071, 2011.

[CO15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. Cryptology ePrint Archive, Report 2015/267, 2015. `http://eprint.iacr.org/`.

[Cou97] P. Cousot. Types as abstract interpretations. In *Proc 24th ACM Symp. Principles of Programming Languages, POPL*, pages 316–331, 1997.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[Dut14] B. Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 513–524, 2012.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[LHS$^+$14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient ram-model secure computation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 623–638, 2014.

[MKG14] Alex J. Malozemoff, Jonathan Katz, and Matthew 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.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 1999.

[NP01] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms, SODA*, pages 448–457, 2001.

[OZ15] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proc. 36th ACM SIGPLAN Conf on Programming Language Design and Implementation*, pages 619–630, 2015.

[Smi01] Geoffrey Smith. A new type system for secure information flow. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 115–125. IEEE Computer Society, 2001.