Wys*: A DSL for Verified Secure Multi-party Computations

Aseem Rastogi[1(✉)], Nikhil Swamy[2], and Michael Hicks[3]

[1] Microsoft Research, Bangalore, India
aseemr@microsoft.com
[2] Microsoft Research, Redmond, USA
nswamy@microsoft.com
[3] University of Maryland, College Park, USA
mwh@cs.umd.edu

Abstract. Secure multi-party computation (MPC) enables a set of mutually distrusting parties to cooperatively compute, using a cryptographic protocol, a function over their private data. This paper presents Wys*, a new domain-specific language (DSL) for writing *mixed-mode* MPCs. Wys* is an embedded DSL hosted in F*, a verification-oriented, effectful programming language. Wys* source programs are essentially F* programs written in a custom MPC effect, meaning that the programmers can use F*'s logic to verify the correctness and security properties of their programs. To reason about the distributed runtime semantics of these programs, we formalize a deep embedding of Wys*, also in F*. We mechanize the necessary metatheory to prove that the properties verified for the Wys* source programs carry over to the distributed, multi-party semantics. Finally, we use F*'s extraction to extract an interpreter that we have proved matches this semantics, yielding a partially verified implementation. Wys* is the first DSL to enable formal verification of MPC programs. We have implemented several MPC protocols in Wys*, including private set intersection, joint median, and an MPC-based card dealing application, and have verified their correctness and security.

1 Introduction

Secure multi-party computation (MPC) enables two or more parties to compute a function f over their private inputs x_i so that parties don't see each others' inputs, but rather only see the output $f(x_1, ..., x_n)$. Using a trusted third party to compute f would achieve this goal, but in fact we can achieve it using one of a variety of cryptographic protocols carried out only among the participants [12,26,58,65]. One example use of MPC is private set intersection (PSI): the x_i could be individuals' personal interests, and the function f computes their intersection, revealing which interests the group has in common, but not any interests that they don't. MPC has also been used for auctions [18], detecting tax fraud [16], managing supply chains [33], privacy preserving statistical analysis [31], and more recently for machine learning tasks [19,21,30,38,44].

F. Nielson and D. Sands (Eds.): POST 2019, LNCS 11426, pp. 99–122, 2019.
https://doi.org/10.1007/978-3-030-17138-4_5

Typically, cryptographic protocols expect f to be specified as a boolean or arithmetic circuit. Programming directly with circuits and cryptography is painful, so starting with the Fairplay project [40] many researchers have designed higher-level domain-specific languages (DSLs) for programming MPCs [6,14,17,19,23,27,29,34,37,39,45,48,49,52,56,61]. These DSLs compile source code to circuits which are then given to the underlying cryptographic protocol. While doing this undoubtedly makes it easier to program MPCs, these languages still have several drawbacks regarding both security and usability.

This paper presents Wys*, a new MPC DSL that addresses several problems in prior DSLs. Unlike most previous MPC DSLs, Wys* is not a standalone language, but is rather an embedded DSL hosted in F* [59], a full-featured, verification-oriented, effectful programming language. Wys* has the following two distinguishing elements:

1. A program logic for MPC (Sects. 2 and 3). In their most general form, MPC applications are *mixed-mode*: they consist of parties performing (potentially different) local, in-clear computations (e.g. I/O, preprocessing inputs) interleaved with joint, secure computations. Wys* is the first MPC DSL to provide a program logic to formally reason about the *correctness and security* of such applications, e.g., to prove that the outputs will not reveal too much information about a party's inputs [41].[1]

To avoid reasoning about separate programs for each party, Wys* builds on the basic programming model of the Wysteria MPC DSL [52] that allows applications to be written as a single specification. Wys* presents a *shallow embedding* of the Wysteria programming model in F*. When writing Wys* source programs, programmers essentially write F* programs in a new Wys effect, against a library of MPC combinators. The pre- and postcondition specifications on the combinators encode a program logic for MPC. The logic provides *observable traces*—a novel addition to the Wysteria semantics—which programmers can use to specify security properties such as delimited release [55]. Since Wys* programs are F* programs, F* computes verification conditions (VCs) for them which are discharged using Z3 [2] as usual.

We prove the soundness of the program logic—that the properties proven about the Wys* source programs carry over when these programs are run by multiple parties in a distributed manner—also in F*. The proof connects the pre- and postconditions of the Wys* combinators to their distributed semantics in two steps. First, we implement the combinators in F*, proving the validity of their pre- and postconditions against their implementation. Next, we reason about this implementation and the distributed runtime semantics through a deep embedding of Wys* in F*. Essentially, we deep-embed the Wys* combinator abstract syntax trees (ASTs) as an F* datatype and formalize two operational semantics for them: a conceptual single-threaded semantics that models their

[1] Our attacker model is the "honest-but-curious" model where the attackers are the participants themselves, who play their roles in the protocol faithfully, but are motivated to infer as much as they can about the other participants' secrets by observing the protocol. Section 2.3 makes the security model of Wys* more precise.

F* implementation, and the actual distributed semantics that models the multi-party runs of the programs. We prove, in F*, that the single-threaded semantics is sound with respect to the distributed semantics (Sect. 3). While we use F*, the program logic is general and it should be possible to embed it in other verification frameworks (e.g., in Coq, in the style of Hoare Type Theory [46]).

2. A full-featured, partially verified implementation (Sect. 3). Wys*'s implementation is, in part, formally verified. The hope is that formal verification will reduce the occurrence of security threatening bugs, as it has in prior work [15,36,50,63,64].

We define an interpreter in F* that operates over the Wys* ASTs produced by a custom F* extraction for the Wys effect. While the local computations are executed locally by the interpreter, the interpreter compiles secure-computation ASTs to circuits, on the fly, and executes them using the Goldreich, Micali and Wigderson (GMW) multi-party computation protocol [26]. The Wys* AST (and hence the interpreter) does not "bake in" standard F* constructs like numbers and lists. Rather, inherited language features appear abstractly in the AST, and their semantics is handled by a foreign function interface (FFI). This permits Wys* programs to take advantage of existing code and libraries available in F*.

To prove the interpreter behaves correctly, we prove, in F*, that it correctly implements the formalized distributed semantics. The circuit library and the GMW implementation are not verified—while it is possible to verify the circuit library [4], verifying a GMW implementation is an open research question. But the stage is set for verified versions to be plugged into the Wys* codebase. We characterize the Trusted Computing Base (TCB) of the Wys* toolchain in Sect. 3.5.

Using Wys* we have implemented several programs, including PSI, joint median, and a card dealing application (Sect. 4). For PSI and joint median we implement two versions: a straightforward one and an optimized one that improves performance but increases the number of adversary-observable events. We formally prove that the optimized and unoptimized versions are equivalent, both functionally and w.r.t. privacy of parties' inputs. Our card dealing application relies on Wys*'s support for secret shares [57]. We formally prove that the card dealing algorithm always deals a fresh card.

In sum, Wys* constitutes the first DSL that supports proving security and correctness properties about MPC programs, which are executed by a partially verified implementation of a full-featured language. No prior DSL provides these benefits (Sect. 5). The Wys* implementation, example programs, and proofs are publicly available on Github at https://github.com/FStarLang/FStar/tree/stratified_last/examples/wysteria.[2]

2 Verifying and Deploying Wys* Programs

We illustrate the main concepts of Wys* by showing, in several stages, how to program, optimize, and verify the two-party joint median example [32,53].

[2] This development was done on an older F* version, but the core ideas of what we present here apply to the present version as well.

In this example, two parties, Alice and Bob, each have a set of n distinct, locally sorted integers, and they want to compute the median of the union of their sets without revealing anything else; our running example fixes $n = 2$, for simplicity.

2.1 Secure Computations with as_sec

In Wys*, as in its predecessor Wysteria [52], an MPC is written as a single specification that executes in one of the two *computation modes*. The primary mode is called sec mode. In it, a computation is carried out using an MPC protocol among multiple principals. Here is the joint median in Wys*:

```
1 let median a b in_a in_b =
2   as_sec {a, b} (fun () → let cmp = fst (reveal in_a) > fst (reveal in_b) in
3                           let x3 = if cmp then fst (reveal in_a) else snd (reveal in_a) in
4                           let y3 = if cmp then snd (reveal in_b) else fst (reveal in_b) in
5                           if x3 > y3 then y3 else x3)
```

The four arguments to median are, respectively, principal identifiers for Alice and Bob, and Alice and Bob's secret inputs expressed as tuples. In Wys*, values specific to each principal are *sealed* with the principal's name (which appears in the sealed container's type). As such, the types of in_a and in_b are, respectively, sealed {a} (int ∗ int) and sealed {b} (int ∗ int). The as_sec ps f construct indicates that thunk f should be run in sec mode among principals in the set ps. In this mode, the code has access to the secrets of the principals ps, which it can reveal using the reveal coercion. As we will see later, the type of reveal ensures that parties cannot reveal each others' inputs outside sec mode.[3] Also note that the code freely uses standard F* library functions like fst and snd. The example extends naturally to $n > 2$ [3].

To run this program, both Alice and Bob would start a Wys* interpreter at their host and direct it to run the median function Upon reaching the as_sec thunk, the interpreters coordinate with each other to compute the result using the underlying MPC protocol. Section 2.5 provides more details.

2.2 Optimizing median with as_par

Although median gets the job done, it can be inefficient for large n. However, it turns out if we reveal the result of comparison on line 2 to both the parties, then the computation on line 3 (resp. line 4) can be performed locally by Alice (resp. Bob) without the need of cryptography. Doing so can massively improve performance: previous work [32] has observed a 30× speedup for $n = 64$.

This optimized variant is a *mixed-mode* computation, where participants perform some local computations interleaved with small, jointly evaluated secure computations. Wys*'s second computation mode, par mode, supports such mixed-mode computations. The construct as_par ps f states that each principal in ps should locally execute the thunk f, simultaneously; any principal not in

[3] The runtime representation of sealed a v at b's host is an opaque constant • (Sect. 2.5).

the set ps simply skips the computation. Within f, while running in par mode, principals may engage in secure computations via as_sec.

Here is an optimized version of median using as_par:

```
1 let median_opt a b in_a in_b =
2   let cmp = as_sec {a, b} (fun () → fst (reveal in_a) > fst (reveal in_b)) in
3   let x3 = as_par {a} (fun () → if cmp then fst (reveal in_a) else snd (reveal (in_a))) in
4   let y3 = as_par {b} (fun () → if cmp then snd (reveal in_b) else fst (reveal (in_b))) in
5   as_sec {a, b} (fun () → if reveal x3 > reveal y3 then reveal y3 else reveal x3)
```

The secure computation on line 2 *only* computes cmp and returns the result to both the parties. Line 3 is then a par mode computation involving only Alice in which she discards one of her inputs based on cmp. Similarly, on line 4, Bob discards one of his inputs. Finally, line 5 compares the remaining inputs using as_sec and returns the result as the final median.

One might wonder whether the par mode is necessary. Could we program the local parts of a mixed-mode program in normal F*, and use a special compiler to convert the sec mode parts to circuits and pass them to a GMW MPC service? We could, but it would complicate both writing MPCs and formally reasoning that the whole computation is correct and secure. In particular, programmers would need to write one program for each party that performs a different local computation (as in median_opt). The potential interleaving among local computations and their synchronization behavior when securely computing together would be a source of possible error and thus must be considered in any proof. For example, Alice's code might have a bug in it that prevents it from reaching a synchronization point with Bob, to do a GMW-based MPC. For Wys*, the situation is much simpler. Programmers may write and maintain a single program. This program can be formally reasoned about directly using a SIMD-style, "single-threaded" semantics, per the soundness result from Sect. 3.4. This semantics permits reasoning about the coordinated behavior of multiple principals, without worry about the effects of interleavings or wrong synchronizations. Thanks to par mode, invariants about coordinated local computations are directly evident since we can soundly assume the lockstep behavior (e.g., loop iterations in the PSI example in Sect. 4).

2.3 Embedding a Type System for Wys* in F*

Designing high-level, multi-party computations is relatively easy using Wysteria's abstractions. Before trying to run such a computation, we might wonder:

1. Is it *realizable*? For example, does a computation that is claimed to be executed only by some principals ps (e.g., using an as_par ps or an as_sec ps) only ever access data belonging to ps?
2. Is it *correct?* For example, does median_opt correctly compute the median of Alice and Bob's inputs?
3. Is it *secure*? For example, do the optimizations in median_opt, which produce more visible outputs, potentially leak more about the inputs?

By embedding Wys* in F* and leveraging its extensible, monadic, dependent type-and-effect system, we address each of these three questions. We define a new indexed monad called Wys for computations that use MPC combinators as_sec and as_par. Using Wys along with the sealed type, we can ensure that protocols are realizable. Using F*'s capabilities for formal verification, we can reason about a computation's correctness. By characterizing observable events as part of Wys, we can define trace properties of MPC programs to reason about their security.

To elaborate on the last: we are interested in *application-level* security properties, assuming that the underlying cryptographic MPC protocol (GMW [26] in our implementation) is secure. In particular, the Wys monad models the *ideal* behavior of sec mode—a secure computation reveals only the final output and nothing else. Thus the programmer could reason, for example, that optimized MPC programs reveal no more than their unoptimized versions. To relate the proofs over ideal functionality to the actual implementation, as is standard, we rely on the security of the cryptographic protocol and the composition theorem [20] to postulate that the implementation securely realizes the ideal specification.

The Wys monad. The Wys monad provides several features. First, all DSL code is typed in this monad, encapsulating it from the rest of F*. Within the monad, computations and their specifications can make use of two kinds of *ghost state*: *modes* and *traces*. The mode of a computation indicates whether the computation is running in an as_par or in an as_sec context. The trace of a computation records the sequence and nesting structure of outputs of the jointly executed as_sec expressions—the result of a computation and its trace constitute its observable behavior. The Wys monad is, in essence, the product of a reader monad on modes and a writer monad on traces [43,62].

Formally, we define the following F* types for modes and traces. A mode Mode m ps is a pair of a mode tag (either Par or Sec) and a set of principals ps. A trace is a forest of trace element (telt) trees. The leaves of the trees record messages TMsg x that are received as the result of executing an as_sec thunk. The tree structure represented by the TScope ps t nodes record the set of principals that are able to observe the messages in the trace t.

```
type mtag = Par | Sec
type mode = Mode: m:mtag → ps:prins → mode
type telt = TMsg : x:α → telt | TScope: ps:prins → t:list telt → telt
type trace = list telt
```

Every Wys* computation e has a monadic computation type Wys t pre post. The type indicates that e is in the Wys monad (so it may perform multi-party computations); t is its result type; pre is a precondition on the mode in which e may be executed; and post is a postcondition relating the computation's mode, its result value, and its trace of observable events. When run in a context with mode m satisfying the precondition predicate pre m, e may produce the trace tr, and if and when it returns, the result is a t-typed value v validating post m v tr. The style of indexing a monad with a computation's pre- and postcondition is a standard technique [7,47,59]—we defer the definition of the monad's bind and return to

the actual implementation and focus instead on specifications of Wys* specific combinators. We describe as_sec, reveal, and as_par, and how we give them types in F*, leaving the rest to the online technical report [54]. By convention, any free variables in the type signatures are universally prenex quantified.

Defining as_sec in Wys*

```
1 val as_sec: ps:prins → f:(unit → Wys a pre post) → Wys a
2   (requires (fun m → m=Mode Par ps ∧ pre (Mode Sec ps)))
3   (ensures (fun m r tr → tr=[TMsg r] ∧ ∃t. post (Mode Sec ps) r t)))
```

The type of as_sec is *dependent* on the first parameter, ps. Its second argument f is the thunk to be evaluated in sec mode. The result's computation type has the form Wys a (requires ϕ) (ensures ψ), for some precondition and postcondition predicates ϕ and ψ, respectively. We use the requires and ensures keywords for readability—they are not semantically significant.

The precondition of as_sec is a predicate on the mode m of the computation in whose context as_sec ps f is called. For all the ps to jointly execute f, we require all of them to transition to perform the as_sec ps f call simultaneously, i.e., the current mode must be Mode Par ps. We also require the precondition pre of f to be valid once the mode has transitioned to Mode Sec ps—line 2 says just this.

The postcondition of as_sec is a predicate relating the initial mode m, the result r:a, and the trace tr of the computation. Line 3 states that the trace of a secure computation as_sec ps f is just a singleton [TMsg r], reflecting that its execution reveals only result r. Additionally, it ensures that the result r is related to the mode in which f is run (Mode Sec ps) and some trace t according to post, the postcondition of f. The API models the "ideal functionality" of secure computation protocols (such as GMW) where the participants only observe the final result.

Defining reveal in Wys*. As discussed earlier, a value v of type sealed ps t encapsulates a t value that can be accessed by calling reveal v. This call should only succeed under certain circumstances. For example, in par mode, Bob should not be able to reveal a value of type sealed {Alice} int. The type of reveal makes the access control rules clear:

```
val unseal: sealed ps α → Ghost α

val reveal: x:sealed ps α → Wys α
  (requires (fun m → m.mode=Par ⟹ m.ps ⊆ ps ∧ m.mode=Sec ⟹ m.ps ∩ ps ≠ ∅))
  (ensures (fun m r tr → r=unseal x ∧ tr=[]))
```

The unseal function is a Ghost function, meaning that it can only be used in specifications for reasoning purposes. On the other hand, reveal can be called in the concrete Wys* programs. Its precondition says that when executing in Mode Par ps', *all* current participants must be listed in the seal, i.e., ps' ⊆ ps. However, when executing in Mode Sec ps', only a subset of current participants is required: ps' ∩ ps ≠ ∅. This is because the secure computation is executed jointly by all of ps', so it can access any of their individual data. The postcondition of reveal relates the result r to the argument x using the unseal function.

Defining as_par in WYS*

```
1 val as_par: ps:prins → (unit → Wys a pre post) → Wys (sealed ps a)
2   (requires (fun m → m.mode=Par ∧ ps ⊆ m.ps ∧ can_seal ps a ∧ pre (Mode Par ps)))
3   (ensures (fun m r tr → ∃t. tr=[TScope ps t] ∧ post (Mode Par ps) (unseal r) t)))
```

The type of as_par enforces the current mode to be Par, and ps to be a subset of current principals. Importantly, the API scopes the trace t of f to model the fact that any observables of f are only visible to the principals in ps. Note that as_sec did not require such scoping, as there ps and the set of current principals in m are the same. The can_seal predicate enforces that a is a zero-order type (i.e. closures cannot be sealed), and that in case a is already a sealed type, its set of principals is a subset of ps.

2.4 Correctness and Security Verification

Using the Wys monad and the sealed type, we can write down precise types for our median and median_opt programs, proving various useful properties. We discuss the statements of the main lemmas and the overall proof structure. By programming the protocols as a single specification using the high-level abstractions provided by WYS*, our proofs are relatively straightforward—in all the proofs of this section, F* required no additional hints. In particular, we rely heavily on the view that both parties execute (different fragments of) the same code, thus avoiding the unwieldy task of reasoning about low-level message passing.

Correctness and Security of median. We first define a pure specification of median of two int tuples:

```
let median_of (x1, x2) (y1, y2) = let (_, m, _, _) = sort x1 x2 y1 y2 in m
```

Further, we capture the preconditions using the following predicate:

```
let median_pre (x1, x2) (y1, y2) = x1 < x2 ∧ y1 < y2 ∧ distinct x1 x2 y1 y2
```

Using these, we prove the following top-level specification for median:

```
val median: in_a:sealed {a} (int * int) → in_b:sealed {b} (int * int) → Wys int
  (requires (fun m → m = Mode Par {a, b})) (* should be called in the Par mode *)
  (ensures (fun m r tr → let in_a, in_b = unseal in_a, unseal in_b in
      (median_pre in_a in_b ⟹ r = median_of in_a in_b) ∧
(* functional correctness *)
      tr = [TMsg r])) (* trace is just the final value *)
```

This signature establishes that when Alice and Bob simultaneously execute median (in Par mode), with secrets in_a and in_b, then, if and when the protocol terminates, (a) if their inputs satisfy the precondition median_pre, then the result is the joint median of their inputs and (b) the observable trace consists only of the final result, as there is but a single as_sec thunk in median, i.e., it is *secure*.

Correctness and Security of median_opt. The security proof of median_opt is particularly interesting, because the program intentionally reveals more than just

the final result, i.e., the output of the first comparison. We would like to verify that this additional information does not compromise the privacy of the parties' inputs. To do this, we take the following approach.

First, we characterize the observable trace of median_opt as a pure, specification-only function. Then, using relational reasoning, we prove a *noninteference with delimited release* property [55] on these traces. Essentially we prove that, for two runs of median_opt where Bob's inputs and the output median are the same, the observable traces are also the same irrespective of Alice's inputs. Thus, from Alice's perspective, the observable trace does not reveal more to Bob than what the output already does. We prove this property symmetrically for Bob.

We start by defining a trace function for median_opt:

```
let opt_trace a b (x1, _) (y1, _) r = [
  TMsg (x1 > y1); (* observable from the first as_sec *)
  TScope {a} []; TScope {b} []; (* observables from two local as_par *)
  TMsg r ] (* observable from the final as_sec *)
```

A trace will have four elements: output of the first as_sec computation, two empty scoped traces for the two local as_par computations, and the final output.

Using this function, we prove correctness of median_opt, thus:

```
val median_opt: in_a:sealed {a} (int * int) → in_b:sealed {b} (int * int) → Wys int
  (requires (fun m → m = Mode Par {a, b})) (* should be called in the Par mode *)
  (ensures (fun m r tr → let in_a = unseal in_a in let in_b = unseal in_b in
    (median_pre in_a in_b ⟹ r = median_of in_a in_b) ∧
(* functional correctness *)
    tr = opt_trace a b in_a in_b r
(* opt_trace precisely describes the observable trace *)
```

The delimited release property is then captured by the following lemma:

```
val median_opt_is_secure_for_alice: a:prin → b:prin
  → in_a1:(int * int) → in_a2:(int * int) → in_b:(int * int) (* possibly diff a1, a2 *)
  → Lemma (requires (median_pre in_a1 in_b ∧ median_pre in_a2 in_b ∧
                     median_of in_a1 in_b = median_of in_a2 in_b)) (* but same median *)
          (ensures (opt_trace a b in_a1 in_b (median_of in_a1 in_b) = (* ensures .. *)
                    opt_trace a b in_a2 in_b (median_of in_a2 in_b))) (* .. same trace *)
```

The lemma proves that for two runs of median_opt where Bob's input and the final output remain same, but Alice's inputs vary arbitrarily, the observable traces are the same. As such, no more information about information leaks about Alice's inputs via the traces than what is already revealed by the output. We also prove a symmetrical lemma median_opt_is_secure_for_bob.

In short, because the Wys monad provides programmers with the observable traces in the logic, they can then be used to prove properties, relational or otherwise, in the pure fragment of F* outside the Wys monad. We present more examples and their verification details in Sect. 4.

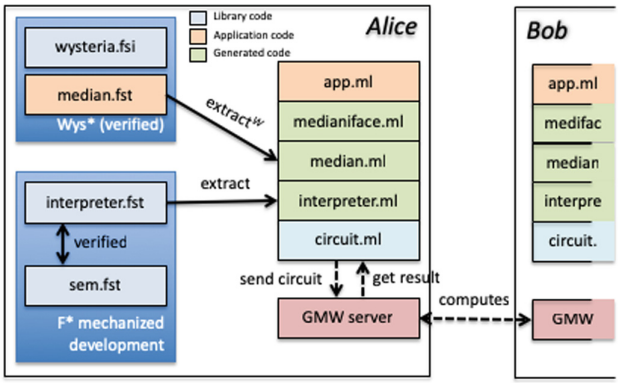

Fig. 1. Architecture of an Wys* deployment

2.5 Deploying Wys* Programs

Having defined a proved-secure MPC program in Wys*, how do we run it? Doing so requires the following steps (Fig. 1). First, we run the F* compiler in a special mode that *extracts* the Wys* code (say psi.fst), into the Wys* AST as a data structure (in psi.ml). Except for the Wys* specific nodes (as_sec, as_par, etc.), the rest of the program is extracted into *FFI nodes* that indicate the use of, or calls into, functionality provided by F* itself.

The next step is for each party to run the extracted AST using the Wys* interpreter. This interpreter is written in F* and we have proved (see Sect. 3.5) that it implements a deep embedding of the Wys* semantics, also specified in F* (Figs. 5 and 6, Sect. 3). The interpreter is extracted to OCaml by the usual F* extraction. Each party's interpreter executes the AST locally until it reaches an as_sec ps f node, where the interpreter's back-end compiles f, on-the-fly, for particular values of the secrets in f's environment, to a boolean circuit. First-order, loop-free code can be compiled to a circuit; Wys* provides specialized support for several common combinators (e.g., fst, snd, list combinators such as List.intersect, List.mem, List.nth etc.).

The circuit is handed to a library by Choi et al. [22] that implements the GMW [26] MPC protocol. Running the GMW protocol involves the parties in ps generating and communicating (XOR-based) secret shares [57] for their secret inputs, and then cooperatively evaluating the boolean circuit for f over them. While our implementation currently uses the GMW protocol, it should be possible to plugin other MPC protocols as well.

One obvious question is how both parties are able to get this process off the ground, given that they don't know some of the inputs (e.g., other parties' secrets). The sealed abstraction helps here. Recall that for median, the types of the inputs are of the form sealed {a} (int $*$ int) and sealed {b} (int $*$ int). When the program is run on Alice's host, the former will be a pair of Alice's values, whereas the latter will be an opaque constant (which we denote as $\bullet$). The reverse will

$$
\begin{array}{rl}
\text{Principal} & p \qquad \text{Principal set} \quad s \qquad \text{FFI const} \quad \mathsf{c}, \mathsf{f} \\
\text{Constant } c ::= & p \mid s \mid () \mid \top \mid \bot \mid \mathsf{c} \\
\text{Expression } e ::= & \mathsf{as\_par}\ e_1\ e_2 \mid \mathsf{as\_sec}\ e_1\ e_2 \mid \mathsf{seal}\ e_1\ e_2 \mid \mathsf{reveal}\ e \mid \mathsf{ffi}\ \mathsf{f}\ \bar{e} \\
& \mid \mathsf{mkmap}\ e_1\ e_2 \mid \mathsf{project}\ e_1\ e_2 \mid \mathsf{concat}\ e_1\ e_2 \\
& \mid c \mid x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{fix}\ f.\lambda x.e \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3
\end{array}
$$

Fig. 2. Wys* syntax

be true on Bob's host. When the circuit is constructed, each principal links their non-opaque inputs to the relevant input wires of the circuit. Similarly, the output map component of each party is derived from their output wires in the circuit, and thus, each party only gets to see their own output.

3 Formalizing and Implementing Wys*

In the previous section, we presented examples of verifying properties about Wys* programs using F*'s logic. However, these programs are not executed using the F* (single-threaded) semantics; they have a distributed semantics involving multiple parties. So, how do the properties that we verify using F* carry over?

In this section, we present the metatheory that answers this question. First, we formalize the Wys* single-threaded (ST) semantics, that faithfully models the F* semantics of the Wys* API presented in Sect. 2. Next, we formalize the distributed (DS) semantics that multiple parties use to run Wys* programs. Then we prove the former is *sound* with respect to the latter, so that properties proved of programs under ST apply when run under DS. We have mechanized the proof of this theorem in F*.

3.1 Syntax

Figure 2 shows the complete syntax of Wys*. Principals and principal sets are first-class values, and are denoted by p and s respectively. Constants in the language also include () (unit), booleans ($\top$ and $\bot$), and FFI constants c. Expressions e include the regular forms for functions, applications, let bindings, etc. and the Wys*-specific constructs. Among the ones that we have not seen in Sect. 2, expression mkmap e_1 e_2 creates a map from principals in e_1 (which is a principal set) to the value computed by e_2. project e_1 e_2 projects the value of principal e_1 from the map e_2, and concat e_1 e_2 concatenates the two maps. The maps are used if an as_sec computation returns different outputs to the parties.

Host language (i.e., F*) constructs are also part of the syntax of Wys*, including constants c for strings, integers, lists, tuples, etc. Likewise, host language functions/primitives can be called from Wys*—ffi f $\bar{e}$ is the invocation of a host-language function f with arguments $\bar{e}$. The FFI confers two benefits. First, it simplifies the core language while still allowing full consideration of security relevant properties. Second, it helps the language scale by incorporating many of the standard features, libraries, etc. from the host language.

$$
\begin{array}{rl}
\text{Map } m & ::= \cdot \mid m[p \mapsto v] \\
\text{Value } v & ::= p \mid s \mid () \mid \top \mid \bot \mid m \mid \mathsf{v} \mid (L, \lambda x.e) \mid (L, \mathsf{fix}\ f.\lambda x.e) \mid \mathsf{sealed}\ s\ v \mid \bullet \\
\text{Mode } M & ::= \mathsf{Par}\ s \mid \mathsf{Sec}\ s \\
\text{Context } E & ::= \langle\rangle \mid \mathsf{as\_par}\ \langle\rangle\ e \mid \mathsf{as\_par}\ v\ \langle\rangle \mid \mathsf{as\_sec}\ \langle\rangle\ e \mid \mathsf{as\_sec}\ v\ \langle\rangle \mid \ldots \\
\text{Frame } F & ::= (M, L, E, T) \\
\text{Stack } X & ::= \cdot \mid F, X \\
\text{Environment } L & ::= \cdot \mid L[x \mapsto v] \\
\text{Trace element } t & ::= \mathsf{TMsg}\ v \mid \mathsf{TScope}\ s\ T \\
\text{Trace } T & ::= \cdot \mid t, T \\
\text{Configuration } C & ::= M; X; L; T; e \\
\\
\text{Par component } P & ::= \cdot \mid P[p \mapsto C] \\
\text{Sec component } S & ::= \cdot \mid S[s \mapsto C] \\
\text{Protocol } \pi & ::= P; S
\end{array}
$$

Fig. 3. Runtime configuration syntax

S-ASPAR

$$\frac{e_1 = \mathsf{as\_par}\ s\ (L_1, \lambda x.e) \quad M = \mathsf{Par}\ s_1 \quad s \subseteq s_1 \quad X_1 = (M; L; \mathsf{seal}\ s\ \langle\rangle; T), X}{M; X; L; T; e_1 \rightarrow \mathsf{Par}\ s; X_1; L_1[x \mapsto ()]; \cdot; e}$$

S-PARRET

$$\frac{X = (M_1; L_1; \mathsf{seal}\ s\ \langle\rangle; T_1), X_1 \quad \mathsf{can\_seal}\ s\ v \quad T_2 = \mathsf{append}\ T_1\ [\mathsf{TScope}\ s\ T]}{M; X; L; T; v \rightarrow M_1; X_1; L_1; T_2; \mathsf{sealed}\ s\ v}$$

S-ASSEC

$$\frac{e_1 = \mathsf{as\_sec}\ s\ (L_1, \lambda x.e) \quad M = \mathsf{Par}\ s \quad X_1 = (M; L; \langle\rangle\ T), X}{M; X; L; T; e_1 \rightarrow \mathsf{Sec}\ s; X_1; L_1[x \mapsto ()]; \cdot; e}$$

S-SECRET

$$\frac{M = \mathsf{Sec}\ \_ \quad X = (M_1; L_1; \langle\rangle; T), X_1 \quad T_1 = \mathsf{append}\ T\ [\mathsf{TMsg}\ v]}{M; X; L; \cdot; v \rightarrow M_1; X_1; L_1; T_1; v}$$

Fig. 4. WYS* ST semantics (selected rules)

3.2 Single-Threaded Semantics

We formalize the semantics in the style of Hieb and Felleisen [24], where the redex is chosen by (standard, not shown) *evaluation contexts* E, which prescribe left-to-right, call-by-value evaluation order. The ST semantics, a model of the F* semantics and the WYS* API, defines a judgment $C \rightarrow C'$ that represents a single step of an abstract machine (Fig. 4). Here, C is a *configuration* $M; X; L; T; e$. This five-tuple consists of a mode M, a stack X, a local environment L, a trace T, and an expression e. The syntax for these elements is given in Fig. 3. The value form v represents the host language (FFI) values. The stack and environment are standard; trace T and mode M were discussed in the previous section.

For space reasons, we focus on the two main WYS* constructs as_par and as_sec. Our technical report [54] shows other WYS* specific constructs.

Rules S-ASPAR and S-PARRET (Fig. 4) reduce an as_par expression once its arguments are fully evaluated—its first argument s is a principal set, while the second argument $(L_1, \lambda x.e)$ is a closure where L_1 captures the free variables of thunk $\lambda x.e$. S-ASPAR first checks that the current mode M is Par and contains all the principals from the set s. It then pushes a seal $s\ \langle\rangle$ frame on the stack, and

$$\text{P-PAR}\ \frac{C \rightsquigarrow C'}{P[p \mapsto C]; S \longrightarrow P[p \mapsto C']; S} \qquad \text{P-ENTER}\ \frac{\forall p \in s.\ P[p].e = \mathsf{as\_sec}\ s\ (L_p, \lambda x.e) \quad s \notin \mathsf{dom}(S) \quad L = \mathsf{combine}\ \bar{L}_p}{P; S \longrightarrow P; S[s \mapsto \mathsf{Sec}\ s; \cdot; L[x \mapsto ()]; \cdot; e]}$$

$$\text{P-SEC}\ \frac{C \rightarrow C'}{P; S[s \mapsto C] \longrightarrow P; S[s \mapsto C']} \qquad \text{P-EXIT}\ \frac{S[s] = \mathsf{Sec}\ s; \cdot; L; T; v \quad P' = \forall p \in s.\ P[p \mapsto P[p] \triangleleft (\mathsf{slice\_v}\ p\ v)] \quad S' = S \setminus s}{P; S \longrightarrow P'; S'}$$

Fig. 5. Distributed semantics, multi-party rules

$$\text{L-ASPAR1}\ \frac{e_1 = \mathsf{as\_par}\ s\ (L_1, \lambda x.e) \quad p \in s \quad X_1 = (M; L; \mathsf{seal}\ s\ \langle\rangle; T), X}{\mathsf{Par}\ p; X; L; T; e_1 \rightsquigarrow \mathsf{Par}\ p; X_1; L_1[x \mapsto ()]; \cdot; e}$$

$$\text{L-PARRET}\ \frac{X = (M; L_1; \mathsf{seal}\ s\ \langle\rangle; T_1), X_1 \quad T_2 = \mathsf{append}\ T_1\ T \quad v_1 = \mathsf{sealed}\ s\ v}{\mathsf{Par}\ p; X; L; T; v \rightsquigarrow \mathsf{Par}\ p; X_1; L_1; T_2; v_1}$$

$$\text{L-ASPAR2}\ \frac{p \notin s}{\mathsf{Par}\ p; X; L; T; \mathsf{as\_par}\ s\ (L_1, \lambda x.e) \rightsquigarrow \mathsf{Par}\ p; X; L; T; \mathsf{sealed}\ s\ \bullet}$$

Fig. 6. Distributed semantics, selected local rules (the mode M is always $\mathsf{Par}\ p$)

starts evaluating e under the environment $L_1[x \mapsto ()]$. The rule S-ASPARRET pops the frame and seals the result, so that it is accessible only to the principals in s. The rule also creates a trace element $\mathsf{TScope}\ s\ T$, essentially making observations during the reduction of e (i.e., T) visible only to principals in s.

Turning to as_sec, the rule S-ASSEC checks the precondition of the API, and the rule S-ASSECRET generates a trace observation $\mathsf{TMsg}\ v$, as per the post-condition of the API. As mentioned before, as_sec semantics models the ideal, trusted third-party semantics of secure computations where the participants only observe the final output. We can confirm that the rules implement the types of as_par and as_sec shown in Sect. 2.

3.3 Distributed Semantics

In the DS semantics, principals evaluate the same program locally and asynchronously until they reach a secure computation, at which point they synchronize to jointly perform the computation. The semantics consists of two parts: (a) a judgment of the form $\pi \longrightarrow \pi'$ (Fig. 5), where a protocol π is a tuple $(P; S)$ such that P maps each principal to its local configuration and S maps a set of principals to the configuration of an ongoing, secure computation; and (b) a local evaluation judgment $C \rightsquigarrow C'$ (Fig. 6) to model how a single principal behaves while in par mode.

Rule P-PAR in Fig. 5 models a single party taking a step, per the local evaluation rules. Figure 6 shows these rules for as_par. (See technical report [54] for more local evaluation rules.) A principal either participates in the as_par

computation, or skips it. Rules L-ASPAR1 and L-PARRET handle the case when $p \in s$, and so, the principal p participates in the computation. The rules closely mirror the corresponding ST semantics rules in Fig. 4. One difference in the rule L-ASPARRET is that the trace T is not scoped. In the DS semantics, traces only contain TMsg elements; i.e., a trace is the (flat) list of secure computation outputs observed by that active principal. If $p \notin s$, then the principal skips the computation with the result being a sealed value containing the opaque constant $\bullet$ (rule L-ASPAR2). The contents of the sealed value do not matter, since the principal will not be allowed to unseal the value anyway.

As should be the case, there are no local rules for as_sec—to perform a secure computation parties need to combine their data and jointly do the computation. Rule P-ENTER in Fig. 5 handles the case when principals enter a secure computation. It requires that all the principals $p \in s$ must have the expression form as_sec s $(L_p, \lambda x.e)$, where L_p is their local environment associated with the closure. Each party's local environment contains its secret values (in addition to some public values). Conceptually, a secure computation *combines* these environments, thereby producing a joint view, and evaluates e under the combination. We define an auxiliary combine function for this purpose:

combine_v (•, v) = v
combine_v (v, •) = v
combine_v (sealed s v_1, sealed s v_2) = sealed s (combine_v v_1 v_2)
...

The rule P-ENTER combines the principals' environments, and creates a new entry in the S map. The principals are now waiting for the secure computation to finish. Rule P-SEC models a stepping rule inside the sec mode.

The rule P-EXIT applies when a secure computation has completed and returns results to the waiting principals. If the secure computation terminates with value v, each principal p gets the value slice_v p v. The slice_v function is analogous to combine, but in the opposite direction—it strips off the parts of v that are not accessible to p:

slice_v p (sealed s v) = sealed s •, if p ∉ s
slice_v p (sealed s v) = sealed s (slice_v p v), if p ∈ s
...

In the rule P-EXIT, the $\triangleleft$ notation is defined as:
$M; X; L; T; \_ \triangleleft v = M; X; L;$ append T [TMsg v]$; v$
That is, the returned value is also added to the principal's trace to note their observation of the value.

3.4 Metatheory

Our goal is to show that the ST semantics faithfully represents the semantics of WYS* programs as they are executed by multiple parties, i.e., according to the DS semantics. We do this by proving *simulation* of the ST semantics by the DS semantics, and by proving *confluence* of the DS semantics. Our F* development mechanizes all the metatheory presented in this section.

Simulation. We define a slice s C function that returns the corresponding protocol π_C for an ST configuration C. In the P component of π_C, each principal $p \in s$ is mapped to their *slice* of the protocol. For slicing values, we use the same slice_v function as before. Traces are sliced as follows:

```
slice_tr p (TMsg v) = [TMsg (slice_v p v)]
slice_tr p (TScope s T) = slice_tr p T, if p ∈ s
slice_tr p (TScope s T) = [], if p ∉ s
```

The slice of an expression (e.g., the source program) is itself. For all other components of C, slice functions are defined analogously.

We say that C is *terminal* if it is in Par mode and is fully reduced to a value (i.e. when $C = \_; X; \_; \_; e$, e is a value and X is empty). Similarly, a protocol $\pi = (P, S)$ is terminal if S is empty and all the local configurations in P are terminal. The simulation theorem is then the following:

Theorem 1 (Simulation of ST by DS). *Let s be the set of all principals. If $C_1 \rightarrow^* C_2$, and C_2 is terminal, then there exists some derivation* (slice s C_1) $\longrightarrow^*$ (slice s C_2) *such that* (slice s C_2) *is terminal.*

To state *confluence*, we first define the notion of *strong termination*.

Definition 1 (Strong termination). *If all possible runs of protocol π terminate at π_t, we say π* strongly terminates in π_t, *written $\pi \Downarrow \pi_t$.*

Our confluence result then says:

Theorem 2 (Confluence of DS). *If $\pi \longrightarrow^* \pi_t$ and π_t is terminal, then $\pi \Downarrow \pi_t$.*

Combining the two theorems, we get a corollary that establishes the soundness of the ST semantics w.r.t. the DS semantics:

Corollary 1 (Soundness of ST semantics). *Let s be the set of all principals. If $C_1 \rightarrow^* C_2$, and C_2 is terminal, then* (slice s C_1) $\Downarrow$ (slice s C_2).

Now suppose that for a Wys* source program, we prove in F* a postcondition that the result is sealed alice n, for some $n > 0$. By the soundness of the ST semantics, we can conclude that when the program is run in the DS semantics, it may diverge, but if it terminates, alice's output will also be sealed alice n, and for all other principals their outputs will be sealed alice •. Aside from the correspondence on results, our semantics also covers correspondence on traces. Thus the correctness and security properties that we prove about a Wys* program using F*'s logic, hold for the program that actually runs.

3.5 Implementation

The formal semantics presented in the prior section is mechanized as an inductive type in F*. This style is useful for proving properties, but does not directly translate to an implementation. Therefore, we implement an interpretation function step in F* and prove that it corresponds to the rules; i.e., that for all input

configurations C, $\mathsf{step}(C) = C'$ implies that $C \rightarrow C'$ according to the semantics. Then, the core of each principal's implementation is an F* stub function `tstep` that repeatedly invokes `step` on the AST of the source program (produced by the F* extractor run in a custom mode), unless the AST is an `as_sec` node. Functions `step` and `tstep` are extracted to OCaml by the standard F* extraction process.

Local evaluation is not defined for the `as_sec` node, so the stub implements what amounts to P-ENTER and P-EXIT from Fig. 5. When the stub notices the program has reached an `as_sec` expression, it calls into a circuit library we have written that converts the AST of the second argument of `as_sec` to a boolean circuit. This circuit and the encoded inputs are communicated to a co-hosted server that implements the GMW MPC protocol [22]. The server evaluates the circuit, coordinating with the GMW servers of the other principals, and sends back the result. The circuit library decodes the result and returns it to the stub. The stub then carries on with the local evaluation. Our FFI interface currently provides a form of monomorphic, first-order interoperability between the (dynamically typed) interpreter and the host language.

Our F* formalization of the Wys* semantics, including the AST specification, is 1900 lines of code. This formalization is used both by the metatheory as well as by the (executable) interpreter. The metatheory that connects the ST and DS semantics (Sect. 3) is 3000 lines. The interpreter and its correctness proof are another 290 lines of F* code. The interpreter `step` function is essentially a big switch-case on the current expression, that calls into the functions from the semantics specification. The `tstep` stub is another 15 lines. The size of the circuit library, not including the GMW implementation, is 836 lines. The stub, the implementation of GMW, the circuit library, and F* toolchain (including the custom Wys* extraction mode) are part of our Trusted Computing Base (TCB).

4 Applications

In addition to joint median, presented in Sect. 2, we have implemented and proved properties of two other MPC applications, *dealing for online card games* and *private set intersection* (PSI).

Card Dealing. We have implemented an MPC-based card dealing application in Wys*. Such an application can play the role of the dealer in a game of online poker, thereby eliminating the need to trust the game portal for card dealing. The application relies on Wys*'s support for *secret shares* [57]. Using secret shares, the participating parties can share a value in a way that none of the parties can observe the actual value individually (each party's share consists of some random-looking bytes), but they can recover the value by combining their shares in `sec` mode.

In the application, the parties maintain a list of secret shares of already dealt cards (the number of already dealt cards is public information). To deal a new card, each party first generates a random number locally. The parties then perform a secure computation to compute the sum of their random numbers modulo 52, let's call it n. The output of the secure computation is secret shares

of n. Before declaring n as the newly dealt card, the parties needs to ensure that the card n has not already been dealt. To do so, they iterate over the list of secret shares of already dealt cards, and for each element of the list, check that it is different from n. The check is performed in a secure computation that simply combines the shares of n, combines the shares of the list element, and checks the equality of the two values. If n is different from all the previously dealt cards, it is declared to be the new card, else the parties repeat the protocol by again generating a fresh random number each.

Wys* provides the following API for secret shares:

```
type Sh: Type → Type
type can_sh: Type → Type
assume Cansh_int: can_sh int

val v_of_sh: sh:Sh α → Ghost α
val ps_of_sh: sh:Sh α → Ghost prins

val mk_sh: x:α → Wys (Sh α)
    (requires (fun m → m.mode = Sec ∧ can_sh α))
    (ensures (fun m r tr → v_of_sh r = x ∧ ps_of_sh r = m.ps ∧ tr = []))
val comb_sh: x:Sh α → Wys α (requires (fun m → m.mode = Sec ∧ ps_of_sh x = m.ps))
                            (ensures (fun m r tr → v_of_sh x = r ∧ tr = []))
```

Type Sh α types the shares of values of type α. Our implementation currently supports shares of int values only; the can_sh predicate enforces this restriction on the source programs. Extending secret shares support to other types (such as pairs) should be straightforward (as in [52]). Functions v_of_sh and ps_of_sh are marked Ghost, meaning that they can only be used in specifications for reasoning purposes. In the concrete code, shares are created and combined using the mk_sh and comb_sh functions. Together, the specifications of these functions enforce that the shares are created and combined by the same set of parties (through ps_of_sh), and that comb_sh recovers the original value (through v_of_sh). The Wys* interpreter transparently handles the low-level details of extracting shares from the GMW implementation of Choi et al. (mk_sh), and reconstituting the shares back (comb_sh).

In addition to implementing the card dealing application in Wys*, we have formally verified that the returned card is fresh. The signature of the function that checks for freshness of the newly dealt card is as follows (abc is the set of three parties in the computation):

```
val check_fresh: l:list (Sh int){∀ s'. mem s' l ⟹ ps_of_sh s' = abc}
  → s:Sh int{ps_of_sh s = abc}
  → Wys bool (requires (fun m → m = Mode Par abc))
    (ensures (fun _ r _ → r ⟺ (∀ s'. mem s' l ⟹ not (v_of_sh s' = v_of_sh s))))
```

The specification says that the function takes two arguments: l is the list of secret shares of already dealt cards, and s is the secret shares of the newly dealt card. The function returns a boolean r that is true iff the concrete value (v_of_sh) of s is different from the concrete values of all the elements of the list l. Using F*, we verify that the implementation of check_fresh meets this specification.

PSI. Consider a dating application that enables its users to compute their common interests without revealing all of them. This is an instance of the more general private set intersection (PSI) problem [28].

We implement a straightforward version of PSI in Wys*:

```
let psi a b (input_a:sealed {a} (list int)) (input_b:sealed {b} (list int)) (l_a:int) (l_b:int) =
  as_sec {a,b} (fun () → List.intersect (reveal input_a) (reveal input_b) l_a l_b)
```

where the input sets are expressed as lists with public lengths.

Huang et al. [28] provide an optimized PSI algorithm that performs much better when the density of common elements in the two sets is high. We implement their algorithm in Wys*. The optimized version consists of two nested loops – an outer loop for Alice's set and an inner loop for Bob's – where an iteration of the inner loop compares the current element of Alice's set with the current element of Bob's. The nested loops are written using as_par so that both Alice and Bob execute the loops in lockstep (note that the set sizes are public), while the comparison in the inner loop happens using as_sec. Instead of naive $l\_a * l\_b$ comparisons, Huang et al. [28] observe that once an element of Alice's set ax matches an element of Bob's set bx, the inner loop can return immediately, skipping the comparisons of ax with the rest of Bob's set. Furthermore, bx can be removed from Bob's set, excluding it from any further comparisons with other elements in Alice's set. Since there are no repeats in the input sets, all the excluded comparisons are guaranteed to be false. We show the full code and its performance comparison with psi in the technical report [54].

As with the median example from Sect. 2, the optimized PSI intentionally reveals more for performance gains. As such, we would like to verify that the optimizations do not reveal more about parties' inputs. We take the following stepwise refinement approach. First, we characterize the trace of the optimized implementation as a pure function trace_psi_opt la lb (omitted for space reasons), and show that the trace of psi_opt is precisely trace_psi_opt la lb.

Then, we define an intermediate PSI implementation that has the same nested loop structure, but performs $l\_a * l\_b$ comparisons without any optimizations. We characterize the trace of this intermediate implementation as the pure function trace_psi, and show that it precisely captures the trace.

To show that trace_psi does not reveal more than the intersection of the input sets, we prove the following lemma.

$\Psi\ la_0\ la_1\ lb_0\ lb_1 \stackrel{\text{def}}{=}$ *(* possibly diff input sets, but with *)*
$la_0 \cap lb_0 = la_1 \cap lb_1 \wedge$ *(* intersections the same *)*
$\text{length}\ la_0 = \text{length}\ la_1 \wedge \text{length}\ lb_0 = \text{length}\ lb_1$ *(* lengths the same *)*

```
val psi__interim_is_secure: la0:_ → lb0:_ → la1:_ → lb1:_ → Lemma
  (requires (Ψ la0 la1 lb0 lb1))
  (ensures (permutation (trace_psi la0 lb0) (trace_psi la1 lb1)))
```

The lemma essentially says that for two runs on same length inputs, if the output is the same, then the resulting traces are permutation of each other.[4] We can reason about the traces of psi_interim up to permutation because Alice has no prior knowledge of the choice of representation of Bob's set (Bob can shuffle his list), so cannot learn anything from a permutation of the trace.[5] This establishes the security of psi_interim.

Finally, we can connect psi_interim to psi_opt by showing that there exists a function f, such that for any trace tr=trace_psi la lb, the trace of psi_opt, trace_psi_opt la lb, can be computed by f (length la) (length lb) tr. In other words, the trace produced by the optimized implementation can be computed using a function of information already available to Alice (or Bob) when she (or he) observes a run of the secure, unoptimized version psi_interim la lb. As such, the optimizations do not reveal further information.

5 Related Work

Source MPC Verification. While the verification of the underlying crypto protocols has received some attention [4,5], verification of the correctness and security properties of MPC source programs has remained largely unexplored, surprisingly so given that the goal of MPC is to preserve the privacy of secret inputs. The only previous work that we know of is Backes et al. [9] who devise an applied pi-calculus based abstraction for MPC, and use it for formal verification. For an auction protocol that computes the min function, their abstraction comprises about 1400 lines of code. Wys*, on the other hand, enables direct verification of the higher-level MPC source programs, and not their models, and in addition provides a partially verified toolchain.

Wysteria. Wys*'s computational model is based on the programming abstractions of a previous MPC DSL, Wysteria [52]. Wys*'s realization as an embedded DSL in F* makes important advances. In particular, Wys* (a) enhances the Wysteria semantics to include a notion of observable traces, and provides the novel capability to prove security and correctness properties about mixed-mode MPC source programs, (b) expands the programming constructs available by drawing on features and libraries of F*, and (c) adds assurance via a (partially) proved-correct interpreter.

Verified MPC Toolchain. Almeida et al. [4] build a verified toolchain consisting of (a) a verified circuit compiler from (a subset of) C to boolean circuits, and (b) a verified implementation of Yao's [65] garbled circuits protocol for 2-party MPC. They use CompCert [36] for the former, and EasyCrypt [11] for the latter. These are significant advances, but there are several distinctions from our work. The MPC programs in their toolchain are not *mixed-mode*, and thus it cannot express

[4] Holding Bob's (resp. Alice's) inputs fixed and varying Alice's (resp. Bob's) inputs, as done for median in Sect. 2.4, is covered by this more general property.

[5] We could formalize this observation using a probabilistic, relational variant of F* [10].

examples like median_opt and the optimized PSI. Their framework does not enable formal verification of source programs like Wys* does. It may be possible to use other frameworks for verifying C programs (e.g. Frama-C [1]), but it is inconvenient as one has to work in the subset of C that falls in the intersection of these tools. Wys* is also more general as it supports general n-party MPC; e.g., the card dealing application in Sect. 4 has 3 parties. Nevertheless, Wys* may use their verified Yao implementation for the special case of 2 parties.

MPC DSLs and DSL Extensions. In addition to Wysteria several other MPC DSLs have been proposed in the literature [14,17,27,29,34,37,39,48,49,52,56,61]. Most of these languages have standalone implementations, and the (usability/scalability) drawbacks that come with them. Like Wys*, a few are implemented as language extensions. Launchbury et al. [35] describe a Haskell-embedded DSL for writing low-level "share protocols" on a multi-server "SMC machine". OblivC [66] is an extension to C for two-party MPC that annotates variables and conditionals with an obliv qualifier to identify private inputs; these programs are compiled by source-to-source translation. The former is essentially a shallow embedding, and the latter is compiler-based; Wys* is unique in that it combines a shallow embedding to support source program verification and a deep embedding to support a non-standard target semantics. Recent work [19,21] compiles to cryptographic protocols that include both arithmetic and boolean circuits; the compiler decides which fragments of the program fall into which category. It would be interesting work to integrate such a backend in Wys*.

Mechanized Metatheory. Our verification results are different from a typical verification result that might either mechanize metatheory for an idealized language [8], or might prove an interpreter or compiler correct w.r.t. a formal semantics [36]—we do both. We mechanize the metatheory of Wys* establishing the soundness of the conceptual ST semantics w.r.t. the actual DS semantics, and mechanize the proof that the interpreter implements the correct DS semantics.

General DSL Implementation Strategies. DSLs (for MPC or other purposes) are implemented in various ways, such as by developing a standalone compiler/interpreter, or by shallow or deep embedding in a host language. Our approach bears relation to the approach taken in LINQ [42], which embeds a query language in normal C# programs, and implements these programs by extracting the query syntax tree and passing it to a *provider* to implement for a particular backend. Other researchers have embedded DSLs in verification-oriented host languages (e.g., Bedrock [13] in Coq [60]) to permit formal proofs of DSL programs. Low* [51] is a shallow embedding of a small, sequential, well-behaved subset of C in F* that extracts to C using a F*-to-C compiler. Low* has been used to verify and implement several cryptographic constructions. Fromherz et al. [25] present a deep embedding of a subset of x64 assembly in F* that allows efficient verification of assembly and its interoperation with C code generated from Low*. They design (and verify) a custom VC generator for the deeply embedded DSL, that allows for the proofs of assembly crypto routines to scale.

6 Conclusions

This paper has presented Wys*, the first DSL to enable formal verification of efficient source MPC programs as written in a full-featured host programming language, F*. The paper presented examples such as joint median, card dealing, and PSI, and showed how the DSL enables their correctness and security proofs. Wys* implementation, examples, and proofs are publicly available on Github.

Acknowledgments. We would like to thank the anonymous reviewers, Catalin Hriţcu, and Matthew Hammer for helpful comments on drafts of this paper. This research was funded in part by the U.S. National Science Foundation under grants CNS-1563722, CNS-1314857, and CNS-1111599.

References

1. Frama-c. https://frama-c.com/
2. Z3 theorem prover. z3.codeplex.com
3. Aggarwal, G., Mishra, N., Pinkas, B.: Secure computation of the k^{th}-ranked element. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 40–55. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_3
4. Almeida, J.B., et al.: A fast and verified software stack for secure function evaluation. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017 (2017)
5. Almeida, J.B., et al.: Verified implementations for secure and verifiable computation (2014)
6. Araki, T., et al.: Generalizing the SPDZ compiler for other protocols. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018 (2018)
7. Atkey, R.: Parameterised notions of computation. J. Funct. Program. **19**, 335–376 (2009). https://doi.org/10.1017/S095679680900728X. http://journals.cambridge.org/article_S095679680900728X
8. Aydemir, B.E., et al.: Mechanized metatheory for the masses: the PoplMark challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_4
9. Backes, M., Maffei, M., Mohammadi, E.: Computationally sound abstraction and verification of secure multi-party computations. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010) (2010)
10. Barthe, G., Fournet, C., Grégoire, B., Strub, P., Swamy, N., Béguelin, S.Z.: Probabilistic relational verification for cryptographic implementations. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 193–206 (2014). https://doi.org/10.1145/2535838.2535847
11. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_5

12. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols. In: STOC (1990)
13. Bedrock, a coq library for verified low-level programming. http://plv.csail.mit.edu/bedrock/
14. Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: CCS (2008)
15. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y.: Implementing TLS with verified cryptographic security. In: IEEE Symposium on Security & Privacy, Oakland, pp. 445–462 (2013). http://www.ieee-security.org/TC/SP2013/papers/4977a445.pdf
16. Bogdanov, D., Jõemets, M., Siim, S., Vaht, M.: How the Estonian Tax and Customs Board Evaluated a tax fraud detection system based on secure multi-party computation. In: Böhme, R., Okamoto, T. (eds.) FC 2015. LNCS, vol. 8975, pp. 227–234. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47854-7_14
17. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: a framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88313-5_13
18. Bogetoft, P., et al.: Secure multiparty computation goes live. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 325–343. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03549-4_20
19. Büscher, N., Demmler, D., Katzenbeisser, S., Kretzmer, D., Schneider, T.: HyCC: compilation of hybrid protocols for practical secure computation. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018 (2018)
20. Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptol. **13**(1), 143–202 (2000). https://doi.org/10.1007/s001459910006
21. Chandran, N., Gupta, D., Rastogi, A., Sharma, R., Tripathi, S.: EzPC: programmable, efficient, and scalable secure two-party computation for machine learning. Cryptology ePrint Archive, Report 2017/1109 (2017). https://eprint.iacr.org/2017/1109
22. Choi, S.G., Hwang, K.W., Katz, J., Malkin, T., Rubenstein, D.: Secure multi-party computation of Boolean circuits with applications to privacy in on-line marketplaces (2011). http://eprint.iacr.org/
23. Crockett, E., Peikert, C., Sharp, C.: Alchemy: a language and compiler for homomorphic encryption made easy. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018 (2018)
24. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoret. Comput. Sci. **103**(2), 235–271 (1992)
25. Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A., Swamy, N.: A verified, efficient embedding of a verifiable assembly language. In: 46th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2019 (2019)
26. Goldreich, O., Micali, S., Wigderson, A.: How to play ANY mental game. In: STOC (1987)
27. Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure two-party computations in ANSI C. In: CCS (2012)
28. Huang, Y., Evans, D., Katz, J.: Private set intersection: are garbled circuits better than custom protocols? In: NDSS (2012)
29. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: USENIX (2011)

30. Juvekar, C., Vaikuntanathan, V., Chandrakasani, A.: GAZELLE: a low latency framework for secure neural network inference. In: USENIX Security 2018 (2018)
31. Kamm, L.: Privacy-preserving statistical analysis using secure multi-party computation. Ph.D. thesis, University of Tartu (2015)
32. Kerschbaum, F.: Automatically optimizing secure computation. In: CCS (2011)
33. Kerschbaum, F., et al.: Secure collaborative supply-chain management. Computer **44**(9), 38–43 (2011)
34. Laud, P., Randmets, J.: A domain-specific language for low-level secure multiparty computation protocols. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015 (2015)
35. Launchbury, J., Diatchki, I.S., DuBuisson, T., Adams-Moran, A.: Efficient lookup-table protocol in secure multiparty computation. In: ICFP (2012)
36. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
37. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.: Automating efficient RAM-model secure computation. In: IEEE Symposium on Security and Privacy, Oakland (2014)
38. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via MiniONN transformations. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017 (2017)
39. Malka, L.: VMCrypt: modular software architecture for scalable secure computation. In: CCS (2011)
40. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay: a secure two-party computation system. In: USENIX Security (2004)
41. Mardziel, P., Hicks, M., Katz, J., Hammer, M., Rastogi, A., Srivatsa, M.: Knowledge inference for optimizing and enforcing secure computations. In: Proceedings of the Annual Meeting of the US/UK International Technology Alliance (2013)
42. Meijer, E., Beckman, B., Bierman, G.: LINQ: reconciling object, relations and xml in the .net framework. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD 2006, p. 706. ACM, New York (2006). https://doi.org/10.1145/1142473.1142552
43. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4
44. Mohassel, P., Zhang, Y.: SecureML: a system for scalable privacy-preserving machine learning. In: IEEE S&P (2017)
45. Mood, B., Gupta, D., Carter, H., Butler, K.R.B., Traynor, P.: Frigate: a validated, extensible, and efficient compiler and interpreter for secure computation. In: IEEE EuroS&P (2016)
46. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: dependent types for imperative programs. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP (2008)
47. Nanevski, A., Morrisett, J.G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. **18**(5–6), 865–911 (2008). http://ynot.cs.harvard.edu/papers/jfpsep07.pdf
48. Nielsen, J.D.: Languages for secure multiparty computation and towards strongly typed macros. Ph.D. thesis (2009)
49. Nielsen, J.D., Schwartzbach, M.I.: A domain-specific programming language for secure multiparty computation. In: PLAS (2007)
50. PolarSSL verification kit (2015). http://trust-in-soft.com/polarssl-verification-kit/
51. Protzenko, J., et al.: Verified low-level programming embedded in F* (ICFP) (2017)

52. Rastogi, A., Hammer, M.A., Hicks, M.: Wysteria: a programming language for generic, mixed-mode multiparty computations. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy (2014)
53. Rastogi, A., Mardziel, P., Hammer, M., Hicks, M.: Knowledge inference for optimizing secure multi-party computation. In: PLAS (2013)
54. Rastogi, A., Swamy, N., Hicks, M.: WYS*: a DSL for verified secure multi-party computations (2019). https://arxiv.org/abs/1711.06467
55. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 174–191. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-37621-7_9
56. Schropfer, A., Kerschbaum, F., Muller, G.: L1 - an intermediate language for mixed-protocol secure computation. In: COMPSAC (2011)
57. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
58. Shamir, A., Rivest, R.L., Adleman, L.M.: Mental poker. In: Klarner, D.A. (ed.) The Mathematical Gardner, pp. 37–43. Springer, Boston (1981). https://doi.org/10.1007/978-1-4684-6686-7_5
59. Swamy, N., et al.: Dependent types and multi-monadic effects in F*. In: POPL (2016)
60. The Coq Development Team: The Coq proof assistant. http://coq.inria.fr
61. VIFF, the virtual ideal functionality framework. http://viff.dk/
62. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59451-5_2. http://dl.acm.org/citation.cfm?id=647698.734146
63. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010 (2010)
64. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (2011)
65. Yao, A.C.C.: How to generate and exchange secrets. In: FOCS (1986)
66. Zahur, S., Evans, D.: Obliv-C: a language for extensible data-oblivious computation. Unpublished (2015). http://oblivc.org/downloads/oblivc.pdf