Wys*: A Verified Language Extension for Secure Multi-party Computations

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) implementation for writing MPCs. Wys* is a *Verified, Domain-Specific Integrated Language Extension* (VDSILE), a new kind of embedded DSL hosted in F*, a full-featured, verification-oriented programming language. Wys* source programs are essentially F* programs written against an MPC library, meaning that programmers can use F*'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*, 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 mechanism to extract an interpreter that we have proved matches this semantics, yielding a verified implementation. Wys* 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* 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 $x_1, ..., x_n$ so that no party sees any of the others' inputs, but rather only sees the output $f(x_1, ..., x_n)$. 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 [1]–[4]. 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 [5], detecting tax fraud [6], managing supply chains [7], and performing privacy preserving statistical analysis [8].

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 [9] many researchers have designed higher-level domain-specific languages (DSLs) in which to program MPCs [10]–[21]. 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 [22]. The goal of an MPC DSL is secure computations, and such reasoning gives assurance that this goal is being achieved.[1] Yet, only a few DSLs (Sharemind DSL [21], Wysteria [19], and SCVM [20]) 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 machine-checked proofs. A middle ground might be a mechanization of the semantics and its metatheory [23] (e.g., in a system like Coq, Agda, or Isabelle), which adds greater assurance that it is correct [24], but so far no MPC DSL has even had a mechanized semantics.

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 [25]–[29], but no existing MPC DSL implementation has been (even partially) formally verified—this should not be surprising since, as mentioned earlier, these DSLs have lacked 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*, a new MPC DSL that addresses these problems. Unlike most previous MPC DSLs, Wys* 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

1. 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.

be hosted by F⋆ [30], a full-featured, verification-oriented programming language.

Wys⋆ has the following distinguishing elements:

• *Integrated language extension* (Section 3). Programmers can write Wys⋆ MPC source programs in what is essentially an extended dialect of F⋆. Wys⋆ inherits the basic programming model from Wysteria [19] (see Section 2 for comparison with Wysteria). Like so-called *shallow* domain-specific language embeddings, Wys⋆ embeds the Wysteria-specific combinators in normal F⋆ syntax, with prescriptions on their correct use expressed with F⋆'s dependent type-and-effect system. This arrangement has two benefits. Firstly, Wys⋆ programs can use, with no extra effort, standard constructs such as datatypes and libraries directly from F⋆. Secondly, 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 source MPC programs.

• *Deep embedding of domain-specific semantics* (Section 4). 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 Wys⋆ this is impossible because its core semantics cannot be directly encoded in F⋆'s semantics. A Wys⋆ program is like a SIMD program whose execution alternates between in-parallel computations at each party, privately, and joint computations involving all parties, securely. 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, 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 novel foreign function interface (FFI) that is easy to use, both when programming and when verifying.

• *Verified implementation, modulo a trusted crypto library* (Sections 4 and 5). Within F⋆, we mechanize two operational semantics for deep-embedded Wys⋆ ASTs: a conceptual single-threaded semantics that formalizes the SIMD view and a distributed semantics that formalizes the actual multi-party runs of the programs. We prove that (a) the conceptual single-threaded semantics is *sound* with respect to the actual distributed semantics (including the semantics of the FFI calls), and (b) the distributed semantics is correctly implemented by our interpreter. These proofs are checked by F⋆'s proof checking algorithm. As a result, we have verified that the properties formally proven about the Wys⋆ 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, Micali and Wigderson (GMW) multi-party computation protocol [3], 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.

Using Wys⋆ we have implemented several programs, including PSI, joint median, and an MPC-based card dealing application (Section 6). For PSI and joint median we implement two versions: 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 versions' traces provide neither participant with any additional information about the other's secrets. Performance experiments confirm that the optimized versions do indeed perform better. Our card dealing application relies on Wys⋆'s support for secret shares [31]. We formally prove that our card dealing algorithm always deals a fresh card.

In summary, this paper's main contribution is Wys⋆, a new Verified, Domain-Specific Integrated Language Extension (VDSILE) for supporting secure multiparty computation. Wys⋆ is unique in its use of formal methods to ensure that both its underlying implementation and the programs written in it behave according to important correctness and security properties. The Wys⋆ implementation, example programs, and proofs are publicly available online on github.

2. Related work

Wys⋆'s computational model is based on programming abstractions of a previous domain-specific language, Wysteria [19]. But Wys⋆ offers several new contributions. First, while Wysteria is a standalone language, Wys⋆ is implemented as an extension to the language F⋆. As such, Wys⋆ programs can freely use datatypes and libraries from the host language via a novel verification-friendly FFI mechanism (outlined in Section 5.2). This architecture allows programs to scale more easily (rather than requiring constant extension and reimplementation of the standalone language, and its verification results). Second, relying on F⋆'s verification features, the Wys⋆ implementation has also been verified, except for its cryptographic libraries. Third, since Wys⋆ is also an embedded DSL in F⋆, MPC programs can be formally verified to satisfy correctness and security properties. In support of security verification, Wys⋆'s semantics includes a notion of *observable traces*, which we use to state and prove information flow properties. Although it is our point of departure, Wysteria enjoys none of these benefits.

MPC DSLs and DSL extensions. In addition to Wysteria, several other MPC DSLs have been proposed in the literature [10]–[21]. Most of these languages have standalone implementations, and the drawbacks that come with these. Like Wys⋆, a few are implemented as language extensions. Launchbury et al. [32] describe a Haskell-embedded DSL for writing low-level "share protocols" on a multi-server "SMC machine". OblivC [33] 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 its use of the VDSILE strategy, that combines a shallow embedding to support source program verification and a deep embedding to support a non-standard target semantics.

Mechanized metatheory. Our verification results are different from a typical verification result that might either "mechanize metatheory" using a proof assistant for an idealized language [23], or might prove an interpreter or compiler correct w.r.t. a formal semantics [26]—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.

Source MPC verification. While the verification of the underlying crypto protocols has received some attention [34], the verification of the MPC source programs has remained largely unexplored. The only previous work that we know of is Backes et. al. [35] 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 in addition provides a verified toolchain.

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 embedding the DSL (shallowly or deeply) in a host language. VDSILE's language-integrated syntax bears relation to the approach taken in LINQ [36], 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 [37] in Coq [38]) to permit formal proofs of DSL programs. F⋆ provides some advantage 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.

In sum, WYS⋆ is the first DSL to enable formal verification of efficient source MPC programs as written in a full-featured host programming language. WYS⋆ is also the first MPC DSL to provide a (partially) verified interpreter.

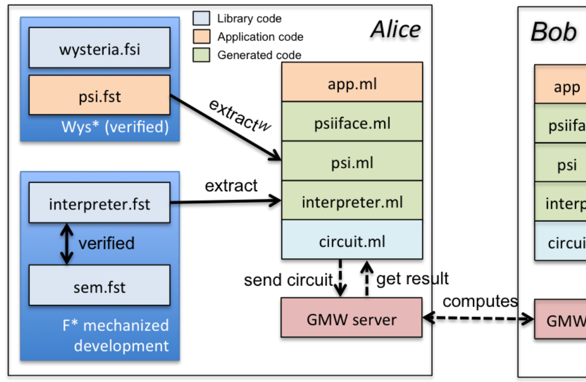

Figure 1. Architecture of an WYS⋆ deployment

3. 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.

3.1. Secure computations with as_sec

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⋆:

```
let psi a b input_a input_b l_a l_b =
    as_sec {a,b} (fun () →
        let r = List.intersect (reveal input_a) (reveal input_b) l_a l_b
        give a r ++ give b r)
```

The six arguments to psi are, respectively, principal identifiers for Alice and Bob, Alice and Bob's secret inputs, expressed as lists, and their (public) lengths. 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 secret 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 4), 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 4). 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.intersect, List.mem, List.nth etc.). In the current example, the lengths of the input lists are required to be public in order for Alice and Bob to be able to create boolean circuits.

The circuit is handed to a library by Choi et al. [39] that implements the GMW [3] 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 [31] 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 this program of six inputs, when only five of the inputs are known to them (the principals, their own inputs, and the size of other party's 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 l_a of Alice's values, whereas the latter will be a list of l_b 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, the output map component of each party is derived from their output wires in the circuit, thus, each party only gets to see their own output.

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.

3.2. Optimizing PSI with as_par

Although psi gets the job done, it turns out to be inefficient for some cases (as shown in §6). Better implementations of PSI for such cases 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*'s 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 [40], 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).

```
1  let rec for_each_alice a b la lb =
2      if la=[] then []
3      else let lb, r = check_each_bob a b (List.hd la) lb in
4          r::for_each_alice a b (List.tl la) lb
5  and check_each_bob a b ax lb =
6      if lb=[] then [], []
7      else let bx = List.hd lb in
8          let r = as_sec {a,b} (fun () → reveal ax = reveal bx) in
9          if r then List.tl lb, [r]
10         else let lb', r' = check_each_bob a b ax (List.tl lb) in
11             bx::lb', r::r'
12 let psi_opt a b la lb = as_par {a,b} (fun () →
13  let bs = build_matrix (for_each_alice la lb) in
14  let ia = as_par {a} (fun () → filteri (contains true ∘ row bs) la) in
15  let ib = as_par {b} (fun () → filteri (contains true ∘ col bs) lb) in
16  give a ia ++ give b 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 in 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.

3.3. 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.

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. 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 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.

Formally, we define the following types for modes and traces. A mode Mode m ps is 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 block. 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 pre-condition on the mode in which e may be executed; and post is a post-condition relating the computation's mode, its result value, and its trace of observable events. When run in a context with mode m satisfying the pre-condition predicate pre m, e may send and receive message according to some trace tr, and if and when it returns, the result is a t-typed value v validating the post-condition predicate post m v tr. The style of indexing a monad with a computation's pre- and post-condition is a standard technique [30], [41], [42]—we defer the definition of the monad's bind and return to the actual implementation and focus instead on specifications of combinators specific to WYS*.

We now describe two of the Wysteria-specific combinators in WYS*, as_sec and reveal, and how we give them types in F*.

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] ∧ post (Mode Sec ps) 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.[2] 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.

2. This is the "ideal functionality" ensured by the backend, e.g., GMW.

***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: #ps:prins → sealed ps α → Ghost α

val reveal: #ps:prins → 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' $\subseteq$ ps. However, when executing in Mode Sec ps', only a subset of current participants is required: ps' $\cap$ ps $\neq \emptyset$. 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.

3.4. 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 actual implementation. By programming the protocols using the high-level abstractions provided by WYS*, our proofs are relatively straightforward. In particular, we rely heavily on the 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 4, 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:

```
val psi_opt: a:prin → b:prin
      → la:list (sealed {a} int) → lb:list (sealed {b} int)
      → Wys (map {a, b} (list int))
  (requires (fun m → m=Mode Par {a, b}
                   ∧ no_dups la ∧ no_dups lb))
  (ensures (fun m r tr →
      let ia = as_set (Map.get r a) in
      let ib = as_set (Map.get r b) in
      ia = ib ∧ ia = (as_set la ∩ as_set lb) ∧ tr=psi_opt_trace la lb))
```

The signature above establishes that when Alice and Bob simultaneously execute psi_opt (they start together in Par mode), with lists la and lb containing their secrets (without any duplicates), then if and when the protocol terminates, they both obtain that same results ia and ib 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 la lb is described by the function psi_opt_trace la lb. This is a purely specificational 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 la lb in terms of its inputs, we can prove optimizations correct using relational reasoning [43] and we can also prove security hyperproperties [44] by relating traces from multiple runs of the protocol.

Our goal is to prove a noninterference with delimited release [45] property for psi_opt. Since our attacker model is the "honest-but-curious" model (as mentioned in Section 1), 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 the protocol trace than what she is allowed to. Covering Bob's perspective symmetrically, we show that in two runs of psi la_0 lb_0 and psi la_1 lb_1 that satisfy formula Ψ below, the traces observed by Alice and Bob are indistinguishable, up to permutation, where la_0, la_1, lb_0, lb_1 have type lset int, the type of integer sets represented as lists.

```
Ψ la₀ la₁ lb₀ lb₁ =
        intersect la₀ lb₀ = intersect la₁ lb₁
      ∧ length la₀ = length la₁ ∧ length lb₀ = length lb₁
```

In other words, Alice and Bob learn no more than the intersection of their sets and the size of the other's set; Ψ 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 loops. We prove the following relational security property for psi, relating the traces trace_psi la_0 lb_0 and trace_psi la_1 lb_1—the formal statement of the lemma we prove in F* is shown below.

```
val psi_is_secure: la₀:_ → lb₀:_ → la₁:_ → lb₁:_ → Lemma
  (requires (Ψ la₀ la₁ lb₀ lb₁))
  (ensures (permutation (trace_psi la₀ lb₀)
                        (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* [46], but have yet to do so.

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.

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

Figure 2. WYS⋆ syntax

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=trace_psi la lb, the trace of psi_opt, trace_psi_opt la lb, can be computed by f (length la) tr. In other words, the trace produced psi_opt la lb 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.

We present more examples and their verification details in Section 6.

4. 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; 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 3. 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⋆.

4.1. 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 3, expression $\mathsf{mkmap}\ e_1\ e_2$ creates a map from principals in e_1 (which is a principal set) to the value computed by e_2. $\mathsf{project}\ e_1\ e_2$ projects the value of principal e_1 from the map e_2, and $\mathsf{concat}\ e_1\ e_2$ concatenates the two maps.

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

Figure 3. Runtime configuration syntax

Host language (i.e., F⋆) constructs are also part of the syntax of WYS⋆, including constants c include strings, integers, lists, tuples, etc. Likewise, host language functions/primitives can be called from WYS⋆—$\mathsf{ffi}\ \mathsf{f}\ \bar{e}$ 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.

4.2. Single-threaded semantics

The ST semantics is a model of the F⋆ semantics and the WYS⋆ API. The ST semantics defines a judgment $C \to C'$ that represents a single step of an abstract machine. Here, C is a *configuration* $M; X; L; T; e$. This five-tuple consists of a mode M, a stack X, an environment L, a trace T, and an expression e. The syntax for these elements is given in Figure 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.

The ST semantics is formalized in the style of Hieb and Felleisen [47], where the redex is chosen by (standard) *evaluation contexts* E, which prescribe left-to-right, call-by-value evaluation order. A few of the core rules are given in Figure 4. In essence, the semantics extends a standard reduction machinery for a call-by-value, lambda calculus (in direct correspondence with a pure fragment of F⋆), with several Wysteria-specific constructs. We argue, by inspection, that the Wysteria-specific constructs are in 1-1 correspondence with their specifications in the Wys monad. Despite the "eyeball closeness", there is room for formal discrepancy between the ST semantics and its static model within F⋆'s Wys monad. We leave to future work formally proving a correspondence between the ST semantics and μF⋆, the official semantics of F⋆ in F⋆ [30].

$$\textsc{S-Let}\ \frac{X_1 = (M; L; \mathsf{let}\ x\ =\ \langle\rangle\ \mathsf{in}\ e_2; T), X}{M; X; L; T; \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 \rightarrow M; X_1; L; \cdot; e_1} \qquad \frac{}{M; X; L; T; (L_1, \lambda x.e)\ e_1 \rightarrow M; X; L_1[x \mapsto e_1]; T; e}\ \textsc{S-App}$$

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

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

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

$$\textsc{S-AsSecRet}\ \frac{\mathsf{is\_sec}\ M \qquad X = (M_1; L_1; \langle\rangle; T), X_1 \quad T_1 = \mathsf{append}\ T\ [\mathsf{TMsg}\ v]}{M; X; L; \cdot; v \rightarrow M_1; X_1; L_1; T_1; v}$$

$$\textsc{S-Seal}\ \frac{M = \_\ s_1 \quad s \subseteq s_1}{M; X; L; T; \mathsf{seal}\ s\ v \rightarrow M; X; L; T; \mathsf{sealed}\ s\ v} \qquad \frac{M = \mathsf{Par}\ s_1 \Rightarrow s_1 \subseteq s \qquad M = \mathsf{Sec}\ s_1 \Rightarrow s_1 \cap s \neq \phi}{M; X; L; T; \mathsf{reveal}\ (\mathsf{sealed}\ s\ v) \rightarrow M; X; L; T; v}\ \textsc{S-Reveal}$$

$$\textsc{S-MkMap}\ \frac{M = \mathsf{Par}\ s_1 \Rightarrow v = \mathsf{sealed}\ s_2\ v_2 \wedge s \subseteq s_1 \wedge s \subseteq s_2 \qquad M = \mathsf{Sec}\ s_1 \Rightarrow s \subseteq s_1 \wedge v_2 = v}{M; X; L; T; \mathsf{mkmap}\ s\ v \rightarrow M; X; L; T; [s \mapsto v_2]}$$

$$\textsc{S-Proj}\ \frac{M = \mathsf{Par}\ s \Rightarrow s = \mathsf{singleton}\ p \quad M = \mathsf{Sec}\ s \Rightarrow p \in s \quad m[p] = v}{M; X; L; T; \mathsf{project}\ m\ p \rightarrow M; X; L; T; v}$$

$$\textsc{S-Concat}\ \frac{\mathsf{dom}(m_1) \cap \mathsf{dom}(m_1) = \phi}{M; X; L; T; \mathsf{concat}\ m_1\ m_2 \rightarrow M; X; L; T; m_1 \uplus m_2} \qquad \frac{v = \mathsf{exec\_ffi}\ \mathsf{f}\ \bar{v}}{M; X; L; T; \mathsf{ffi}\ \mathsf{f}\ \bar{v} \rightarrow M; X; L; T; v}\ \textsc{S-FFI}$$

Figure 4. WYS⋆ ST semantics (selected rules)

The standard constructs such as let bindings ($\mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2$), applications ($e_1\ e_2$), etc. evaluate as usual (see rules S-LET and S-APP), where the mode and traces play no role. Rules S-ASPAR and S-ASPARRET reduce an as_par expression once its arguments are fully evaluated. S-ASPAR first checks that the current mode is Par and contains all the principals from the set s. It then pushes a seal s $\langle\rangle$ frame on the stack, and starts evaluating e. 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 TScope s T, essentially making observations during the reduction of e (i.e., T) visible only to the principals in s.

To see that these rules faithfully model the F⋆ API, consider the F⋆ type of as_par, shown below.

```
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 ∧
3                      can_seal ps a ∧ pre (Mode Par ps)))
4   (ensures (fun m r tr → ∃t. tr=[TScope ps t] ∧
5                           post (Mode Par ps) (unseal r) t)))
```

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.[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.

4. For technical reasons, function closures may not be sealed; see the technical report for details.

Next consider the rules S-ASSEC and S-ASSECRET. Again, we can see that the rules implement the type of as_sec (shown in §3). The rule S-ASSEC checks the pre-condition 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 3.

Rules S-MKMAP, S-PROJ, 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 ($s \subseteq s_2$). The rule also requires (for both Par and Sec mode) that all the parties in the map domain are present in the current mode ($s \subseteq s_1$). In rule S-PROJ, 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 ($\uplus$) of the two maps.

The rule S-FFI implements the FFI call by calling a host-language function exec_ffi. As expected, calling a host-language function has no effect on the WYS⋆-specific state. Concretely, this is enforced by F⋆'s monadic encapsulation of effects. We present more details of exec_ffi in Section 5.2. The remaining rules are straightforward.

$$\text{L-LET}\ \frac{X_1 = (M; L; \mathsf{let}\ x\ =\ \langle\rangle\ \mathsf{in}\ e_2; T), X}{M; X; L; T; \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 \to M; X_1; L; \cdot; e_1} \qquad \frac{}{M; X; L; T; (L_1, \lambda x.e)\ e_1 \to M; X; L_1[x \mapsto e_1]; T; e}\ \text{L-APP}$$

$$\text{L-ASPAR1}\ \frac{p \in s \quad X_1 = (M; L; \mathsf{seal}\ s\ \langle\rangle; T), X}{M; X; L; T; \mathsf{as\_par}\ s\ (L_1, \lambda x.e) \rightsquigarrow M; X_1; L_1[x \mapsto ()]; \cdot; e} \qquad \text{L-ASPARRET}\ \frac{X = (M; L_1; \mathsf{seal}\ s\ \langle\rangle; T_1), X_1 \quad T_2 = \mathsf{append}\ T_1\ T}{M; X; L; T; v \rightsquigarrow M; X_1; L_1; T_2; \mathsf{sealed}\ s\ v}$$

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

$$\text{L-SEAL}\ \frac{p \in s \Rightarrow v_1 = \mathsf{seal}\ s\ v \quad p \notin s \Rightarrow v_1 = \mathsf{seal}\ s\ \bullet}{M; X; L; T; \mathsf{seal}\ s\ v \rightsquigarrow M; X; L; T; v_1} \qquad \frac{p \in s}{M; X; L; T; \mathsf{reveal}\ (\mathsf{sealed}\ s\ v) \rightsquigarrow M; X; L; T; v}\ \text{L-REVEAL}$$

$$\text{L-MKMAP}\ \frac{v = \mathsf{sealed}\ s_2\ v_2 \quad p \in s \Rightarrow p \in s_2 \wedge m = [p \mapsto v] \quad p \notin s \Rightarrow m = \cdot}{M; X; L; T; \mathsf{mkmap}\ s\ v \to M; X; L; T; m} \qquad \frac{p = p_1 \quad m = [p \mapsto v]}{M; X; L; T; \mathsf{project}\ m\ p_1 \to M; X; L; T; v}\ \text{L-PROJ}$$

$$\text{L-CONCAT}\ \frac{\mathsf{dom}(m_1) \cap \mathsf{dom}(m_1) = \phi}{M; X; L; T; \mathsf{concat}\ m_1\ m_2 \to M; X; L; T; m_1 \uplus m_2}$$

Figure 5. Distributed semantics, selected local rules (the mode M is always Par (singleton p))

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

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

Figure 6. Distributed semantics, multi-party rules

4.3. Distributed semantics

The DS semantics implements judgments of the form $\pi \longrightarrow \pi'$, 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. 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 (non-deterministically) chooses a principal's configuration and evaluates it according to the *local evaluation* judgment $C \rightsquigarrow C'$, 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 P-ENTER and P-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 evaluation agrees with the ST semantics for the standard language constructs (e.g. rules L-LET and L-APP) and differs for WYS*-specific constructs.[5]

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 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 garbage $\bullet$ (rule L-ASPAR2). The contents of the sealed value

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".

do not matter, since the principal will not be allowed to unseal the value anyway.

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_sec—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_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_v function on values as follows:

```
combine_v (•, v) = v
combine_v (v, •) = v
combine_v (p, p) = p
combine_v (sealed s v1, sealed s v2) = sealed s (combine_v v1 v2)
...
```

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:

```
let x = as_par alice (fun x → 2) in
let y = as_par bob (fun x → 3) in
let z = as_sec (alice, bob) (fun z → (unseal x) + (unseal 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 •. Similarly, in alice's environment y will be mapped to sealed bob •, 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 v. 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:

```
slice_v p p' = 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
```

As an example, consider the following code:

```
let x = as_sec (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 •, produced using the slice_v function on the result of seal alice y.

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

$$M; X; L; T; \_ \triangleleft v = M; X; L; \text{append } T \text{ [TMsg } v]; v$$

That is, the returned value is also added to the principal's trace to note their observation of the value.

4.4. 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., $C.e$ 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 (slice s C_1) $\longrightarrow^*$ (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 π strongly terminates in the terminal protocol π_t, written as $\pi \Downarrow \pi_t$, if all possible runs of π terminate in some number of steps in π_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 $(\mathsf{slice}\ s\ C_1) \Downarrow (\mathsf{slice}\ s\ C_2)$.

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*, the correctness and security properties that we prove about a WYS* program using F*'s logic, hold for the program that actually runs.

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

5. Implementation

This section describes our WYS* implementation. We begin by describing our interpreter; we have proved that its core implements our formal semantics, adding confidence that bugs have not been introduced in the translation from formalism to implementation. Then we describe our novel FFI by which WYS* programs can easily take advantage of features and libraries of WYS*'s host language, F*.

5.1. WYS* interpreter

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 $C \rightsquigarrow C'$ 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 as_sec, so the stub implements what amounts to P-ENTER and P-EXIT from Figure 6. 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-located server, written using a library due to Choi et. al. [39] 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* 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 (Section 4.4) 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 the F* extractor (including our custom WYS* 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.

5.2. FFI

When writing a source WYS* program (in F*), the programmer can access definitions from an FFI module,[6] which exports F* datatypes and library functions to WYS* programs. Below we explain WYS*'s extensible FFI mechanism, which enables programmers to add new datatypes and functions to the FFI module, while ensuring that the WYS* metatheory remains applicable.

Source programming with FFI. Datatypes and functions are added to the FFI module as usual:

```
type list α =
  | Nil: list α
  | Cons: hd:α → tl:list α → list α

val append: l1:list α → l2:list α → Tot (list α)
let rec append l1 l2 = match l1 with
  | [] → l2
  | hd::tl → hd::(append tl l2)
```

In addition to these usual definitions, WYS* requires the programmer to define the corresponding slice (and combine) functions (Section 4.3) for the new datatype, e.g.:

6. How F* programs call into WYS* functions was described in Section 3.1.

```
val slice_list: (prin → α → Tot α) → prin → list α → Tot (list α)
let rec slice_list f p l = match l with
  | [] → []
  | hd::tl → (f p hd)::(slice_list f p tl)
```

Once the datatype is defined, an WYS⋆ program is free to use it by importing the FFI module, and can prove properties with it using standard F⋆ reasoning. F⋆'s monadic encapsulation of effects ensures that the FFI functions do not interfere with WYS⋆-specific state, i.e., the mode and trace. For example, the effect annotation Tot in append above indicates that append is a pure function, and hence does not modify or use the the Wys monad's state.

FFI metatheory. Recall from Section 4 that we formalize FFI calls using the expression form ffi, FFI values using the value form v, and the semantics of FFI calls using the meta function exec_ffi. The WYS⋆ metatheory (Section 4.4) needs to relate the ST and DS semantics of FFI calls. For this purpose, the FFI programmer must prove that the added definitions meet certain obligations. For example, one such obligation is that the slice of the value returned from an FFI call must match the return value of the FFI call on the slice of the arguments. Formally, for an FFI function f,

$$\forall p, \bar{v}.\ \text{slice } p\ (\text{exec\_ffi f } \bar{v}) = \text{exec\_ffi f } \overline{\text{slice } p\ v}$$

To fulfill this obligation for the append function, for example, the programmer is required to prove the following lemma:

```
val slice_append_lemma:
    f:(prin → α → Tot α) → p:prin → l1:list α → l2:list α → Lemma
    (ensures (slice_list f p (append l1 l2) =
               append (slice_list f p l1) (slice_list f p l2)))
```

This lemma is easily provable in F⋆.

Once all obligations are proven, WYS⋆ metatheory guarantees that the theorems of Section 4.4 extend to the new datatypes and functions in the FFI.

FFI implementation. The FFI module itself is extracted to OCaml via regular F⋆ extraction. We rely on the F⋆ metatheory to conclude that the extracted code executes per its specification.

The custom F⋆ extraction mode that we have implemented for WYS⋆, identifies integer, string, and other constants in a WYS⋆ source program, and extracts them to the V_ffi AST form, which is part of the AST value type. The FFI calls, that can take as arguments and return F⋆ datatypes, are extracted to an E_ffi AST form, which is part of the AST expression type:

```
type exp = ...
  | E_ffi: f:α → args:list exp → inj:β → exp
```

The f argument is extracted to be the name of the FFI function, that links to the extracted OCaml function. We explain the inj argument shortly.

When evaluating the WYS⋆ AST, the interpreter may reach an E_ffi f args inj node. As we saw in Section 4.2, the interpreter calls a library function exec_ffi with the list of values; in addition, it passes a inj argument. The exec_ffi function first un-embeds any embedded host-language arguments. The un-embedding function is straightforward (the values shown below are from the value AST form):

```
unembed V_unit = ()
unembed (V_ffi v) = v (* values in the host language *)
unembed (V_seal s v) = V_seal s v
```

Interpreter-specific values, such as V_seal, are passed as is. The FFI module does not have access to the WYS⋆ API in F⋆, and hence it must treat these values abstractly. exec_ffi then calls the OCaml function f with the un-embedded arguments. The OCaml function returns some result which needs to be embedded back into the AST. F⋆ programs cannot inspect the type of values at run-time, but fortunately the compiler has enough information during extraction to know how to re-embed the result.

In particular, when the extractor compiles an FFI call in the source program to an E_ffi node, it has the type information for the return value of the FFI call. Using this information, it instruments the E_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 (), the injection is (an OCaml function) fun x → V_unit. If the return value is an interpreter value (e.g. V_seal), the injection is the identity. If the return value is some host F⋆ value (such as a list, tuple, or int), the injection creates an V_ffi node. exec_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 this approach to be higher order by using coercions [48].

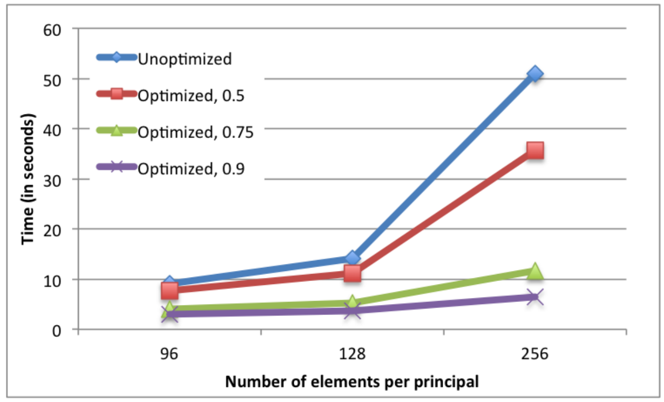

Figure 7. Time to run (in secs) normal and optimized PSI for varying per-party set sizes and intersection densities.

6. 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 3. 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 [49]. 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 up the computation, revealing some intermediate results and off-loading some parts to the local hosts (much like PSI). We refer the reader to [49] for more details of the algorithms.

For both the versions, we prove functional correctness:

```
val median:
  x:sealed Alice (int ∗ int) → y:sealed Bob (int ∗ int) → Wys int
     (requires (fun m → m = Mode Par {Alice, Bob}))
     (ensures (fun _ r t → (pre (unseal x) (unseal x) ⟹
                                        r = median_spec (unseal x) (unseal y))))
```

Here, pre captures the preconditions on the inputs as mentioned above, and median_spec is an idealized median specification. For both the versions, F⋆ is able to prove the specification without any extra hints in the code. For example, the monolithic version is programmed as:

```
let median x y =
  let g: unit → Wys int
    (requires (fun m → m = Mode Sec {Alice, Bob}))
    (ensures (fun _ r _ → (pre (unseal x) (unseal y) ⟹
                                        r = median_spec (unseal x) (unseal y))))
         = fun _ → ... in //monolithic median algorithm
  as_sec {Alice, Bob} g
```

For proving the security properties, for the unoptimized version, we prove that the observable trace is [TMsg r], where r is the result of the computation, basically reflecting that both the parties only see the final result. For the optimized version, we first prove that the observable trace is opt_trace (reveal x) (reveal y), where opt_trace is a purely specificational function that takes as arguments Alice's (reveal x) and Bob's (reveal y) inputs, and returns the trace generated by the optimized median algorithm. Then, we prove that the trace does not reveal more than the final output, by proving the following relational lemma:

```
val optimized_median_is_secure_for_alice:
  x1:(int ∗ int) → x2:(int ∗ int) → y:(int ∗ int)
  → Lemma (requires (pre x1 y ∧ pre x2 y ∧
                              median_spec x1 y = median_spec x2 y))
              (ensures (opt_trace x1 y = opt_trace x2 y))
```

The lemma says that for two runs of the optimized median with arbitrarily different inputs from Alice (x1 and x2), if Bob's input and output are the same, i.e. median_spec x1 y = median_spec x2 y, then the observable trace is also same. Essentially, the trace does not reveal more about Alice's inputs, beyond what is already revealed by the output. We also prove a symmetrical lemma for Bob where we vary Bob's inputs and keep Alice's input and output the same. Once again, for both the proofs, F⋆ is able to prove them automatically.

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* [31]. 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 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 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:

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

val v_of_sh: #a:Type → sh:Sh a → Ghost a
val ps_of_sh: #a:Type → sh: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 → v_of_sh r = x ∧ ps_of_sh r = m.ps ∧
tr = []))
val comb_sh: #a:Type → x:Sh a → Wys a
    (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 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. Extending secret shares support to 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):

```
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.

Other applications and secure server. We have implemented some more applications in WYS*, including a geo-location sharing application. At the moment, we have only run these applications using a *secure server* backend. In this backend, as_sec 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 [50]). We conjecture that such a server could be useful for a trusted hardware based deployment scenario.

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 WYS*, a new MPC DSL that is hosted in F*. WYS* 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. WYS* is the first DSL to enable formal verification of the source MPC programs. WYS* is also the first MPC DSL to provide a (partially) verified interpreter. Furthermore, WYS* 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 WYS*, and verified for correctness and security.

References

[1] A. Shamir, R. L. Rivest, and L. M. Adleman, *Mental poker*. Springer, 1980.

[2] A. C.-C. Yao, "How to generate and exchange secrets," in *FOCS*, 1986.

[3] O. Goldreich, S. Micali, and A. Wigderson, "How to play ANY mental game," in *STOC*, 1987.

[4] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *STOC*, 1990.

[5] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft, "Financial cryptography and data security," 2009, ch. Secure Multiparty Computation Goes Live.

[6] D. Bogdanov, M. Jemets, S. Siim, and M. Vaht, "How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation," in *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2015.

[7] F. Kerschbaum, A. Schroepfer, A. Zilli, R. Pibernik, O. Catrina, S. de Hoogh, B. Schoenmakers, S. Cimato, and E. Damiani, "Secure collaborative supply-chain management," *Computer*, 2011.

[8] L. Kamm, "Privacy-preserving statistical analysis using secure multi-party computation," Ph.D. dissertation, University of Tartu, 2015.

[9] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay: a secure two-party computation system," in *USENIX Security*, 2004.

[10] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits," in *USENIX*, 2011.

[11] "VIFF, the virtual ideal functionality framework," http://viff.dk/.

[12] L. Malka, "Vmcrypt: modular software architecture for scalable secure computation," in *CCS*, 2011.

[13] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: a system for secure multi-party computation," in *CCS*, 2008.

[14] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ANSI C," in *CCS*, 2012.

[15] J. D. Nielsen and M. I. Schwartzbach, "A domain-specific programming language for secure multiparty computation," in *PLAS*, 2007.

[16] J. D. Nielsen, "Languages for secure multiparty computation and towards strongly typed macros," Ph.D. dissertation, 2009.

[17] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *Computer Security - ESORICS 2008*, 2008.

[18] A. Schropfer, F. Kerschbaum, and G. Muller, "L1 - an intermediate language for mixed-protocol secure computation," in *COMPSAC*, 2011.

[19] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[20] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks, "Automating efficient ram-model secure computation," in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[21] P. Laud and J. Randmets, "A domain-specific language for low-level secure multiparty computation protocols," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1492–1503. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813664

[22] P. Mardziel, M. Hicks, J. Katz, M. Hammer, A. Rastogi, and M. Srivatsa, "Knowledge inference for optimizing and enforcing secure computations," in *Proceedings of the Annual Meeting of the US/UK International Technology Alliance*, 2013, this short paper consists of coherent excerpts from several prior papers.

[23] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic, "Mechanized metatheory for the masses: The poplmark challenge," in *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, ser. TPHOLs'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 50–65.

[24] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler, "Run your research: On the effectiveness of lightweight mechanization," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 285–296.

[25] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation*, 2011.

[26] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, 2009.

[27] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with verified cryptographic security," in *IEEE Symposium on Security & Privacy (Oakland)*, 2013, pp. 445–462. [Online]. Available: http://www.ieee-security.org/TC/SP2013/papers/4977a445.pdf

[28] "PolarSSL verification kit," http://trust-in-soft.com/polarssl-verification-kit/, 2015.

[29] J. Yang and C. Hawblitzel, "Safe to the last instruction: Automated verification of a type-safe operating system." Association for Computing Machinery, Inc., June 2010.

[30] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Z. Beguelin, "Dependent types and multi-monadic effects in F*," in *POPL*, 2016.

[31] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979.

[32] J. Launchbury, I. S. Diatchki, T. DuBuisson, and A. Adams-Moran, "Efficient lookup-table protocol in secure multiparty computation," in *ICFP*, 2012.

[33] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation," Unpublished, 2015, http://oblivc.org/downloads/oblivc.pdf.

[34] J. B. Almeida, M. Barbosa, G. Barthe, G. Davy, F. Dupressoir, B. Grgoire, and P.-Y. Strub, "Verified implementations for secure and verifiable computation," 2014.

[35] M. Backes, M. Maffei, and E. Mohammadi, "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.

[36] E. Meijer, B. Beckman, and G. Bierman, "Linq: Reconciling object, relations and xml in the .net framework," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 706–706. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142552

[37] "Bedrock, a coq library for verified low-level programming," http://plv.csail.mit.edu/bedrock/.

[38] The Coq development team, *The Coq proof assistant*. [Online]. Available: http://coq.inria.fr

[39] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein, "Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces," 2011, http://eprint.iacr.org/.

[40] Y. Huang, D. Evans, and J. Katz, "Private set intersection: Are garbled circuits better than custom protocols?" in *NDSS*, 2012.

[41] R. Atkey, "Parameterised notions of computation," *Journal of Functional Programming*, vol. 19, pp. 335–376, 2009. [Online]. Available: http://journals.cambridge.org/article_S095679680900728X

[42] A. Nanevski, J. G. Morrisett, and L. Birkedal, "Hoare type theory, polymorphism and separation," *J. Funct. Program.*, vol. 18, no. 5-6, pp. 865–911, 2008. [Online]. Available: http://ynot.cs.harvard.edu/papers/jfpsep07.pdf

[43] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '04. New York, NY, USA: ACM, 2004, pp. 14–25. [Online]. Available: http://doi.acm.org/10.1145/964001.964003

[44] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1891823.1891830

[45] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, 2003.

[46] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Z. Béguelin, "Probabilistic relational verification for cryptographic implementations," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 193–206. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535847

[47] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theoretical computer science*, vol. 103, no. 2, pp. 235–271, 1992.

[48] F. Henglein, "Dynamic typing: Syntax and proof theory," *Sci. Comput. Program.*, vol. 22, no. 3, pp. 197–230, 1994. [Online]. Available: http://dx.doi.org/10.1016/0167-6423(94)00004-2

[49] A. Rastogi, P. Mardziel, M. Hammer, and M. Hicks, "Knowledge inference for optimizing secure multi-party computation," in *PLAS*, 2013.

[50] C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular code-based cryptographic verification," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.