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\textsuperscript{⋆}, a new domain-specific language (DSL) implementation for writing MPCs. WYS\textsuperscript{⋆} is a Verified, Domain-Specific Integrated Language Extension (VDSILE), a new kind of embedded DSL hosted in F\textsuperscript{*}, a full-featured, verification-oriented programming language. WYS\textsuperscript{⋆} source programs are essentially F\textsuperscript{*} programs written against an MPC library, meaning that the programmers can use F\textsuperscript{*}'s logic to verify the correctness and security properties of their programs. To reason about the distributed semantics of these programs, we formalize a deep embedding of WYS\textsuperscript{⋆}, also in F\textsuperscript{*}. We mechanize the necessary metatheory to prove that the properties verified for the WYS\textsuperscript{⋆} source programs carry over to the distributed, multi-party semantics. Finally, we use F\textsuperscript{*}'s extraction mechanism to extract an interpreter that we have proved matches this semantics, yielding a verified implementation. Indeed, WYS\textsuperscript{⋆} is the first DSL to enable formal verification of source MPC programs, and also the first MPC DSL to provide a verified implementation. With WYS\textsuperscript{⋆} we have implemented several MPC protocols, including private set intersection, joint median, and an MPC-based card dealing application, and have verified their security and correctness.

1. Introduction

Secure multi-party computation (MPC) is a framework that enables two or more parties to compute a function \( f \) over their private inputs \( x_1, \ldots, x_n \), so that no party sees any of the others’ inputs, but rather only sees the output \( f(x_1, \ldots, x_n) \). Utilizing 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 [6, 18, 43, 49]. 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. Among other applications, MPC has been used for auctions [13], detecting tax fraud [12], managing supply chains [24], and performing privacy preserving statistical analysis [23].

Typically, cryptographic protocols expect \( f \) to be specified as a boolean or arithmetic circuit. Programming directly with circuits and cryptography via a host-language API is painful, so starting with the Fairplay project [31] many researchers have designed higher-level domain-specific languages (DSLs) in which to program MPCs [8, 11, 20, 21, 26, 29, 30, 35, 36, 39, 41, 46]. These DSLs compile source code to circuits which are then given to the underlying protocol. While doing this undoubtedly makes it easier to program MPCs, these languages still have several drawbacks regarding both security and usability.

First, MPC participants should be able to reason that \( f \) is sufficiently privacy preserving, i.e., that its output will not reveal too much information about the inputs [32]. The goal of an MPC DSL is secure computations, and such reasoning gives assurance that this goal is being achieved. Yet, only a few DSLs (Sharemind DSL [26], Wysteria [39], and SCVM [29]) have a mathematical semantics that can serve as a basis for formal reasoning.

Second, those languages that do have a semantics lack support for mechanized reasoning about MPC programs: only by-hand proofs are possible, which provide less assurance than formally verified proofs. A middle ground might be a mechanization of the semantics and its metatheory [3], which adds greater assurance that it is correct [25]; but, so far, no MPC DSL has even had a mechanized semantics, let alone supporting mechanized reasoning about DSL source code.

Third, there is a gap between the semantics, if there is one, and the actual implementation. Within that gap is the potential for security holes. Formal verification of the MPC DSL’s toolchain can significantly reduce the occurrence of security-threatening bugs [10, 28, 37, 47, 48], but no existing MPC DSL implementation has been (even partially) formally verified—this should not be surprising since, as mentioned earlier, these DSLs lack a formal semantics on which to base a verification effort.

Finally, there is the practical problem that existing DSLs do not scale up, because they lack the infrastructure of a full-featured language. Adding more features (to both the language and the formalization) would help, but doing so quickly becomes unwieldy and frustrating, especially when the added features are “standard” and do not have much to do with MPC. We want access to libraries and frameworks for I/O, GUIs, etc. in a way that easily adds to functionality without adding complexity or compromising security.

This paper presents WYS\textsuperscript{⋆}, a new MPC DSL that addresses these problems. Unlike most previous MPC DSLs, WYS\textsuperscript{⋆} is not a standalone language, but is rather what we call a Verified, Domain-Specific Integrated Language Extension (VDSILE), a new kind of embedded DSL that can be hosted by F\textsuperscript{*} [44], a full-featured, verification-oriented programming language.

WYS\textsuperscript{⋆} (indeed, any VDSILE) has three distinguishing elements:

- **Integrated language extension** (Section 2). Programmers can write WYS\textsuperscript{⋆} MPC source programs in what is essentially an extended dialect of F\textsuperscript{*}. WYS\textsuperscript{⋆} inherits the basic programming model from Wysteria [39] (see related work in Section 6 for comparison with Wysteria). Like so-called shallow domain-specific language embeddings, WYS\textsuperscript{⋆} embeds the Wysteria-specific combinators in normal F\textsuperscript{*} syntax, with prescriptions on their correct use expressed with F\textsuperscript{*}’s dependent type-and-effect system. This arrangement has two benefits. Firstly, WYS\textsuperscript{⋆} programs can use, with no extra effort, standard constructs such as datatypes and libraries directly from F\textsuperscript{*}. Secondly, the programmers can formally verify properties, such as those related to correctness and security, about their MPC program.
using F’ s semi-automated verification facilities. WYS* is the first DSL to enable formal verification of the source MPC programs.

- Deep embedding of domain-specific semantics (Section 3). A shallow embedding implements the semantics of a DSL using the abstraction facilities of the host language, e.g., as a kind of library. However, for Wysteria (and hence WYS*) this is impossible because its core semantics cannot be directly encoded in F’ s semantics. A Wysteria program is like a SIMD program in which many parties alternate between computing locally and in parallel (private computations), and computing securely and jointly (secure computations). While such a program can be conceptually viewed as having a single thread of control, it is not directly implemented that way. As such, we take the approach of a typical deep embedding: We define an interpreter in F’ that operates over WYS* abstract syntax trees (ASTs), defined as an F’ data type; these trees are produced by running the F’ compiler (in a special mode) on the extended source program. Importantly, our interpreter does not need to understand all F’ constructs that might be extracted. Even though their use is intermixed with Wysteria-specific constructs, their semantics is handled by a lightweight FFI, mostly hidden from the source programmer.1

- Partially verified implementation (Sections 3 and 4). Within F’ we mechanize two operational semantics for WYS*: a conceptual single-threaded semantics that formalizes the SIMD view, mentioned above, and a distributed semantics, that formalizes the actual multi-party runs of the programs. Importantly, we have machine-checked proofs that (a) the conceptual single-threaded semantics is sound with respect to the actual distributed semantics, and (b) the distributed semantics is correctly implemented by our interpreter. As a result, we have verified that the properties formally proven about the Wysteria-extended F’ source programs carry over when these programs are run by multiple parties in a distributed manner. There is an important caveat though: Our interpreter makes use of a circuit library to compile ASTs to circuits and then execute them using the Goldreich, Miccianci, and Wigderson (GMW) multi-party computation protocol [18], but at present this library is not formally verified. Formal verification of GMW (which is, at present, an open problem) would add even greater assurance. WYS* is the first MPC DSL to provide a (partially) verified interpreter.

Scope and novelty of our verification results. Our verification results are different from a typical verification result that might either “mechanize metatheory” using a proof assistant for an idealized language, or might prove an interpreter or compiler correct w.r.t. a formal semantics—we do both. We mechanize the metatheory of WYS* establishing the soundness of the conceptual single-threaded semantics w.r.t. the actual distributed semantics, and we also mechanize the proof that the WYS* interpreter implements the correct WYS* semantics. In addition, WYS* also provides formal verification capabilities for the source MPC programs.

Summary of experimental evaluation. Using WYS* we have implemented several programs, including PSI, joint median, and an MPC-based card dealing application (Section 5). For PSI and joint median we implement two versions each: a straightforward one and a version that composes several small MPCs, which improves performance but increases the number of visible outputs. We formally prove, for both PSI and median, that the optimized and unoptimized versions are equivalent, both functionally and with respect to the privacy of parties’ inputs. In particular, WYS* enhances the Wysteria target semantics to “instrument” it with a trace of observations, and we prove that the visible events in the optimized ver-

---

1 This is similar to what happens with C#’s language integrated queries (LINQ), but is a bit more flexible, as discussed in Section 6.

---

2. Verified programming in WYS*

Consider a dating application that enables its users to compute their common interests without revealing all their private interests to one another. This is an instance of the private set intersection (PSI) problem. We illustrate the main concepts of WYS* by showing, in several stages, how to program, optimize, verify and deploy this application—Figure 1 provides an overview.

2.1 Secure computations with as_secc

In WYS*, an MPC is written as a single specification which executes in one of two computation modes. The primary mode is called sec mode, and it specifies a secure computation to be carried out among multiple parties. Here is the private set intersection example written in WYS*:

```plaintext
let psi a b input_a input_b = ...
```
The four arguments to psi are, respectively, principal identifiers for Alice and Bob, and Alice and Bob’s inputs, expressed as lists. The `as_sec ps f` construct indicates that thunk f should be run in sec mode. In this mode, the code may jointly access the secrets of the principals ps. In this case, we jointly intersect input_a and input_b, the inputs of a and b, and then return the same result r to both a and b. Outside of sec mode, Alice would not be permitted to see Bob’s input, and vice versa, but inside both can be made visible using the `reveal` coercion. Finally, the code constructs a map, associating a result for each principal (which in this case is the same)—give p v builds a singleton map p → v and ++ concatenates disjoint maps.

Running this code requires the following steps. First, we run the F* compiler in a special mode that `extracts` the above code. psi fst, into the WYS’ AST as a data structure in psi.ml. WYS’ has only a few constructs of its own, like `as_sec` (the full syntax is in Figure 2 in Section 3), and these are extracted to Wysteria-specific nodes. The rest of a program’s code 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, Alice and Bob, to run the extracted program using the WYS’ interpreter. This interpreter is written in F* and provably implements a deep embedding of the F* semantics, also specified in F* (shown in Figures 4, 5, and 6 in Section 3). This interpreter is extracted to OCaml code by a standard F* process. When each party reaches `as_sec ps f`, 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., `List.filter`).

The circuit is handed to a library by Choi et al. [14] that implements the GMW [18] multi-party computation protocol. Running the protocol at each party starts by confirming that they wish to run the same circuit, and then proceeds by generating (XOR-based) secret shares [42] for each party’s secret inputs. Running the GMW protocol involves evaluating the boolean circuit for f over the secret shares, involving communication between the parties for each AND-gate.

One obvious question is how both parties are able to get this process off the ground, running the program of four inputs, when only three of the inputs are known to them (the principals and their own input). 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 `input_a` and `input_b` are, respectively, list (sealed `{a}` int) and list (sealed `{b}` int). When the program is run on Alice’s host, the former will be a list of n of Alice’s values, whereas the latter will be a list of n garbage values (which we denote as ⊥). The reverse will be true on Bob’s host. When the circuit is constructed, each principal links their non-garbage values to the relevant input wires of the circuit. Likewise, they populate a copyable local of the output map with what is returned to them, mapping the other principals’ outputs to ⊥.

We would like MPC’s like psi to be called from normal F* programs. For example, we would like the logic for a dating application, which involves reading inputs, displaying results, etc. to be able to call into psi to compute common interests. To achieve this, WYS’ provides a way to compute a “single-party projection” of multi-party functions, i.e., a version of psi that can be called with just a single party’s inputs. The other party’s inputs are filled in with sealed garbage values, as described above. Calling this function from F* code also kicks off the WYS* interpreter, so that it can run psi as described above. When the interpreter completes, the result is returned and the F* program can continue.

2.2 Optimizing PSI with as_par

Although psi gets the job done, it turns out to be inefficient (as shown in §5). Better implementations of PSI involve performing a mixed-mode computation, where each participant evaluates some local computations in parallel (e.g., iterating over the elements of their sets) interleaved with small amounts of jointly evaluated, cryptographically secure computations. WYS’ second computation mode, called par mode, supports such mixed-mode computation. In particular, the construct `as_par ps f` states that each principal in ps should locally execute the thunk f, simultaneously (any principal not in the set ps simply skips the expression). Within f, principals may engage in secure computations via `as_sec`.

Below is an optimized version of PSI, based on an algorithm by Huang et al. [22], which uses as_par. The function psi_opt (line 12) begins by using as_par involving Alice and Bob. In the provided thunk, each principal calls for each,alice la lb, which in turn calls check_each bob a lb, for each element a of Alice’s list la. Secure computation occurs at the use of as_sec at line 8. Within the circuit, Alice and Bob securely compare their values ax and bx, and gather a list (list bool). There is one outer list for each of Alice’s elements, the ith inner list contains comparisons of Alice’s ith value with some of Bob’s values—rather than comparing each of Alice’s elements with all of Bob’s, the code is optimized (as described below) to omit redundant comparisons. At line 13, both parties build a matrix of comparisons from the boolean lists. Alice inspects the rows of the matrix (line 14) to determine which of her elements are in the intersection; Bob inspects the columns (line 15); and the joint function gives a result to each principal (line 16).

```
let rec for_each_alice a b la lb =
  if lb=[] then []
  else let lb, r = check_each bob a b (List.hd la) lb in
  r::for_each_alice a b (List.tl la) lb
and check_each bob a b ax lb =
  if lb=[] then [], []
  else let bx = List.hd lb in
  let r = as_sec `{a}` (fun () -> reveal ax = reveal bx) in
  if r then List.tl lb, [r]
  else let lb', r' = check_each bob a b ax (List.tl lb) in
  bx::lb', r'
let psi_opt a b la lb = as_par `{a}` (fun () ->
  let bs = build_matrix (for_each_alice a la) lb in
  let ia = as_par `{a}` (fun () -> filteri (contains true o row bs) ia) in
  let ib = as_par `{b}` (fun () -> filteri (contains true o col bs) lb) in
  give ia ++ give ib
```

The optimizations are at line 9. Once we detect that element ax is in the intersection, we return immediately instead of comparing ax against the remaining elements of lb. Furthermore, we remove bx from lb, excluding it from any future comparisons with other elements of Alice’s set la. Since la and lb are representations of sets (no repeats), all the excluded comparisons are guaranteed to be false.

One might wonder whether we could have programmed most of this code normal F*, relying on just sec mode for the circuit evaluation. However, recalling that our goal is to formally reason about the code and prove it correct and secure, par mode provides significant benefits. In particular, the SIMD model provided by WYS* enables us to capture many invariants for free. For example, proving the correctness of psi_opt requires reasoning that both participants iterate their loops in lock step—WYS* assures this by construction. Besides, the code would be harder to write (and read) if it were split across multiple functions or files. As a general guideline, we use F* for code written from the view of a single principal, and WYS* when programming for all principals at once, and rely on the FFI to mediate between the two.
2.3 Embedding a type system for WYS* in F*

Using the abstractions provided by Wysteria, designing various high-level, multi-party computation protocols is relatively easy. However, before deploying such protocols, three important questions arise. b

1. Is the protocol 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. Does the protocol correctly implement the desired functionality? For example, does it correctly compute the intersection of Alice and Bob’s sets?
3. Is the protocol secure? For example, do the optimizations of the previous section that omit certain comparisons inadvertently also release information besides the final answer?

By embedding WYS* in F* and leveraging its type system, we address each of these three questions. Our strategy is to make use of F*’s extensible, monadic dependent type-and-effect system to define a new indexed monad (called Wys) and use it to describe precise trace properties of Wysteria multi-party computations. Additionally, we make use of an abstract type, sealed ps t, representing a value accessible only to the principals in ps. Combining the Wys monad with the sealed type, we encode a form of information-flow control to ensure that protocols are realizable.

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 an as_sec context. The trace of a computation records the sequence and nesting structure of messages exchanged between parties as they jointly execute 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.

We now describe two of the Wysteria-specific combinators in WYS*, as_sec and reveal, and how we give them types in F* (a complete reference to WYS* API is provided in Figure 8 in the Appendix).

Defining as_sec in WYS*

val as_par: ps:prins → f(unit → Wys a pre post) → Wys a
(val (requires (fun m → m=Mode Par ps ∧ pre (Mode Sec ps)))
(val (ensures (fun m r tr → tr=[TMsg r] ∧ post (Mode Sec ps r) r]))

The type of as_sec is dependent on the first parameter, ps. Its second argument f is the thunk to be evaluated in as_sec mode. The result’s computation type has the form Wys a (requires φ) (ensures ψ), for some pre-condition and post-condition predicates φ and ψ, respectively. The free variables in the type (a, pre and post) are implicitly universally quantified (at the front); we use the requires and ensures keywords for readability—they are not semantically significant.

The pre-condition 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 pre-condition pre of f to be valid once the mode has transitioned to Mode Sec ps—line 2 says just this.

The post-condition 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 the empty trace [] (since f has no observables) according to post, the post-condition of f.

Defining reveal in WYS*—As discussed earlier, a value v of type sealed ps t encaptulates 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: #ps:prins → sealed ps α → Ghost α

val reveal: #ps:prins → x:sealed ps α → Wys α
(val (requires (fun m m.modesPar m.→ m.pss ps ∧ m.mode=Sec ps ∧ m.pss ps ∩ ps ≠ ϕ)
(ensures (fun m r tr → r.unseal x ∧ t r 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 pre-condition 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 ≠ ϕ. This is because the secure computation is executed for 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.

2.4 Correctness and security verification

Using the Wys monad and the sealed type, we can write down precise types for our psi program, proving various useful properties. For lack of space, we discuss the statements of the main lemmas we prove and the proof structure—the details of their machine-checked proofs are left to the supplement. By programming the protocols using the high-level abstractions provided by WYS*, our proofs are relatively straightforward. In particular, we rely heavily on the

2This is the “ideal functionality” ensured by the backend, e.g., GMW.
view that both parties execute (different fragments of) the same code. In contrast, reasoning directly against the low-level message passing semantics would be much more unwieldy. In Section 3, by formalizing the connection between the high- and low-level semantics, we justify our source-level reasoning.

We present the structure of the security and correctness proof for psi_opt by showing the top-level specification for psi_opt:

\[
\text{val} \ psi_{opt} : \text{la}_{list} (\text{sealed Alice int}) \rightarrow \text{lb}_{list} (\text{sealed Bob int}) \\
\rightarrow \text{Wys} (\text{map} \ (\text{Alice,Bob}) \ (\text{list int}))
\]

\[
(\text{requires} \ (\text{fun} \ m \rightarrow \text{mMode Par} \ (\text{Alice,Bob}) \ (\text{list int})) \\
\wedge \text{no_dups} \text{ia} \wedge \text{no_dups} \text{lb})
\]

\[
(\text{ensures} \ (\text{fun} \ m \rightarrow \text{r Par}) \ (\text{Alice,Bob}))
\]

The signature above establishes that when Alice and Bob simultaneously execute psi_opt (they start together in Par mode), with lists ia and lb containing their secrets (without any duplicates), then if and when the protocol terminates, they both obtain that same result is and lb corresponding to the intersection of their sets, i.e., the protocol is functionally correct.

To prove properties beyond functional correctness, we also prove that the trace of observable events from a run of psi_opt is described by the function psi_opt_trace ia lb. This is a purely specification function that, in effect, records each of the boolean results of every as_sec comparison performed during a run of psi—it has the same structure as for each alice and check each bob.

Given a full characterization of the observable behavior of psi_opt_trace ia lb in terms of its inputs, we can prove optimizations correct using relational reasoning [9] and we can also prove security hyperproperties [15] by relating traces from multiple runs of the protocol.

Our goal is to prove a noninterference with delimited release [40] property for psi_opt. Our attacker model is the “honest-but-curious” model where the attackers are the participants in the protocol themselves. That is, we assume that the participants in the protocol play their roles faithfully, but they are motivated to deduce as much as they can about the other participants’ secrets by observing the protocol. We do not aim to prove security properties against a third-party network adversary.

For psi, from the perspective of Alice as the attacker, we aim prove that for two runs of the protocol in which Alice’s input is constant but Bob’s varies, Alice learns no more by observing the protocol trace than what she is allowed to. Covering Bob’s perspective symmetrically, we show that in two runs of psi la0 lb0 and psi la1 lb1 that satisfy formula \( \Psi \) below, the traces observed by Alice and Bob are indistinguishable, up to permutation, where la0, la1, lb0, lb1 have type list int, the type of integer sets represented as lists.

\[
\Psi la0 la1 lb0 lb1 = \\
\text{intersect la0 lb0 = intersect la1 lb1} \\
\wedge \text{length la0 = length la1} \\
\wedge \text{length lb0 = length lb1}
\]

In other words, Alice and Bob learns no more than the intersection of their sets and the size of the other’s set; \( \Psi \) is the predicate that delimits the information released by the protocol. As far as we are aware, this is the first formal proof of correctness and security of Huang et al.’s optimized, private-set-intersection protocol.3

The proof is in the style of a step-wise refinement, via psi, an inefficient variant of the psi_opt program. Running psi la lb always involves doing exactly length la + length lb comparisons in two nested

3 In carrying out this proof, it becomes evident that Alice and Bob learn the size of each other’s sets. One can compose psi_opt with other protocols to partially hide the size—Wys’ makes it easy to compose protocols simply by composing their functions.

<table>
<thead>
<tr>
<th>Principal</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal set</td>
<td>s</td>
</tr>
<tr>
<td>FFI const</td>
<td>c, f</td>
</tr>
<tr>
<td>Constant</td>
<td>e</td>
</tr>
<tr>
<td>Expression</td>
<td>e</td>
</tr>
<tr>
<td>\text{val} psi_is_secure : la0 la1 lb0 lb1 \rightarrow Lemma</td>
<td></td>
</tr>
</tbody>
</table>

\[
(\text{requires} \ (\Psi la0 la1 lb0 lb1)) \\
(\text{ensures} \ (\text{permutation} \ (\text{trace psi la0 lb0}) \\
(\text{trace psi la1 lb1})))
\]

We reason about the traces of psi only up to permutation. Given that Alice has no prior knowledge of the choice of representation of Bob’s set (Bob can shuffle his list), the traces Alice observes are equivalent up to permutation—we can formalize this observation using a probabilistic, relational variant of F* [5], but have yet to do so.

As a next step, we prove that optimizing psi to psi_opt is secure by showing that there exists a function f, such that for any trace \( tr = \text{trace psi la lb} \), the trace of psi_opt, \( \text{trace psi_opt la lb} \), can be computed by \( f(\text{length la} \ tr) \). In other words, the trace produced by psi_opt 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 la lb. As such, the optimizations do not reveal further information.

3. Formalizing 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; instead they have a distributed semantics carried out by multiple parties. So, how do the properties that we verify using F* carry over to the actual runs?

In this section, we present the metatheory that answers this question. First, we formalize the Wys’ single-threaded (ST) semantics, arguing that it faithfully realizes the F* semantics, including the Wys’ API presented in Section 2. Next, we formalize the distributed (DS) semantics that the multiple parties use to run Wys’ programs. Our theorems establish the correspondence between the two semantics, thereby ensuring that the properties that we verify using F* carry over to the actual protocol runs. We have mechanized all the metatheory presented in this section in F*.

3.1 Syntax

Figure 2 shows the complete syntax of Wys*. Principal and principal sets are first-class values, and are denoted by p and s respectively. Constants in the language also include \( () \) (unit), booleans, 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 Section 2, expression mkmap e1 e2 creates a map from principals in e1 (which is a principal set) to the value computed by e2, project e1 e2 projects the value of principal e1 from the map e2, and concat e1 e2 concatenates the two maps.

Draft 5

2016/8/23
Rule S-ASPAR implements the pre-condition on line 2. For the pre-condition on line 3, rule S-ASPARRET checks that the returned value can be sealed.\textsuperscript{4} The rule also generates a trace element TScope s T, as per the post-condition on line 4, and returns the sealed value, as per the return type of the API and the post-condition on line 5.

Next consider the rules S-ASSEC and S-ASSECRET. Again, we can see that the rules implement the type of assec\footnote{For technical reasons, function closures may not be sealed; see the technical report for details.} (shown in \S2). The rule S-ASSEC checks the precondition of the API, and the rule S-ASSECRET generates a trace observation TMsg v, as per the postcondition of the API.

In a similar manner, we can easily see that the rule S-REVEAL implements the corresponding pre- and postconditions as given in Section 2.

Rules S-MKMAP, S-PROI, and S-CONCAT implement map creation, projection, and concatenation respectively. For map creation, if the current mode is Par, the rule ensures that the parties in s can access the value v by requiring v to be a sealed value that all parties in s can reveal (s ⊆ s₂). The rule also requires (for both Par and Sec mode) that all the parties in the map domain are present in the current mode (s ⊆ s₁). In rule S-PROI, if the current mode is Par then the current party set must be a singleton equal to the index of the map projection, whereas if the current mode is Sec, then the index of the map projection must be present in the current party set. Rule S-CONCAT simply checks that the two maps have disjoint range, and returns the disjoint union (∪) of the two maps.

The rule S-FFI implements the FFI call by calling a host-language function esec.fii. As expected, calling a host-language function has no effect on the WYS\textsuperscript{+}-specific state. Concretely, this is enforced by F\textsuperscript{*}'s monadic encapsulation of effects. The remaining rules are straightforward.

### 3.3 Distributed semantics

The DS semantics implements judgments of the form \( \pi \rightarrow \pi' \), where a protocol \( \pi \) 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. Both kinds of configurations (local and secure) have the form C (per Figure 3).

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. This semantics is expressed with four rules, given in Figure 6, which state that either: (1) a principal can take a step in their local configuration, (2) a secure computation can take a step, (3) some principals can enter a new secure computation, and finally, (4) a secure computation can return the result to the (waiting) participants.

The first case is covered by rule P-PAR, which (nondeterministically) chooses a principal’s configuration and evaluates it according to the local evaluation judgment C \( \rightsquigarrow \) C\textsuperscript{\prime}, which is given in Figure 5 (discussed below). The second case is covered by P-SEC, which evaluates using the ST semantics. The last two cases are covered by F-ENTER and F-EXIT, also discussed below.

**Local evaluation.** The rules in Figure 5 present the local evaluation semantics. These express how a single principal behaves while in par-mode; as such, mode \( M \) will always be Par (p). Local eval-

---

\[ \text{val aspar} \text{\ : } \text{ps:} \text{prins} \rightarrow (\text{unit} \rightarrow \text{Wys} \text{ a pre post}) \rightarrow \text{Wys} \text{ (sealed ps a)} \]
Draft

Our formal development actually shares the code for both sets of rules, and differs for WYS* specific constructs. For an as_par expression, a principal either participates in the computation, or skips it. Rules L-ASPAR1 and L-ASPARRET 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. One difference in the rule L-ASPARRET is that the trace \( T \) is not scoped. In the DS semantics, traces only contain TMgs elements; i.e., a trace is the (flat) list of secure computation outputs observed by that active principal. If \( p \not\in s \), then the principal skips the computation with the result being a sealed value containing garbage \( \bullet \) (rule L-ASPAR2). The contents of the sealed value do not matter, since the principal will not be allowed to unseal the value anyway.

---

5 Our formal development actually shares the code for both sets of rules, using an extra flag to indicate whether a rule is “local” or “joint”.
Rule L-SEAL has the same intuition as above. Rule L-REVEAL allows principal $p$ to reveal the value sealed $s\ v$, only if $p \in s$. Rule L-MKMAP requires value $v$ to be a sealed value. In case the current principal $p$ is in the set $s$, the rule requires that $p$ can access the contents of the sealed value ($p \in s_2$) and creates a singleton map that maps $p$ to the contents of the sealed value. In case the current principal is not in the set $s$, the rule simply creates an empty map. Rule L-PROJ projects the current principal’s mapping from the map $m$. The rule for map concatenation is straightforward.

As should be the case, there are no local rules for as_sealed—to perform a secure computation parties need to combine their data and jointly do the computation.

**Entering/exiting secure computations.** Returning to Figure 6, Rule P-ENTER handles the case when principals enter a secure computation. It requires that all the principals $p \in s$ must have the expression form as_sealed $s$ ($L_p, \lambda x.e$), 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$_v$ function on values as follows:

$$
\begin{align*}
\text{combine}_v(v, v) & = v \\
\text{combine}_v(v, \bullet) & = v \\
\text{combine}_v(\bullet, v) & = v \\
\text{combine}_v(\text{sealed } s_{v_1}, \text{sealed } s_{v_2}) & = \text{sealed } s (\text{combine}_v(v_1, v_2)) \\
\end{align*}
$$

The first two rules handle the case when one of the values is garbage; in these cases, the function picks the other value. For sealed values, if the set $s$ is the same, the function recursively combines the contents. The combine function for the environments combines the mappings pointwise. The combine functions for $n$ values and environments is a folding of the corresponding function.

So now, consider the following code:

```plaintext
let x = as_par alice (fun x -> 2) in
let y = as_par bob (fun x -> 3) in
let z = as_sealed (alice, bob) (fun z -> (unsealed x) + (unsealed y)) in ...
```

In alice’s environment $x$ will be mapped to sealed alice 2, whereas in bob’s environment it will be mapped to sealed alice $\bullet$. Similarly, in alice’s environment $y$ will be mapped to sealed bob $\bullet$, whereas in bob’s environment it will be mapped to sealed bob 3. Before the secure computation, their environments will be combined, producing an environment with $x$ mapped to sealed alice 2 and $y$ mapped to sealed bob 3, and then, the secure computation function will be evaluated in this new environment.

Although the combine$_v$ function as written is a partial function, our metatheory guarantees that at runtime, the function always succeeds. Since the principals are computing the same program over their view of the data, these views are structurally similar.

So, 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.

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

$$
\begin{align*}
\text{slice}_v (p \ u) & = p' \\
\text{slice}_v (\text{sealed } s \ u) & = \text{sealed } s \bullet, \text{ if } p \notin s \\
\text{slice}_v (p \ \text{sealed } s \ u) & = \text{sealed } s \ (\text{slice}_v (p \ u), \text{ if } p \in s)
\end{align*}
$$

As an example, consider the following code:

```plaintext
let x = as_sealed (alice, bob) (fun x -> let y = ... in seal alice y)
```

Since the return value of the secure computation is sealed for alice, bob will get a sealed alice $\bullet$, produced using the slice$_v$ function on the result of seal alice $y$.

In the rule P-EXIT, the $\triangleleft$ notation is defined as:

$$
M; X; L; T; \triangleleft v = M; X; L; \triangleleft \text{append } T; \text{TMag } \triangleleft v
$$

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 $\pi C$ for an ST configuration $C$. In the $P$ component of $\pi C$, 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:

$$
\begin{align*}
\text{slice}_{tr} \ p \ (TMsg v) & = [TMsg (\text{slice}_v (p \ v))] \\
\text{slice}_{tr} \ p \ (TScope s T) & = \text{slice}_{tr} \ p \ T, \text{ if } p \in s \\
\text{slice}_{tr} \ p \ (TScope s T) & = [], \text{ if } p \notin s
\end{align*}
$$

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., $C.e$ is a value and $C.X$ 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 (alice $s C_1$) $\rightarrow^*$ (slice $s C_2$) such that (slice $s C_2$) is terminal.

Notably, each principal’s value and trace in protocol (slice $s C_2$) is the slice of the value and trace in $C_2$.

**Confluence**

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

**Definition 1** (Strong termination). A protocol $\pi$ strongly terminates in the terminal protocol $\pi_t$, written as $\pi \Downarrow \pi_t$, if all possible runs of $\pi$ terminate in some number of steps in $\pi_t$.

Our confluence result then says:

**Theorem 2** (Confluence of DS). If $\pi \rightarrow^* \pi_t$ and $\pi_t$ 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 post-condition 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 $\bullet$. Aside from the correspondence on results, our semantics also covers
correspondence on traces. Thus, via our VDSILE embedding of Wysteria in F\(^\ast\), the correctness and security properties that we prove about a WYS\(^\ast\) program using F\(^\ast\)’s logic, hold for the program that actually runs.

Of course, this statement is caveatied by how we produce an actual implementation from the DS semantics; details are presented in the next section.

4. Implementation

This section describes our WYS\(^\ast\) interpreter. We have proved that the core of this interpreter implements our formal semantics, adding confidence that bugs have not been introduced in the translation from formalism to implementation.

4.1 WYS\(^\ast\) interpreter

The formal semantics presented in the prior section is mechanized as an inductive type in F\(^\ast\). This style is useful for proving properties, but does not directly translate to an implementation. Therefore, we implement an interpretation function step in F\(^\ast\) and prove that it corresponds to the rules; i.e., that for all input configurations \(C, \text{step}(C) = C'\) implies that \(C \rightarrow C'\) according to the semantics. Then, the core of each principal’s implementation is an F\(^\ast\) stub function \(\text{tstep}\) that repeatedly invokes \(\text{step}\) on the AST of the source program (produced by the F\(^\ast\) extractor run in a custom mode), unless the AST is an as\(\text{sec}\) node. Functions \(\text{step}\) and \(\text{tstep}\) are extracted to OCaml by the standard process.

Local evaluation is not defined for as\(\text{sec}\), so the stub implements what amounts to P-\textsc{enter} and P-\textsc{exit} from Figure 6. When the stub notices the program has reached an as\(\text{sec}\) expression, it calls into a circuit library we have written that converts the AST of the second argument of as\(\text{sec}\) to a boolean circuit. This circuit and the encoded inputs are communicated to a co-located server, written using a library due to Choi et. al. [14] that implements the GMW MPC protocol. 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 F\(^\ast\) formalization of WYS\(^\ast\) is 5000 lines of code, including all the metatheory. It makes abundant use of F\(^\ast\)’s dependent types to state and prove invariants. The implementation of the (verified) step function is essentially a big switch-case on the current expression, and is 60 lines of code. The \(\text{tstep}\) 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 the F\(^\ast\) extractor (including our custom WYS\(^\ast\) mode for it) are part of our trusted computing base. As such, bugs in them could constitute security holes. Verifying these components as well (especially the circuit library and the GMW implementation, which are open problems to our knowledge) is interesting future work.

4.2 FFI

When writing a source WYS\(^\ast\) program (in F\(^\ast\)), the programmer can call functions from an FFI module.\(^6\) During compilation, the FFI module is extracted to OCaml using the regular F\(^\ast\) extraction. The custom mode of the F\(^\ast\) extraction that we have implemented, identifies the FFI calls in the WYS\(^\ast\) program, and extracts them to an E\(\text{ffi}\) AST form, which is part of the AST expression type.

\[
\text{type exp = } \ldots \\
\quad | \text{E}_\text{ffi}: f:\alpha \rightarrow \text{list}\ exp \rightarrow \text{inj}\beta \rightarrow \exp
\]

\(^6\) How F\(^\ast\) programs call into WYS\(^\ast\) functions was described in Section 2.1.

\[\text{Interpreter specific values, such as V}_\text{seal}, \text{are passed as is. The FFI module does not have access to the WYS\(^\ast\) API in F\(^\ast\), and hence it can only use these values parametrically. exec}\_\text{ffi} \text{then calls the OCaml function f with the un-embedded arguments. The OCaml function returns some result, that needs to be embedded back to the AST. So, the question is how can we embed the result at runtime? Inspecting the type of the result is not an option. The custom F\(^\ast\) extraction mode comes to our rescue.}

When the extractor compiles an FFI call in the source program to an E\_\text{ffi} node, it has the type information for the return value of the FFI call. Using this information, it instruments the E\_\text{ffi} node with an injection, a function that can be used at runtime to embed the FFI call result back to the AST. For example, if the result is \(\text{()\), the injection is (an OCaml function) fun x \rightarrow V\_\text{unit}. If the return value is an interpreter value (e.g. V\_\text{seal}), the injection is the identity. If the return value is some host value (such as a list, tuple, or int), the injection creates an E\_\text{ffi} node. exec\_\text{ffi} uses the injection to embed the result back to the AST, and returns it to the interpreter.}

Our interface essentially provides a form of monomorphic, first-order interoperability between the (dynamically typed) interpreter and the host language. We do not foresee any problems extending our current work to higher order with coercions [19].

5. Applications

Private set intersection. We evaluate the performance of the psi (computing intersection in a single secure computation), and the psi\_opt (the optimized version) algorithms from Section 2. The programs that we benchmark are slightly different than the ones presented there, in that the local col and row functions are not the verified ones. The results are shown in Figure 7. We measure the time (in seconds) for per party set sizes 96, 128, and 256, and intersection densities (i.e. the fraction of elements that are common) 0.5, 0.75, and 0.9.
The time taken by the unoptimized version is independent of the intersection density since it always compares all pairs of values. However, as the intersection density increases, the optimized version performs far better—it is able to skip many comparisons. For lower densities (<0.35), the optimization does not improve performance, as the algorithm essentially becomes quadratic, and the setup cost for each secure computation takes over.

**Joint median.** We program unoptimized and optimized versions of the two-party joint median [38]. The programs take two distinct, sorted inputs from alice, \(x_1\) and \(x_2\), and two distinct, sorted inputs from bob, \(y_1\) and \(y_2\) and return the median of all four. In the unoptimized version, the whole computation takes place as a monolithic secure computation, whereas the optimized version breaks the computation, revealing some intermediate results, and off-loading some parts to the local hosts (much like PSI). We refer the reader to [38] for more details of the algorithms.

For both the versions, we prove functional correctness:

```plaintext
val median
x:sealed Alice (int * int) \rightarrow y:sealed Bob (int * int) \rightarrow Wys int
(requires fun m \rightarrow m = Mode Par (Alice, Bob))
(ensures fun r t \rightarrow (pre unseal x) (unseal y) \Rightarrow
r = median_spec (unseal x) (unseal y))
```

where median_spec is an idealized median specification. For the unoptimized version, we prove that the trace \(\tau\), where \(\tau\) is the result of the computation, basically reflecting that the principals only see the final result. We prove the optimized version to be secure using a relational argument that the trace does not reveal more than the output.

**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 [42]. 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 a secure block.

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 block is secret shares of \(n\). Before declaring \(n\) as the newly dealt card, the parties need 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 block 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* exports the following API for secret shares:

```plaintext
type Sh: Type \rightarrow Type
val Sh a: Type \rightarrow Sh a
val can_sh: Type \rightarrow Type
assume Can_sh int: can_sh int
val v_of_sh: #a:Type \rightarrow sh:Sh a \rightarrow Ghost a
val ps_of_sh: #a:Type \rightarrow sh:Sh a \rightarrow Ghost ints
val mk_sh: #a:Type \rightarrow x: a \rightarrow Wys (Sh a)
(requires fun m \rightarrow m = Mode Sec ∧ can_sh a)
(ensures fun m r t \rightarrow v_of_sh r = x ∧ ps_of_sh r = m.ps ∧ \tau = [])
val comb_sh: #a:Type \rightarrow x:Sh a \rightarrow Wys a
(val comb_sh: #a:Type \rightarrow x:Sh a \rightarrow Wys a
(requires fun m \rightarrow m.mode = Sec ∧ ps_of_sh x = m.ps)
(ensures fun m r tr \rightarrow v_of_sh x = r ∧ \tau = [])
```

Type Sh a types the shares of values of type a. Our implementation currently supports shares of int values only; the can_sh predicate enforces this restriction on the source programs. Extend secret shares support for other types (such as pairs) should be straightforward. 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 parties):

```plaintext
val check_fresh:
Ilist (Sh int) \forall s', mem s'1 \rightarrow ps_of_sh s' = abc
⇒ \rightarrow s:Sh int \rightarrow ps_of_sh s = abc
⇒ \rightarrow Wys bool (requires fun m \rightarrow m = Mode Par abc)
(ensures fun r t \rightarrow r \Leftrightarrow (\forall s': mem s'1 \Rightarrow not (v_of_sh s' = v_of_sh s)))
```

The specification says that the function takes two arguments: \(I\) 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. Figure 9 (in the Appendix) shows the complete code for the card dealing application.

**Other applications and secure server.** We have implemented some more applications in Wys*, including a geolocation sharing application. At the moment, we have only run these applications using a secure server backend. In this backend, as see works by literally sending code and inputs to a separate server that implements the ST semantics directly. The server returns the result with a cryptographic proof of correctness to each party (we have verified the use of cryptography using a technique similar to Fournet et al. [17]). We conjecture that such a server could be useful for a trusted hardware-based deployment scenario.

6. Related work

**Source MPC verification.** While the verification of the underlying cryptographic protocols has received some attention [1], the verification of the MPC source programs has remained largely unexplored. The only previous work that we know of is Backes et. al. [4] 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* permits direct verification of higher-level MPC source programs and provides a verified toolchain.

**Wysteria.** The Wys* API (Figure 8) is based on programming abstractions of Wysteria [39]. However, unlike Wysteria which is a standalone language, Wys* is implemented as a VDSILE in F*. Wysteria lacks the source program verification capabilities, and has an unverified implementation. In addition, being a standalone language, it lacks the support for standard language constructs such as datatypes and libraries. Wys* programs, on the other hand, can freely use datatypes and libraries from the host language F*.
through a transparent and lightweight FFI mechanism outlined in Section 4.2. At the language level, Wy’s⋆ enhances the Wysteria target semantics with observable traces that can be used to verify the security properties of Wy’s⋆ programs.

**MPC DSL extensions.** As listed in the introduction, several MPC DSLs have been proposed in the literature [8, 11, 20, 21, 26, 29, 30, 35, 36, 39, 41, 46]. Most of these languages have standalone implementations, but like Wy’s⋆, a few are implemented as language extensions. Launchbury et al. [27] define a Haskell-embedded DSL for writing low-level “share protocols” on a multi-server “SMC machine”. OblivC [50] 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; Wy’s⋆ is unique in its use of the VDSILE strategy.

**DSL implementation strategies.** DSLs are implemented in various ways, such as by developing a standalone compiler/interpreter, or by embedding the DSL (shallowly or deeply) in a host language. VDSILE’s language-integrated syntax bears relation to the approach taken in LINQ [33], 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 [7] in Coq [45]) to permit formal proofs of DSL programs. F⋆ provides some advantages as a host language since it is both higher-order and effectful, making it easier to write DSL combinators for effectful languages while still proving that DSL programs have good properties, and being able to (easily) extract those programs to runnable code.

## 7. Conclusions

This paper has proposed Verified, Domain-Specific Integrated Language Extensions (VDSILE) as a new way to implement a domain-specific language. The paper specifically applies the idea to design and implement Wy’s⋆, a new MPC DSL that is hosted in F⋆. Wy’s⋆ inherits the basic programming model from Wysteria. However, by virtue of being implemented as a VDSILE, it provides several novel capabilities missing from all previous MPC DSLs, including Wysteria. Wy’s⋆ is the first DSL to enable formal verification of the source MPC programs. Wy’s⋆ is also the first MPC DSL to provide a (partially) verified interpreter. Furthermore, Wy’s⋆ programs can freely use standard constructs such as datatypes and libraries directly from F⋆, thereby making it more scalable and usable. All these capabilities constitute a significant step towards making MPC more practical and trustworthy. The paper has reported on several MPC applications programmed in Wy’s⋆, and verified for correctness and security.

## References


A. Appendix
\begin{figure}
\begin{verbatim}
type as_mode =
  | Par
  | Sec

type mode =
  | Mode: m:as_mode → ps:prins → mode

type telt =
  | TM: #a:Type → x:a → telt
  | TS: ps:prins → t:telt list → telt

type trace = list telt

kind Requires = mode → Type
kind Ensures (a:Type) = mode → a → trace → Type

effect Wys (a:Type) (req:Requires) (ens:Ensures a)

val as_sec: ps:prins → f:(unit → Wys a) pre post
  → Wys a (requires (fun m → m=Mode Par ps ∧ pre (Mode Sec ps))) (ensures (fun m r tr → tr=[TM r] ∧ post (Mode Sec ps) r []))

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

type sealed : prins → Type → Type

val unseal: #ps:prins → sealed ps a → Ghost a

val seal: ps:prins → x:α → Wys (sealed ps α) (requires (fun m → m.ps ⊆ m.ps)) (ensures (fun m r tr → x=unseal r ∧ tr=[]))

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

type map : prins → Type → Type

(* we omit the signatures and axioms for ghost select, contains, join, and const_map functions on maps *)
(* type eprins is also a set of principals, but unlike prins, it admits the empty set *)

val mkmap_p: ps:prins → eps:epsprins → x:sealed α ps1
  → Wys (map α eps) (requires (fun m → m.mode=Par ∧ pre m.ps1 ∧ eps ⊆ m.ps))
      (ensures (fun m r tr → r = const_map eps (unseal x) ∧ tr = []))

val mkmap_s: eps:epsprins → x:α
  → Wys (map α eps) (requires (fun m → m.mode=Sec ∧ eps ⊆ m.ps)) (ensures (fun m r tr → r = const_map eps x ∧ tr = []))

val project: eps:epsprins → p:prin → x:map α eps{contains p x}
  → Wys α (requires (fun m → m.mode=Par ⇒ m.ps → singleton p ∧ m.mode=Sec ⇒ mem p m.ps))
      (ensures (fun m r tr → r = select p x ∧ tr = []))

val concat: eps:epsprins → eps:epsprins → y:map α eps_y → y:map α eps_y
  → Wys (map α (eps_x ∪ eps_y)) (requires (fun m → disjoint (dom x) (dom y))) (ensures (fun m r tr → r = join x y ∧ tr = []))

val of_sh: #a:Type → sh:Sh a → Ghost a

val of_ps: Sh:Type → Sh a → Ghost prins

val mk_sh: #a:Type → x:a
  → Wys (Sh a) (requires (fun m → m.mode = Sec ∧ can_sh a)) (ensures (fun m r tr → of_sh r = x ∧ ps_of_sh r = m.ps ∧ tr = []))

val comb_sh: #a:Type → Sh a
  → Wys a (requires (fun m → m.mode = Sec ∧ ps_of_sh x = m.ps)) (ensures (fun m r tr → of_sh x = r ∧ tr = []))
\end{verbatim}
\end{figure}

\begin{center}
Figure 8. WYS* API
\end{center}
val check_fresh : list (Sh int) → Wys bool → \forall s. FStar.List.mem s l \implies ps_of_sh s = abc → s : Sh int \{ ps_of_sh s = abc \} → Wys bool (\forall m = Mode Par abc) (\forall . \forall x ∈ s) (\forall s'. FStar.List.mem s' l \implies (\forall . \forall s'' ∈ s') = (\forall s'. FStar.List.mem s' l \implies not (\forall . \forall s'' ∈ s')))

let rec check_fresh l s = if l = mk_nil () then true else
  let x = hd_of_cons l in
  let check_one : unit → Wys bool (\forall m = Mode Sec abc) (\forall . \forall x ∈ s) (\forall s. FStar.List.mem s l \implies ps_of_sh s = abc)

let deal ps : print \{ ps = abc \} → share/list (Sh int) \{ \forall x. FStar.List.mem x shares \implies ps_of_sh s = abc \} → Wys unit (\forall m = Mode Par abc) (\forall . \forall x ∈ s) (\forall s. FStar.List.mem s l \implies ps_of_sh s = abc)

let add : add \{ \forall m = Mode Sec abc \} \{ \forall x ∈ s \implies ps_of_sh s = abc \} → Wys unit \{ \forall m = Mode Sec abc \} \{ \forall x ∈ s \implies ps_of_sh s = abc \}

let new_sh = as_sec abc add in

(* shares of new card *)

let new_sh = as_sec abc add in

(* compute % 52 *)

let max = 52 in

let conditional_sub : s : Sh int \{ ps_of_sh s = abc \} → Wys (Sh int) \{ \forall m = Mode Sec abc \} \{ \forall x ∈ s \implies ps_of_sh s = abc \}

let new_sh = as_sec abc (conditional_sub new_sh) in

let fresh = check_fresh shares new_sh in

if fresh then
  let card : unit → Wys unit \{ \forall m = Mode Sec abc \} \{ \forall x ∈ s \implies ps_of_sh s = abc \}

let mk_tuple (mk_cons new_sh shares) c else
mk_tuple shares max

Figure 9. Card dealing for 3 parties in WYS∗