WYSTERIA: A Programming Language for Generic, Mixed-Mode Multiparty Computations

Aseem Rastogi
University of Maryland, College Park
aseem@cs.umd.edu

Matthew A. Hammer
University of Maryland, College Park
hammer@cs.umd.edu

Michael Hicks
University of Maryland, College Park
mwh@cs.umd.edu

Abstract*—In a Secure Multiparty Computation (SMC), mutually distrusting parties use cryptographic techniques to cooperatively compute over their private data; in the process each party learns only explicitly revealed outputs. In this paper, we present WYSTERIA, a high-level programming language for writing SMCs. As with past languages, like Fairplay, WYSTERIA compiles secure computations to circuits that are executed by an underlying engine. Unlike past work, WYSTERIA provides support for *mixed-mode* programs, which combine local, private computations with synchronous SMCs. WYSTERIA complements a standard feature set with built-in support for secret shares and with *wire bundles*, a new abstraction that supports generic n-party computations. We have formalized WYSTERIA, its refinement type system, and its operational semantics. We show that WYSTERIA programs have an easy-to-understand single-threaded interpretation and prove that this view corresponds to the actual multi-threaded semantics. We also prove type soundness, a property we show has security ramifications, namely that information about one party's data can only be revealed to another via (agreed upon) secure computations. We have implemented WYSTERIA, and used it to program a variety of interesting SMC protocols from the literature, as well as several new ones. We find that WYSTERIA's performance is competitive with prior approaches while making programming far easier, and more trustworthy.

I. INTRODUCTION

Secure multi-party computation (SMC) protocols [1], [2], [3] enable two or more parties $p_1, ..., p_n$ to cooperatively compute a function f over their private inputs $x_1, ..., x_n$ in a way that every party directly sees only the output $f(x_1, ..., x_n)$ while keeping the variables x_i private. Some examples are

- the x_i are arrays of private data and f is a statistical function (e.g., median) [4], [5];
- (private set intersection) the x_i are private sets (implemented as arrays) and f is the $\cap$ set-intersection operator [6], [7]; one use-case is determining only those friends, interests, etc. that individuals have in common;
- (second-price auction) the x_i are bids, and f determines the winning bidders [8], [9]

Of course, there are many other possibilities.

An SMC for f is typically implemented using garbled circuits [1], homomorphic encryption [10], or cooperative computation among a group of servers (e.g., using secret shares) [8], [11], [2], [12]. Several libraries and intermediate languages have been designed that provide efficient building blocks for constructing SMCs [13], [14], [15], [16], but their use can be tedious and error prone. As such, there have been several efforts, led by the Fairplay project [17], to define higher-level languages from which an SMC protocol can be compiled. In Fairplay, a compiler accepts a Pascal-like imperative program and compiles it to a garbled circuit. More recent efforts by Holzer et al [18] and Kreuter et al [16] support subsets of ANSI C, and follow-on work has expanded Fairplay's expressiveness to handle $n > 2$ parties [19].

We are interested in making SMC a building block for realistic programs. Such programs will move between "normal" (i.e., per-party local) computations and "secure" (i.e., joint, multi-party secure) modes repeatedly, resulting overall in what we call *mixed mode* computations. For example, we might use SMCs to implement the role of the dealer in a game of mental poker [20]—the game will be divided into rounds of local decision-making and joint interaction (shuffling, dealing, bidding, etc.). Mixed-mode computations are also used to improve performance over monolithic secure computations. As one example, we can perform private set intersection by having each party iteratively compare elements in their private sets, with only the comparisons (rather than the entire looping computation) carried out securely [7]. Computing the joint median can similarly be done by restricting secure computations only to comparisons, which Kerschbaum [5] has shown can net up to a 30× performance improvement.

Existing projects, like Fairplay, focus on how to compile normal-looking code into a representation like a boolean circuit that can be run by an SMC engine. In most cases, such compilation is done in advance. As such, mixed-mode programming is implemented by writing local computations in some host language (like C or Java) that call out to the SMC engine to evaluate the generated code. However, writing mixed-mode computations in a *single* language has some compelling benefits over the multi-lingual approach. First, it is easier to write mixed-mode programs since the programmer can see all of the interactions in one place, and not have to navigate foreign function interfaces, which are hard to use [21]. Second, for the same reason, programs are easier to understand, and thus answers to security concerns (could the overall computation leak too much information about my secrets?) will be more evident. Third, there is an opportunity for more dynamic execution models, e.g., compiling secure blocks on the fly where the participants or elements of the computation may be based on the results of prior interactions. Finally, there is an opportunity for greater code reuse, as a single language can encapsulate mixed mode protocols as reusable library functions.

This paper presents a new programming language for writing mixed-mode secure computations called WYSTERIA that realizes all of these benefits. WYSTERIA is the first language to support writing mixed-mode, multiparty computations in a generic manner, supporting any number of participants. It does this by offering several compelling features:

Conceptual single thread of control: All WYSTERIA programs operate in a combination of *parallel* and *secure* modes, where the former identifies local computations taking place on one or more hosts (in parallel), and the latter identifies secure computations occurring jointly among parties. Importantly, WYSTERIA mixed-mode computations can be viewed as having a single thread of control, with all communication between hosts expressed as variable bindings accessed within secure computations. Single threadedness makes programs far easier to write and reason about (whether by humans or by automated analyses [5], [22]). We formalize WYSTERIA's single-threaded semantics and prove a simulation theorem from single- to multi-threaded semantics. We also prove a theorem for correspondence of final configurations in the other direction, for successfully terminating multi-threaded executions. In both semantics, it is evident that all communication among parties occurs via secure blocks, and thus, information flows are easier to understand.

Generic support for more than two parties: WYSTERIA programs may involve an arbitrary number of parties, such that which parties, and their number, can be determined dynamically rather than necessarily at compile-time. To support such programs, WYSTERIA provides a notion of *principals as data* which can be used to dynamically determine computation participants or their outcomes (e.g., to identify winners in a tournament proceeding to the next round). WYSTERIA also implements a novel feature called *wire bundles* that are used to represent the inputs and outputs of secure computations such that a single party's view of a wire bundle is his own value, while the shared view makes evident all possible values. A secure computation, having the shared view, may iterate over the contents of a bundle. The length of such a bundle may be unspecified initially. The WYSTERIA compiler employs *dynamic circuit generation* to produce circuits when unknowns (like wire bundle lengths) become available. Despite this dynamism, Wysteria's meta-theoretical properties guarantee that participants proceed synchronously, i.e., they will always agree on the protocol they are participating in.

Secret shares: Many interesting programs interleave local and secure computations where the secured outcomes are revealed later. For example, in mental poker, each party must maintain a representation of the deck of cards whose contents are only revealed as cards are dealt. To support such programs, WYSTERIA provides secret shares as first-class objects. Secret shares resemble wire bundles in that each party has a local view (copy) and these views are combined in secure blocks to recover the original value. The WYSTERIA type system ensures shares are used properly; e.g., programs cannot inadvertently combine the shares from different objects.

Refinement type system: WYSTERIA is a functional programming language that comes equipped with a *refinement type system* to express the expectations and requirements of computations in the language. In particular, the types for wire bundles and shares are *dependent*, and directly identify the parties involved. For example, suppose we have a function is_richer that takes a list of principals and their net worths and returns who is richest. The logical refinement on the function's return type will state that the returned principal is one from the original set. Our type system also provides *delegation effects* for expressing in which context a function can be called; e.g., a function that computes in parallel mode cannot be called from within a secure computation, while the reverse is possible in certain circumstances. In general, our type system ensures the standard freedom from type errors: there will be no mistake of Alice communicating a string to a secure computation which expects an integer. WYSTERIA's mixed-mode design enables such reasoning easily: separating the host and SMC languages would make a proof of type soundness for mixed-mode programs far more difficult.

We have implemented a WYSTERIA interpreter which executes secure blocks by compiling them to boolean circuits, executed by Choi et al's implementation [23] of the Goldreich, Micali, and Wigderson protocol [2]. We have used WYSTERIA to build a broad array of mixed-mode programs proposed in the literature, along with some new ones. Our experimental results demonstrate three key points. First WYSTERIA's performance is competitive with prior approaches; e.g., we can reproduce the mixed-mode performance gains reported previously. Second, generic protocols for n-principals can be expressed with ease in WYSTERIA, and executed efficiently. Finally, WYSTERIA's novel high-level abstractions, e.g. secure state, enables expressing novel protocols not present in the existing literature.

Related work: WYSTERIA is not the first language for mixed mode SMC, but is unique in its high-level design, generality, and formal guarantees. For example, languages like L1 [24] and SMCL [8] permit some mixed-mode computations to be expressed directly. However, these languages lack WYSTERIA's single-threaded semantics, exposing more low-level details, e.g., for performing communication or constructing secret shares. As such, there are more opportunities for mistakes; e.g., one party may fail to always receive a sent message (or may receive the wrong one), or may not provide the right protocol shares. L1 is also limited to only two parties, and neither language has a type system expressing the requirements for well-formed mixed-mode compositions (which is handled by our delegation effects). No prior system of which we are aware has formalized its operational semantics and type system and shown them to be sensible (with the notable exception of Liu et al. [25], discussed later). Similar pitfalls exist for other languages (Section IX surveys related work).

The next section begins by presenting an overview of WYSTERIA's design and features by example. Section III presents λ_{Wy}, a formalization of WYSTERIA, and Sections IV and V formalize λ_{Wy}'s type system and operational semantics, both single- and multi-threaded versions. Section VI proves the type system is sound and that the two semantics correspond. Sections VII and VIII describe our implementation and experimental results, and we finish with related work and conclusions.

A full formal development of λ_{Wy} with proofs is available in a supplemental technical report [26]. The WYSTERIA implementation is available at http://bitbucket.org/aseemr/wysteria.

II. Overview of Wysteria

Wysteria is a functional programming language for performing mixed-mode secure multiparty computations. It has many features inherited directly from functional programming, such as first-class (higher order) functions, variable binding with let, tuples (aka records), sums (aka variants or tagged unions), and standard primitive values like integers and arrays. In this section we introduce Wysteria's novel features.

Computation Modes: Wysteria defines two computation modes: *secure mode* in which secure (multiparty) computations take place, and *parallel mode* in which one or more parties compute locally, in parallel. Here is a version of the so-called *millionaires' problem* that employs both modes:[1]

```
let a   =par({Alice})=       read() in
let b   =par({Bob})=         read() in
let out =sec({Alice,Bob})=  a>b in
out
```

Ignoring the par() and sec() annotations, this program is just a series of let-bindings: it first reads Alice's value, then reads Bob's value, computes which is bigger (who is richer?), and returns the result. The annotations indicate *how* and *where* these results should be computed. The par({ Alice }) annotation indicates that the read() (i.e., the rhs computation) will be executed locally (and normally) at Alice's location, while the par({Bob}) annotation indicates that the second read() will be executed at Bob's location. The sec({ Alice , Bob}) annotation indicates that a > b will be executed as a secure multiparty computation between Alice and Bob. Notice communication from local nodes (Bob and Alice) to the secure computation is done implicitly, as variable binding: the secure block "reads in" values a and b from each site. Our compiler sees this and compiles it to actual communication. In general, Wysteria programs, though they will in actuality run on multiple hosts, can be viewed as having the apparent single-threaded semantics, i.e., as if there were no annotations; we have proved a simulation theorem from single- to multi-threaded semantics.

In the example we used parallel mode merely as a means to acquire and communicate values to a secure-mode computation (which we sometimes call a *secure block*). As we show throughout the paper, there are many occasions in which parallel mode is used as a substantial component of the overall computation, often as a way to improve performance. On these occasions we specify the mode par(w) where w is a *set* of principals, rather than (as above) a single principal. In this case, the rhs of the let binding executes the given code *at every principal* in the set. Indeed, the code in the example above is implicitly surrounded by the code let result =par({ Alice , Bob})= e in result (where e is the code given above). This says that Alice and Bob both, in parallel, run the entire program. In doing so, they will *delegate* to other computation modes, e.g., to a mode in which only Alice performs a computation (to bind to a) or in which only Bob does one, or in which they jointly and securely compute the a>b. The delegation rules stipulate that parallel mode computations among a set of principals may delegate to any subset of those principals for either secure or parallel computations. We will see several more examples as we go.

Wires: In the above example, we are implicitly expressing that a is Alice's (acquired in par({ Alice }) mode) and b is Bob's. But suppose we want to make the computation of out into a function: what should the function's type be, so that the requirements of input sources are expressed? We do this with a novel language feature we call *wires*, as follows:

```
is_richer = λa: W {Alice} nat. λb: W {Bob} nat.
   let out =sec({Alice,Bob})= a[Alice] > b[Bob] in
   out
```

Here, the is_richer function takes two arguments a and b, each of which is a wire. The wires express that the data "belongs to" a particular principal: a value of type W { Alice } t is accessible *only* to Alice, which is to say, inside of par({ Alice }) computations or sec({ Alice } ∪ w) computations (where w can be any set of principals); notably, it is *not* accessible from within computations par({ Alice } ∪ w}) where w is a nonempty set. Note that wires are given *dependent types* which refer to principal *values*, in this case the values Alice and Bob; we will see interesting uses of such types shortly. Here is how we can call this function:

```
let a   =par({Alice})=      read() in
let b   =par({Bob})=        read() in
let out = is_richer (wire {Alice} a) (wire {Bob} b) in
out
```

This code is creating wires from Alice and Bob's private values and passing them to the function. Note that the output is not a wire, but just a regular value, and this is because it should be accessible to both Alice and Bob.

Delegation effects: Just as we use types to ensure that the inputs to a function are from the right party, we can use *effects* on a function's type to ensure that the caller is in a legal mode. For example, changing the third line in the above program to let out =par({ Alice })= is_richer ··· would be inappropriate, as we would be attempting to invoke the is_richer function only from Alice even though we also require Bob to participate in the secure computation in the function body. The requirement of joint involvement is expressed as a *delegation effect* in our type system, which indicates the expected mode of the caller. The delegation effect for is_richer function is sec({ Alice , Bob}), indicating that it must be called from a mode involving at least Alice and Bob (e.g., par({ Alice , Bob})). The effect annotates the function's type; the type of is_richer is thus W { Alice } nat → W {Bob}nat −sec({Alice ,Bob})→ bool; i.e., is_richer takes Alice's wire and Bob's wire, delegates to a secure block involving the two of them, and produces a boolean value. Delegation effects like par({ Alice , Bob}) are also possible.

Wire bundles: So far we have used single wires, but Wysteria also permits bundling wires together, which (as we will see later) is particularly useful when parties are generic over which and how many principals can participate in a secure computation. Here is our example modified to use bundling:

```
is_richer = λv: W {Alice,Bob} nat.
   let out =sec({Alice,Bob})= v[Alice] > v[Bob] in
   out
```

This code says that the input is a wire bundle whose values are from *both* Alice and Bob. We extract the individual values from the bundle v inside of the secure block using array-like projection syntax. To call this function after reading the inputs a and b we write is_richer ((wire { Alice } a) ++ (wire {Bob} b)).

[1] This program does not actually type check in Wysteria, but it is useful for illustration; the corrected program (using "wires") is given shortly.

Here the calling code concatenates together, using ⧺, the two wires from Alice and Bob into a bundle containing both of their inputs. Of course, this is just an abstraction, and does not literally represent what is going on at either party's code when this is compiled. For principal p, an empty wire bundle $\cdot$ ("dot") is used for a wire bundle when p is not in its domain, so that for Alice the above wire bundle would be represented as {Alice :a} ⧺ $\cdot$ while for Bob it would be $\cdot$ ⧺ {Bob:b}. When the secure computation begins, each party contributes its own value for every input bundle, and receives only its own value for every output bundle. The type system's accessibility rules for bundles generalize what was stated above: if v has type W ({A} ∪ w1) nat, where w1 may or may not be empty, then v[A] is only allowed in par({A}) mode or in sec({A} ∪ w2) mode (s.t. v is accessible to sec({A} ∪ w2), and nowhere else.

First-class principals and n-party computation: Now suppose we would like to generalize our function to operate over an arbitrary number of principals. At the moment we would have to write one function for two principals, a different one for three, and yet another one for four. To be able to write just one function for n parties we need two things, (1) a way to abstract which principals might be involved in a computation, and (2) a way to iterate over wire bundles. Then we can write a function that takes a wire bundle involving multiple arbitrary principals and iterate over it to find the highest value, returning the principal who has it. Here is the code to do this:

```
1 richest_of = λms:ps. λv: W ms nat.
2   let out =sec(ms)=
3     wfold(None, v,
4       λrichest. λp. λn. match richest with
5         | None   ⇒ Some p
6         | Some q ⇒ if n > v[q] then Some p
7                                else Some q) ) )
8   in (wire ms out)
```

The idea is that ms abstracts an unknown *set* of principals (which has type ps), and wfold permits iterating over the wire bundle for those principals: notice how ms appears in the type of v. The wfold construct takes three arguments. Its first argument None is the initial value for the loop's accumulator (None is a value of optional type). The second argument v is the wire bundle to iterate over. The wfold's body above is an anonymous function with three parameters: richest is the current accumulator, whose value is the richest principal thus far (or None), p is the current principal under consideration, and n is p's wire value, a nat. On line 5, no principal has yet been considered so the first becomes a candidate to be richest. Otherwise, on line 6, the protocol compares the current maximum with the present principal's worth and updates the accumulator. When the loop terminates, it yields the value of the accumulator, which is placed in a wire bundle and returned.

In addition to wfold, WYSTERIA also provides a way to apply a function to every element of a wire bundle, producing a new bundle (like the standard functional map).

The richest_of function can be applied concretely as follows (where the variables ending in _networth we assume are read from each party's console):

```
let all = {Alice,Bob,Charlie} in
let r : W all (ps{singl∧⊆all} option) =
          richest_of all (  wire {Alice}   alice_networth
                          ⧺ wire {Bob}     bob_networth
                          ⧺ wire {Charlie} charlie_networth )
```

The richest_of function illustrates that first-class principals are useful as the object of computation: the function's result r is a wire bundle carrying principal options. The type for values in r is a *refinement type* of the form tϕ where ϕ is a formula that refines the base type t. The particular type states that every value in r is either None or Some(s) where not only s has type ps (a set of principals), but that it is a singleton set (the singl part), and this set is a subset of all (the ⊆all part); i.e., s is exactly one of the set of principals involved in the computation. WYSTERIA uses refinements to ensure delegation requirements, e.g., to ensure that if ps0 is the set of principals in a nested parallel block, then the set of principals ps1 in the enclosing block is a superset, i.e., that ps1 ⊇ ps0. Refinements capture relationships between principal sets in their types to aid in proving such requirements during type checking.

Secret shares: Secure computations are useful in that they only reveal the final outcome, and not any intermediate results. However, in interactive settings we might not be able to perform an entire secure computation at once but need to do it a little at a time, hiding the intermediate results until the very end. To support this sort of program, WYSTERIA provides *secret shares*. Within a secure computation, e.g., involving principals A and B, we can encode a value of type t into shares having type Sh w t where w is the set of principals involved in the computation (e.g., {A,B}). When a value of this type is returned, each party gets its own encrypted share (similar to how wire bundles work, except that the contents are abstract). When the principals engage in a subsequent secure computation their shares can be recombined into the original value.

To illustrate the utility of secret shares in the context of mixed mode protocols, we consider a simple two-player, two-round bidding game. In each of the two rounds, each player submits a private bid. A player "wins" the game by having the higher average of two bids. The twist is that after the first round, both players learn the identity of the higher bidder, but not their bid. By learning which initial bid is higher, the players can adjust their second bid to be either higher or lower, depending on their preferences and strategy.

The protocol below implements the game as a mixed-mode computation that consists of two secure blocks, one per round. In order to force players to commit to their initial bid while not revealing it directly, the protocol stores the initial bids in secret shares. In the second secure block, the protocol recovers these bids in order to compute the final winning bidder:

```
/* Bidding round 1 of 2: */
let a1  =par({Alice})= read () in
let b1  =par({Bob})= read () in
let in1 = (wire {Alice} a1) ⧺ (wire {Bob} b1) in
let (higher1, sa, sb) =sec({Alice,Bob})=
  let c = if in1[Alice] > in2[Bob] then Alice else Bob in
  (c, makesh in1[Alice], makesh in1[Bob])
in
print higher1 ;

/* Bidding round 2 of 2: */
let a2 =par({Alice})= read () in
let b2 =par({Bob})= read () in
let in2 = (wire {Alice} a2) ⧺ (wire {Bob} b2) in
let higher2 =sec({Alice,Bob})=
  let (a1, b1) = (combsh sa, combsh sb) in
  let bid_a = (a1 + in2[Alice]) / 2 in
  let bid_b = (b1 + in2[Bob]) / 2 in
  if bid_a > bid_b then Alice else Bob
in
print higher2
```

$$\begin{array}{llll}
\text{Principal} & p,\ q & ::= & \textbf{Alice} \mid \textbf{Bob} \mid \textbf{Charlie} \mid \cdots \\
\text{Value} & v,\ w & ::= & x \mid n \mid \textbf{inj}_i\ v \mid (v_1, v_2) \mid p \mid \{w\} \mid w_1 \cup w_2
\end{array}$$

$$\begin{array}{l}
\text{Expression} \\
e ::= v_1 \oplus v_2 \mid \textbf{case}\,(v, x_1.e_1, x_2.e_2) \mid \textbf{fst}\,(v) \mid \textbf{snd}\,(v) \mid \lambda x.e \mid v_1\ v_2 \\
\quad \mid \ \textbf{fix}\ x.\lambda y.e \mid \textbf{array}(v_1, v_2) \mid \textbf{select}(v_1, v_2) \mid \textbf{update}(v_1, v_2, v_3) \\
\quad \mid \ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{let}\ x \overset{M}{=} e_1\ \textbf{in}\ e_2 \mid \textbf{wire}_w(v) \mid e_1 \mathbin{+\!\!+} e_2 \mid v[w] \\
\quad \mid \ \textbf{wfold}_w(v_1, v_2, v_3) \mid \textbf{wapp}_w(v_1, v_2) \mid \textbf{waps}_w(v_1, v_2) \\
\quad \mid \ \textbf{wcopy}_w(v) \mid \textbf{makesh}(v) \mid \textbf{combsh}(v) \mid v
\end{array}$$

$$\begin{array}{llll}
\text{Type environment} & \Gamma & ::= & . \mid \Gamma, x :_M \tau \mid \Gamma, x : \tau \\
\text{Mode} & M,\ N & ::= & m(w) \mid \top \\
\text{Modal operator} & m & ::= & \mathsf{p} \mid \mathsf{s} \\
\text{Effect} & \epsilon & ::= & \cdot \mid M \mid \epsilon_1, \epsilon_2
\end{array}$$

$$\begin{array}{llll}
\text{Refinement} & \phi & ::= & \mathsf{true} \mid \textbf{singl}(\nu) \mid \nu \subseteq w \mid \nu = w \mid \phi_1 \wedge \phi_2 \\
\text{Type} & \tau & ::= & \textbf{nat} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \textbf{ps}\ \phi \mid \textbf{W}\ w\ \tau \\
 & & & \mid \textbf{Array}\ \tau \mid \textbf{Sh}\ w\ \tau \mid x{:}\tau_1 \xrightarrow{\epsilon} \tau_2
\end{array}$$

Fig. 1. Program syntax: values, expressions and types.

The first secure block above resembles the millionaires' protocol, except that it returns not only the principal c with the higher input but also secret shares of both inputs: sa has type Sh {Alice ,Bob} int , and both Alice and Bob will have a different value for sa, analogous to wire bundles; the same goes for sb. The second block recovers the initial bids by combining the players' shares and computes the final bid as two averages.

Unlike past SMC languages that expose language primitives for secret sharing (e.g., [24]), in WYSTERIA the type system ensures that shares are not misused, e.g., shares for different underlying values may not be combined.

Expressive power: We have used WYSTERIA to program a wide variety of mixed-mode protocols proposed in the literature, e.g., all of those listed in the introduction, as well as several of our own invention. In all cases, WYSTERIA's design arguably made writing these protocols simpler than their originally proposed (often multi-lingual) implementations, and required few lines of code. We discuss these examples in more detail in Section VIII and in the supplemental technical report [26].

III. FORMAL LANGUAGE

In this section we introduce λ_{Wy}, the formal core calculus that underpins the language design of WYSTERIA. Sections IV and V present λ_{Wy}'s type system and operational semantics, respectively, and Section VI proves type soundness and a correspondence theorem.

Figure 1 gives the λ_{Wy} syntax for values v, expressions e, and types τ. λ_{Wy} contains standard values v consisting of variables x, natural numbers n (typed as **nat**), sums $\textbf{inj}_i\ v$ (typed by the form $\tau_1 + \tau_2$),[2] and products (v_1, v_2) (typed by the form $\tau_1 \times \tau_2$). In addition, λ_{Wy} permits principals p to be values, as well as sets thereof, constructed from singleton principal sets $\{w\}$ and principal set unions $w_1 \cup w_2$. These are all given type $\textbf{ps}\ \phi$, where ϕ is a *type refinement*; the type system ensures that if a value w has type $\textbf{ps}\ \phi$, then $\phi[w/\nu]$ is valid.[3] Refinements in λ_{Wy} are relations in set theory. Refinements $\nu \subseteq w$ and $\nu = w$ capture subset and equality relationships, respectively, with another value w. The refinement $\textbf{singl}(\nu)$ indicates that the principal set is a singleton.

λ_{Wy} expressions e are, for simplicity, in so-called *administrative normal form* (ANF), where nearly all sub-expressions are values, as opposed to arbitrary nested expressions. ANF form can be generated from an unrestricted WYSTERIA program with simple compiler support.

Expressions include arithmetic operations ($v_1 \oplus v_2$), **case** expressions (for computing on sums), and **fst** and **snd** for accessing elements of a product. Expressions $\lambda x.e$ and $v_1\ v_2$ denote abstractions and applications respectively. λ_{Wy} also includes standard **fix** point expressions (which encode loops) and mutable arrays: $\textbf{array}(v_1, v_2)$ creates an array (of type $\textbf{Array}\ \tau$) whose length is v_1, and whose elements (of type τ) are each initialized to v_2; array accesses are written as $\textbf{select}(v_1, v_2)$ where v_1 is an array and v_2 is an index; and array updates are written as $\textbf{update}(v_1, v_2, v_3)$, updating array v_1 at index v_2 with value v_3.

Let bindings in λ_{Wy} can optionally be annotated with a *mode* M, which indicates that expression e_1 should be executed in mode M as a delegation from the present mode. Modes are either secure (operator s) or parallel (operator p), among a set of principals w. Mode $\top$ represents is a special parallel mode among all principals; at run-time, $\top$ is replaced with $\mathsf{p}(w)$ where w is the set of all principals participating in the computation. Once the execution of e_1 completes, e_2 then executes in the original mode. Unannotated let bindings execute in the present mode. λ_{Wy} has dependent function types, written $x{:}\tau_1 \xrightarrow{\epsilon} \tau_2$, where x is bound in ϵ and τ_2; the ϵ annotation is an *effect* that captures all the delegations inside the function body. An effect is either empty, a mode, or a list of effects.

Wire bundle creation, concatenation, and folding are written $\textbf{wire}_w(v)$, $w_1 \mathbin{+\!\!+} w_2$, and $\textbf{wfold}_w(v_1, v_2, v_3)$, respectively (the w annotation on **wfold** and other combinators denotes the domain of the wire bundle being operated on). Wire bundles carrying a value of type τ for each principal in a set w are given the (dependent) type $\textbf{W}\ w\ \tau$. We also support mapping a wire bundle by a either a single function ($\textbf{waps}_w(v_1, v_2)$), or another wire bundle of per-principal functions ($\textbf{wapp}_w(v_1, v_2)$). Finally, the form $\textbf{wcopy}_w(v)$ is a coercion that allows wire bundles created in delegated computations to be visible in computations that contain them (operationally, $\textbf{wcopy}_w(v)$ is a no-op). λ_{Wy} also models support for secret shares, which have type $\textbf{Sh}\ w\ \tau$, analogous to the type of wire bundles. Shares of value v are created (in secure mode) with $\textbf{makesh}(v)$ and reconstituted (also in secure mode) with $\textbf{combsh}(v)$.

IV. TYPE SYSTEM

At a high level, the λ_{Wy} type system enforces the key invariants of a mixed-mode protocol: (a) each variable can only be used in an appropriate mode, (b) delegated computations require that all participating principals are present in the current mode, (c) parallel local state (viz., arrays) must remain consistent across parallel principals, and (d) code in the secure blocks must be restricted so that it can be compiled to a boolean circuit in our implementation. In this section, we present the typing rules, and show how these invariants are maintained.

[2] Sums model *tagged unions* or *variant* types

[3] We write $\phi[w/\nu]$ to denote the result of substituting w for ν in ϕ.

$\boxed{\Gamma \vdash_M v : \tau}$ *(Value typing)*

$$\textsc{T-var}\quad \frac{x :_M \tau \in \Gamma \vee x : \tau \in \Gamma \qquad \Gamma \vdash \tau}{\Gamma \vdash_M x : \tau} \qquad \textsc{T-nat}\quad \frac{}{\Gamma \vdash_M n : \textbf{nat}} \qquad \textsc{T-inj}\quad \frac{\Gamma \vdash_M v : \tau_i \qquad j \in \{1,2\} \qquad \tau_j \ \texttt{IsFlat} \qquad \Gamma \vdash \tau_j}{\Gamma \vdash_M \textbf{inj}_i\, v : \tau_1 + \tau_2}$$

$$\textsc{T-prod}\quad \frac{\Gamma \vdash_M v_i : \tau_i}{\Gamma \vdash_M (v_1, v_2) : \tau_1 \times \tau_2} \qquad \textsc{T-princ}\quad \frac{}{\Gamma \vdash_M p : \textbf{ps}\ (\nu = \{p\})}$$

$$\textsc{T-psone}\quad \frac{\Gamma \vdash_M w : \textbf{ps}\ (\textbf{singl}(\nu))}{\Gamma \vdash_M \{w\} : \textbf{ps}\ (\nu = \{w\})} \qquad \textsc{T-psunion}\quad \frac{\Gamma \vdash_M w_i : \textbf{ps}\ \phi_i}{\Gamma \vdash_M w_1 \cup w_2 : \textbf{ps}\ (\nu = w_1 \cup w_2)}$$

$$\textsc{T-psvar}\quad \frac{\Gamma \vdash_M x : \textbf{ps}\ \phi}{\Gamma \vdash_M x : \textbf{ps}\ (\nu = x)} \qquad \textsc{T-msub}\quad \frac{\Gamma \vdash M \qquad \Gamma \vdash_M x : \tau \qquad \Gamma \vdash M \rhd N \qquad N = \mathsf{s}(\_) \Rightarrow \tau\ \texttt{IsSecIn}}{\Gamma \vdash_N x : \tau} \qquad \textsc{T-sub}\quad \frac{\Gamma \vdash_M v : \tau_1 \qquad \Gamma \vdash \tau_1 <: \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash_M v : \tau}$$

Fig. 2. Value typing judgement.

$\boxed{\Gamma \vdash M \rhd N}$ *(Mode M can delegate to mode N)*

$$\textsc{D-refl}\quad \frac{\Gamma \vdash w_2 : \textbf{ps}\ (\nu = w_1)}{\Gamma \vdash m(w_1) \rhd m(w_2)} \qquad \textsc{D-top}\quad \frac{\Gamma \vdash w : \textbf{ps}\ \phi}{\Gamma \vdash \top \rhd m(w)} \qquad \textsc{D-par}\quad \frac{\Gamma \vdash w_2 : \textbf{ps}\ (\nu \subseteq w_1)}{\Gamma \vdash \mathsf{p}(w_1) \rhd \mathsf{p}(w_2)}$$

$$\textsc{D-sec}\quad \frac{\Gamma \vdash w_2 : \textbf{ps}\ (\nu = w_1)}{\Gamma \vdash \mathsf{p}(w_1) \rhd \mathsf{s}(w_2)}$$

$\boxed{\Gamma \vdash \tau_1 <: \tau_2}$ *(Subtyping)*

$$\textsc{S-refl}\quad \frac{}{\Gamma \vdash \tau <: \tau} \qquad \textsc{S-trans}\quad \frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \qquad \textsc{S-sum}\quad \frac{\Gamma \vdash \tau_i <: \tau_i'}{\Gamma \vdash \tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$

$$\textsc{S-prod}\quad \frac{\Gamma \vdash \tau_i <: \tau_i'}{\Gamma \vdash \tau_1 \times \tau_2 <: \tau_1' \times \tau_2'} \qquad \textsc{S-princs}\quad \frac{[\![\Gamma]\!] \vDash \phi_1 \Rightarrow \phi_2}{\Gamma \vdash \textbf{ps}\ \phi_1 <: \textbf{ps}\ \phi_2}$$

$$\textsc{S-wire}\quad \frac{\Gamma \vdash w_2 : \textbf{ps}\ (\nu \subseteq w_1) \qquad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash \textbf{W}\, w_1\, \tau_1 <: \textbf{W}\, w_2\, \tau_2} \qquad \textsc{S-array}\quad \frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash \textbf{Array}\, \tau_1 <: \textbf{Array}\, \tau_2}$$

$$\textsc{S-share}\quad \frac{\Gamma \vdash w_2 : \textbf{ps}\ (\nu = w_1) \qquad \Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash \textbf{Sh}\, w_1\, \tau_1 <: \textbf{Sh}\, w_2\, \tau_2} \qquad \textsc{S-arrow}\quad \frac{\Gamma \vdash \tau_1' <: \tau_1 \qquad \Gamma, x : \tau_1' \vdash \tau_2 <: \tau_2'}{\Gamma \vdash x{:}\tau_1 \xrightarrow{\epsilon} \tau_2 <: x{:}\tau_1' \xrightarrow{\epsilon} \tau_2'}$$

Fig. 3. Subtyping and delegation judgements.

Value typing: Figure 2 shows the value typing judgement $\Gamma \vdash_M v : \tau$, read as *under* Γ *and in current mode M, value v has type τ*. Variable bindings in Γ are of two forms: the usual $x : \tau$, and $x :_M \tau$ where M is the mode in which x is defined. Rule T-VAR looks up the binding of x in Γ, and checks that either x is bound with no mode annotation, or matches the mode M in the binding to the current mode. It uses an auxiliary judgement for type well-formedness, $\Gamma \vdash \tau$, which enforces invariants like, for a wire bundle type $\textbf{W}\ w\ \tau$, w should have a $\textbf{ps}\ \phi$ type. The rule T-INJ uses another auxiliary judgement τ `IsFlat` which holds for types τ that lack wire bundles and shares. Both auxiliary judgments are defined in our technical report [26]. WYSTERIA disallows wire bundles and shares in sum values, since it hides their precise sizes; the size information of wires and shares is required for boolean circuit generation. The rules T-PROD, T-PRINC, T-PSONE, T-PSUNION, and T-PSVAR are unsurprising.

Delegations: Rule T-MSUB is used to permit reading a variable in the present mode N, though the variable is bound in mode M. This is permitted under two conditions: when mode M is well-formed $N = \mathsf{s}(\_) \Rightarrow \tau\ \texttt{IsSecIn}$, and when $\Gamma \vdash M \rhd N$, which is read as *under type environment* Γ*, mode M can delegate computation to mode N*. The former condition enforces that variables accessible in secure blocks do not include functions known only to some parties as they can't be compiled to circuits by all the parties. As such, the condition excludes wire bundles that carry functions. The condition $\Gamma \vdash M \rhd N$ captures the intuitive idea that a variable defined in a larger mode can be accessed in a smaller mode. It is defined at the top of Figure 3. In addition to capturing valid variable accesses across different modes, the same relation also checks when it is valid for a mode to delegate computation to another mode (**let** $x \overset{N}{=} e_1$ **in** e_2). The rule D-REFL type checks the reflexive case, where the refinement $\nu = w_1$ captures that $w_2 = w_1$ at run-time. The special mode $\top$, that we use to type check generic library code written in WYSTERIA, can delegate to any mode (rule D-TOP). Recall that at run-time, $\top$ is replaced with $\mathsf{p}(w)$, where w is the set of all principals. A parallel mode $\mathsf{p}(w_1)$ can delegate computation to another parallel mode $\mathsf{p}(w_2)$ only if all the principals in w_2 are present at the time of delegation, i.e. $w_2 \subseteq w_1$. The rule D-PAR enforces this check by typing w_2 with the $\nu \subseteq w_1$ refinement. Finally, principals in parallel mode can begin a secure computation; rule D-SEC again uses refinements to check this delegation. We note that uses of rule D-PAR and rule D-SEC can be combined to effectively delegate from parallel mode to secure block consisting of a *subset* of the ambient principal set. Secure modes are only allowed to delegate to themselves (via rule D-REFL), since secure blocks are implemented using monolithic boolean circuits.

Subtyping: Rule T-SUB is the (declarative) subsumption rule, and permits giving a value of type τ_1 the type τ if the τ is a subtype of τ_1. More precisely, the subtyping judgement $\Gamma \vdash \tau_1 <: \tau_2$ is read as *under* Γ *type* τ_1 *is a subtype of* τ_2. The rules for this judgement are given at the bottom of Figure 3. Rules S-REFL, S-TRANS, S-SUM, S-PROD, S-ARRAY, and S-ARROW are standard. Rule S-PRINCS offloads reasoning about refinements to an auxiliary judgement written $[\![\Gamma]\!] \vDash \phi_1 \Rightarrow \phi_2$, which reads *assuming variables in* Γ *satisfy their refinements, the refinement* ϕ_1 *entails* ϕ_2. We elide the details of this ancillary judgement, as it can be realized with an SMT solver; our implementation uses Z3 [27], as described in Section VII. For wire bundles, the domain of the supertype, a principal set, must be a subset of domain of the subtype, i.e. type $\textbf{W}\ w_1\ \tau_1$ is a subtype of $\textbf{W}\ w_2\ \tau_2$ if $w_2 \subseteq w_1$, and $\tau_1 <: \tau_2$ (rule S-WIRE). As mentioned earlier, value w_2 is typed with no mode annotation. Rule S-SHARE is similar but requires τ_1 and τ_2 to be invariant since circuit generation requires an fixed size (in bits) for the shared value.

$\boxed{\Gamma \vdash_M e : \tau; \epsilon}$ *(**Expression typing**: “Under Γ, expression e has type τ, and may be run at M. ”)*

T-BINOP
$$\frac{\Gamma \vdash_M v_i : \mathbf{nat}}{\Gamma \vdash_M v_1 \oplus v_2 : \mathbf{nat}; \cdot}$$

T-FST
$$\frac{\Gamma \vdash_M v : \tau_1 \times \tau_2}{\Gamma \vdash_M \mathbf{fst}\,(v) : \tau_1; \cdot}$$

T-SND
$$\frac{\Gamma \vdash_M v : \tau_1 \times \tau_2}{\Gamma \vdash_M \mathbf{snd}\,(v) : \tau_2; \cdot}$$

T-CASE
$$\frac{\begin{array}{c}(M = \mathsf{p}(\_) \wedge \epsilon = \mathsf{p}(\cdot)) \vee (\tau \ \mathtt{IsFO} \wedge \epsilon = \cdot) \\ \Gamma \vdash v : \tau_1 + \tau_2 \qquad \Gamma, x_i : \tau_i \vdash_M e_i : \tau; \epsilon_i \\ \Gamma \vdash \tau \qquad \Gamma \vdash M \rhd \epsilon_i\end{array}}{\Gamma \vdash_M \mathbf{case}\,(v, x_1.e_1, x_2.e_2) : \tau; \epsilon, \epsilon_1, \epsilon_2}$$

T-LAM
$$\frac{\Gamma \vdash \tau \qquad \Gamma, x : \tau \vdash_M e : \tau_1; \epsilon}{\Gamma \vdash_M \lambda x.e : (x{:}\tau \xrightarrow{\epsilon} \tau_1); \cdot}$$

T-APP
$$\frac{\begin{array}{c}\Gamma \vdash_M v_1 : x{:}\tau_1 \xrightarrow{\epsilon} \tau_2 \\ \Gamma \vdash v_2 : \tau_1 \qquad \Gamma \vdash \tau_2[v_2/x] \\ \Gamma \vdash M \rhd \epsilon[v_2/x] \qquad M = \mathsf{s}(\_) \Rightarrow \tau_2 \ \mathtt{IsFO}\end{array}}{\Gamma \vdash_M v_1\ v_2 : \tau_2[v_2/x]; \epsilon[v_2/x]}$$

T-LET1
$$\frac{\begin{array}{c}\Gamma \vdash_M e_1 : \tau_1; \epsilon_1 \\ \Gamma, x : \tau_1 \vdash_M e_2 : \tau_2; \epsilon_2 \\ \Gamma \vdash \tau_2 \qquad \Gamma \vdash M \rhd \epsilon_2\end{array}}{\Gamma \vdash_M \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 : \tau_2; \epsilon_1, \epsilon_2}$$

T-LET2
$$\frac{\begin{array}{c}M = m(\_) \qquad N = \_(w) \\ \Gamma \vdash M \rhd N \qquad \Gamma \vdash_N e_1 : \tau_1; \epsilon_1 \\ \Gamma, x :_{m(w)} \tau_1 \vdash_M e_2 : \tau_2; \epsilon_2 \\ \Gamma \vdash \tau_2 \qquad \Gamma \vdash M \rhd \epsilon_2\end{array}}{\Gamma \vdash_M \mathbf{let}\ x \overset{N}{=} e_1\ \mathbf{in}\ e_2 : \tau_2; N, \epsilon_1, \epsilon_2}$$

T-FIX
$$\frac{\begin{array}{c}M = \mathsf{p}(\_) \\ \Gamma \vdash (y{:}\tau_1 \xrightarrow{\epsilon, \mathsf{p}(\cdot)} \tau_2) \qquad \Gamma \vdash M \rhd \epsilon \\ \Gamma, x : (y{:}\tau_1 \xrightarrow{\epsilon, \mathsf{p}(\cdot)} \tau_2) \vdash_M \lambda y.e : (y{:}\tau_1 \xrightarrow{\epsilon, \mathsf{p}(\cdot)} \tau_2); \cdot\end{array}}{\Gamma \vdash_M \mathbf{fix}\ x.\lambda y.e : (y{:}\tau_1 \xrightarrow{\epsilon, \mathsf{p}(\cdot)} \tau_2); \cdot}$$

T-ARRAY
$$\frac{\begin{array}{c}M = \mathsf{p}(\_) \\ \Gamma \vdash_M v_1 : \mathbf{nat} \qquad \Gamma \vdash_M v_2 : \tau\end{array}}{\Gamma \vdash_M \mathbf{array}(v_1, v_2) : \mathbf{Array}\, \tau; \cdot}$$

T-SELECT
$$\frac{\Gamma \vdash_M v_1 : \mathbf{Array}\, \tau \qquad \Gamma \vdash_M v_2 : \mathbf{nat}}{\Gamma \vdash_M \mathbf{select}(v_1, v_2) : \tau; \cdot}$$

T-UPDATE
$$\frac{\begin{array}{c}M = \mathsf{p}(\_) \\ \mathsf{mode}(v_1, \Gamma) = M \qquad \Gamma \vdash_M v_1 : \mathbf{Array}\, \tau \\ \Gamma \vdash_M v_2 : \mathbf{nat} \qquad \Gamma \vdash_M v_3 : \tau\end{array}}{\Gamma \vdash_M \mathbf{update}(v_1, v_2, v_3) : \mathbf{unit}; \cdot}$$

T-WIRE
$$\frac{\begin{array}{c}\Gamma \vdash w_1 : \mathbf{ps}\ (\nu \subseteq w_2) \\ m = \mathsf{s} \Rightarrow N = \mathsf{s}(w_2) \\ m = \mathsf{p} \Rightarrow N = \mathsf{p}(w_1) \qquad \Gamma \vdash_N v : \tau \\ m = \mathsf{s} \Rightarrow \tau \ \mathtt{IsFO} \qquad \tau \ \mathtt{IsFlat}\end{array}}{\Gamma \vdash_{m(w_2)} \mathbf{wire}_{w_1}(v) : \mathbf{W}\, w_1\, \tau; \cdot}$$

T-WPROJ
$$\frac{\begin{array}{c}m = \mathsf{p} \Rightarrow \phi = (\nu = w_1) \\ m = \mathsf{s} \Rightarrow \phi = (\nu \subseteq w_1) \\ \Gamma \vdash_{m(w_1)} v : \mathbf{W}\, w_2\, \tau \\ \Gamma \vdash w_2 : \mathbf{ps}\ (\phi \wedge \mathbf{singl}(\nu))\end{array}}{\Gamma \vdash_{m(w_1)} v[w_2] : \tau; \cdot}$$

T-WIREUN
$$\frac{\Gamma \vdash_M v_1 : \mathbf{W}\, w_1\, \tau \qquad \Gamma \vdash_M v_2 : \mathbf{W}\, w_2\, \tau}{\Gamma \vdash_M v_1 +\!\!+\, v_2 : \mathbf{W}\, (w_1 \cup w_2)\, \tau; \cdot}$$

T-WFOLD
$$\frac{\begin{array}{c}M = \mathsf{s}(\_) \\ \tau_2 \ \mathtt{IsFO} \qquad \phi = (\nu \subseteq w \wedge \mathbf{singl}(\nu)) \\ \Gamma \vdash_M v_1 : \mathbf{W}\, w\, \tau \qquad \Gamma \vdash_M v_2 : \tau_2 \\ \Gamma \vdash_M v_3 : \tau_2 \overset{\cdot}{\rightarrow} \mathbf{ps}\ \phi \overset{\cdot}{\rightarrow} \tau \overset{\cdot}{\rightarrow} \tau_2\end{array}}{\Gamma \vdash_M \mathbf{wfold}_w(v_1, v_2, v_3) : \tau_2; \cdot}$$

T-WAPP
$$\frac{\begin{array}{c}M = \mathsf{p}(\_) \qquad \Gamma \vdash_M v_1 : \mathbf{W}\, w\, \tau_1 \\ \Gamma \vdash_M v_2 : \mathbf{W}\, w\, (\tau_1 \overset{\cdot}{\rightarrow} \tau_2)\end{array}}{\Gamma \vdash_M \mathbf{wapp}_w(v_1, v_2) : \mathbf{W}\, w\, \tau_2; \cdot}$$

T-WAPS
$$\frac{\begin{array}{c}M = \mathsf{s}(\_) \qquad \tau_2 \ \mathtt{IsFO} \qquad \tau_2 \ \mathtt{IsFlat} \\ \Gamma \vdash_M v_1 : \mathbf{W}\, w\, \tau_1 \qquad \Gamma \vdash_M v_2 : \tau_1 \overset{\cdot}{\rightarrow} \tau_2\end{array}}{\Gamma \vdash_M \mathbf{waps}_w(v_1, v_2) : \mathbf{W}\, w\, \tau_2; \cdot}$$

T-WCOPY
$$\frac{\begin{array}{c}M = \mathsf{p}(w_1) \qquad \Gamma \vdash w_2 : \mathbf{ps}\ (\nu \subseteq w_1) \\ \Gamma \vdash_{\mathsf{p}(w_2)} v : \mathbf{W}\, w_2\, \tau\end{array}}{\Gamma \vdash_M \mathbf{wcopy}_{w_2}(v) : \mathbf{W}\, w_2\, \tau; \cdot}$$

T-MAKESH
$$\frac{\begin{array}{c}M = \mathsf{s}(w) \\ \tau \ \mathtt{IsFO} \qquad \tau \ \mathtt{IsFlat} \qquad \Gamma \vdash_M v : \tau\end{array}}{\Gamma \vdash_M \mathbf{makesh}(v) : \mathbf{Sh}\, w\, \tau; \cdot}$$

T-COMBSH
$$\frac{M = \mathsf{s}(w) \qquad \Gamma \vdash_M v : \mathbf{Sh}\, w\, \tau}{\Gamma \vdash_M \mathbf{combsh}(v) : \tau; \cdot}$$

T-SUBE
$$\frac{\Gamma \vdash_M e : \tau'; \epsilon \qquad \Gamma \vdash \tau' <: \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash_M e : \tau; \epsilon}$$

Fig. 4. Expression typing judgements. Judgment $\Gamma \vdash v : \tau$, for typing values appearing in dependent types, is defined in the technical report .

Expression typing: Figure 4 gives the typing judgement for expressions, written $\Gamma \vdash_M e : \tau; \epsilon$ and read *under* Γ *at mode* M*, the expression* e *has type* τ *and delegation effects* ϵ. The rules maintain the invariant that $\Gamma \vdash M \rhd \epsilon$, i.e. if e is well-typed, then M can perform all the delegation effects in e.

The rules T-BINOP, T-FST, and T-SND are standard. Rule T-CASE is mostly standard, except for two details. First, the effect in the conclusion contains the effects of both branches. The second detail concerns secure blocks. We need to enforce that **case** expressions in secure blocks do not return functions, since it won't be possible to inline applications of such functions when generating circuits. So, the premise $(M = \mathsf{p}(\_) \wedge \epsilon = M) \vee (\tau \ \mathtt{IsFO} \wedge \epsilon = \cdot)$ enforces that either the current mode is parallel, in which case there are no restrictions on τ, but we add an effect $\mathsf{p}(\cdot)$ so that secure blocks cannot reuse this code, or the type returned by the branches is first order (viz., not a function).

Rule T-LAM is the standard rule for typing a dependent effectful function: the variable x may appear free in the type of function body τ_1 and its effect ϵ, and ϵ appears on the function type. Rule T-APP is the function application rule. It checks that v_1 is a function type, v_2 matches the argument type of the function,[4] and that the application type $\tau_2[v_2/x]$ is well-formed in Γ. The rule performs two additional checks. First, it ensures that the current mode M can perform the delegation effects inside the function body ($\Gamma \vdash M \rhd \epsilon[v_2/x]$). Second, it disallows partial applications in secure blocks ($M = \mathsf{s}(\_) \Rightarrow \tau \ \mathtt{IsFO}$) to prevent a need to represent functional values as circuits; our implementation of secure blocks inlines all function applications before generating circuits.

[4] We restrict the values appearing in dependent types (v_2 in this case) to be typed without any mode annotation. We use an auxiliary judgment (similar to value-typing) $\Gamma \vdash v : \tau$ to type such values (see technical report). This judgment can only access those variables in Γ that are bound without any mode.

Rule T-LET1 is the typing rule for regular let bindings. Rule T-LET2 type checks the let bindings with delegation annotations. The rule differs from rule T-LET1 in several aspects. First, it checks that the delegation is legal (premise $\Gamma \vdash M \rhd N$). Second, it checks e_1 in mode N, instead of current mode M. Third, it checks e_2 with variable x bound in mode $m(w)$ in Γ. This mode consists of the outer modal operator m and the delegated party set w, meaning that x is available only to principals in w, but within the ambient (secure or parallel) block.

Rule T-FIX type checks unguarded recursive loops (which are potentially non-terminating). The rule is standard, but with the provisos that such loops are only permitted in parallel blocks ($M = \mathsf{p}(\_)$) and the current mode can perform all the effects of the loop ($\Gamma \vdash M \rhd \epsilon$). It also adds an effect $\mathsf{p}(\cdot)$ to the final type, so that secure blocks cannot use such loops defined in parallel blocks.

The rules T-ARRAY, T-SELECT, and T-UPDATE type array creation, reading, and writing, respectively, and are standard with the proviso that the first and third are only permitted in parallel blocks. For array writes, the premise $\mathsf{mode}(v_1, \Gamma) = M$ also requires the mode of the array to exactly match the current mode, so that all principals who can access the array do the modification. We elide the definition of this function, which simply extracts the mode of the argument from the typing context (note that since the type of v_1 is an array, and since programmers do not write array locations in their programs directly, the value v_1 must be a variable and not an array location).

Rule T-WIRE introduces a wire bundle for the given principal set w_1, mapping each principal to the given value v. The first premise $\Gamma \vdash w_1 : \mathbf{ps}\ (\nu \subseteq w_2)$ requires that $w_1 \subseteq w_2$, i.e., all principals contributing to the bundle are present in the current mode. The mode under which value v is typed is determined by the modal operator m of the current mode. If it is p, v is typed under $\mathsf{p}(w_1)$. However, if it is s, v is typed under the current mode itself. In parallel blocks, where parties execute locally, the wire value can be typed locally. However, in secure blocks, the value needs to be typable in secure mode. The next premise $m = \mathsf{s} \Rightarrow \tau$ `IsFO` ensures that wire bundles created in secure mode do not contain functions, again to prevent private code execution in secure blocks. Finally, the premise τ `IsFlat` prevents creation of wire bundles containing shares.

Rule T-WPROJ types wire projection $v[w_2]$. The premise $\Gamma \vdash_{m(w_1)} v : \mathbf{W}\, w_2\, \tau$ ensures that v is a wire bundle having w_2 in its domain. To check w_2, there are two subcases, distinguished by current mode being secure or parallel. If the latter, there must be but one participating principal, and that principal value needs to be equal to the index of wire projection: the refinement ϕ enforces that $w_2 = w_1$, and w_2 is a singleton. If projecting in a secure block, the projected principal need only be a member of the current principal set. Intuitively, this check ensures that principals are participants in any computation that uses their private data.

The rule T-WIREUN concatenates two wire bundle arguments, reflecting the concatenation in the final type $\mathbf{W}\,(w_1 \cup w_2)\,\tau$. The rules T-WFOLD, T-WAPP, T-WAPS, and T-WCOPY type the remaining primitives for wire bundles. In rule T-WFOLD, the premise enforces that **wfold** is only permitted in secure blocks, that the folding function is pure (viz., its set of effects is empty), and that the types of the wire bundle, accumulator and function agree. As with general function application in secure blocks, the rule enforces that the result of the fold is first order. The rules T-WAPP and T-WAPS are similar to each other, and to rule T-WFOLD. In both rules, a function v_2 is applied to the content of a wire bundle v_1. rule T-WAPP handles parallel mode applications where the applied functions reside in a wire bundle (i.e., each principal can provide their own function). Rule T-WAPS handles a more restricted case where the applied function is not within a wire bundle; this form is required in secure mode because to compile a secure block to a boolean circuit at run-time each principal must know the function being applied. As with rule T-WFOLD, the applied functions must be pure. Rule T-WCOPY allows copying of a wire bundle value v from $\mathsf{p}(w_2)$ to $\mathsf{p}(w_1)$ provided $w_2 \subseteq w_1$. This construct allows principals to carry over their private values residing in a wire bundle from a smaller mode to a larger mode while maintaining the privacy – principals in larger mode that are not in the wire bundle see an empty wire bundle $(\cdot)$.

Rules T-MAKESH and T-COMBSH introduce and eliminate secret share values of type $\mathbf{Sh}\, w\, \tau$, respectively. In both rules, the type of share being introduced or eliminated carries the principal set of the current mode, to enforce that all sharing participants are present when the shares are combined. Values within shares must be flat and first-order so that their bit-level representation size can be determined. Finally, rule T-SUBE is the subsumption rule for expressions.

V. Operational semantics

We define two distinct operational semantics for λ_{Wy} programs, each with its own role and advantages. The *single-threaded semantics* of λ_{Wy} provides a deterministic view of execution that makes apparent the *synchrony* of multi-party protocols. The *multi-threaded semantics* of λ_{Wy} provides a non-deterministic view of execution that makes apparent the relative *parallelism* and *privacy* of multi-party protocols. In Section VI, we state and prove correspondence theorems between the two semantics.

A. Single-threaded semantics

WYSTERIA's single-threaded semantics defines a transition relation for *machine configurations* (just *configurations* for short). A configuration $\mathcal{C}$ consists of a designated *current mode* M, and four additional run-time components: a store σ, a stack κ, an environment ψ and a program counter e (which is an expression representing the code to run next).

$$
\begin{array}{llcl}
\text{Configuration} & \mathcal{C} & ::= & M\{\sigma; \kappa; \psi; e\} \\
\text{Store} & \sigma & ::= & \cdot \mid \sigma\{\ell :_M v_1, \ldots, v_k\} \\
\text{Stack} & \kappa & ::= & \cdot \mid \kappa :: \langle M; \psi; x.e \rangle \mid \kappa :: \langle \psi; x.e \rangle \\
\text{Environment} & \psi & ::= & \cdot \mid \psi\{x \mapsto_M v\} \mid \psi\{x \mapsto v\}
\end{array}
$$

A store σ models mutable arrays, and consists of a finite map from (array) locations ℓ to value sequences $v_1, .., v_k$. Each entry in the store additionally carries the mode M of the allocation, which indicates which parties have access. A stack κ consists of a (possibly empty) list of stack frames, of which there are two varieties. The first variety is introduced by delegations (let bindings with mode annotations) and consists of a mode, an environment (which stores the values of local variables), and

$\boxed{\mathcal{C}_1 \longrightarrow \mathcal{C}_2}$ **Configuration stepping**: “Configuration $\mathcal{C}_1$ steps to $\mathcal{C}_2$”

$$\begin{array}{lrcl}
\textsc{stpC-Local} & M\{\sigma_1;\kappa;\psi_1;e_1\} & \longrightarrow & M\{\sigma_2;\kappa;\psi_2;e_2\}\ \ \text{when } \sigma_1;\psi_1;e_1 \xrightarrow{M} \sigma_2;\psi_2;e_2 \\
\textsc{stpC-Let} & M\{\sigma;\kappa;\psi;\mathbf{let}\,x=e_1\,\mathbf{in}\,e_2\} & \longrightarrow & M\{\sigma;\kappa :: \langle\psi;x.e_2\rangle;\psi;e_1\} \\
\textsc{stpC-delPar} & \mathsf{p}(w_1\cup w_2)\{\sigma;\kappa;\psi;\mathbf{let}\ x \overset{\mathsf{p}(w')}{=} e_1\ \mathbf{in}\ e_2\} & \longrightarrow & \mathsf{p}(w_2)\{\sigma;\kappa :: \langle\mathsf{p}(w_1\cup w_2);\psi;x.e_2\rangle;\psi;e_1\}\ \ \text{when } \psi\llbracket w'\rrbracket = w_2 \\
\textsc{stpC-delSSec} & \mathsf{s}(w)\{\sigma;\kappa;\psi;\mathbf{let}\ x \overset{\mathsf{s}(w')}{=} e_1\ \mathbf{in}\ e_2\} & \longrightarrow & s(w)\{\sigma;\kappa :: \langle s(w);\psi;x.e_2\rangle;\psi;e_1\}\ \ \text{when } \psi\llbracket w'\rrbracket = w \\
\textsc{stpC-delPSec} & \mathsf{p}(w)\{\sigma;\kappa;\psi;\mathbf{let}\ x \overset{\mathsf{s}(w')}{=} e_1\ \mathbf{in}\ e_2\} & \longrightarrow & \mathsf{p}(w)\{\sigma;\kappa :: \langle\mathsf{p}(w);\psi;x.e_2\rangle;\psi;\mathbf{secure}_{w'}(e_1)\} \\
\textsc{stpC-secEnter} & \mathsf{p}(w)\{\sigma;\kappa;\psi;\mathbf{secure}_{w'}(e)\} & \longrightarrow & \mathsf{s}(w)\{\sigma;\kappa;\psi;e\}\ \ \text{when } \psi\llbracket w'\rrbracket = w \\
\textsc{stpC-popStk1} & N\{\sigma;\kappa :: \langle M;\psi_1;x.e\rangle;\psi_2;v\} & \longrightarrow & M\{\sigma;\kappa;\psi_1\{x\mapsto_{m(w)}(\psi_2\llbracket v\rrbracket_N)\};e\}\ \ \text{when } M=m(\_) \text{ and } N=\_(w) \\
\textsc{stpC-popStk2} & M\{\sigma;\kappa :: \langle\psi_1;x.e\rangle;\psi_2;v\} & \longrightarrow & M\{\sigma;\kappa;\psi_1\{x\mapsto(\psi_2\llbracket v\rrbracket_M)\};e\}
\end{array}$$

$\boxed{\sigma_1;\psi_1;e_1 \xrightarrow{M} \sigma_2;\psi_2;e_2}$ **Local stepping**: “Under store σ_1 and environment ψ_1, expression e_1 steps at mode M to σ_2, ψ_2 and e_2”

$$\begin{array}{lrcll}
\textsc{stpL-Case} & \sigma;\psi;\mathbf{case}\,(v,x_1.e_1,x_2.e_2) & \xrightarrow{M} & \sigma;\psi\{x_i\mapsto v'\};e_i & \text{when } \psi\llbracket v\rrbracket_M = \mathbf{inj}_i\ v' \\
\textsc{stpL-Fst} & \sigma;\psi;\mathbf{fst}\,(v) & \xrightarrow{M} & \sigma;\psi;v_1 & \text{when } \psi\llbracket v\rrbracket_M = (v_1,v_2) \\
\textsc{stpL-Snd} & \sigma;\psi;\mathbf{snd}\,(v) & \xrightarrow{M} & \sigma;\psi;v_2 & \text{when } \psi\llbracket v\rrbracket_M = (v_1,v_2) \\
\textsc{stpL-BinOp} & \sigma;\psi;v_1\oplus v_2 & \xrightarrow{M} & \sigma;\psi;v' & \text{when } \psi\llbracket v_1\rrbracket_M = v_1',\ \psi\llbracket v_2\rrbracket_M = v_2' \text{ and } v_1'\oplus v_2' = v' \\
\textsc{stpA-Lambda} & \sigma;\psi;\lambda x.e & \xrightarrow{M} & \sigma;\psi;\mathbf{clos}\,(\psi;\lambda x.e) & \\
\textsc{stpA-Apply} & \sigma;\psi_1;v_1\ v_2 & \xrightarrow{M} & \sigma;\psi_2\{x\mapsto v'\};e & \text{when } \psi_1\llbracket v_1\rrbracket_M = \mathbf{clos}\,(\psi_2;\lambda x.e) \text{ and } \psi_1\llbracket v_2\rrbracket = v' \\
\textsc{stpL-Fix} & \sigma;\psi;\mathbf{fix}\ x.\lambda y.e & \xrightarrow{\mathsf{p}(w)} & \sigma;\psi;\mathbf{clos}\,(\psi;\ \mathbf{fix}\,x.\lambda y.e) & \\
\textsc{stpA-FixApply} & \sigma;\psi_1;v_1\ v_2 & \xrightarrow{M} & \sigma;\psi';e & \text{when } \psi_1\llbracket v_1\rrbracket_M = \mathbf{clos}\,(\psi;\ \mathbf{fix}\,x.\lambda y.e) \text{ and } \psi_1\llbracket v_2\rrbracket = v' \\
 & & & & \quad\text{and } \psi' = \psi\{x\mapsto\mathbf{clos}\,(\psi;\ \mathbf{fix}\,x.\lambda y.e);y\mapsto v'\} \\
\textsc{stpL-array} & \sigma;\psi;\mathbf{array}(v_1,v_2) & \xrightarrow{M} & \sigma\{\ell:_M w^k\};\psi;\ell & \text{when } \psi\llbracket v_1\rrbracket_M = k,\ \psi\llbracket v_2\rrbracket_M = w \text{ and } \mathsf{next}_M(\sigma) = \ell \\
\textsc{stpL-select} & \sigma;\psi;\mathbf{select}(v_1,v_2) & \xrightarrow{M} & \sigma;\psi;w_i & \text{when } \psi\llbracket v_1\rrbracket_M = \ell,\ \psi\llbracket v_2\rrbracket_M = i,\ i\in[1..k] \text{ and } \sigma(\ell) = \{\bar{w}\}_k \\
\textsc{stpL-sel-err} & \sigma;\psi;\mathbf{select}(v_1,v_2) & \xrightarrow{M} & \sigma;\psi;\mathsf{error} & \text{when } \psi\llbracket v_1\rrbracket_M = \ell,\ \psi\llbracket v_2\rrbracket_M = i,\ i\notin[1..k] \text{ and } \sigma(\ell) = \{\bar{w}\}_k \\
\textsc{stpL-update} & \sigma;\psi;\mathbf{update}(v_1,v_2,v_3) & \xrightarrow{M} & \sigma';\psi;\mathsf{()} & \text{when } \psi\llbracket v_1\rrbracket_M = \ell,\ \psi\llbracket v_2\rrbracket_M = i \text{ and } \psi\llbracket v_3\rrbracket_M = w_i' \\
 & & & & \quad\text{and } \sigma(\ell) = \{\bar{w}\}_k,\ j\in[1..k] \text{ and } w_j' = w_j \text{ for } j\neq i \\
 & & & & \quad\text{and } \sigma' = \sigma\{\ell:_M\{\bar{w'}\}_k\} \\
\textsc{stpL-upd-err} & \sigma;\psi;\mathbf{update}(v_1,v_2,v_3) & \xrightarrow{M} & \sigma;\psi;\mathsf{error} & \text{when } \psi\llbracket v_1\rrbracket_M = \ell,\ \psi\llbracket v_2\rrbracket_M = i,\ i\notin[1..k] \text{ and } \sigma(\ell) = \{\bar{w}\}_k \\
\textsc{stpL-Makesh} & \sigma;\psi;\mathbf{makesh}(v) & \xrightarrow{\mathsf{s}(w)} & \sigma;\psi;\mathbf{sh}\ w\ v' & \text{when } \psi\llbracket v\rrbracket_{\mathsf{s}(w)} = v' \\
\textsc{stpL-Combsh} & \sigma;\psi;\mathbf{combsh}(v) & \xrightarrow{\mathsf{s}(w)} & \sigma;\psi;v' & \text{when } \psi\llbracket v\rrbracket_{\mathsf{s}(w)} = \mathbf{sh}\ w\ v' \\
\textsc{stpL-Wire} & \sigma;\psi;\mathbf{wire}_w(v) & \xrightarrow{M} & \sigma;\psi;\{\!|(\psi\llbracket v\rrbracket_N)|\!\}^{\mathsf{wires}}_{w'} & \text{where } \{\!|v|\!\}^{\mathsf{wires}}_{w_1\cup w_2} = \{\!|v|\!\}^{\mathsf{wires}}_{w_1} \mathbin{+\!\!+} \{\!|v|\!\}^{\mathsf{wires}}_{w_2} \\
\textsc{stpL-Wcopy} & \sigma;\psi;\mathbf{wcopy}_w(v) & \xrightarrow{M} & \sigma;\psi;v & \quad\text{and } \{\!|v|\!\}^{\mathsf{wires}}_{\{p\}} = \{p:v\} \text{ and } \{\!|v|\!\}^{\mathsf{wires}}_{\cdot} = \cdot \\
 & & & & \quad\text{and } \psi\llbracket w\rrbracket = w' \text{ and } M = m(_) \\
 & & & & m = \mathsf{s} \Rightarrow N = M \text{ and } m = \mathsf{p} \Rightarrow N = \mathsf{p}(w') \\
\textsc{stpL-parProj} & \sigma;\psi;v_1[v_2] & \xrightarrow{\mathsf{p}(\{p\})} & \sigma;\psi;v' & \text{when } \psi\llbracket v_2\rrbracket_{\mathsf{p}(\{p\})} = p \text{ and } \psi\llbracket v_1\rrbracket_{\mathsf{p}(\{p\})} = \{p:v'\} \mathbin{+\!\!+} w' \\
\textsc{stpL-secProj} & \sigma;\psi;v_1[v_2] & \xrightarrow{\mathsf{s}(\{p\}\cup w)} & \sigma;\psi;v' & \text{when } \psi\llbracket v_2\rrbracket_{\mathsf{s}(\{p\}\cup w)} = p \text{ and } \psi\llbracket v_1\rrbracket_{\mathsf{s}(\{p\}\cup w)} = \{p:v'\} \mathbin{+\!\!+} w' \\
\textsc{stpL-WireUn} & \sigma;\psi;v_1 \mathbin{+\!\!+} v_2 & \xrightarrow{M} & \sigma;\psi;v_1' \mathbin{+\!\!+} v_2' & \text{when } \psi\llbracket v_1\rrbracket_M = v_1' \text{ and } \psi\llbracket v_2\rrbracket_M = v_2' \\
\textsc{stpL-wapp1} & \sigma;\psi;\mathbf{wapp}_w(v_1,v_2) & \xrightarrow{M} & \sigma;\psi;\cdot & \text{when } \psi\llbracket w\rrbracket = \cdot \\
\textsc{stpL-wapp2} & \sigma;\psi;\mathbf{wapp}_w(v_1,v_2) & \xrightarrow{M} & \sigma;\psi;e & \text{when } \psi\llbracket w\rrbracket = \{p\}\cup w' \text{ and } M = \mathsf{p}((\{p\}\cup w')\cup w_1) \\
 & & & & \quad\text{and } \psi\llbracket v_1\rrbracket_M = v_1' \text{ and } \psi\llbracket v_2\rrbracket_M = v_2' \\
\multicolumn{5}{r}{\text{where } e = \mathbf{let}\ z_1 \overset{\mathsf{p}(\{p\})}{=} \mathbf{let}\,z_2 = v_1'[p]\,\mathbf{in}\,\mathbf{let}\,z_3 = v_2'[p]\,\mathbf{in}\,z_2\ z_3\ \mathbf{in}\ \mathbf{let}\,z_4 = \mathbf{wapp}_{w'}(v_1',v_2')\,\mathbf{in}\,((\mathbf{wire}_{\{p\}}(z_1)) \mathbin{+\!\!+} z_4)} \\
\textsc{stpL-waps1} & \sigma;\psi;\mathbf{waps}_w(v_1,v_2) & \xrightarrow{M} & \sigma;\psi;\cdot & \text{when } \psi\llbracket w\rrbracket = \cdot \\
\textsc{stpL-waps2} & \sigma;\psi;\mathbf{waps}_w(v_1,v_2) & \xrightarrow{M} & \sigma;\psi;e & \text{when } \psi\llbracket w\rrbracket = \{p\}\cup w' \text{ and } M = \mathsf{s}((\{p\}\cup w')\cup w_1) \\
 & & & & \quad\text{and } \psi\llbracket v_1\rrbracket_M = v_1' \text{ and } \psi\llbracket v_2\rrbracket_M = v_2' \\
 & & & \multicolumn{2}{l}{\text{where } e = \mathbf{let}\,z_1 = v_1'[p]\,\mathbf{in}\,\mathbf{let}\,z_2 = v_2'\ z_1\,\mathbf{in}\,\mathbf{let}\,z_3 = \mathbf{waps}_{w'}(v_1',v_2')\,\mathbf{in}\,((\mathbf{wire}_{\{p\}}(z_2)) \mathbin{+\!\!+} z_3)} \\
\textsc{stpL-wfold1} & \sigma;\psi;\mathbf{wfold}_w(v_1,v_2,v_3) & \xrightarrow{M} & \sigma;\psi;v' & \text{when } \psi\llbracket w\rrbracket = \cdot \text{ and } \psi\llbracket v_2\rrbracket_M = v' \\
\textsc{stpL-wfold2} & \sigma;\psi;\mathbf{wfold}_w(v_1,v_2,v_3) & \xrightarrow{M} & \sigma;\psi;e & \text{when } \psi\llbracket w\rrbracket = \{p\}\cup w' \text{ and } M = \mathsf{s}((\{p\}\cup w')\cup w_1) \\
 & & & & \quad\text{and } \psi\llbracket v_1\rrbracket_M = v_1',\ \psi\llbracket v_2\rrbracket_M = v_2' \text{ and } \psi\llbracket v_3\rrbracket_M = v_3' \\
 & & & \multicolumn{2}{l}{\text{where } e = \mathbf{let}\,z_1 = v_1'[p]\,\mathbf{in}\,\mathbf{let}\,z_2 = v_3'\ v_2'\ p\ z_1\,\mathbf{in}\,\mathbf{wfold}_{w'}(v_1',z_2,v_3')}
\end{array}$$

Fig. 5. λ_{Wy}: operational semantics of single-threaded configurations a return continuation, which is an expression containing one free variable standing in for the return value supplied when the frame is popped. The second variety is introduced by regular let-bindings where the mode is invariant; it consists of only an environment and a return continuation.

An environment ψ consists of a finite map from variables to closed values. As with stores, each entry in the environment additionally carries a mode M indicating which principals have access and when (whether in secure or in parallel blocks). Note that we extend the definition of values v from Figure 1 to include several new forms that appear only during execution. During run-time, we add forms to represent empty party sets (written $\cdot$), empty wire bundles (written $\cdot$), single-principal wire bundles ($\{p : v\}$), wire bundle concatenation ($v_1 +\!\!+\, v_2$), array locations (ℓ), and closures (written $\textbf{clos}\,(\psi; \lambda x.e)$ and $\textbf{clos}\,(\psi;\, \textbf{fix}\, f.\lambda x.e)$).

Our "environment-passing" semantics (in contrast to a semantics based on substitution) permits us to directly recover the multi-threaded view of each principal in the midst of single-threaded execution. Later, we exploit this ability to show that the single and multi-threaded semantics enjoy a precise correspondence. If we were to substitute values for variables directly into the program text these distinct views of the program's state would be tedious or impossible to recover.

Figure 5 gives the single-threaded semantics in the form of three judgements. The main judgement $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ is located at the figure's top and can be read as saying *configuration $\mathcal{C}_1$ steps to $\mathcal{C}_2$*. Configuration stepping uses an ancillary judgement for local stepping (lowest in figure). The local stepping judgement $\sigma_1; \psi_1; e_1 \xrightarrow{M} \sigma_2; \psi_2; e_2$ covers all (common) cases where neither the stack nor the mode of the configuration change. This judgement can be read as *under store σ_1 and environment ψ_1, expression e_1 steps at mode M to σ_2, ψ_2 and e_2*. Most configuration stepping rules are local, in the sense that they stay within one mode and do not affect the stack.

Configurations manage their current mode in a stack-based discipline, using their stack to record and recover modes as the thread of execution enters and leaves nested parallel and secure blocks. The rules of $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ handle eight cases: Local stepping only (STPC-LOCAL), regular let-binding with no delegation annotation (STPC-LET), delegation to a parallel block (STPC-DELPAR), delegation to secure block from a secure or parallel block (STPC-DELSSEC and STPC-DELPSEC, respectively), entering a secure block (STPC-SECENTER) and handling return values using the two varieties of stack frames (STPC-POPSTK1 and STPC-POPSTK2). We discuss local stepping shortly, considering the other rules first.

Both delegation rules move the program counter under a let, saving the returning context on the stack for later (to be used in STPC-POPSTK). Parallel delegation is permitted when the current principals are a superset of the delegated principal set; secure delegation is permitted when the sets coincide. Secure delegation occurs in two phases, which is convenient later when relating the single-threaded configuration semantics to the multi-threaded view; STPC-SECENTER handles the second phase of secure delegation.

The standard language features of WYSTERIA are covered by the first thirteen local stepping rules, including five rules

$\boxed{\psi[\![v]\!]_M = v'}$ **Environment lookup**: "Closing v for M under ψ is v'"

$$\begin{array}{rcll}
\psi[\![(v_1, v_2)]\!]_M & = & (v_1', v_2') & \text{when } \psi[\![v_1]\!]_M = v_1' \text{ and } \psi[\![v_2]\!]_M = v_2' \\
\psi[\![x]\!]_M & = & v & \text{when } x \mapsto_N v \in \psi \text{ and } . \vdash N \rhd M \\
\psi[\![x]\!]_M & = & v & \text{when } x \mapsto v \in \psi
\end{array}$$

Fig. 6. λ_{Wy}: Environment lookup judgments (selected rules).

to handle primitive array operations. We explain the first rule in detail, as a model for the rest. Case analysis (STPL-CASE) branches based on the injection, stepping to the appropriate branch and updating the environment with the payload value of the injected value v'. The incoming environment ψ *closes* the (possibly open) scruntinee value v of the case expression using value bindings for the free variables accessible at the current mode M. We write the closing operation as $\psi[\![v]\!]_M$ and show selected rules in Figure 6 (complete rules are in the the technical report). Note the second rule makes values bound in larger modes available in smaller modes. The rule STPL-CASE updates the environment using the closed value, adding a new variable binding at the current mode. The remainder of the rules follow a similar pattern of environment usage.

Projection of pairs (STPL-FST,STPL-SND) gives the first or second value of the pair (respectively). Binary operations close the operands before computing a result (STPL-BINOP). Lambdas and fix-points step similarly. In both cases a rule closes the expression, introducing a closure value with the current environment (STPA-LAMBDA and STPA-FIX). Their application rules restore the environment from the closure, update it to hold the argument binding, and in the case of the fix-points, a binding for the fix expression. The mode $\mathsf{p}(w)$ enforces that (potentially unguarded) recursion via STPL-FIX may not occur in secure blocks.

The array primitives access or update the store, which remained invariant in all the cases above. Array creation (STPL-ARRAY) updates the store with a fresh array location ℓ; the location maps to a sequence of v_1 copies of initial value v_2. Array selection (STPL-SELECT,STPL-SEL-ERR) projects a value from an array location by its index. The side conditions of STPL-SELECT enforce that the index is within range. When out of bounds, STPL-SEL-ERR applies instead, stepping the program text to error, indicating that a fatal (out of bounds error) has occurred. We classify an error program as *halted* rather than stuck. As with ("successfully") halted programs that consist only of a return value, the error program is intended to lack any applicable stepping rules. Array updating (STPL-UPDATE,STPL-UPD-ERR) is similar to projection, except that it updates the store with a new value at one index (and the same values at all other indices). As with projection, an out-of-bounds array index results in the error program.

For secret sharing, the two rules STPL-MAKESH and STPL-COMBSH give the semantics of **makesh** and **combsh**, respectively. Both rules require secure computation, indicated by the mode above the transition arrow. In STPL-MAKESH, the argument value is closed and "distributed" as a share value $\textbf{sh}\, w\, v'$ associated with the principal set w of the current mode $\mathsf{s}(w)$. STPL-COMBSH performs the reverse operation, "combining" the share values of each principal of w to recover the original value.

The remaining local stepping rules support wire bundles and their combinators. STPL-WIRE introduces a wire bundle for given party set w, mapping each principal in the set to the argument value (closed under the current environment). STPL-WCOPY is a no-op: it yields its argument wire bundle. STPL-PARPROJ projects from wire bundles in a parallel mode when the projected principal is present and alone. STPL-SECPROJ projects from wire bundles in a secure mode when the projected principal is present (alone or not).

The wire combinator rules follow a common pattern. For each of the three combinators, there are two cases for the party set w that indexes the combinator: either w is empty, or it consists of at least one principal. In the empty cases, the combinators reduce according to their respective base cases: Rules STPL-WAPP1 and STPL-WAPS1 both reduce to the empty wire bundle, and STPL-WFOLD1 reduces to the accumulator value v_2. In the inductive cases there is at least one principal p. In these cases, the combinators each unfold once for p (we intentionally keep the order of this unfolding non-deterministic, so all orderings of principals are permitted). Rule STPL-WAPP2 unfolds a parallel-mode wire application for $\mathsf{p}(\{p\})$, creating let-bindings that project the argument and function from the two wire bundle arguments, perform the function application and recursively process the remaining principals; finally, the unfolding concatenates the resulting wire bundle for the other principals with that of p. Rule STPL-WAPS2 is similar to STPL-WAPP2, except that the function being applied v_2 is *not* carried in a wire bundle (recall that secure blocks forbid functions within wire bundles). Finally, STPL-WFOLD2 projects a value for p from v_1 and applies the folding function v_3 to the current accumulator v_2, the principal p, the projected value. The result of this application is used as the new accumulator in the remaining folding steps.

B. Multi-threaded semantics

Whereas the single-threaded semantics of λ_{Wy} makes synchrony evident, in actuality a WYSTERIA program is run as a distributed program involving distinct computing principals. We make this multi-agent view apparent in a multi-threaded semantics of λ_{Wy} which defines the notion of a *protocol*:

$$\begin{array}{llll} \text{Protocol} & \pi & ::= & \varepsilon \mid \pi_1 \cdot \pi_2 \mid A \\ \text{Agent} & A & ::= & p\,\{\sigma;\kappa;\psi;e\} \mid \mathsf{s}\binom{w_1}{w_2}\,\{\sigma;\kappa;\psi;e\} \end{array}$$

A protocol π consists of a (possibly empty) sequence of agents A. There are two varieties of agents. First, *principal agents* are written $p\,\{\sigma;\kappa;\psi;e\}$ and correspond to the machine of a single principal p. Second, *secure agents* are written $\mathsf{s}\binom{w_1}{w_2}\,\{\sigma;\kappa;\psi;e\}$ and correspond to a secure block for principals w_2, where w_1 is the subset of these principals still waiting for their result. Both varieties of agents consist of a store, stack, environment and expression—the same components as the single-threaded configurations described above. We note that in the rules discussed below, we treat protocols as commutative monoids, meaning that the order of composition does not matter, and that empty protocols can be freely added and removed without changing the meaning.

Figure 7 defines the stepping judgement for protocols $\pi_1 \longrightarrow \pi_2$, read as *protocol* π_1 *steps to protocol* π_2. Rule STPP-PRIVATE steps principal p's computing agent in mode $\mathsf{p}(\{p\})$ according to the single-threaded semantics. We note that this rule covers nearly all parallel mode code, by virtue of parallel mode meaning "each principal does the same thing in parallel." However, single-threaded rules involving delegation effects, STPC-DELPAR and STPC-SECENTER are different in multi-threaded semantics.

Parallel delegation reduces by case analysis on the agent's principal p. Rule STPP-PRESENT applies when p is a member of the delegated set. In that case, p simply reduces by pushing e_2 on the stack and continue to e_1. When p is not a member of the delegated set, rule STPP-ABSENT entirely skips the first nested expression and continues with e_2. The type system ensures that in this case, p never uses x, and hence does not needs its binding in the environment.

To see the effect of these rules, consider the following code, which is like the millionaires' example we considered earlier.

$$e = \begin{cases} \textbf{let}\ x_1 \overset{\mathsf{p}(\{\textbf{Alice}\})}{=} \textbf{read}\,()\ \textbf{in} \\ \textbf{let}\ x_2 \overset{\mathsf{p}(\{\textbf{Bob}\})}{=} \textbf{read}\,()\ \textbf{in} \\ \textbf{let}\, x_3 = (\textbf{wire}_{\{\textbf{Alice}\}}(x_1)) +\!\!+ (\textbf{wire}_{\{\textbf{Bob}\}}(x_2))\,\textbf{in} \\ \textbf{let}\ x_4 \overset{\mathsf{s}(\{\textbf{Alice},\textbf{Bob}\})}{=} x_3[\textbf{Alice}] > x_3[\textbf{Bob}]\ \textbf{in} \\ x_4 \end{cases}$$

To start, both **Alice** and **Bob** start running the program (call it e) in protocol **Alice** $\{\cdot;\cdot;\cdot;e\} \cdot$ **Bob** $\{\cdot;\cdot;\cdot;e\}$. Consider evaluation for **Bob**'s portion. The protocol will take a step according to STPP-ABSENT (via STPP-FRAME) since the first let binding is for **Alice** (so **Bob** is not permitted to see the result). Rule STPP-PRESENT binds the x_2 to whatever is read from **Bob**'s console; suppose it is 5. Then, **Bob** will construct the wire bundle x_3, where **Alice**'s binding is absent and **Bob**'s binding x_2 is 5.

At the same time, **Alice** will evaluate her portion of the protocol similarly, eventually producing a wire bundle where her value for x_1 is whatever was read in (6, say), and **Bob**'s binding is absent. (Of course, up to this point each of the steps of one party might have been interleaved with steps of the other.) The key is that elements of the joint protocol that are private to one party are hidden from the others, in fact totally absent from others' local environments. Now, both are nearly poised to delegate to a secure block.

Secure delegation reduces in a series of phases that involve multi-agent coordination. In the first phase, the principal agents involved in a secure block each reduce to a secure expression $\textbf{secure}_w(e)$, using STPC-DELPSEC via STPP-PRIVATE. At any point during this process, rule STPP-BEGIN (nondeterministically) creates a secure agent with a matching principal set w and expression e. After which, each principal agent can enter their input into the secure computation via STPP-SECENTER, upon which they begin to wait, blocking until the secure block completes. Their input takes the form of their current store σ and environment ψ, which the rule *combines* with that of the secure agent. We explain the combine operation below. Once all principals have entered their inputs into the secure agent, the secure agent can step via STPP-SECSTEP. The secure agent halts when its stack is empty and its program is a value.

Once halted, the secure block's principals can leave with the output value via STPP-SECLEAVE. As values may refer to variables defined in the environment, the rule first closes the value with the secure block's environment, and then each

$\boxed{\pi_1 \longrightarrow \pi_2}$ **Protocol stepping**: "Protocol π_1 steps to π_2"

$$\begin{array}{lrcll}
\textsc{stpP-Private} & p\,\{\sigma_1;\kappa_1;\psi_1;e_1\} & \longrightarrow & p\,\{\sigma_2;\kappa_2;\psi_2;e_2\} & \text{when } \mathsf{p}(\{p\})\{\sigma_1;\kappa_1;\psi_1;e_1\} \longrightarrow \mathsf{p}(\{p\})\{\sigma_2;\kappa_2;\psi_2;e_2\} \\
\textsc{StpP-Present} & p\left\{\sigma;\kappa;\psi;\textbf{let } x \overset{\mathsf{p}(w)}{=} e_1 \textbf{ in } e_2\right\} & \longrightarrow & p\,\{\sigma;\kappa_1;\psi;e_1\} & \text{when } \{p\} \subseteq \psi[\![w]\!] \text{ and } \kappa_1 = \kappa :: \langle \mathsf{p}(\{p\});\psi;x.e_2\rangle \\
\textsc{StpP-Absent} & p\left\{\sigma;\kappa;\psi;\textbf{let } x \overset{m(w)}{=} e_1 \textbf{ in } e_2\right\} & \longrightarrow & p\,\{\sigma;\kappa;\psi;e_2\} & \text{when } \{p\} \not\subseteq \psi[\![w]\!] \\
\textsc{stpP-Frame} & \pi_1 \cdot \pi_2 & \longrightarrow & \pi_1' \cdot \pi_2 & \text{when } \pi_1 \longrightarrow \pi_1' \\
\textsc{StpP-SecBegin} & \varepsilon & \longrightarrow & \mathsf{s}(^{\cdot}_{w})\,\{\cdot;\cdot;\cdot;e\} &
\end{array}$$

$$\begin{array}{ll}
\textsc{StpP-SecEnter} & \mathsf{s}(^{w_1}_{w_2})\,\{\sigma;\cdot;\psi;e\} \cdot p\,\{\sigma';\kappa;\psi';\textbf{secure}_{w_2}(e)\} \longrightarrow \mathsf{s}(^{w_1\cup\{p\}}_{w_2})\,\{\sigma\circ\sigma';\cdot;\psi\circ\psi';e\} \cdot p\,\{\sigma';\kappa;\cdot;\textbf{wait}\} \\
\textsc{StpP-SecStep} & \mathsf{s}(^{w}_{w})\,\{\sigma_1;\kappa_1;\psi_1;e_1\} \;\longrightarrow\; \mathsf{s}(^{w}_{w})\,\{\sigma_2;\kappa_2;\psi_2;e_2\} \quad \text{when } \mathsf{s}(w)\{\sigma_1;\kappa_1;\psi_1;e_1\} \longrightarrow \mathsf{s}(w)\{\sigma_2;\kappa_2;\psi_2;e_2\} \\
\textsc{StpP-SecLeave} & \mathsf{s}(^{w_1\cup\{p\}}_{w_2})\,\{\sigma';\cdot;\psi;v\} \cdot p\,\{\sigma;\kappa;\cdot;\textbf{wait}\} \longrightarrow \mathsf{s}(^{w_1}_{w_2})\,\{\sigma';\cdot;\psi;v\} \cdot p\,\{\sigma;\kappa;\cdot;v'\} \quad \text{when } \mathsf{slice}_p(\psi[\![v]\!]_{\mathsf{s}(w_2)}) \rightsquigarrow v' \\
\textsc{StpP-SecEnd} & \mathsf{s}(^{\cdot}_{w_2})\,\{\sigma;\cdot;\psi;v\} \;\longrightarrow\; \varepsilon
\end{array}$$

Fig. 7. λ_{Wy}: operational semantics of multi-threaded target protocols

$\boxed{\mathsf{slice}_p(v_1) \rightsquigarrow v_2}$ **Value slicing**: "Value v_1 sliced for p is v_2"

$$\begin{array}{lcll}
\mathsf{slice}_p((v_1,v_2)) & \rightsquigarrow & (v_1',v_2') & \text{when } \mathsf{slice}_p(v_i) \rightsquigarrow v_i' \\
\mathsf{slice}_p(\{p:v\} +\!\!+ v_1) & \rightsquigarrow & \{p:v\} & \\
\mathsf{slice}_p(v_1 +\!\!+ v_2) & \rightsquigarrow & \cdot & \text{when } p \notin \mathsf{dom}(v_1 +\!\!+ v_2)
\end{array}$$

$\boxed{\mathsf{slice}_p(\psi) \rightsquigarrow \psi'}$ **Environment slicing**: "Environment ψ sliced for p is ψ'"

$$\begin{array}{lll}
\mathsf{slice}_p(\psi\{x \mapsto_{\mathsf{p}(w)} v\}) & \rightsquigarrow \mathsf{slice}_p(\psi)\{x \mapsto_{\mathsf{p}(\{p\})} \mathsf{slice}_p(v)\} & ,\ p \in w \\
\mathsf{slice}_p(\psi\{x \mapsto_{\mathsf{p}(w)} v\}) & \rightsquigarrow \mathsf{slice}_p(\psi) & ,\ p \notin w \\
\mathsf{slice}_p(\psi\{x \mapsto v\}) & \rightsquigarrow \mathsf{slice}_p(\psi)\{x \mapsto \mathsf{slice}_p(v)\} &
\end{array}$$

Fig. 8. λ_{Wy}: Slicing judgments (selected rules).

party receives the *slice* of the closed value that is relevant to him. We show selected slicing rules in Figure 8. We elide the complete definition (see our technical report), but intuitively, the slice of a value is just a congruence, except in the case of wire bundle values, where each principal of the bundle gets its own component; other principals get $\cdot$, the empty wire bundle. The *combine* operation mentioned above is analogous to slice, but in the reverse direction – it combines the values in a congruent way, but for wire bundles it concatenates them (see the technical report).

Returning to our example execution, we can see that **Bob** and **Alice** will step to $\textbf{secure}_{\{\textbf{Alice},\textbf{Bob}\}}(e')$ (with the result poised to be bound to x_4 in both cases) where e' is $x_3[\textbf{Alice}] > x_3[\textbf{Bob}]$. At this point we can begin a secure block for e' using StpP-SecBegin and both **Bob** and **Alice** join up with it using StpP-SecEnter. This causes their environments to be merged, with the important feature that for wire bundles, each party contributes his/her own value, and as such x_3 in the joined computation is bound to $\{\textbf{Alice} : 6\} +\!\!+ \{\textbf{Bob} : 5\}$. Now the secure block performs this computation while **Alice** and **Bob**'s protocols wait. When the result 1 (for "true") is computed, it is passed to each party via StpP-SecLeave, along with the sliced environment ψ' (as such each party's wire bundle now just contains his/her own value). At this point each party steps to 1 as his final result.

VI. Meta theory

We prove several meta-theoretical results for λ_{Wy} that are relevant for mixed-mode multi-party computations. Proofs for these results can be found in the technical report. First, we show that well-typed Wysteria programs always make progress (and stay well-typed). In particular, they never get *stuck*: they either complete with a final result, or reach a well-defined error state (due to an out of bounds array access or update, but for no other reason).

Theorem 6.1 (Progress): If $\Sigma \vdash \mathcal{C}_1 : \tau$ then either $\mathcal{C}_1$ **halted** or there exists configuration $\mathcal{C}_2$ such that $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$.

Theorem 6.2 (Preservation): If $\Sigma_1 \vdash \mathcal{C}_1 : \tau$ and $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, then there exists $\Sigma_2 \supseteq \Sigma_1$ s.t. $\Sigma_2 \vdash \mathcal{C}_2 : \tau$.

The premise $\Sigma \vdash \mathcal{C}_1 : \tau$ appears in both theorems and generalizes the notion of well-typed expressions to that of well-typed configurations; it can be read as *under store typing* Σ*, configuration* $\mathcal{C}_1$ *has type* τ. This definition involves giving a notion of well-typed stores, stacks and environments, which we omit here for space reasons (see technical report).

Next, turning to the relationship between the single- and multi-threaded semantics, the following theorem shows that every transition in the single-threaded semantics admits corresponding transitions in the multi-threaded semantics:

Theorem 6.3 (Sound forward simulation): Suppose that $\Sigma \vdash \mathcal{C}_1 : \tau$ and that $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$. Then there exist π_1 and π_2 such that $\pi_1 \longrightarrow^* \pi_2$ and $\mathsf{slice}_w(\mathcal{C}_i) \rightsquigarrow \pi_i$ (for $i \in \{1,2\}$), where w is the set of all principals.

The conclusion of the theorem uses the auxiliary slicing judgement to construct a multi-threaded protocol from a (single-threaded) configuration.

Turning to the multi-threaded semantics, the following theorem states that the non-determinism of the protocol semantics always resolves to the same outcome, i.e., given any two pairs of protocol steps that take a protocol to two different configurations, there always exists two more steps that bring these two intermediate states into a common final state:

Theorem 6.4 (Confluence): Suppose that $\pi_1 \longrightarrow \pi_2$ and $\pi_1 \longrightarrow \pi_3$, then there exists π_4 such that $\pi_2 \longrightarrow \pi_4$ and $\pi_3 \longrightarrow \pi_4$.

A corollary of confluence is that every terminating run of the (non-deterministic) multi-threaded semantics yields the same result.

For correspondence in the other direction (multi- to single-threaded), we can prove the following lemma.

Lemma 6.1 (Correspondence of final configurations): Let $\Sigma \vdash \mathcal{C} : \tau$ and $\mathsf{slice}_w(\mathcal{C}) \rightsquigarrow \pi$, where w is the set of all principals. If $\pi \longrightarrow^* \pi'$, where π' is an error-free terminated protocol, then there exists an error-free terminated $\mathcal{C}'$ s.t. $\mathcal{C} \longrightarrow^* \mathcal{C}'$ and $\mathsf{slice}_w(\mathcal{C}') \rightsquigarrow \pi'$.

One of the most important consequences of these theorems is that principals running parallel to one another, and whose computations sucessfully terminate in the single-threaded semantics, will be properly synchronized in the multi-threaded semantics; e.g., no principal will be stuck waiting for another one that will never arrive.

We would like to prove a stronger, *backward simulation* result that also holds for non-terminating programs, but unfortunately it does not hold because of the possibility of errors and divergence. For example, when computing **let** $x \overset{M}{=} e_1$ **in** e_2, the single-threaded semantics could diverge or get an array access error in e_1, and therefore may never get to compute e_2. However, in multi-threaded semantics, principals not in M are allowed to make progress in e_2. Thus, for those steps in the multi-threaded semantics, we cannot give a corresponding source configuration. We plan to take up backward simulation (e.g., by refining the semantics) as future work.

Security: As can be seen in Figure 7, the definition of the multi-threaded semantics makes apparent that all inter-principal communication (and thus information leakage) occurs via secure blocks. As such, all information flows between parties must occur via secure blocks. These flows are made more apparent by WYSTERIA's single-threaded semantics, and are thus easier to understand.

VII. IMPLEMENTATION

We have implemented a tool chain for WYSTERIA, including a frontend, a type checker, and a run-time interpreter. Our implementation is written in OCaml, and is roughly 6000 lines of code. Our implementation supports the core calculus features (in gentler syntax) and also has named records and conditionals. To run a WYSTERIA program, each party invokes the interpreter with the program file and his principal name. If the program type checks it is interpreted. The interpreter dynamically generates boolean circuits from secure blocks it encounters, running them using Choi et al.'s implementation [23] of the Goldreich, Micali, and Wigderson (GMW) protocol [2], which provably simulates (in the semi-honest setting) a trusted third party. Figure 9 gives a high-level overview of WYSTERIA running for four clients (labeled A, B, C and D). The remainder of this section discusses the implementation in detail.

Type checker: The WYSTERIA type checker uses standard techniques to turn the declarative type rules presented earlier into an algorithm (e.g., inlining uses of subsumption to make the typing rules syntax-directed). We use the Z3 SMT solver [27] to discharge the refinement implications, encoding sets using Z3's theory of arrays. Since Z3 cannot reason about the cardinality of sets encoded this way, we add support for **singl**(ν) as an uninterpreted logical function `single` that maps sets to booleans. Facts about this function are added to Z3 in the rule T-PRINC.

Interpreter: When the interpreter reaches a secure block it compiles that block to a circuit in several steps. First, it must convert the block to straight-line code. It does this by expanding **wfold** and **waps** expressions (according to the now available principal sets), inlining function calls, and selectively substituting in non-wire and non-share variables from its environment. The type system ensures that, thanks to synchrony, each party will arrive at the same result.

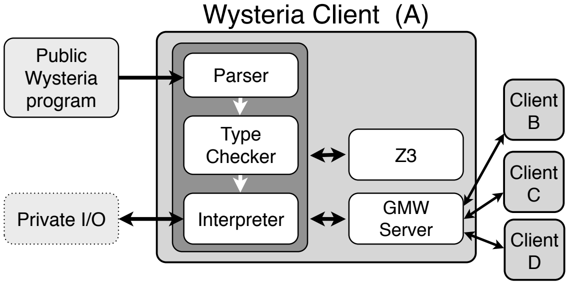

Fig. 9. Overview of WYSTERIA system with four interacting clients.

Next, the interpreter performs a type-directed translation to a boolean circuit, taking place in two phases. In the first phase, it assigns a set of *wire IDs* to each value and expression, where the number of wires depends on the corresponding type. The wires are stitched together using high-level operators (e.g., `ADD r1 r2 r3`, where `r1`, `r2`, and `r3` are ranges of wire IDs). As usual, we generate circuits for both branches of **case** expressions and feed their results to a multiplexer switched by the compiled guard expression's output wire. Records and wire bundles are simply an aggregation of their components. Wire IDs are also assigned to the input and output variables of each principal—these are the free wire and share variables that remained in the block after the first step.

In the second phase, each high-level operator (e.g. `ADD`) is translated to low-level AND and XOR gates. Once again, the overall translation is assured to produce exactly same circuit, including wire ID assignments, at each party. Each host's interpreter also translates its input values into bit representations. Once the circuit is complete it is written to disk.

At this point, the interpreter signals a local server process originally forked when the interpreter started. This server process implements the secure computation using the GMW library. This process reads in the circuit from disk and coordinates with other parties' GMW servers using network sockets. At the end of the circuit execution, the server dumps the final output to a file, and signals back the interpreter. The interpreter then reads in the result, converts it to the internal representation, and carries on with parallel mode execution (or terminates if complete).

Secure computation extensions and optimizations: The GMW library did not originally support secret shares, but they were easy to add. We extended the circuit representation to designate particular wires as making shares (due to **makesh**(v) expressions) and reconstituting shares (due to **combsh**(v) expressions). For the former, we modified the library to dump the designated (random) wire value to disk—already a share—and for the latter we do the reverse, directing the result into the circuit. We also optimized the library's Oblivious Transfer (OT) extension implementation for the mixed-mode setting.

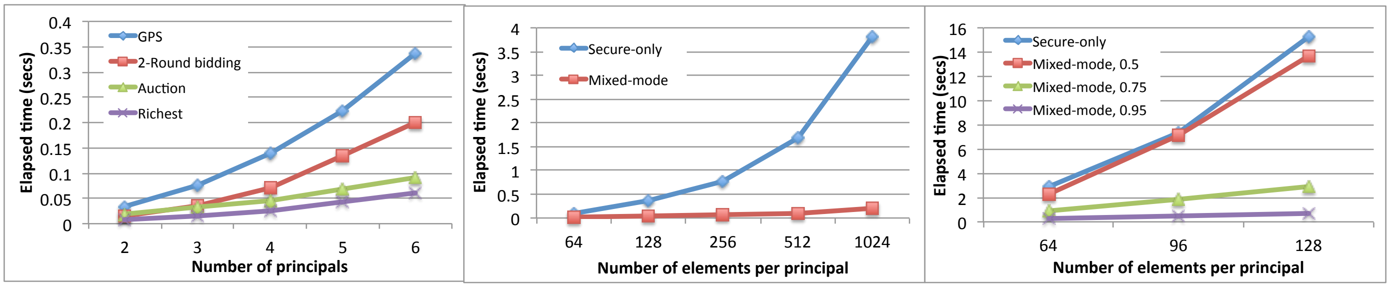

Fig. 10. (a) n-party SMC examples. (b) Secure median vs mixed-mode median. (c) Secure PSI vs mixed-mode PSI for different density.

VIII. Experiments

We conduct two sets of experiments to study Wysteria's empirical performance. First we measure the performance of several n-party example programs of our own design and drawn from the literature. We find that these programs run relatively quickly and scale well with the number of principals. Second, we reproduce two experiments from the literature that demonstrate the performance advantage of mixed-mode vs. monolithic secure computation.[5]

Secure computations for n parties: We have implemented several n-party protocols as Wysteria functions that are *generic* in the participating principal set (whose identity and size can both can vary). The Richest protocol computes the richest principal, as described in Section II. The GPS protocol computes, for each participating principal, the other principal that is nearest to their location; everyone learns their nearest neighbor without knowing anyone's exact location. The Auction protocol computes the high bidder among a set of participating principals, as well as the second-highest bid, which is revealed to everyone; only the auction holder learns who is the winning bidder. Finally, we have implemented the two-round bidding game from Section II for multiple principals. Recall that this example crucially relies on Wysteria's notion of secret shares, a high-level abstraction that existing SMC languages lack.

Figure 10(a) shows, for varying numbers of principals, the elapsed time to compute these functions. We can see each of these computations is relatively fast and scales well with increasing numbers of parties.

Mixed-mode secure computations: To investigate the performance advantages of mixed-mode secure computations, we study two functions that mix modes: two-party median computes the median of two principals' elements, and two-party intersect is a PSI protocol that computes the intersection of two principals' elements. In both cases, we compare the *mixed-mode* version of the protocol with the *secure-only versions*, which like FairPlayMP, only use a single monolithic secure block. We chose these protocols because they have been studied in past literature on secure computation [5], [7], [22]; both protocols enjoy the property that by mixing modes, certain computation steps in the secure-only version can either be off-loaded to local computation (as in median) or avoided altogether (as in intersect), while providing the same privacy guarantees.

Mixed-mode median: Here is a simplified version of median that accepts two numbers from each party:

```
let m =sec({A,B})=
  let x1 = (fst w1[A]) in let x2 = (snd w1[A]) in
  let y1 = (fst w2[B]) in let y2 = (snd w2[B]) in
  let b1 = x1 ≤ y1 in
  let x3 = if b1 then x2 else x1 in
  let y3 = if b1 then y1 else y2 in
  let b2 = x3 ≤ y3 in
  if b2 then x3 else y3
in m
```

The participating principals A and B store their (sorted, distinct) input pairs in wire bundles w_1 and w_2 such that w_1 contains A and B's smaller numbers and w_2 contains their larger ones. First, the protocol compares the smaller numbers. Depending on this comparison, the protocol discards one input for each principal. Then, it compares the remaining two numbers and the smaller one is chosen as the median (thus preferring the lower-ranked element when there is an even number).

Under certain common assumptions [22], the following mixed-mode version equivalently computes median with the same security properties.

```
1 let w1 =par(A,B)= (wire {A} x1) ++ (wire {B} y1) in
2 let b1 =sec(A B)= (w1[A] ≤ w1[B]) in
3 let x3 =par(A)=    if b1 then x2 else x1 in
4 let y3 =par(B)=    if b1 then y1 else y2 in
5 let w2 =par(A,B)= (wire {A} x3) ++ (wire {B} y3) in
6 let b2 =sec(A,B)= (w2[A] ≤ w2[B]) in
7 let m  =sec(A,B)= if b2 then w2[A] else w2[B] in
8 m
```

The key difference compared with the secure-only version is that the conditional assignments on lines 3 and 4 need not be done securely. Rather, the protocol reveals b1 and b2, allowing each principal to perform these steps locally. Past work as shown that this change still preserves the final *knowledge profile* of each party, and is thus equally secure in the semi-honest setting [22].

Figure 10(b) compares the performance of mixed-mode median over secure-only median for varying sizes of inputs (generalizing the program above).

We can see that the elapsed time for mixed-mode median remains comparatively fixed, even as input sizes increase exponentially. By comparison, secure-only median scales poorly with increasing input sizes. This performance difference illustrates the (sometimes dramatic) benefit of supporting mixed-mode computations.

[5] We ran all our experiments on Mac OS X 10.9, with 2.8 GHz Intel Core Duo processor and 4GB memory. To isolate the performance of Wysteria from that of I/O, all the principals run on the same host, and network communication uses local TCP/IP sockets.

Private set intersection: In intersect , two principals compute the intersection of their private sets. The set sizes are assumed to be public knowledge. As with median, the intersect protocol can be coded in two ways: a secure-only pairwise comparison protocol performs $n_1 \times n_2$ comparisons inside the secure block which result from the straight-line expansion of two nested loops. Huang et. al. [7] propose two optimizations to this naive pairwise comparison protocol. First, when a matching element is found, the inner loop can be short circuited, avoiding its remaining iterations. Second, once an index in the inner loop is known to have a match, it need not be compared in the rest of the computation. We refer the reader to their paper for further explanation. We note that WYSTERIA allows programmers to easily express these optimizations in the language, using built-in primitives for expressing parallel-mode loops and arrays.

Figure 10(c) compares the secure-only and mixed-mode versions of intersect . For the mixed-mode version, we consider three different densities of matching elements: 0.5, 0.75, and 0.95 (where half, three-quarters, and 95% of the elements are held in common). For the unoptimized version, these densities do not affect performance, since it always executes all program paths, performing comparisons for every pair of input elements. As can be seen in the figure, as the density of matching elements increases, the mixed-mode version is far more performant, even for larger input sizes. By contrast, the optimization fails to improve performance at lower densities, as the algorithm starts to exhibit quadratic-time behavior (as in the secure-only version).

All the examples presented here, and more (including a prototype WYSTERIA program to deal cards for mental card games), are given in full in the the technical report.

IX. RELATED WORK

Several research groups have looked at compiling multiparty programs in high-level languages to secure protocols. Our work is distinguished from all of these in several respects.

Support for multi-party ($n > 2$) computations: Our language design has carefully considered support for secure computations among more than two parties. Most prior work has focused on the two-party case. Fairplay [17] compiles a garbled circuit from a Pascal-like imperative program. The entry point to this program is a function whose two arguments are *players*, which are records defining each participant's expected input and output type. More recently, Holzer et al [18] developed a compiler for programs with similarly specified entry points, but written in (a subset of) ANSI C. Alternatively, at the lowest level there are libraries for building garbled circuits directly, e.g., those developed by Malka [13], Huang et al [14], and Mood et al [15]. These lines of work provide important building blocks for the back end of our language (in the two party case).

The only language of which we are aware that supports $n > 2$ parties is FairplayMP [19]. Its programs are similar to those of FairPlay, but now the entry point can contain many player arguments, including arrays of players, where the size of the array is statically known. Our *wire bundles* have a similar feel to arrays of players. Just as FairPlayMP programs can iterate over (arbitrary-but-known-length) arrays of players in a secure computation, we provide constructs for iterating over wire bundles. Unlike the arrays in FairplayMP, however, our wire bundles have the possibility of representing different *subsets* of principals' data, rather than assume that all principals are always present; moreover, in WYSTERIA these subsets can themselves be treated as variable.

Support for mixed-mode computations: All of the above languages specify secure computations in their entirety, e.g., a complete garbled circuit, but much recent work (including this work) has focused on the advantage of *mixed-mode* computations.

As first mentioned in Section I, L1 [24] is an intermediate language for mixed-mode SMC, but is limited to two parties. Compared to L1, WYSTERIA provides more generality and ease of use. Further, WYSTERIA programmers need not be concerned with the low-level mechanics of inter-party communication and secret sharing, avoiding entire classes of potential misuse (e.g., when two parties wait to receive from each other at the same time, or when they attempt to combine shares of distinct objects).

PCF [16] is a circuit format language for expressing mixed-mode secure computations for two parties. As with L1, it allows programmers to distinguish secure computation from surrounding computation that is run locally, in the clear. It also suffers from limitations that are similar to those of L1, in that it is (by design) very low level, and in that it lacks abstractions for supporting multiple parties, as well as a formal semantics.

SMCL [8] is a language for secure computations involving a replicated client and a shared "server" which represents secure multiparty computations. Our approach is less rigid in its specification of roles: we have secure and parallel computations involving arbitrary numbers of principals, rather than all of them, or just one, as in SMCL. SMCL provides a type system that aims to enforce some information flow properties modulo declassification. SMCL's successor, VIFF [28], reflects the SMCL computational model as a library/DSL in Python, but lacks type-based guarantees.

Liu et al. define a typed intermediate language for mixed-mode SMC protocols [25]. However, their approach is limited to two parties and their proposed language is simplistic in comparison to WYSTERIA, e.g., it lacks function abstractions and is thus not suitable for expressing reusable library code. Given each variable's classification as public, secret (e.g., used only within an SMC), or private, their compiler can produce a mixed mode protocol guaranteed to have the same security as a monolithic secure protocol. They use an information flow control-style type system to prohibit illegal flows of information between modal computations. By contrast, WYSTERIA makes no attempt to relate the security properties of mixed-mode and non-mixed-mode versions of a protocol; instead, one must use a separate analysis for this purpose (e.g. [22]). We note that WYSTERIA could safely use Liu et al.'s novel "RAM-model secure" computation protocol as a means to implement secure blocks among two parties.

SMCs as cloud computations: Another line of research in SMCs deals with a client-server setting, where client wants to run a function over his private input using untrusted servers (e.g. in a cloud). To protect confidentiality of his data, the client distributes secret shares of his input among the servers. The servers run *same* function, but use their own shares. Finally, they send the output shares to the client, who then recovers the clear output value. Launchbury et. al. [12] present a table-lookup based optimization for such SMC protocols, that aims at minimizing the cost incurred by expensive operations such as multiplication and network communication between servers. Mitchell et. al. [11] give an expressive calculus for writing such functions. Their calculus is mixed-mode, but only in terms of data—the programs can use both encrypted (private) and non-encrypted (public) values. They give an extended information flow type system that rejects programs that cannot be run on a *secure computation platform* (such as homomorphic encryption). In WYSTERIA, the above client-server setting can be expressed as a monolithic secure block to be run by the servers, each of which holds secret shares of client's input. As we have shown in the paper, we can express more general mixed-mode SMCs.

Other language proposals: The TASTY compiler produces secure computations that may combine homomorphic encryption and garbled circuits [29]. Its input language, TASTYL, requires explicit specification of communication between parties, as well as the means of secure computation, whereas in our approach, such concerns are handled automatically (during run-time compilation of generated circuits). Kerschbaum et al. [30] explore automatic selection of mixed protocols consisting of garbled circuits and homomorphic encryption. Jif/Split enables writing multi-party computations in Java as (conceptually) single-threaded programs [31]. It offers compiler support for dividing Java programs into pieces to run on different hosts, based on information-flow analysis and explicit declassifications. Unlike our work, Jif/Split runs statically (at compile time), depends on real-life trusted third parties, and lacks language abstractions and run-time techniques for employing secure computations without a trusted third party.

X. CONCLUSION

This paper presents WYSTERIA, the first programming language designed for expressing mixed-mode computations for multiple parties. In contrast to prior work, multi-party protocols in WYSTERIA can be expressed *generically*, and may perform dynamic decisions about each principal's role within a protocol. WYSTERIA's type system ensures that well-typed programs never misuse the language's abstractions. Further, WYSTERIA programs are concise and readable, since they can be interpreted through a (conceptually simple) single-threaded semantics, as well as a (more realistic) multi-threaded semantics. We show formally that these two views coincide. We present implementation of WYSTERIA in the form of an interpreter that uses the GMW protocol to realize secure blocks. We show our implementation performs well on new and known protocols.

Acknowledgments: We would like to thank Nikhil Swamy and anonymous reviewers for their helpful comments and suggestions, and Jon Katz for helping us with the GMW library. This research was sponsored by NSF award CNS-1111599 and the US Army Research laboratory and the UK Ministry of Defence under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the U.S. Government, the UK Ministry of Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

REFERENCES

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

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

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

[4] G. Aggarwal, N. Mishra, and B. Pinkas, "Secure computation of the k th-ranked element," in *EUROCRYPT*. Springer, 2004.

[5] F. Kerschbaum, "Automatically optimizing secure computation," in *CCS*, 2011.

[6] M. J. Freedman, K. Nissim, and B. Pinkas, "Efficient private matching and set intersection," in *EUROCRYPT*, 2004.

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

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

[9] 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.

[10] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT*, 1999.

[11] J. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman, "Information-flow control for programming on encrypted data," in *CSF*, 2012.

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

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

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

[15] B. Mood, L. Letaw, and K. Butler, "Memory-efficient garbled circuit generation for mobile devices," in *Financial Cryptography*, 2012.

[16] B. Kreuter, ahbi shelat, B. Mood, and K. Butler, "PCF: A portable circuit format for scalable two-party secure computation," in *USENIX*, 2013.

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

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

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

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

[21] M. Furr and J. S. Foster, "Checking Type Safety of Foreign Function Calls," *TOPLAS*, vol. 30, no. 4, pp. 1–63, July 2008.

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

[23] 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/.

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

[25] 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.

[26] A. Rastogi, M. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," Department of Computer Science, University of Maryland, Tech. Rep. CS-TR-5034, March 2014.

[27] "Z3 theorem prover," z3.codeplex.com.

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

[29] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "Tasty: tool for automating secure two-party computations," in *CCS*, 2010.

[30] F. Kerschbaum, T. Schneider, and A. Schröpfer, "Automatic protocol selection in secure two-party computations," in *NDSS*, 2013.

[31] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM Trans. Comput. Syst.*, 2002.