ABSTRACT

Title of dissertation: Language-based Techniques for Practical and Trustworthy Secure Multi-party Computations

Aseem Rastogi, Doctor of Philosophy, 2016

Dissertation directed by: Professor Michael Hicks
Department of Computer Science

Secure Multi-party Computation (MPC) enables a set of parties to collaboratively compute, using cryptographic protocols, a function over their private data in a way that the participants do not see each other's data, they only see the final output. Typical MPC examples include statistical computations over joint private data, private set intersection, and auctions. While these applications are examples of *monolithic* MPC, richer MPC applications move between "normal" (i.e., per-party local) computations and "secure" (i.e., joint, multi-party secure) modes repeatedly, resulting overall in *mixed-mode* computations. For example, we might use MPC to implement the role of the dealer in a game of mental poker – the game will be divided into rounds of local decision-making (e.g. bidding) and joint interaction (e.g. dealing). Mixed-mode computations are also used to improve performance over monolithic secure computations.

Starting with the Fairplay project, several MPC frameworks have been proposed in the last decade to help programmers write MPC applications in a *high-level* language, while the toolchain manages the low-level details. However, these frameworks are either not expressive enough to allow writing mixed-mode applications or lack formal specifica-

tion, and reasoning capabilities, thereby diminishing the parties' trust in such tools, and the programs written using them. Furthermore, none of the frameworks provides a verified toolchain to run the MPC programs, leaving the potential of security holes that can compromise the privacy of parties' data.

This dissertation presents language-based techniques to make MPC more practical and trustworthy. First, it presents the design and implementation of a new MPC Domain Specific Language, called Wysteria, for writing rich mixed-mode MPC applications. Wysteria provides several benefits over previous languages, including a conceptual single thread of control, generic support for more than two parties, high-level abstractions for secret shares, and a fully formalized type system and operational semantics. Using Wysteria, we have implemented several MPC applications, including, for the first time, a card dealing application.

The dissertation next presents Wys*, an embedding of Wysteria in F*, a full-featured verification oriented programming language. Wys* improves on Wysteria along three lines: (a) It enables programmers to formally verify the correctness and security properties of their programs. As far as we know, Wys* is the first language to provide verification capabilities for MPC programs. (b) It provides a partially verified toolchain to run MPC programs, and finally (c) It enables the MPC programs to use, with no extra effort, standard language constructs from the host language F*, thereby making it more usable and scalable.

Finally, the dissertation develops static analyses that help optimize monolithic MPC programs into mixed-mode MPC programs, while providing similar privacy guarantees as the monolithic versions.

Language-based Techniques for Practical and Trustworthy Secure Multi-party Computations

by

Aseem Rastogi

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2016

Advisory Committee:
Professor Michael Hicks, Chair/Advisor
Dr. Nikhil Swamy
Professor Lawrence C. Washington, Dean's Representative
Professor Jonathan Katz
Professor Charalampos (Babis) Papamanthou

Acknowledgments

TBD

Table of Contents

List of Figures

List of Abbreviations

MPC	Secure Multi-party Computation
DSL	Domain Specific Language
PSI	Private Set Intersection
I/O	Input/Output
GUI	Graphical User Interface
FFI	Foreign Function Interface
VDSILE	Verified, Domain-Specific Integrated Language Extension
SIMD	Single Instruction, Multiple Data
AST	Abstract Syntax Tree
SMT	Satisfiability Modulo Theories
ANF	A-Normal Form
OT	Oblivious Transfer
API	Application Programming Interface
DNF	Disjunctive Normal Form
IL	Intermediate Language

Chapter 1: Introduction

Secure Multi-party Computation (MPC) protocols [1–3] 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 set-intersection operator $\cap$ [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]

In recent years MPC has also been used in detecting tax frauds [9] and collaborative supply chain management [10].

MPC can be used for much more – it eliminates the need for a trusted third party while providing similar data privacy guarantees. Users of an online dating application do not have to trust the application server to match the compatible profiles, they can use MPC and retain the control of their data. A web browser plugin can use MPC with the ad-providers to compute relevant ads for a user without revealing all user preferences to the ad-

provider. An email client plugin can use MPC with other email clients to collaboratively build a spam filter without revealing emails to each other. In a game of online poker, the players do not have to trust the game portal for card dealing, they can use MPC. In summary, MPC provides a general purpose solution to address increasing data privacy concerns.

The examples above clearly go beyond the secure computation of a single function f. Instead, they 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, an online poker MPC application is divided into rounds of local computations (for strategizing and bidding) and joint secure computations (for card dealing). In addition, secure computations use and modify a *secret state* consisting of the deck of cards. The parties secretly share the deck in a way that no single party can see the current deck in clear, but in secure computations they can combine their shares to recover it. Such a secure state is also known as *secret shares*.

Mixed-mode computations are also used to improve the 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 prior work has shown can net up to a 30x performance improvement [5].

MPC for a function f is typically implemented using cryptographic protocols that expect f to be represented as a boolean or arithmetic circuit [1–3, 11]. Several libraries and intermediate languages have been designed that provide efficient building blocks for

constructing circuits [12–14]. Programming directly with circuits and cryptography via a host-language API can be painful, so starting with the Fairplay project [15] many researchers have designed higher-level domain-specific languages (DSLs) in which to program MPCs. In Fairplay, a compiler accepts a Pascal-like imperative program and compiles it to a garbled circuit [1]. More recent efforts by Holzer et al. [16] and Kreuter et al. [17] support subsets of ANSI C, and follow-on work has expanded Fairplay's expressiveness to handle $n > 2$ parties [18]. While such languages undoubtedly make it easier to program MPCs, they still have several drawbacks regarding both security and usability.

First, projects like Fairplay focus *only* on how to compile normal-looking code into a representation like a boolean circuit that can be run by an MPC 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 MPC engine to evaluate the generated code. However, writing mixed-mode computations in a *single* language has some compelling benefits over the multi-lingual approach. For one, 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 [19]. 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. Also, 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.

Second, MPC participants should be able to *reason that f is correct and sufficiently privacy preserving*, i.e., that it computes the intended function and its output will not reveal too much information about the inputs [20]. For example, for the optimized private set intersection [7], parties should be able to formally prove that the extra messages reveal nothing beyond the final result. The goal of an MPC DSL is secure computations, and such reasoning gives assurance that this goal is being achieved. Yet, only a few DSLs (e.g. Sharemind DSL [21]) have a mathematical semantics that can serve as a basis for formal reasoning.

Third, those languages that do have a semantics lack support for (semi-) *automated reasoning of MPC programs*: only by-hand proofs are possible, which provide less assurance than formally verified proofs. A middle ground might be a mechanization of the semantics and its metatheory [22], which adds greater assurance that it is correct [23], but no DSL has a mechanized semantics.

Fourth, there is a gap between the semantics, if there is one, and the actual implementation. Within that gap is the potential for security holes. *Formal verification of the MPC DSL's toolchain* can significantly reduce the occurrence of security-threatening bugs [24–31], but no existing MPC DSL implementation has been (even partially) verified.

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

This dissertation makes significant advances in solving each of the problems above. In particular, our thesis is the following:

> *Formal methods from programming language design and program verification can enable the development of rich and practical Secure Multi-party Computation applications, with strong correctness and security guarantees.*

1.1 Overview

In this section, we present an overview of the contributions of this dissertation, relating to how it addresses the problems mentioned above.

1.1.1 Wysteria: A programming language for mixed-mode MPC

We first present a new MPC DSL, called Wysteria, for writing mixed-mode secure computations [32]. Wysteria is the first language to support writing mixed-mode, multiparty computations in a generic manner, supporting any number of participants. We summarize its several compelling features below.

Conceptual single-threaded semantics. 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 [5,33]). We formalize WYSTERIA's single-threaded semantics and prove a simulation theorem from single- to multi-threaded semantics. 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 of *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 their 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 wire bundle. The domain of such a wire bundle may be unspecified initially. The WYSTERIA compiler employs *dynamic circuit generation* to produce circuits when unknowns (like wire bundle domains) 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 single-threaded view ensures that 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 MPC languages would make a proof of type soundness for mixed-mode programs far more difficult.

WYSTERIA is not the first language for mixed mode MPC, but is unique in its high-level design, generality, and formal guarantees. For example, languages like L1 [34] and SMCL [35] 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 mes-

sage (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.

We have implemented a Wysteria interpreter which executes secure blocks by compiling them to boolean circuits, executed by Choi et al.'s implementation [11] of the Goldreich, Micali, and Wigderson (GMW) protocol [2]. We have used Wysteria to build a broad array of mixed-mode programs proposed in the literature, along with some new ones, most notably a card dealing application. 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.

1.1.2 Wys*: A verified language extension for MPC

With Wysteria's easy-to-use, high-level abstractions in hand, the dissertation next focuses on improving the usability and security of Wysteria programs and its toolchain. While Wysteria significantly advances the state-of-the-art of the MPC DSLs, it does not provide formal reasoning capabilities, and being a standalone DSL, lacks the features of a rich programming language. To that end, we present Wys* [36], an embedding of

WYSTERIA in F* [37], a verification-oriented, full-featured programming language. WYS* is what we call a *Verified, Domain-Specific Integrated Language Extension*, a new kind of embedded DSL exemplified by following three elements.

Integrated language extension. Programmers can write WYS* MPC source programs in what is essentially an extended dialect of F*. Like so-called *shallow* domain-specific language embeddings, the WYSTERIA-specific combinators are expressed in normal F* syntax, with prescriptions on their correct use expressed with F*'s dependent type-and-effect system. This arrangement means that programmers can use F*'s semi-automated verification facilities to prove correctness and security properties about their MPC programs.

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

Partially verified implementation. Within F⋆ we mechanize two operational semantics for Wysteria: a single-threaded semantics that formalizes the SIMD view, mentioned above, and a distributed semantics, that formalizes programs as they are actually run by multiple parties. Importantly, we have machine-checked proofs that the single-threaded semantics is *sound* with respect to the distributed semantics, and that the distributed semantics is correctly implemented by our interpreter. As a result, we have verified that the properties we prove about the WYSTERIA-extended F⋆ source programs hold for the multi-party programs that actually run. There is an important caveat, though: Our interpreter makes use of a circuit library to compile ASTs to circuits and then execute them using the Goldreich, Micali and Wigderson (GMW) multi-party computation protocol [2], but at present this library is not formally verified. Formal verification of GMW (which is, at present, an open problem) would add even greater assurance.

To enable reasoning about the security properties of WYS⋆ programs, WYS⋆ semantics is instrumented with *traces* that record observations made by the parties during the execution of a program. The WYS⋆ API makes these traces available to the programmers, using which they can state and prove security properties about their programs.

Using WYS⋆ we have implemented several programs, including private set intersection (PSI) and joint median. For PSI and joint median we implemented two versions each, a straightforward one, but which achieves poor performance, and a more optimized version. We formally proved that the optimized and unoptimized versions are equivalent, both functionally and with respect to privacy. In particular, we prove that the visible events in the optimized version's trace provides neither participant with any additional information about the other's secrets. Performance experiments confirm that the optimized versions

do indeed perform better.

1.1.3 Knowledge inference for optimizing MPC

Wys*, with its Foreign Function Interface (FFI), enables MPC programs to inherit standard functionality such as UI libraries, datatypes, etc. directly from the host language F*. However, another usability concern with MPC has to do with the performance – MPC cryptographic protocols are known to be prohibitively expensive. Researchers have shown that in some cases, such as PSI [7] and median computation [5], a monolithic MPC program can be optimized to a mixed-mode MPC program while retaining the privacy characteristics of the monolithic version. In particular, the mixed-mode version reveals exactly the values of those intermediate variables that would have been known in the monolithic version as well.

The final part of this dissertation explores methods for inferring when and if variables in an MPC can be inferred by the participants, and thus may enable protocol optimizations [33]. Specifically, we consider two related problems: knowledge inference and constructive knowledge inference. Both problems are specified by giving an MPC as a *single* program that uses multiple parties' variables. From this program, a solution to the knowledge inference problem states which parties can learn which additional variables (other than the inputs and outputs), if any, from a cooperative run of the unoptimized protocol. We call a knowledge inference solution constructive if, in addition to correctly asserting that a party p knows a variable y, the solution also gives an evidence of party p's knowledge of y in the form of a program that computes y from p's private data and

the final output. We provide a sound and complete solution for the knowledge inference problem, and a sound and conditionally-complete solution to the constructive knowledge inference problem. Our solutions build on previous work on template-based program verification [38, 39] formulating the knowledge problem as satisfiability of predicates which are discharged using the Z3 SMT-solver [40] in the backend.

1.2 Threat model

Throughout this dissertation, we assume the *semi-honest* (also known as honest-but-curious) threat model [41]. In this model, parties follow the protocol properly to its completion, but keep a record of all the intermediate computations (internal coin tosses and messages received from other parties). In other words, the parties are *honest* (they follow the proper protocol) but *curious* (they are interested in learning other parties' secrets). Previous work has also considered mixed-mode MPC protocols in the context of semi-honest threat model [5, 7]. While this model yields weaker security than the malicious model, it is useful if information leakage is the only concern (e.g. two corporations who would be wary of negative reactions if they do not comply with the protocol). See Section 6.2 for discussion on supporting malicious threat model.

1.3 Summary

In summary, this dissertation makes the following contributions:

1. We describe the design and implementation of a new MPC DSL, called Wysteria. Wysteria is the first language to support writing mixed-mode MPCs in a generic

manner, supporting any number of participants. It provide easy-to-use, high-level abstractions with a conceptual single-threaded semantics that simplifies the reasoning about the correctness and security properties of MPC programs.

- We formalize two semantics for WYSTERIA, a single-threaded specification semantics and a distributed, actual semantics, and prove that the two correspond.
- We implement a WYSTERIA interpreter that executes secure blocks by compiling them at runtime to boolean circuits, executed by Choi et al.'s implementation of the Goldreich, Micali, and Wigderson protocol.
- We program several MPC programs in WYSTERIA, including for the first time, a card dealing application, and illustrate that WYSTERIA's novel high-level abstractions, e.g. secure state, enable expressing novel protocols not considered previously in the literature.

2. We describe WYS*, an embedding of WYSTERIA in F*. The embedding is a *Verified, Domain-Specific Integrated Language Extension* providing the benefits of (a) formal verification of the correctness and security properties of the MPC programs, (b) partially verified implementation of the WYSTERIA toolchain, and (c) a Foreign Function Interface that enables MPC programs to inherit standard language constructs directly from the host language F*. The programs we have written with WYS* constitute the first MPC programs to be formally verified, and the WYS* implementation itself is the first MPC DSL to be be constructed with formal assurance.

3. Finally, we formalize the problem of *knowledge inference* and *constructive knowl-*

edge inference, that can help optimize monolithic secure computations into mixed-mode computations, with similar privacy characteristics as the monolithic versions. We present solutions for both these problems and formally prove their soundness and relative completeness.

Chapter 2: Wysteria: A Programming Language for Mixed-mode MPC

This chapter presents Wysteria, a new MPC DSL that provides high-level abstractions to write mixed-mode secure computation programs. Wysteria provides a conceptual single-threaded semantics that can be used to reason about the correctness and security properties of the MPC programs, while the Wysteria metatheory guarantees that the reasoning also applies to the actual protocol runs. Whereas Wysteria allows for informal, lightweight reasoning about the MPC programs, in Chapter 3, we make such reasoning *formal* and mechanize the Wysteria metatheory.

We have used Wysteria to program several MPC applications from the literature, as well as some novel ones, including a card dealing application (Section 2.7.3). In our experience, we have found Wysteria abstractions to be expressive and easy-to-use.

We first present an overview of Wysteria features with the help of several examples (2.1), followed by the formal specification of the Wysteria syntax, type system, and operational semantics (2.2, 2.3, 2.4). We then present the Wysteria metatheoretic properties (2.5), including progress and preservation theorems (Theorem 1 and Theorem 2) and a forward-simulation theorem that establishes the correspondence between the conceptual single-threaded semantics and the actual protocol semantics (Theorem 3). We conclude the chapter with Wysteria implementation (2.6) and evaluation (2.7).

2.1 Wysteria overview

Wysteria is a functional programming language with 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. The main novelty of Wysteria is that it allows the programmers to write a distributed MPC as a *single* program with a conceptual single-threaded semantics, while the runtime takes care of the actual distributed semantics.

2.1.1 Computation modes for secure and local computations

Wysteria defines two computation modes: *secure mode* in which secure (multi-party) 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 *only*, while the par({Bob}) an-

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

notation indicates that the second read() will be executed at Bob's location *only*. 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. WYSTERIA 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*). 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.

2.1.2 Wires for inputs and outputs

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 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* [42, 43] 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.

2.1.3 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 $\rightarrow$ W {Bob}nat $-$sec({Alice,Bob})$\rightarrow$ 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)). 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} $\cup$w$_1$) nat, where w$_1$ may or may not be empty, then v[A] is only allowed in par({A}) mode or in sec({A} $\cup$w$_2$) mode (s.t. v is accessible to sec({A} $\cup$w$_2$), and nowhere else.

2.1.4 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:

```
richest_of = λms:ps. λv: W ms nat.
               let out =sec(ms)=
                 wfold(None, v,
                       λrichest. λp. λn. match richest with
                           | None   ⇒ Some p
                           | Some q ⇒ if n > v[q] then Some p
                                      else Some q)
               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. In the None case of the match, no principal has yet been considered so the first becomes a candidate to be richest. Otherwise, in the Some case, 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 $t\{\phi\}$ 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.

2.1.5 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.

Figure 2.1 shows 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:

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 MPC languages that expose language primitives for secret sharing (e.g., [34]), in Wysteria the type system ensures that shares are not misused, e.g., shares for different underlying values may not be combined.

```
(* 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
```

Figure 2.1: Two round bidding game in WYSTERIA

2.2 Formal syntax

In this section we introduce λ_{Wy}, the formal core calculus that underpins the language design of WYSTERIA.

Figure 2.2 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 $\mathbf{inj}_i\ v$ (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 **ps** ϕ, where ϕ is a *type refinement*; the type system ensures that if a value w has type **ps** ϕ,

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

Principal	$p,\ q$	$::=$	**Alice** | **Bob** | **Charlie** | $\cdots$
Value	$v,\ w$	$::=$	$x \mid n \mid \mathbf{inj}_i\ v \mid (v_1, v_2) \mid p \mid \{w\} \mid w_1 \cup w_2$
Expression	e	$::=$	$v_1 \oplus v_2 \mid \mathbf{case}\,(v, x_1.e_1, x_2.e_2) \mid \mathbf{fst}\,(v) \mid \mathbf{snd}\,(v) \mid \lambda x.e \mid v_1\ v_2 \mid \mathbf{fix}\ x.\lambda y.e$
			$\mid \mathbf{array}(v_1, v_2) \mid \mathbf{select}(v_1, v_2) \mid \mathbf{update}(v_1, v_2, v_3) \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$
			$\mid \mathbf{let}\ x \overset{M}{=} e_1\ \mathbf{in}\ e_2 \mid \mathbf{wire}_w(v) \mid v_1 +\!\!+ v_2 \mid v[w] \mid \mathbf{wfold}_w(v_1, v_2, v_3)$
			$\mid \mathbf{wapp}_w(v_1, v_2) \mid \mathbf{waps}_w(v_1, v_2) \mid \mathbf{wcopy}_w(v) \mid \mathbf{makesh}(v) \mid \mathbf{combsh}(v)$
			$\mid v$

Type environment	Γ	$::=$	$. \mid \Gamma, x :_M \tau \mid \Gamma, x : \tau$
Mode	$M,\ N$	$::=$	$m(w) \mid \top$
Modal operator	m	$::=$	$\mathsf{p} \mid \mathsf{s}$
Effect	ϵ	$::=$	$\cdot \mid M \mid \epsilon_1, \epsilon_2$

Refinement	ϕ	$::=$	$\mathsf{true} \mid \mathbf{singl}(\nu) \mid \nu \subseteq w \mid \nu = w \mid \phi_1 \wedge \phi_2$
Type	τ	$::=$	$\mathbf{nat} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mathbf{ps}\ \phi \mid \mathbf{W}\ w\ \tau \mid \mathbf{Array}\ \tau \mid \mathbf{Sh}\ w\ \tau \mid x{:}\tau_1 \xrightarrow{\epsilon} \tau_2$

Figure 2.2: λ_{Wy} syntax

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 **singl**(ν) indicates that the principal set is a singleton.

λ_{Wy} expressions e are, for simplicity, in so-called *A-normal form* (ANF) [44], 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: **array**(v_1, v_2) creates an array (of type **Array** τ) whose length is v_1, and whose elements (of type τ) are each initialized to v_2; array accesses are written as **select**(v_1, v_2) where v_1 is an array and v_2 is an index; and array updates are written as **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

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

inside the function body. An effect is either empty, a mode, or a list of effects.

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

2.3 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.

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

T-VAR

$$\frac{x :_M \tau \in \Gamma \vee x : \tau \in \Gamma}{\Gamma \vdash \tau}$$
$$\Gamma \vdash_M x : \tau$$

T-NAT

$$\Gamma \vdash_M n : \mathbf{nat}$$

T-INJ

$$\frac{\Gamma \vdash_M v : \tau_i \qquad j \in \{1, 2\} \qquad \tau_j \ \texttt{IsFlat} \qquad \Gamma \vdash \tau_j}{\Gamma \vdash_M \mathbf{inj}_i \, v : \tau_1 + \tau_2}$$

T-PROD

$$\frac{\Gamma \vdash_M v_i : \tau_i}{\Gamma \vdash_M (v_1, v_2) : \tau_1 \times \tau_2}$$

T-PRINC

$$\Gamma \vdash_M p : \mathbf{ps}\ (\nu = \{p\})$$

T-PSONE

$$\frac{\Gamma \vdash_M w : \mathbf{ps}\ (\mathbf{singl}(\nu))}{\Gamma \vdash_M \{w\} : \mathbf{ps}\ (\nu = \{w\})}$$

T-PSUNION

$$\frac{\Gamma \vdash_M w_i : \mathbf{ps}\ \phi_i}{\Gamma \vdash_M w_1 \cup w_2 : \mathbf{ps}\ (\nu = w_1 \cup w_2)}$$

T-PSVAR

$$\frac{\Gamma \vdash_M x : \mathbf{ps}\ \phi}{\Gamma \vdash_M x : \mathbf{ps}\ (\nu = x)}$$

T-MSUB

$$\frac{\Gamma \vdash M \qquad \Gamma \vdash_M x : \tau \qquad \Gamma \vdash M \rhd N \qquad N = \mathsf{s}(\_) \Rightarrow \tau \ \texttt{IsSecIn}}{\Gamma \vdash_N x : \tau}$$

T-SUB

$$\frac{\Gamma \vdash_M v : \tau_1 \qquad \Gamma \vdash \tau_1 <: \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash_M v : \tau}$$

Figure 2.3: Value typing judgment

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

D-REFL

$$\frac{\Gamma \vdash w_2 : \mathbf{ps}\ (\nu = w_1)}{\Gamma \vdash m(w_1) \rhd m(w_2)}$$

D-TOP

$$\frac{\Gamma \vdash w : \mathbf{ps}\ \phi}{\Gamma \vdash \top \rhd m(w)}$$

D-PAR

$$\frac{\Gamma \vdash w_2 : \mathbf{ps}\ (\nu \subseteq w_1)}{\Gamma \vdash \mathsf{p}(w_1) \rhd \mathsf{p}(w_2)}$$

D-SEC

$$\frac{\Gamma \vdash w_2 : \mathbf{ps}\ (\nu = w_1)}{\Gamma \vdash \mathsf{p}(w_1) \rhd \mathsf{s}(w_2)}$$

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

S-REFL

$$\frac{}{\Gamma \vdash \tau <: \tau}$$

S-TRANS

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3}$$

S-SUM

$$\frac{\Gamma \vdash \tau_i <: \tau_i'}{\Gamma \vdash \tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$

S-PROD

$$\frac{\Gamma \vdash \tau_i <: \tau_i'}{\Gamma \vdash \tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}$$

S-PRINCS

$$\frac{[\![\Gamma]\!] \vDash \phi_1 \Rightarrow \phi_2}{\Gamma \vdash \mathbf{ps}\ \phi_1 <: \mathbf{ps}\ \phi_2}$$

S-WIRE

$$\frac{\Gamma \vdash w_2 : \mathbf{ps}\ (\nu \subseteq w_1) \qquad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash \mathbf{W}\ w_1\ \tau_1 <: \mathbf{W}\ w_2\ \tau_2}$$

S-ARRAY

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash \mathbf{Array}\ \tau_1 <: \mathbf{Array}\ \tau_2}$$

S-SHARE

$$\frac{\Gamma \vdash w_2 : \mathbf{ps}\ (\nu = w_1) \qquad \Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash \mathbf{Sh}\ w_1\ \tau_1 <: \mathbf{Sh}\ w_2\ \tau_2}$$

S-ARROW

$$\frac{\Gamma \vdash \tau_1' <: \tau_1 \qquad \Gamma, x : \tau_1' \vdash \tau_2 <: \tau_2'}{\Gamma \vdash x{:}\tau_1 \xrightarrow{\epsilon} \tau_2 <: x{:}\tau_1' \xrightarrow{\epsilon} \tau_2'}$$

Figure 2.4: Subtyping and delegation judgments

2.3.1 Value typing

Figure 2.3 shows the value typing judgment $\Gamma \vdash_M v : \tau$, 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 judgment for type well-formedness, $\Gamma \vdash \tau$, which enforces invariants like, for a wire bundle type $\mathbf{W}\, w\, \tau$, w should have a $\mathbf{ps}\ \phi$ type. The rule T-INJ uses another auxiliary judgment τ `IsFlat` which holds for types τ that lack wire bundles and shares. Both auxiliary judgments are defined in the Appendix A (Figures A.3, A.4). 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.

2.3.2 Delegations typing judgments

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 $N = \mathtt{s}(\_) \Rightarrow \tau$ `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 2.4. 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.

2.3.3 Subtyping judgments

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 judgment $\Gamma \vdash \tau_1 <: \tau_2$ is read as *under* Γ *type* τ_1 *is a subtype of* τ_2. The rules for this judgment are given at the bottom of Figure 2.4. 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 judgment

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 judgment, as it can be realized with an SMT solver; our implementation uses Z3 [40], as described in 2.6. For wire bundles, the domain of the supertype, a principal set, must be a subset of domain of the subtype, i.e. type $\mathbf{W}\ w_1\ \tau_1$ is a subtype of $\mathbf{W}\ w_2\ \tau_2$ 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.

2.3.4 Expression typing

Figures 2.5 gives the typing judgment 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 $(M = \mathsf{p}(\_) \wedge \epsilon = M) \vee (\tau\ \texttt{IsFO} \wedge \epsilon = \cdot)$ 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).

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

T-BINOP

$$\frac{\Gamma \vdash_M v_i : \textbf{nat}}{\Gamma \vdash_M v_1 \oplus v_2 : \textbf{nat}; \cdot}$$

T-FST

$$\frac{\Gamma \vdash_M v : \tau_1 \times \tau_2}{\Gamma \vdash_M \textbf{fst}\,(v) : \tau_1; \cdot}$$

T-SND

$$\frac{\Gamma \vdash_M v : \tau_1 \times \tau_2}{\Gamma \vdash_M \textbf{snd}\,(v) : \tau_2; \cdot}$$

T-CASE

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

T-LAM

$$\frac{\begin{array}{c}\Gamma \vdash \tau \\ \Gamma, x : \tau \vdash_M e : \tau_1; \epsilon \end{array}}{\Gamma \vdash_M \lambda x.e : (x{:}\tau \xrightarrow{\epsilon} \tau_1); \cdot}$$

T-APP

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

T-LET1

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

T-LET2

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

T-FIX

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

Figure 2.5: Expression typing judgments

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 ($M = \mathsf{s}(\_) \Rightarrow \tau$ `IsFO`) to prevent a need to represent functional values as circuits; our implementation of secure blocks inlines all function applications before generating circuits.

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 ($\Gamma \vdash M \rhd \epsilon$).

[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 Appendix A Figure A.1). This judgment can only access those variables in Γ that are bound without any mode.

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

T-ARRAY

$$\frac{M = \mathsf{p}(\_) \qquad \Gamma \vdash_M v_1 : \textbf{nat} \qquad \Gamma \vdash_M v_2 : \tau}{\Gamma \vdash_M \textbf{array}(v_1, v_2) : \textbf{Array}\,\tau; \cdot}$$

T-SELECT

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

T-UPDATE

$$\frac{M = \mathsf{p}(\_) \qquad \mathsf{mode}(v_1, \Gamma) = M \qquad \Gamma \vdash_M v_1 : \textbf{Array}\,\tau \qquad \Gamma \vdash_M v_2 : \textbf{nat} \qquad \Gamma \vdash_M v_3 : \tau}{\Gamma \vdash_M \textbf{update}(v_1, v_2, v_3) : \textbf{unit}; \cdot}$$

Figure 2.6: Expression typing judgments for arrays

It also adds an effect $\mathsf{p}(\cdot)$ to the final type, so that secure blocks cannot use such loops defined in parallel blocks.

Figure 2.6 shows the typing judgments for arrays. 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).

Figure 2.7 shows the typing judgments for wires. 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 $\Gamma \vdash w_1 : \textbf{ps}\ (\nu \subseteq w_2)$ requires that $w_1 \subseteq w_2$, i.e., all principals contributing

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

T-WIRE

$$\frac{\Gamma \vdash w_1 : \mathbf{ps}\ (\nu \subseteq w_2) \quad m = \mathsf{s} \Rightarrow N = \mathsf{s}(w_2) \quad m = \mathsf{p} \Rightarrow N = \mathsf{p}(w_1) \quad \Gamma \vdash_N v : \tau \quad m = \mathsf{s} \Rightarrow \tau\ \texttt{IsFO} \quad \tau\ \texttt{IsFlat}}{\Gamma \vdash_{m(w_2)} \mathbf{wire}_{w_1}(v) : \mathbf{W}\, w_1\, \tau; \cdot}$$

T-WPROJ

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

T-WIREUN

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

T-WFOLD

$$\frac{M = \mathsf{s}(\_) \quad \tau_2\ \texttt{IsFO} \quad \phi = (\nu \subseteq w \wedge \mathbf{singl}(\nu)) \quad \Gamma \vdash_M v_1 : \mathbf{W}\, w\, \tau \quad \Gamma \vdash_M v_2 : \tau_2 \quad \Gamma \vdash_M v_3 : \tau_2 \dot{\to} \mathbf{ps}\ \phi \dot{\to} \tau \dot{\to} \tau_2}{\Gamma \vdash_M \mathbf{wfold}_w(v_1, v_2, v_3) : \tau_2; \cdot}$$

T-WAPP

$$\frac{M = \mathsf{p}(\_) \quad \Gamma \vdash_M v_1 : \mathbf{W}\, w\, \tau_1 \quad \Gamma \vdash_M v_2 : \mathbf{W}\, w\, (\tau_1 \dot{\to} \tau_2)}{\Gamma \vdash_M \mathbf{wapp}_w(v_1, v_2) : \mathbf{W}\, w\, \tau_2; \cdot}$$

T-WAPS

$$\frac{M = \mathsf{s}(\_) \quad \tau_2\ \texttt{IsFO} \quad \tau_2\ \texttt{IsFlat} \quad \Gamma \vdash_M v_1 : \mathbf{W}\, w\, \tau_1 \quad \Gamma \vdash_M v_2 : \tau_1 \dot{\to} \tau_2}{\Gamma \vdash_M \mathbf{waps}_w(v_1, v_2) : \mathbf{W}\, w\, \tau_2; \cdot}$$

T-WCOPY

$$\frac{M = \mathsf{p}(w_1) \quad \Gamma \vdash w_2 : \mathbf{ps}\ (\nu \subseteq w_1) \quad \Gamma \vdash_{\mathsf{p}(w_2)} v : \mathbf{W}\, w_2\, \tau}{\Gamma \vdash_M \mathbf{wcopy}_{w_2}(v) : \mathbf{W}\, w_2\, \tau; \cdot}$$

Figure 2.7: Expression typing judgments for wires

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

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

T-MAKESH

$$\frac{M = \mathsf{s}(w) \qquad \tau \ \mathtt{IsFO} \qquad \tau \ \mathtt{IsFlat} \qquad \Gamma \vdash_M v : \tau}{\Gamma \vdash_M \mathbf{makesh}(v) : \mathbf{Sh}\ w\ \tau; \cdot}$$

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

$$\frac{\Gamma \vdash_M e : \tau'; \epsilon \qquad \Gamma \vdash \tau' <: \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash_M e : \tau; \epsilon}$$

Figure 2.8: Remaining rules for expression typing

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 $(\cdot)$.

Figure 2.8 shows the remaining expression typing judgments for shares and the subsumption rule. 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-

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

Figure 2.9: WYSTERIA runtime configuration syntax

order so that their bit-level representation size can be determined. Finally, rule T-SUBE is the subsumption rule for expressions.

2.4 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.

2.4.1 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). The definitions are shown in Figure 2.9.

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 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 2.2 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 $\mathbf{clos}\,(\psi; \lambda x.e)$ and $\mathbf{clos}\,(\psi;\ \mathbf{fix}\, f.\lambda x.e)$).

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.

We split the single-threaded semantics in two judgments, one that change the stack

$\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ **Configuration stepping**: "Configuration $\mathcal{C}_1$ steps to $\mathcal{C}_2$"

stpC-Local

$$M\{\sigma_1; \kappa; \psi_1; e_1\} \longrightarrow M\{\sigma_2; \kappa; \psi_2; e_2\} \text{ when } \sigma_1; \psi_1; e_1 \xrightarrow{M} \sigma_2; \psi_2; e_2$$

stpC-Let

$$M\{\sigma; \kappa; \psi; \textbf{let}\ x = e_1\ \textbf{in}\ e_2\} \longrightarrow M\{\sigma; \kappa :: \langle \psi; x.e_2 \rangle; \psi; e_1\}$$

stpC-delPar

$$\mathsf{p}(w_1 \cup w_2)\{\sigma; \kappa; \psi; \textbf{let}\ x \overset{\mathsf{p}(w')}{=} e_1\ \textbf{in}\ e_2\} \longrightarrow \mathsf{p}(w_2)\{\sigma; \kappa :: \langle \mathsf{p}(w_1 \cup w_2); \psi; x.e_2 \rangle; \psi; e_1\} \quad \text{when } \psi[\![w']\!] = w_2$$

stpC-delSSec

$$\mathsf{s}(w)\{\sigma; \kappa; \psi; \textbf{let}\ x \overset{\mathsf{s}(w')}{=} e_1\ \textbf{in}\ e_2\} \longrightarrow s(w)\{\sigma; \kappa :: \langle s(w); \psi; x.e_2 \rangle; \psi; e_1\} \quad \text{when } \psi[\![w']\!] = w$$

stpC-delPSec

$$\mathsf{p}(w)\{\sigma; \kappa; \psi; \textbf{let}\ x \overset{\mathsf{s}(w')}{=} e_1\ \textbf{in}\ e_2\} \longrightarrow \mathsf{p}(w)\{\sigma; \kappa :: \langle \mathsf{p}(w); \psi; x.e_2 \rangle; \psi; \textbf{secure}_{w'}(e_1)\}$$

stpC-secEnter

$$\mathsf{p}(w)\{\sigma; \kappa; \psi; \textbf{secure}_{w'}(e)\} \longrightarrow \mathsf{s}(w)\{\sigma; \kappa; \psi; e\} \text{ when } \psi[\![w']\!] = w$$

stpC-popStk1

$$N\{\sigma; \kappa :: \langle M; \psi_1; x.e \rangle; \psi_2; v\} \longrightarrow M\{\sigma; \kappa; \psi_1\{x \mapsto_{m(w)} (\psi_2[\![v]\!]_N)\}; e\} \quad \text{when } M = m(\_) \text{ and } N = \_(w)$$

stpC-popStk2

$$M\{\sigma; \kappa :: \langle \psi_1; x.e \rangle; \psi_2; v\} \longrightarrow M\{\sigma; \kappa; \psi_1\{x \mapsto (\psi_2[\![v]\!]_M)\}; e\}$$

Figure 2.10: Delegation semantics of single-threaded configurations

and the mode of the current configuration, and one that does not. Figure 2.10 shows the judgments for the first kind. The judgment $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ can be read as saying *configuration $\mathcal{C}_1$ steps to $\mathcal{C}_2$*. 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).

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 second judgment for single-threaded semantics, called local stepping judgment $\sigma_1; \psi_1; e_1 \xrightarrow{M} \sigma_2; \psi_2; e_2$, covers all (common) cases where neither the stack nor the mode of the configuration change. This judgment 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.

Figure 2.11 shows some cases for this judgment. We explain the first rule in detail, as a model for the rest. Case analysis (STPL-CASE) branches based on the injection, step-

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

STPL-CASE

$$\sigma; \psi; \mathbf{case}\,(v, x_1.e_1, x_2.e_2) \xrightarrow{M} \sigma; \psi\{x_i \mapsto v'\}; e_i \quad \text{when } \psi[\![v]\!]_M = \mathbf{inj}_i\, v'$$

STPL-FST

$$\sigma; \psi; \mathbf{fst}\,(v) \xrightarrow{M} \sigma; \psi; v_1 \quad \text{when } \psi[\![v]\!]_M = (v_1, v_2)$$

STPL-SND

$$\sigma; \psi; \mathbf{snd}\,(v) \xrightarrow{M} \sigma; \psi; v_2 \quad \text{when } \psi[\![v]\!]_M = (v_1, v_2)$$

STPL-BINOP

$$\sigma; \psi; v_1 \oplus v_2 \xrightarrow{M} \sigma; \psi; v' \quad \text{when } \psi[\![v_1]\!]_M = v_1', \psi[\![v_2]\!]_M = v_2' \text{ and } v_1' \oplus v_2' = v'$$

STPL-LAMBDA

$$\sigma; \psi; \lambda x.e \xrightarrow{M} \sigma; \psi; \mathbf{clos}\,(\psi; \lambda x.e)$$

STPL-APPLY

$$\sigma; \psi_1; v_1\ v_2 \xrightarrow{M} \sigma; \psi_2\{x \mapsto v'\}; e \quad \text{when } \psi_1[\![v_1]\!]_M = \mathbf{clos}\,(\psi_2; \lambda x.e) \text{ and } \psi_1[\![v_2]\!] = v'$$

STPL-FIX

$$\sigma; \psi; \mathbf{fix}\ x.\lambda y.e \xrightarrow{\mathsf{p}(w)} \sigma; \psi; \mathbf{clos}\,(\psi;\ \mathbf{fix}\ x.\lambda y.e)$$

STPL-FIXAPPLY

$$\sigma; \psi_1; v_1\ v_2 \xrightarrow{M} \sigma; \psi'; e \quad \text{when } \psi_1[\![v_1]\!]_M = \mathbf{clos}\,(\psi;\ \mathbf{fix}\ x.\lambda y.e), \psi_1[\![v_2]\!] = v' \text{ and } \psi' = \psi\{x \mapsto \mathbf{clos}\,(\psi;\ \mathbf{fix}\ x.\lambda y.e); y \mapsto v'\}$$

Figure 2.11: Local stepping of single-threaded configurations

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

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

Figure 2.12: Environment lookup judgments (selected rules)

ping 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 2.12 (complete rules are in the Appendix A Figure A.5). 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 (STPL-LAMBDA and STPL-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.

Figure 2.13 shows the local stepping judgment for arrays and shares. The array primitives access or update the store, which remained invariant in all the cases above. Array

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

STPL-ARRAY

$$\sigma; \psi; \mathbf{array}(v_1, v_2) \xrightarrow{M} \sigma\{\ell :_M w^k\}; \psi; \ell \quad \text{when } \psi[\![v_1]\!]_M = k,\ \psi[\![v_2]\!]_M = w \text{ and } \mathsf{next}_M(\sigma) = \ell$$

STPL-SELECT

$$\sigma; \psi; \mathbf{select}(v_1, v_2) \xrightarrow{M} \sigma; \psi; w_i \quad \text{when } \psi[\![v_1]\!]_M = \ell, \psi[\![v_2]\!]_M = i, i \in [1..k] \text{ and } \sigma(\ell) = \{\bar{w}\}_k$$

STPL-SEL-ERR

$$\sigma; \psi; \mathbf{select}(v_1, v_2) \xrightarrow{M} \sigma; \psi; \mathsf{error} \quad \text{when } \psi[\![v_1]\!]_M = \ell, \psi[\![v_2]\!]_M = i, i \notin [1..k] \text{ and } \sigma(\ell) = \{\bar{w}\}_k$$

STPL-UPDATE

$$\sigma; \psi; \mathbf{update}(v_1, v_2, v_3) \xrightarrow{M} \sigma'; \psi; () \quad \text{when } \psi[\![v_1]\!]_M = \ell, \psi[\![v_2]\!]_M = i, \psi[\![v_3]\!]_M = w'_i \ \sigma(\ell) = \{\bar{w}\}_k, j \in [1..k], w'_j = w_j \text{ for } j \neq i \text{ and } \sigma' = \sigma\{\ell :_M \{\bar{w'}\}_k\}$$

STPL-UPD-ERR

$$\sigma; \psi; \mathbf{update}(v_1, v_2, v_3) \xrightarrow{M} \sigma; \psi; \mathsf{error} \quad \text{when } \psi[\![v_1]\!]_M = \ell, \psi[\![v_2]\!]_M = i, i \notin [1..k] \text{ and } \sigma(\ell) = \{\bar{w}\}_k$$

STPL-MAKESH

$$\sigma; \psi; \mathbf{makesh}(v) \xrightarrow{\mathsf{s}(w)} \sigma; \psi; \mathbf{sh}\ w\ v' \quad \text{when } \psi[\![v]\!]_{\mathsf{s}(w)} = v'$$

STPL-COMBSH

$$\sigma; \psi; \mathbf{combsh}(v) \xrightarrow{\mathsf{s}(w)} \sigma; \psi; v' \quad \text{when } \psi[\![v]\!]_{\mathsf{s}(w)} = \mathbf{sh}\ w\ v'$$

Figure 2.13: Local stepping of single-threaded configurations: arrays and shares

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. The fresh location is chosen (deterministically) by the auxiliary function $\mathsf{next}_M(\sigma)$. 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 **sh** w v' associated with th 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. Figure 2.14 shows the wire bundle creation and projection. 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

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

stpL-Wire

$$\sigma; \psi; \mathbf{wire}_w(v) \xrightarrow{M} \sigma; \psi; \{\!|(\psi[\![v]\!]_N)|\!\}^{\mathsf{wires}}_{w'} \quad \text{where } \{\!|v|\!\}^{\mathsf{wires}}_{w_1 \cup w_2} = \{\!|v|\!\}^{\mathsf{wires}}_{w_1} +\!\!+ \{\!|v|\!\}^{\mathsf{wires}}_{w_2}$$

stpL-Wcopy

$$\sigma; \psi; \mathbf{wcopy}_w(v) \xrightarrow{M} \sigma; \psi; v \quad \text{and } \{\!|v|\!\}^{\mathsf{wires}}_{\{p\}} = \{p : v\}, \{\!|v|\!\}^{\mathsf{wires}}_{\cdot} = \cdot,$$
$$\psi[\![w]\!] = w', M = m(\_),$$
$$m = \mathsf{s} \Rightarrow N = M \text{ and } m = \mathsf{p} \Rightarrow N = \mathsf{p}(w')$$

stpL-parProj

$$\sigma; \psi; v_1[v_2] \xrightarrow{\mathsf{p}(\{p\})} \sigma; \psi; v' \quad \text{when } \psi[\![v_2]\!]_{\mathsf{p}(\{p\})} = p \text{ and}$$
$$\psi[\![v_1]\!]_{\mathsf{p}(\{p\})} = \{p : v'\} +\!\!+ w'$$

stpL-secProj

$$\sigma; \psi; v_1[v_2] \xrightarrow{\mathsf{s}(\{p\} \cup w)} \sigma; \psi; v' \quad \text{when } \psi[\![v_2]\!]_{\mathsf{s}(\{p\} \cup w)} = p \text{ and}$$
$$\psi[\![v_1]\!]_{\mathsf{s}(\{p\} \cup w)} = \{p : v'\} +\!\!+ w'$$

stpL-WireUn

$$\sigma; \psi; v_1 +\!\!+ v_2 \xrightarrow{M} \sigma; \psi; v_1' +\!\!+ v_2' \quad \text{when } \psi[\![v_1]\!]_M = v_1' \text{ and } \psi[\![v_2]\!]_M = v_2'$$

Figure 2.14: Local stepping of single-threaded configurations: wires

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

STPL-WAPP1

$$\sigma; \psi; \mathbf{wapp}_w(v_1, v_2) \xrightarrow{M} \sigma; \psi; \cdot \quad \text{when } \psi[\![w]\!] = \cdot$$

STPL-WAPP2

$$\sigma; \psi; \mathbf{wapp}_w(v_1, v_2) \xrightarrow{M} \sigma; \psi; e \quad \text{when } \psi[\![w]\!] = \{p\} \cup w', M = \mathsf{p}((\{p\} \cup w') \cup w_1),$$
$$\psi[\![v_1]\!]_M = v_1' \text{ and } \psi[\![v_2]\!]_M = v_2'$$
$$\text{where } e = \mathbf{let}\ z_1 \overset{\mathsf{p}(\{p\})}{=} \mathbf{let}\ z_2 = v_1'[p]\ \mathbf{in}\ \mathbf{let}\ z_3 = v_2'[p]\ \mathbf{in}\ z_2\ z_3\ \mathbf{in}$$
$$\mathbf{let}\ z_4 = \mathbf{wapp}_{w'}(v_1', v_2')\ \mathbf{in}\ ((\mathbf{wire}_{\{p\}}(z_1)) +\!\!+ z_4)$$

Figure 2.15: Local stepping of single-threaded configurations: **wapp**

is present and alone. STPL-SECPROJ projects from wire bundles in a secure mode when the projected principal is present (alone or not).

We now turn to the wire combinators. 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. 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).

In Figure 2.15, rule STPL-WAPP1 reduces to the empty wire bundle (base case), and 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 unfold-

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

STPL-WAPS1

$$\sigma; \psi; \mathbf{waps}_w(v_1, v_2) \xrightarrow{M} \sigma; \psi; \cdot \quad \text{when } \psi[\![w]\!] = \cdot$$

STPL-WAPS2

$$\sigma; \psi; \mathbf{waps}_w(v_1, v_2) \xrightarrow{M} \sigma; \psi; e \quad \text{when } \psi[\![w]\!] = \{p\} \cup w', M = \mathsf{s}((\{p\} \cup w') \cup w_1),$$
$$\psi[\![v_1]\!]_M = v_1' \text{ and } \psi[\![v_2]\!]_M = v_2'$$
$$\text{where } e = \mathbf{let}\ z_1 = v_1'[p]\ \mathbf{in\ let}\ z_2 = v_2'\ z_1\ \mathbf{in\ let}\ z_3 = \mathbf{waps}_{w'}(v_1', v_2')\ \mathbf{in}$$
$$((\mathbf{wire}_{\{p\}}(z_2)) +\!\!+ z_3)$$

Figure 2.16: Local stepping of single-threaded configurations: **waps**

ing concatenates the resulting wire bundle for the other principals with that of p.

In Figure 2.16, rule STPL-WAPS1 reduces to the empty wire bundle (base case), while 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, Figure 2.17 shows the rules for the last wire combinator. Rule STPL-WFOLD1 reduces to the accumulator value v_2 (base case), while 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.

2.4.2 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

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

STPL-WFOLD1

$$\sigma; \psi; \mathbf{wfold}_w(v_1, v_2, v_3) \xrightarrow{M} \sigma; \psi; v' \quad \text{when } \psi[\![w]\!] = \cdot \text{ and } \psi[\![v_2]\!]_M = v'$$

STPL-WFOLD2

$$\sigma; \psi; \mathbf{wfold}_w(v_1, v_2, v_3) \xrightarrow{M} \sigma; \psi; e \quad \text{when } \psi[\![w]\!] = \{p\} \cup w', M = \mathsf{s}((\{p\} \cup w') \cup w_1),$$
$$\psi[\![v_1]\!]_M = v_1', \psi[\![v_2]\!]_M = v_2' \text{ and } \psi[\![v_3]\!]_M = v_3'$$
$$\text{where } e = \mathbf{let}\ z_1 = v_1'[p]\ \mathbf{in\ let}\ z_2 = v_3'\ v_2'\ p\ z_1\ \mathbf{in\ wfold}_{w'}(v_1', z_2, v_3')$$

Figure 2.17: Local stepping of single-threaded configurations: **wfold**

Protocol	π	$::=$	$\varepsilon \mid \pi_1 \cdot \pi_2 \mid A$
Agent	A	$::=$	$p\,\{\sigma; \kappa; \psi; e\} \mid \mathsf{s}(^{w_1}_{w_2})\,\{\sigma; \kappa; \psi; e\}$

Figure 2.18: WYSTERIA protocol syntax

principals. We make this multi-agent view apparent in a multi-threaded semantics of λ_{Wy} which defines the notion of a *protocol* (Figure 2.18).

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 $\mathsf{s}(^{w_1}_{w_2})\,\{\sigma; \kappa; \psi; e\}$ 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.

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

STPP-PRIVATE

$$p\,\{\sigma_1;\kappa_1;\psi_1;e_1\} \longrightarrow p\,\{\sigma_2;\kappa_2;\psi_2;e_2\}$$
$$\text{when } \mathsf{p}(\{p\})\{\sigma_1;\kappa_1;\psi_1;e_1\} \longrightarrow \mathsf{p}(\{p\})\{\sigma_2;\kappa_2;\psi_2;e_2\}$$

STPP-PRESENT

$$p\left\{\sigma;\kappa;\psi;\textbf{let } x \overset{\mathsf{p}(w)}{=} e_1 \textbf{ in } e_2\right\} \longrightarrow p\,\{\sigma;\kappa_1;\psi;e_1\}$$
$$\text{when } \{p\} \subseteq \psi[\![w]\!] \text{ and } \kappa_1 = \kappa :: \langle \mathsf{p}(\{p\});\psi;x.e_2\rangle$$

STPP-ABSENT

$$p\left\{\sigma;\kappa;\psi;\textbf{let } x \overset{m(w)}{=} e_1 \textbf{ in } e_2\right\} \longrightarrow p\,\{\sigma;\kappa;\psi;e_2\} \quad \text{when } \{p\} \not\subseteq \psi[\![w]\!]$$

STPP-FRAME

$$\pi_1 \cdot \pi_2 \longrightarrow \pi_1' \cdot \pi_2 \quad \text{when } \pi_1 \longrightarrow \pi_1'$$

STPP-SECSTEP

$$\mathsf{s}\binom{w}{w}\{\sigma_1;\kappa_1;\psi_1;e_1\} \longrightarrow \mathsf{s}\binom{w}{w}\{\sigma_2;\kappa_2;\psi_2;e_2\}$$
$$\text{when } \mathsf{s}(w)\{\sigma_1;\kappa_1;\psi_1;e_1\} \longrightarrow \mathsf{s}(w)\{\sigma_2;\kappa_2;\psi_2;e_2\}$$

STPP-SECBEGIN

$$\varepsilon \longrightarrow \mathsf{s}\binom{\cdot}{w}\{\cdot;\cdot;\cdot;e\}$$

STPP-SECEND

$$\mathsf{s}\binom{\cdot}{w_2}\{\sigma;\cdot;\psi;v\} \longrightarrow \varepsilon$$

STPP-SECENTER

$$\mathsf{s}\binom{w_1}{w_2}\{\sigma;\cdot;\psi;e\} \cdot p\,\{\sigma';\kappa;\psi';\textbf{secure}_{w_2}(e)\} \longrightarrow \mathsf{s}\binom{w_1\cup\{p\}}{w_2}\{\sigma\circ\sigma';\cdot;\psi\circ\psi';e\} \cdot p\,\{\sigma';\kappa;\cdot;\textbf{wait}\}$$

STPP-SECLEAVE

$$\mathsf{s}\binom{w_1\cup\{p\}}{w_2}\{\sigma';\cdot;\psi;v\} \cdot p\,\{\sigma;\kappa;\cdot;\textbf{wait}\} \longrightarrow \mathsf{s}\binom{w_1}{w_2}\{\sigma';\cdot;\psi;v\} \cdot p\,\{\sigma;\kappa;\cdot;v'\}$$
$$\text{when } \mathsf{slice}_p(\psi[\![v]\!]_{\mathsf{s}(w_2)}) \rightsquigarrow v'$$

Figure 2.19: Multi-threaded target protocol semantics

Figure 2.19 defines the stepping judgment 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 (non-deterministically) 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 party receives the *slice* of the closed value that is relevant to him. We show selected slicing rules in Figure 2.20. The

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

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

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

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

Figure 2.20: Slicing judgments (selected rules)

complete definition is in the Appendix A Figures A.7 and A.9. 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 Appendix A Figure A.10).

Returning to our example, we can see that **Bob** and **Alice** will step to $\mathbf{secure}_{\{\mathbf{Alice},\mathbf{Bob}\}}(e')$ (with the result poised to be bound to x_4 in both cases) where e' is $x_3[\mathbf{Alice}] > x_3[\mathbf{Bob}]$. 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 $\{\mathbf{Alice} : 6\} +\!\!+ \{\mathbf{Bob} : 5\}$. 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.

2.5 Metatheory

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 Appendix B. Anticipating on Chapter 3, we have mechanized the proofs of Theorem 3 and Theorem 4 in F* [37].

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 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 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, of which we give the full formal definitions in Appendix A Figure A.12.

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 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 judgment to construct a multi-threaded protocol from a (single-threaded) configuration (Appendix A Figure A.8).

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 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. We formalize this corollary (and mechanize its proof) in Chapter 3.

One of the most important consequences of these theorems is that principals running parallel to one another, and whose computations successfully 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.

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

Lemma 5 (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* $\mathsf{slice}_w(\mathcal{C}') \rightsquigarrow \pi'$*.*

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 2.19, 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. In Chapter 3, we enhance the Wysteria API (and semantics) with *observable traces* that can be used to formally reason about the security properties.

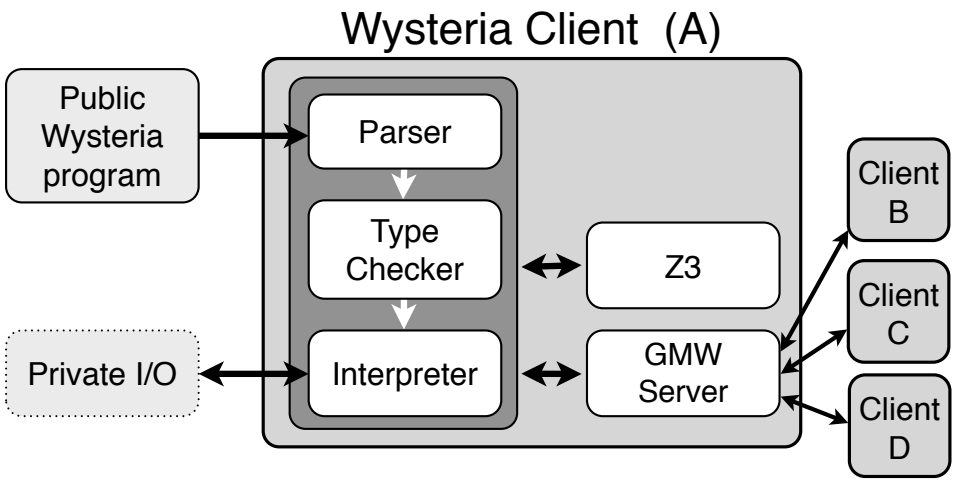

Figure 2.21: Overview of WYSTERIA system with four interacting clients.

2.6 Implementation

We have implemented a tool chain for WYSTERIA, including a front-end, 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 [11] of the Goldreich, Micali, and Wigderson (GMW) protocol [2], which provably simulates (in the semi-honest setting) a trusted third party. Figure 2.21 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.

2.6.1 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 [40] 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.

2.6.2 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.

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).

2.6.3 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.

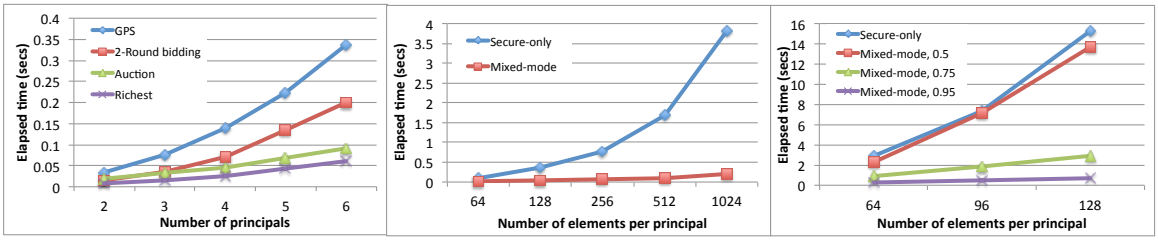

Figure 2.22: (a) n-party MPC examples. (b) Secure median vs mixed-mode median. (c) Secure PSI vs mixed-mode PSI for different density.

We also optimized the library's Oblivious Transfer (OT) extension implementation for the mixed-mode setting.

2.7 Evaluation

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] Finally, we use Wysteria to program an MPC card dealing application.

2.7.1 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

[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.

Richest protocol computes the richest principal, as described in Section 2.1. 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. We have also implemented the two-round bidding game from Section 2.1 for multiple principals. Recall that this example crucially relies on Wysteria's notion of secret shares, a high-level abstraction that existing MPC languages lack.

Figure 2.22(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.

2.7.2 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, 33]; 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

```
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
```

Figure 2.23: Monolithic median example in WYSTERIA

privacy guarantees.

Mixed-mode median. Figure 2.23 shows a simplified version of median that accepts two numbers from each party.

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 [33], the mixed-mode version from Figure 2.24 equivalently computes median with the same security properties.

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. As we show in Chapter 4, this change still preserves the final *knowledge profile* of each party, and is thus equally secure in the semi-honest setting [33].

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

Figure 2.24: Mixed-mode median example in WYSTERIA

Figure 2.22(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.

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 2.22(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).

2.7.3 MPC program for card dealing

We have implemented a generic n-party card dealing MPC application using Wysteria. In the application, each party maintains secret shares of already dealt cards in an array. Secret shares ensure that no single-party can make sense of the cards on their own, yet, the parties can combine their secret shares and recover the dealt cards in secure computations. To deal a new card, parties enter a secure computation with a random number as each party's input. In the computation, the parties compute the sum of their random numbers modulo 52. The computation returns secret shares of this value, which is the potential new card, to each party. To ensure the freshness of the new card, parties enter a loop where in the i-th loop iteration they perform a secure computation to check that the previously dealt i-th card is different from the new card. If the new card is indeed unique,

it is returned in clear to the party it was dealt for (and every party stores secret shares of this card), if the card is not unique, they repeat the algorithm.

To gain more experience with the usability of WYSTERIA, we also built a user interface for the card dealing application. Since WYSTERIA is a standalone DSL with no GUI libraries etc., we had to use some other language for GUI programming. We chose the Racket language [45] for this purpose. We designed the WYSTERIA card dealing program and the Racket GUI to run as separate processes communicating using UNIX pipes. We first added a sysop functionality to WYSTERIA for invoking generic shell commands (using the UNIX command line). When WYSTERIA application is started, it starts the Racket GUI as another process, and sends it the local principal's name. It then waits for a message from the Racket GUI indicating that the user has initiated the card dealing. The Racket GUI sends this message to the WYSTERIA application when the user presses a *Deal* button on the GUI. Once every user has pressed the button on their local ends, their corresponding WYSTERIA program instances then deal five cards each using MPC, and send their own cards to the local Racket GUI process, which then renders the cards.[6]

In building this application, it became clear to us that to make WYSTERIA more usable, we needed to embed it in a rich host language, so that the application code other than the core MPC part (GUI etc.) can be programmed in the host language, and WYSTERIA code can be more seamlessly integrated with it. The next chapter presents such an embedding of WYSTERIA in a more general-purpose programming language.

[6]The Racket code for this application was programmed by Rebecca MacKenzie, an undergraduate visiting student at the time.

2.8 Concluding remarks

This chapter has presented WYSTERIA, a new MPC DSL for writing mixed-mode secure multi-party computations. WYSTERIA provides high-level abstractions with a conceptual single-threaded semantics that can be used to reason about the correctness and security properties of the WYSTERIA programs.

We formalized the WYSTERIA syntax and type system. We also formalized two operational semantics for WYSTERIA, a single-threaded specification semantics and an actual multi-threaded protocol semantics. We proved several theorems about WYSTERIA, particularly a type soundness theorem and a forward simulation theorem that establishes the correspondence between the two WYSTERIA semantics.

We presented an implementation of WYSTERIA, and its evaluation on several benchmarks, including a novel card dealing MPC application. Our experiments show that WYSTERIA abstractions are high-level, easy-to-use, and enable writing rich applications.

In the next chapter, we enhance WYSTERIA capabilities to (a) enable *formal* reasoning about the correctness and security properties of MPC programs, (b) provide a partially verified toolchain, and (c) include more language features such as datatypes and libraries for I/O, GUIs, etc.

Chapter 3: Wys*: A Verified Language Extension for Mixed-mode MPC

Wysteria significantly advances the state-of-the-art of the MPC DSLs. With its formalized conceptual single-threaded semantics, it enables the programmers to write and reason about MPCs like a single-threaded program, while the Wysteria metatheory guarantees that the reasoning also applies to the actual protocol runs. Yet, Wysteria has its limitations. Since an MPC program computes on parties' sensitive, private data, it is desirable for the parties to be able to *formally* reason that the program computes the correct function and is sufficiently privacy preserving, i.e. it does not reveal more than the intended information about their inputs. Wysteria provides only lightweight, informal reasoning capabilities. Furthermore, reasoning about the source MPC programs *only* is not enough. Ultimately, the programs are executed by the MPC toolchain, and security bugs in the toolchain can compromise the privacy of parties' inputs. Formal verification [24–31] significantly reduces the occurrence of correctness and security bugs in the software. Still, the Wysteria toolchain is not verified. Finally, as we pointed out at the end of Chapter 2, Wysteria, being a standalone DSL, lacks the elements of a full-featured programming language such as datatypes, GUI libraries and so on. We also note that none of the existing MPC toolchains provide the verification of source MPC programs and (even partially) verified toolchain.

This chapter presents Wys⋆, which addresses these problems. Wys⋆ is an embedding of Wysteria in F⋆ [37], a verification-oriented, full-featured programming language. Wys⋆ solves the problems above in the following manner.

First, Wys⋆ programs are essentially F⋆ programs written against an MPC library that exports the Wysteria API. And so, programmers can *formally* verify their Wys⋆ MPC programs by using the F⋆'s type-and-effect system to specify the correctness and security properties, and F⋆'s semi-automated verification facilities to prove that the programs satisfy those properties. As an example, for the optimized PSI program [7] and the optimized median program [5], we have proved that the extra rounds of secure computations and message exchanges do not reveal anything more than the final result.

Second, Wys⋆ provides a partially verified interpreter to run the MPC programs. We implement the custom Wysteria semantics (the actual multi-threaded protocol semantics) by defining an interpreter in F⋆ that operates over Wys⋆ abstract syntax trees (ASTs), defined as a datatype; these trees are produced by running the F⋆ compiler (in a special mode) on the extended source program. We formalize the two operational semantics for Wysteria (Section 2.4) in F⋆ and *mechanize* the proofs that the single-threaded semantics is sound with respect to the distributed semantics, and that the distributed semantics is correctly implemented by our interpreter. As a result, we have verified that the properties we prove about the Wysteria-extended F⋆ source programs hold for the multi-party programs that actually run. We have not yet verified the circuit library that compiles Wys⋆ ASTs to circuits and run them using the GMW protocol [11]. Formal verification of GMW is an open problem, and we leave it for future work.

Finally, Wys⋆ embedding allows MPC programs to use, with no extra effort, stan-

dard language features (such as datatypes) and libraries (such as for I/O) directly from F*. WYS* provides a Foreign Function Interface (FFI) that enables MPC programs to seamlessly integrate with the F* code. As a pleasant side-effect, the embedding also simplifies the WYSTERIA language design substantially making it more standard and streamlined, we point out the simplifications as we present WYS* details.

We first present a primer on F* (Section 3.1), followed by an overview of WYS* with the help of some examples (Section 3.2). We then formalize WYS* and its (mechanically verified) metatheory (Section 3.3). Section 3.4 and Section 3.5 present implementation and evaluation of WYS*.

3.1 F* primer

F* is an ML-like functional language, but with a more expressive type system based on dependent refinement types and monadic effects. Programmers can use F* types to express precise and compact specifications, including the correctness and security properties of their programs. The F* type checker then attempts to prove that the programs meet their specifications by generating proof obligations (using a weakest precondition calculus) and discharging them with the help of an SMT solver (e.g. Z3 [40]).

F* supports the verification of effectful code (e.g. code that uses State and Exceptions) by having a monadic type system where each monad is indexed with pre- and post-conditions. F* has primitive support for commonly used effects such as State and Exception, but the programmer can also define *domain-specific* effects and use them for verifying domain-specific programs.

For example, to verify stateful code, F* provides an ST monad. The ST monad is indexed with a return type – the return type of an ST computation, a pre-condition – a predicate on the input state, and a post-condition – a predicate on the input state, the return value, and the output state. Programmers can annotate stateful computations with types in the ST monad, and the F* type checker can type-check the specifications. Consider a function that increments the contents of an int reference:

```
let incr r = r := !r + 1
```

A possible type for the incr function is:

```
val incr: r:ref int → ST unit (fun s0 → True) (fun s0 u s1 → sel s0 r ≥ 0 ⟹ sel s1 r ≥ 0)
```

The type says that the incr function takes a ref int as an argument and performs a stateful computation, we can see the ST monad indexed with the return type of the computation (unit) and a pre- and post-condition. The function fun s0 → True is the (trivial) pre-condition predicate on the input state s0 (fun x → e defines an anonymous function with argument x and body e). (fun s0 u s1 → sel s0 r ≥ 0 ⟹ sel s1 r ≥ 0) is the post-condition predicate on the input state s0, the return value u, and the final state s1. The specification states that if in the input state, reference r contains a non-negative integer (sel s0 r ≥ 0), then in the final state also r contains a non-negative integer (sel s1 r ≥ 0).

This specification is quite weak. Neither does it capture the precise relation between the contents of r before and after the incr function, nor does it capture the fact that incr does not change any other location in the input state. Indeed, we can give incr a stronger type that captures these properties:

```
val incr: r:ref int → ST unit (fun s0 → True) (fun s0 u s1 → s1 = upd s0 r (sel s0 r + 1))
```

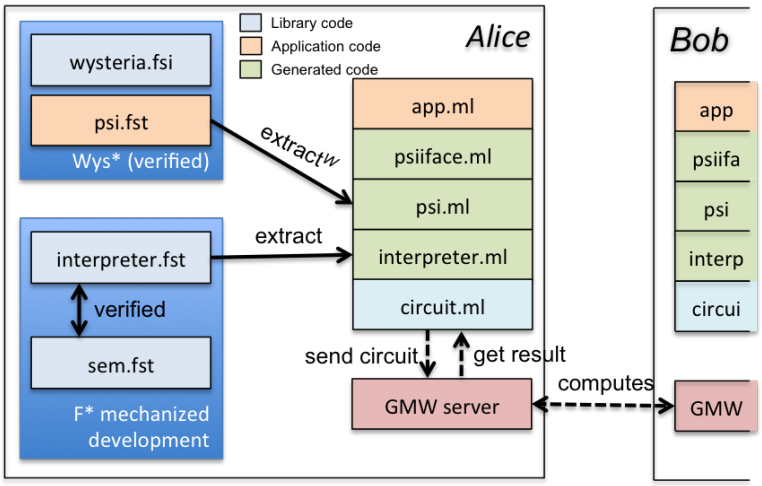

Figure 3.1: Architecture of an Wys⋆ deployment

This specification says that the output state `s1` is same as the input state `s0`, except at location `r` where the content of `r` is incremented by `1`.

For both the specifications above, the F⋆ type checker generates a verification condition as a proof obligation, and proves it using the SMT solver.

3.2 Verified programming in Wys⋆

We now present an overview of Wys⋆. Wys⋆ inherits the basic programming model from Wysteria (Chapter 2). We will illustrate it using an example. Consider a dating application that enables its users to compute their common interests without revealing all their private interests to one another. This is an instance of the private set intersection (PSI) problem. We illustrate the main concepts of Wys⋆ by showing, in several stages, how to program, optimize, verify and deploy this application—Figure 3.1 provides an overview.

3.2.1 Secure computations with as_sec

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

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

The four arguments to psi are, respectively, principal identifiers for Alice and Bob, and Alice and Bob's inputs, expressed as lists. The as_sec ps f construct indicates that thunk f should be run in sec mode. Essentially as_sec construct in WYS* plays the role of *secure delegation* from WYSTERIA. In this mode, the code may jointly access the secrets of the principals ps. In this case, we jointly intersect input_a and input_b, the inputs of a and b, and then return the same result r to both a and b.

Outside of sec mode, Alice would not be permitted to see Bob's input, and vice versa, but inside both can be made visible using the reveal coercion. Note that we did not have such a coercion in WYSTERIA, instead WYSTERIA type system prohibited using the private inputs from other parties using a custom typing judgment form that tracked the *access mode* of variables (Figure 2.3). F*, being a general-purpose language, has a more standard typing judgment. Therefore, WYS* uses explicit coercions to enforce the access control. Specifically, private values of parties are *sealed*, and need to be explicitly revealed before use. During reveal, the WYS* API checks the current mode to ensure that it is allowed to

reveal the value. We provide more details on sealed values later in the section.

Coming back to the example, finally the code constructs a wire bundle, associating a result for each principal (which in this case is the same)—give p v builds a singleton wire bundle p $\mapsto$ v and $+\!\!+$ concatenates disjoint wire bundles. Again, the concept of wire bundles is same as that in WYSTERIA.

Running this code requires the following steps. First, we run the F* compiler in a special mode that *extracts* the above code, psi.fst, into the WYS* AST as a data structure in psi.ml. WYS* has only a few constructs of its own, like as_sec (the full syntax is in Figure 3.5 in Section 3.3), and these are extracted to WYS*-specific nodes. The rest of a program's code is extracted into *FFI nodes* that indicate the use of, or calls into, functionality provided by F* itself.

The next step is for each party, Alice and Bob, to run the extracted program using the WYS* interpreter. This interpreter is written in F* and provably implements a deep embedding of the F* semantics, also specified in F* (shown in Figures 3.7, 3.9, and 3.10 in Section 3.3). This interpreter is extracted to OCaml code by a standard F* process. When each party reaches as_sec ps f, the interpreter's back-end compiles f, on-the-fly, for particular values of the secrets in f's environment, to a boolean circuit. First-order, loop-free code can be compiled to a circuit; WYS* provides specialized support for several common combinators (e.g., List.mem, List.nth etc.).

The circuit is handed to the GMW library, that we used in Chapter 2, by Choi et al. [11]. Running the protocol at each party starts by confirming that they wish to run the same circuit, and then proceeds by generating (XOR-based) secret shares [46] for each party's secret inputs. Running the GMW protocol involves evaluating the boolean circuit

for f over the secret shares, involving communication between the parties for each AND-gate.

One obvious question is how both parties are able to get this process off the ground, running this program of four inputs, when only three of the inputs are known to them (the principals and their own input). In Wys⋆, values specific to each principal are *sealed* with the principal's name (which appears in the sealed container's type). As such, the types of input_a and input_b are, respectively, list (sealed {a} int) and list (sealed {b} int). When the program is run on Alice's host, the former will be a list of n of Alice's values, whereas the latter will be a list of n garbage values (which we denote as •). The reverse will be true on Bob's host. When the circuit is constructed, each principal links their non-garbage values to the relevant input wires of the circuit. Likewise, they populate a local copy of the output wire bundle with what is returned to them, with other principals' components absent from the wire bundle.

We would like MPC's like psi to be called from normal F⋆ programs. For example, we would like the logic for a dating application, which involves reading inputs, displaying results, etc. to be able to call into psi to compute common interests. To achieve this, Wys⋆ provides a way to compute a "single-party projection" of multi-party functions, i.e., a version of psi that can be called with just a single party's inputs. The other party's inputs are filled in with sealed garbage values, as described above. Calling this function from F⋆ code also kicks off the Wys⋆ interpreter, so that it can run psi as described above. When the interpreter completes, the result is returned and the F⋆ program can continue. We note that our Wysteria DSL implementation in Chapter 2 was standalone and lacked elements of a full-featured programming language, such as datatypes and I/O libraries.

3.2.2 Optimizing PSI with as_par

Although psi gets the job done, it turns out to be inefficient (as we have already seen in Section 2.7). Better implementations of PSI involve performing a *mixed-mode* computation, where each participant evaluates some local computations in parallel (e.g., iterating over the elements of their sets) interleaved with small amounts of jointly evaluated, cryptographically secure computations. Wys⋆'s second computation mode, called par mode, supports such mixed-mode computation. In particular, the construct as_par ps f states that each principal in ps should locally execute the thunk f, simultaneously (any principal not in the set ps simply skips the expression). Within f, principals may engage in secure computations via as_sec. Essentially as_par construct in Wys⋆ plays the role of *parallel delegation* in Wysteria.

Figure 3.2 shows the optimized version of PSI [7], which uses as_par. The function psi_opt (line 12) begins by using as_par involving Alice and Bob. In the provided thunk, each principal calls for_each_alice la lb, which in turn calls check_each_bob a lb, for each element a of Alice's list la. Secure computation occurs at the use of as_sec at line 8. Within the circuit, Alice and Bob securely compare their values ax and bx, and gather a list (list bool). There is one outer list for each of Alice's elements, the ith inner list contains comparisons of Alice's ith value with some of Bob's values—rather than comparing each of Alice's elements with all of Bob's, the code is optimized (as described below) to omit redundant comparisons. At line 13, both parties build a matrix of comparisons from the boolean lists. Alice inspects the rows of the matrix (line 14) to determine which of her elements are in the intersection; Bob inspects the columns (line 15); and the joint function gives a

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

Figure 3.2: Optimized PSI example in WYS*

result to each principal (line 16).

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

One might wonder whether we could have programmed most of this code normal F*, relying on just sec mode for the circuit evaluation. However, recalling that our goal is to formally reason about the code and prove it correct and secure, par mode provides significant benefits. In particular, the SIMD model provided by WYS* (and WYSTERIA) enables us to capture many invariants for free. For example, proving the correctness of psi_opt requires reasoning that both participants iterate their loops in lock step—WYS* assures this by construction. Besides, the code would be harder to write (and read) if it were

split across multiple functions or files. As a general guideline, we use F⋆ for code written from the view of a single principal, and Wys⋆ when programming for all principals at once, and rely on the FFI to mediate between the two.

3.2.3 Embedding a type system for Wys⋆ in F⋆

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

1. Is the protocol realizable? For example, does a computation that is claimed to be executed only by some principals ps (e.g., using an as_par ps or an as_sec ps) only ever access data belonging to ps?

2. Does the protocol correctly implement the desired functionality? For example, does it correctly compute the intersection of Alice and Bob's sets?

3. Is the protocol secure? For example, do the optimizations of the previous section that omit certain comparisons inadvertently also release information besides the final answer?

While Wysteria type system addresses the first of the three questions, it provides only *on-paper* reasoning capabilities for the next two. By embedding Wysteria in F⋆ and leveraging its type system, we address each of these three questions.

Our strategy is to make use of F⋆'s extensible, monadic dependent type-and-effect system to define a new indexed monad (called Wys) and use it to describe precise trace properties of Wysteria multi-party computations. Additionally, we make use of an abstract

type, sealed ps t, representing a value accessible only to the principals in ps. Combining the Wys monad with the sealed type, we encode a form of information-flow control to ensure that protocols are realizable.

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

We note that while WYSTERIA (Chapter 2) had the same two computation modes, it lacked the formal notion of traces.

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

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

Every WYS* computation e has a monadic computation type Wys t pre post (similar

to the ST monad from Section 3.1). The type indicates that e is in the Wys monad (so it may perform multi-party computations); t is its result type; pre is a pre-condition on the mode in which e may be executed; and post is a post-condition relating the computation's mode, its result value, and its trace of observable events. When run in a context with mode m satisfying the pre-condition predicate pre m, e may send and receive message according to some trace tr, and if and when it returns, the result is a t-typed value v validating the post-condition predicate post m v tr. The style of indexing a monad with a computation's pre- and post-condition is a standard technique [37,47,48]—we focus on the specifications of combinators specific to WYS*.

We now describe some of the WYSTERIA-specific combinators in WYS*, and how we give them types in F*. The complete API is shown in Figure 3.4.

Defining as_sec in WYS*.

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

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

The pre-condition of as_sec is a predicate on the mode m of the computation in whose context as_sec ps f is called. For all the ps to jointly execute f, we require all of them to transi-

tion to perform the as_sec ps f call simultaneously, i.e., the current mode must be Mode Par ps. We also require the pre-condition pre of f to be valid once the mode has transitioned to Mode Sec ps—line 2 says just this.

The post-condition of as_sec is a predicate relating the initial mode m, the result r:a, and the trace tr of the computation. Line 3 states that the trace of a secure computation as_sec ps f is just a singleton [TMsg r], reflecting that its execution reveals only result r.[1] It also ensures that the result r is related to the mode in which f is run (Mode Sec ps) and the empty trace [] (since f has no observables) according to post, the post-condition of f.

A limitation of our current embedding is that it does not enforce invariants related to the first-order inputs, etc., that help ensure that at runtime the secure computation can be successfully compiled to a boolean circuit (Section 2.3). Instead, Wys⋆ toolchain gives a runtime error.

Access control with sealed types. To ensure that a program never relies on computing with data that it cannot access, we type secrets belonging to a set of principals ps using the type sealed ps t. This is an abstract type manipulated only according to the interface shown in Figure 3.3, a variation on a previous, translucent abstractions pattern used in F⋆ [37].

The type sealed ps t encapsulates a t value. To extract the underlying value, on can either call unseal or reveal. The unseal function is marked Ghost—meaning that it can only be called in specifications. In their types, the first parameter ps is marked as an implicit parameter, using the # notation. On the other hand, reveal can be called in concrete Wysteria programs, although with constraints on the mode. Its precondition says that when executing in Mode Par ps', *all* current participants must be listed in the seal, i.e., ps' ⊆ ps.

[1] This is the "ideal functionality" ensured by the backend, e.g., GMW.

```
type sealed : prins → Type → Type

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

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

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

Figure 3.3: Access control with sealed types

However, when executing in Mode Sec ps', only a subset of current participants is required: ps' ∩ ps ≠ ∅. This is because the secure computation is executed for all of ps', so it can access any of their individual data. Creating sealed values using seal is straightforward.

In both cases, the post-condition says that there are no observable events and that the returned value is the underlying value in the seal (produced by unseal). Recall that Wysteria does not have corresponding constructs. There access control is implemented using mode annotations in type environment bindings (Section 2.3).

We show the complete Wys* API in Figure 3.4 for reference. Wire bundle rules will be explained in the context of single-threaded operational semantics in Section 3.3.3.

3.2.4 Correctness and security verification

Using the Wys monad and the sealed type, we can write down precise types for our psi program, proving various useful properties. We discuss the statements of the main lemmas we prove and the proof structure. By programming the protocols using the high-level abstractions provided by Wys*, our proofs are relatively straightforward. In particular, we

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

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

type sealed : prins → Type → Type

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

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

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

type wire : prins → Type → Type

(* we omit the signatures and axioms for ghost select, contains, join, and const_wire functions on wires *)

(* type eprins is also a set of principals, but unlike prins, it admits the empty set *)

val mkwire_p: #ps1:prins → eps:eprins → x:sealed α ps1 → Wys (wire α eps)
     (requires (fun m → m.mode=Par ∧ eps ⊆ ps1 ∧ eps ⊆ m.ps))
     (ensures (fun m r tr → r = const_wire eps (unseal x) ∧ tr = []))

val mkwire_s: eps:eprins → x:α → Wys (wire α eps)
     (requires (fun m → m.mode=Sec ∧ eps ⊆ m.ps))
     (ensures (fun m r tr → r = const_wire eps x ∧ tr = []))

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

val concat: #epsx:eprins → #epsy:eprins → x:wire α epsx → y:wire α epsy → Wys (wire α (epsx ∪ epsy))
     (requires (fun m → disjoint (dom x) (dom y)))
     (ensures (fun m r tr → r = join x y ∧ tr = [])
```

Figure 3.4: WYS⋆ API in F⋆

rely heavily on the view that both parties execute (different fragments of) the same code. In contrast, reasoning directly against the low-level message passing semantics would be much more unwieldy. In Section 3.3, by formalizing the connection between the high- and low-level semantics, we justify our source-level reasoning.

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

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

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

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

Given a full characterization of the observable behavior of psi_opt_trace la lb in terms of its inputs, we can prove optimizations correct using relational reasoning [49] and we

can also prove security hyperproperties [50] by relating traces from multiple runs of the protocol.

Our goal is to prove a noninterference with delimited release [51] property for psi_opt. Our attacker model is the "honest-but-curious" model where the attackers are the participants in the protocol themselves. That is, we assume that the participants in the protocol play their roles faithfully, but they are motivated to deduce as much as they can about the other participants' secrets by observing the protocol. We do not aim to prove security properties against a third-party network adversary.

For psi, from the perspective of Alice as the attacker, we aim prove that for two runs of the protocol in which Alice's input is constant but Bob's varies, Alice learns no more by observing the the protocol trace than what she is allowed to. Covering Bob's perspective symmetrically, we show that in two runs of psi la_0 lb_0 and psi la_1 lb_1 that satisfy formula Ψ below, the traces observed by Alice and Bob are indistinguishable, up to permutation, where la_0, la_1, lb_0, lb_1 have type lset int, the type of integer sets represented as lists.

$$\Psi\ la_0\ la_1\ lb_0\ lb_1 = \text{intersect}\ la_0\ lb_0 = \text{intersect}\ la_1\ lb_1 \wedge \\ \text{length}\ la_0 = \text{length}\ la_1 \wedge \text{length}\ lb_0 = \text{length}\ lb_1$$

In other words, Alice and Bob learns no more than the intersection of their sets and the size of the other's set; Ψ is the predicate that delimits the information released by the protocol. As far as we are aware, this is the first formal proof of correctness and security of Huang et al.'s optimized, private set-intersection protocol.[2]

The proof is in the style of a step-wise refinement, via psi, an inefficient variant

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

of the psi_opt program. Running psi la lb always involves doing exactly length la $*$ length lb comparisons in two nested loops. We prove the following relational security property for psi, relating the traces trace_psi la_0 lb_0 and trace_psi la_1 lb_1—the formal statement of the lemma we prove in F* is shown below.

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

We reason about the traces of psi only up to permutation. Given that Alice has no prior knowledge of the choice of representation of Bob's set (Bob can shuffle his list), the traces Alice observes are equivalent up to permutation—we can formalize this observation using a probabilistic, relational variant of F* [52], but have yet to do so.

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

3.2.5 Relating security proofs to cryptographic security

The security proofs in Wys* correspond to the idealized model, where the parties directly see only the final output of a secure computation. This is evident from the specification of the as_sec API – only the final output is added to the trace of observable events. To relate the proof to the actual implementation, we assume that the underlying cryptographic protocol (e.g. GMW in our implementation) is secure, and then appeal to the composition

theorem by Canetti [53] to establish that the program securely realizes the mixed-mode (also known as reactive) functionality given by the specification.

3.3 WYS* formalization

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

In this section, we present the metatheory that answers this question. First, we formalize the WYS* single-threaded (ST) semantics, arguing that it faithfully realizes the F* semantics, including the WYS* API presented in Section 3.2. Next, we formalize the distributed (DS) semantics that the multiple parties use to run WYS* programs. Our theorems establish the correspondence between the two semantics, thereby ensuring that the properties that we verify using F* carry over to the actual protocol runs. We have mechanized all the metatheory presented in this section in F*.

3.3.1 Comparison with WYSTERIA formalization

Many of the WYS* constructs are similar to WYSTERIA. As we have already seen, as_par and as_sec in WYS* correspond to the parallel and secure delegations from WYSTERIA. However, WYS* considerably simplifies the language design. To enforce access control, as we have already seen, WYS* uses sealed values, an abstraction that is not present in WYSTERIA. Since sealed values abstraction provides explicit access control, instead of

using non-standard runtime environments that carry a *mode* annotation with each variable to value mapping (Section 2.4.1), Wys* uses standard environments with no mode annotations. Wys* also introduces the formal notion of traces.

In Wys*, we have also considerably simplified the operational semantics rules related to the secure blocks entry and exit. Recall that in Wysteria, we modeled a two-phase secure blocks entry and exit (Section 2.4.2). In Wys*, we instead model a single-phase entry and exit, where each party simultaneously steps in and out of the secure block. We also omit the wire combinators **waps**, **wapp**, and **wfold** in Wys*, and use general fix-points instead.

In terms of metatheory, we have mechanically verified the Wys* metatheory (semantics correspondence theorem, and a new theorem that relates the terminating runs in the single-threaded semantics to terminating runs in the protocol semantics). In contrast, Wysteria metatheoretic proofs were only "on-paper".

Currently Wys* has certain limitations also. In terms of compilation to circuits, as mentioned earlier, Wys* type system does not check prerequisite conditions for successful circuit compilation (e.g. no private function inputs). Wys* circuit library also does not perform general loop unrolling. In metatheory, Wys* does not provide a separate type soundness theorem (as Wysteria did). Doing so would require us to formalize the F* type system and operational semantics. We plan to pursue these in future (also see Section 3.3.3).

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

Figure 3.5: WYS* syntax

3.3.2 Syntax

Figure 3.5 shows the complete syntax of WYS*. Principal and principal sets are first-class values, and are denoted by p and s respectively. Constants in the language also include $()$ (unit), booleans, and FFI constants c. Expressions e include the regular forms for functions, applications, let bindings, etc. and the WYS*-specific constructs. Among the ones that we have not seen in Section 3.2, expression $\mathsf{mkwire}\ e_1\ e_2$ creates a wire bundle from principals in e_1 (which is a principal set) to the value computed by e_2. $\mathsf{project}\ e_1\ e_2$ projects the value of principal e_1 from the wire bundle e_2, and $\mathsf{concat}\ e_1\ e_2$ concatenates the two wire bundles.

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

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

Figure 3.6: Runtime configuration syntax

S-LET

$$\frac{X_1 = (M; L; \mathsf{let}\ x\ =\ \langle\rangle\ \mathsf{in}\ e_2; T), X}{M; X; L; T; \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 \to M; X_1; L; \cdot; e_1}$$

S-APP

$$\frac{}{M; X; L; T; (L_1, \lambda x.e)\ v \to M; X; L_1[x \mapsto v]; T; e}$$

S-ASPAR

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

S-ASPARRET

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

S-ASSEC

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

S-ASSECRET

$$\frac{\mathsf{is\_sec}\ M \quad X = (M_1; L_1; \langle\rangle; T), X_1 \quad T_1 = \mathsf{snoc}\ T\ (\mathsf{TMsg}\ v)}{M; X; L; \cdot; v \to M_1; X_1; L_1; T_1; v}$$

S-SEAL

$$\frac{\mathsf{can\_seal}\ s\ v \quad M = \_\ s_1 \quad s \subseteq s_1}{M; X; L; T; \mathsf{seal}\ s\ v \to M; X; L; T; \mathsf{sealed}\ s\ v}$$

S-REVEAL

$$\frac{M = \mathsf{Par}\ s_1 \Rightarrow s_1 \subseteq s \quad M = \mathsf{Sec}\ s_1 \Rightarrow s_1 \cap s \neq \phi}{M; X; L; T; \mathsf{reveal}\ (\mathsf{sealed}\ s\ v) \to M; X; L; T; v}$$

S-FFI

$$\frac{v = \mathsf{exec\_ffi}\ \mathsf{f}\ \bar{v}}{M; X; L; T; \mathsf{ffi}\ \mathsf{f}\ \bar{v} \to M; X; L; T; v}$$

Figure 3.7: Wys* ST semantics (selected rules)

3.3.3 Single-threaded semantics

The ST semantics is a model of the F* semantics and the WYS* API. The ST semantics defines a judgment $C \rightarrow C'$ that represents a single step of an abstract machine. Here, C is a *configuration* $M; X; L; T; e$. This five-tuple consists of a mode M, a stack X, an environment L, a trace T, and an expression e. The syntax for these elements is given in Figure 3.6. The stack and environment are standard; the trace T and mode M were discussed in the previous section.

The ST semantics is formalized in the style of Hieb and Felleisen [54], where the redex is chosen by (standard) *evaluation contexts* E, which prescribe left-to-right, call-by-value evaluation order. A few of the core rules are given in Figure 3.7. In essence, the semantics extends a standard reduction machinery for a call-by-value, lambda calculus (in direct correspondence with a pure fragment of F*), with several WYSTERIA-specific constructs. We argue, by inspection, that the WYSTERIA-specific constructs are in 1-1 correspondence with their specifications in the Wys monad. Despite the "eyeball closeness", there is room for formal discrepancy between the ST semantics and its static model within F*'s Wys monad. We leave to future work formally proving a correspondence between the ST semantics and μF*, the official semantics of F* in F* [37].

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

rule S-ASPARRET pops the frame and seals the result, so that it is accessible only to the principals in s. The rule also creates a trace element TScope s T, essentially making observations during the reduction of e (i.e., T) visible only to the principals in s.

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

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

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

Next consider the rules S-ASSEC and S-ASSECRET. Again, we can see that the rules implement the type of as_sec (shown in Section 3.2). The rule S-ASSEC checks the precondition of the API, and the rule S-ASSECRET generates a trace observation TMsg v, as per the postcondition of the API.

In a similar manner, we can easily see that the rule S-REVEAL implements the corresponding pre- and postconditions as given in Section 3.2. The rule S-FFI implements the FFI call by calling a host-language function exec_ffi. As expected, calling a host-language function has no effect on the WYS*-specific state. Concretely, this is enforced by F*'s monadic encapsulation of effects. The remaining rules are straightforward.

We show rest of the β-reduction rules for the ST semantics in Figure 3.8. We leave

[3] For technical reasons, function closures may not be sealed; see the end of Section 3.3.4 for details.

S-MKWIREP

$$\frac{M = \mathsf{Par}\ s \quad v = \mathsf{sealed}\ s_2\ v_1 \quad \mathsf{can\_wire}\ v_1 \quad s_1 \subseteq s \quad s_1 \subseteq s_2}{M; X; L; T; \mathsf{mkwire}\ s_1\ v \to M; X; L; T; [s_1 \mapsto v_1]}$$

S-MKWIRES

$$\frac{M = \mathsf{Sec}\ s \quad \mathsf{can\_wire}\ v \quad s_1 \subseteq s}{M; X; L; T; \mathsf{mkwire}\ s_1\ v \to M; X; L; T; [s_1 \mapsto v]}$$

S-PROJWIREP

$$\frac{s = \{p\} \quad m = m'[p \mapsto v]}{\mathsf{Par}\ s; X; L; T; \mathsf{project}\ p\ m \to \mathsf{Par}\ s; X; L; T; v}$$

S-PROJWIRES

$$\frac{p \in s \quad m = m'[p \mapsto v]}{\mathsf{Sec}\ s; X; L; T; \mathsf{project}\ p\ m \to \mathsf{Sec}\ s; X; L; T; v}$$

S-CONCAT

$$\frac{}{M; X; L; T; \mathsf{concat}\ m_1\ m_2 \to M; X; L; T; m_1 \uplus m_2}$$

S-LETB

$$\frac{e = \mathsf{let}\ x\ =\ v_1\ \mathsf{in}\ e_2}{M; X; L; T; e \to M; X; L[x \mapsto v_1]; T; e_2}$$

S-FIX

$$\frac{}{M; X; L; T; \mathsf{fix}\ f.\lambda x.e \to M; X; L; T; (L, \mathsf{fix}\ f.\lambda x.e)}$$

S-FUN

$$\frac{}{M; X; L; T; \lambda x.e \to M; X; L; T; (L, \lambda x.e)}$$

S-FIXAPP

$$\frac{L_2 = L_1[f \mapsto \mathsf{fix}\ f.\lambda x.e][x \mapsto v]}{M; X; L; T; (L_1, \mathsf{fix}\ f.\lambda x.e)\ v \to M; X; L_2; T; e}$$

S-IF

$$\frac{v = \mathsf{true} \Rightarrow e = e_1 \quad v = \mathsf{false} \Rightarrow e = e_2}{M; X; L; T; \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to M; X; L; T; e}$$

Figure 3.8: Wys* ST semantics (remaining β-reduction rules)

out the context rules that are standard and simply move around the evaluation contexts. Rule S-MKWIREP creates a wire bundle in the Par mode. To ensure that the parties in the domain of the wire bundle have access to the mapped value, it requires the second argument to be a sealed value accessible to the parties in the domain ($s_1 \subseteq s_2$). It also checks the current mode party set is a superset of the domain ($s_1 \subseteq s$). Rule S-MKWIRES is similar except it does not require the mapped value to be a sealed value, as the secure context has access to the inputs of all the parties. Both the rules use an auxiliary function can_wire that checks that the value being mapped is not itself a wire bundle or a function closure. These checks are similar to what we had for WYSTERIA in Chapter 2. Rule S-PROJWIREP projects p's mapping from a wire bundle in a Par context. As in WYSTERIA, the rule requires $s = \{p\}$. Rest of the rules are straightforward.

3.3.4 Distributed semantics

The DS semantics implements judgments of the form $\pi \longrightarrow \pi'$, where a protocol π is a tuple $(P; S)$ such that P maps each principal to its local configuration and S maps a set of principals to the configuration of an ongoing, secure computation. Both kinds of configurations (local and secure) have the form C (per Figure 3.6).

In the DS semantics, principals evaluate the same program locally and asynchronously until they reach a secure computation, at which point they synchronize to jointly perform the computation. This semantics is expressed with four rules, given in Figure 3.10, which state that either: (1) a principal can take a step in their local configuration, (2) a secure computation can take a step, (3) some principals can enter a new secure computation, and

$$\textsc{L-aspar1}\ \frac{p \in s \quad X_1 = (M; L; \mathsf{seal}\ s\ \langle\rangle; T), X}{M; X; L; T; \mathsf{as\_par}\ s\ (L_1, \lambda x.e) \rightsquigarrow M; X_1; L_1[x \mapsto ()]; \cdot; e}$$

$$\textsc{L-asparret}\ \frac{\mathsf{can\_seal}\ s\ v \quad X = (M; L_1; \mathsf{seal}\ s\ \langle\rangle; T_1), X_1 \quad T_2 = \mathsf{append}\ T_1\ T}{M; X; L; T; v \rightsquigarrow M; X_1; L_1; T_2; \mathsf{sealed}\ s\ v}$$

$$\textsc{L-aspar2}\ \frac{p \notin s}{M; X; L; T; \mathsf{as\_par}\ s\ (L_1, \lambda x.e) \rightsquigarrow M; X; L; T; \mathsf{sealed}\ s\ \bullet}$$

$$\textsc{L-seal}\ \frac{\mathsf{can\_seal}\ s\ v}{p \in s \Rightarrow v_1 = \mathsf{seal}\ s\ v \quad p \notin s \Rightarrow v_1 = \mathsf{seal}\ s\ \bullet}{M; X; L; T; \mathsf{seal}\ s\ v \rightsquigarrow M; X; L; T; v_1}$$

$$\textsc{L-reveal}\ \frac{p \in s}{M; X; L; T; \mathsf{reveal}\ (\mathsf{sealed}\ s\ v) \rightsquigarrow M; X; L; T; v}$$

$$\textsc{L-mkwire}\ \frac{v = \mathsf{sealed}\ \_\ v_1 \quad \mathsf{can\_wire}\ v_1}{p \in s \Rightarrow m = [p \mapsto v_1] \qquad p \notin s \Rightarrow m = \cdot}{M; X; L; T; \mathsf{mkwire}\ s\ v \rightsquigarrow M; X; L; T; m}$$

$$\textsc{L-projwire}\ \frac{m = [p \mapsto v]}{M; X; L; T; \mathsf{project}\ p\ m \rightsquigarrow M; X; L; T; v}$$

$$\textsc{L-concat}\ \frac{}{M; X; L; T; \mathsf{concat}\ m_1\ m_2 \rightsquigarrow M; X; L; T; m_1 \uplus m_2}$$

Figure 3.9: Distributed semantics, selected local rules (M is always $\mathsf{Par}\ \{p\}$)

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

$$\textsc{P-sec}\ \frac{C \to C'}{P; S[s \mapsto C] \longrightarrow P; S[s \mapsto C']}$$

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

Figure 3.10: Distributed semantics, multi-party rules

finally, (4) a secure computation can return the result to the (waiting) participants.

The first case is covered by rule P-PAR, which (non-deterministically) chooses a principal's configuration and evaluates it according to the *local evaluation* judgment $C \rightsquigarrow C'$, which is given in Figure 3.9 (discussed below). The second case is covered by P-SEC, which evaluates using the ST semantics. The last two cases are covered by P-ENTER and P-EXIT, also discussed below.

Local evaluation. The rules in Figure 3.9 present the local evaluation semantics. These express how a single principal behaves while in par mode; as such, mode M will always be Par {p}. Local evaluation agrees with the ST semantics for the standard language constructs (not shown) and differs for WYS*-specific constructs.[4]

For an as_par expression, a principal either participates in the computation, or skips

[4]Our formal development actually shares the code for both sets of rules, using an extra flag to indicate whether a rule is "local" or "joint".

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

Rule L-SEAL has the same intuition as above. Rule L-REVEAL allows principal p to reveal the value sealed s v, only if $p \in s$. Rule L-MKWIRE creates a wire bundle. If the current mode principal is in the domain of the wire bundle, then it simply maps the contents of the sealed value in a singleton wire bundle, otherwise the rule creates an empty wire bundle. Rule L-PROJWIRE projects current principal's component from a wire bundle. As mentioned earlier, rest of the rules for reducing standard constructs are similar to the ST semantics and hence not shown here. As should be the case, there are no local rules for as_sec—to perform a secure computation parties need to combine their data and jointly do the computation.

Entering/exiting secure computations. Returning to Figure 3.10, Rule P-ENTER handles the case when principals enter a secure computation. It requires that all the principals $p \in s$ must have the expression form as_sec s $(L_p, \lambda x.e)$, where L_p is their local environment associated with the closure. Each party's local environment contains its secret values (in addition to some public values). Conceptually, a secure computation *combines* these environments, thereby producing a joint view, and evaluates e under the combination. We

define an auxiliary combine_v function on values as follows:

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

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

So now, consider the following code:

```
let x = as_par alice (fun x → 2) in
let y = as_par bob (fun x → 3) in
let z = as_sec (alice, bob) (fun z → (unseal x) + (unseal y)) in ...
```

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

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

So, the rule P-ENTER combines the principals' environments, and creates a new entry in the S map. The principals are now waiting for the secure computation to finish.

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

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

As an example, consider the following code:

```
let x = as_sec (alice, bob) (fun x → let y = ... in seal alice y)
```

Since the return value of the secure computation is sealed for alice, bob will get a sealed alice •, produced using the slice_v function on the result of seal alice y.

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

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

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

We now return to the point of not allowing closures to be sealed. Consider the following example:

```
let x = as_par alice 2 in
let y = as_par bob (fun z → x) in
let z = as_sec ab (fun z →
                        let a = reveal y in
                        let b = reveal (a ()) in
                        b) in
z
```

In the source semantics, the program returns 2 to both the parties. In the target semantics, just before the call to the secure block, alice's environment would map x to seal

alice 2 and y to seal bob •, whereas bob's environment would map x to seal alice •, and y to the closure $(L, \mathsf{fun}\ \mathsf{z} \rightarrow \mathsf{x})$, where L maps x to seal alice •. Now, as per the target semantics, their environments are combined, and in the combined environment y gets the value from bob's environment, but the closure for y has a garbage value for x. Thus, running this program in the target semantics fails to make progress.

We found this problem during our effort of mechanizing the semantics. For now, we do not allow closures to be boxed. We plan to fix the problem in future.

3.3.5 Metatheory

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

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

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

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

We say that C is *terminal* if it is in Par mode and is fully reduced to a value (i.e., $C.e$ is a value and $C.X$ is empty). Similarly, a protocol $\pi = (P, S)$ is terminal if S is

empty and all the local configurations in P are terminal. The simulation theorem is then the following:

Theorem 6 (Simulation of ST by DS)**.** *Let s be the set of all principals. If $C_1 \rightarrow^* C_2$, and C_2 is terminal, then there exists some derivation $(\mathsf{slice}\ s\ C_1) \longrightarrow^* (\mathsf{slice}\ s\ C_2)$ such that $(\mathsf{slice}\ s\ C_2)$ is terminal.*

Notably, each principal's value and trace in protocol $(\mathsf{slice}\ s\ C_2)$ is the slice of the value and trace in C_2.

Confluence. We say that a protocol π *strongly terminates* in the terminal protocol π_t, written as $\pi \Downarrow \pi_t$, if all possible runs of π terminate in some number of steps in π_t. Our confluence result then says:

Theorem 7 (Confluence of DS)**.** *If $\pi \longrightarrow^* \pi_t$ and π_t is terminal, then $\pi \Downarrow \pi_t$.*

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

Corollary 8 (Soundness of ST semantics)**.** *Let s be the set of all principals. If $C_1 \rightarrow^* C_2$, and C_2 is terminal, then $(\mathsf{slice}\ s\ C_1) \Downarrow (\mathsf{slice}\ s\ C_2)$.*

Now suppose that for a Wys* source program, we prove in F* a post-condition that the result is sealed alice n, for some $n > 0$. By the soundness of the ST semantics, we can conclude that when the program is run in the DS semantics, it may diverge, but if it terminates, alice's output will also be sealed alice n, and for all other principals their outputs will be sealed alice $\bullet$. Aside from the correspondence on results, our semantics also covers correspondence on traces. Thus, via our embedding of Wysteria in F*, the correctness

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

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

3.4 Implementation

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

3.4.1 WYS* interpreter

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

Local evaluation is not defined for as_sec, so the stub implements what amounts to P-ENTER and P-EXIT from Figure 3.10. When the stub notices the program has reached

an as_sec expression, it calls into a circuit library we have written that converts the AST of the second argument of as_sec to a boolean circuit. This circuit and the encoded inputs are communicated to a co-located server, written using a library due to Choi et al. [11] that implements the GMW MPC protocol. The server evaluates the circuit, coordinating with the GMW servers of the other principals, and sends back the result. The circuit library decodes the result and returns it to the stub. The stub then carries on with the local evaluation.

Our F* formalization of Wys* is 5000 lines of code, including all the metatheory. It makes abundant use of F*'s dependent types to state and prove invariants. The implementation of the (verified) step function is essentially a big switch-case on the current expression, and is 60 lines of code. The tstep stub is another 15 lines. The size of the (unverified) circuit library, not including the GMW implementation, is 836 lines.

The stub, the implementation of GMW, the circuit library, and the F* extractor (including our custom Wys* mode for it) are part of our trusted computing base. As such, bugs in them could constitute security holes. Verifying these components as well (especially the circuit library and the GMW implementation, which are open problems to our knowledge) is interesting future work.

3.4.2 Secure server backend

We also provide a secure server backend for the alternative deployment scenarios where the parties are willing to reveal their inputs to a trusted server (perhaps running inside a secure enclave, such as Intel SGX [55]).

In this backend, when the (local) interpreter reaches an as_sec expression, it sends its inputs and the secure computation AST to a *secure server*. The secure server (written in F⋆), implements the ST semantics directly. It receives the inputs (and the same AST) from all the principals involved, combines their inputs (using the combine function extracted from our F⋆ development), and then interprets the AST in this combined environment using the same semantics as our F⋆ development. When the secure computation finishes, in addition to sending the output to each party, the server also communicates a cryptographic proof of its correctness using cryptographic signatures. We have verified the use of cryptography using a technique similar to Fournet et al. [56]. The local interpreter instances receive the output, verify the accompanying cryptographic signature, and carry on with the local evaluation. We conjecture that such a server could be useful for a trusted hardware based deployment scenario.

3.4.3 FFI

When writing a source Wys⋆ program (in F⋆), the programmer can call functions from an FFI module.[5] During compilation, the FFI module is extracted to OCaml using the regular F⋆ extraction. The custom mode of the F⋆ extraction that we have implemented, identifies the FFI calls in the Wys⋆ program, and extracts them to an E_ffi AST form, which is part of the AST expression type.

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

The f argument is extracted to be the name of the FFI function, that links to the

[5] How F⋆ programs call into Wys⋆ functions was described in Section 3.2.1.

extracted OCaml function. We explain the inj argument shortly.

When evaluating the Wys⋆ AST, the interpreter may reach an E_ffi f args inj node. As we saw in Section 3.3.3, the interpreter calls a library function exec_ffi with the list of values, in addition it also passes the inj argument. The exec_ffi function first un-embeds any embedded host-language arguments. The un-embedding function is straightforward (the values shown below are from the value AST form):

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

Interpreter specific values, such as V_seal, are passed as is. The FFI module does not have access to the Wys⋆ API in F⋆, and hence it can only use these values parametrically. exec_ffi then calls the OCaml function f with the un-embedded arguments. The OCaml function returns some result, that needs to be embedded back to the AST. So, the question is how can we embed the result at runtime? Inspecting the type of the result is not an option. The custom F⋆ extraction mode comes to our rescue.

When the extractor compiles an FFI call in the source program to an E_ffi node, it has the type information for the return value of the FFI call. Using this information, it instruments the E_ffi node with an *injection*, a function that can be used at runtime to embed the FFI call result back to the AST. For example, if the result is (), the injection is (an OCaml function) fun x → V_unit. If the return value is an interpreter value (e.g. V_seal), the injection is the identity. If the return value is some host value (such as a list, tuple, or int), the injection creates an V_ffi node. exec_ffi uses the injection to embed the result back to the AST, and returns it to the interpreter.

Our interface essentially provides a form of monomorphic, first-order interoperabil-

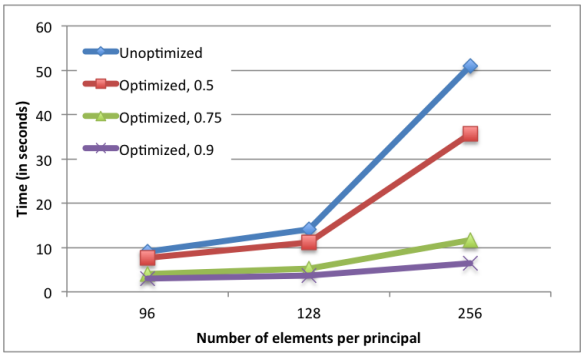

Figure 3.11: Time to run (in secs) normal and optimized PSI for varying per-party set sizes and intersection densities.

ity between the (dynamically typed) interpreter and the host language. We do not foresee any problems extending our current work to higher order with coercions [57].

3.5 Applications

Private set intersection. We evaluate the performance of the psi (computing intersection in a single secure computation), and the psi_opt (the optimized version) algorithms from Section 3.2. The programs that we benchmark are slightly different than the ones presented there, in that the local col and row functions are not the verified ones. The results are shown in Figure 3.11. We measure the time (in seconds) for per party set sizes 96, 128, and 256, and intersection densities (i.e. the fraction of elements that are common) 0.5, 0.75, and 0.9.

The time taken by the unoptimized version is independent of the intersection density since it always compares all pairs of values. However, as the intersection density increases, the optimized version performs far better – it is able to skip many comparisons. For lower densities (< 0.35), the optimization does not improve performance, as the algo-

rithm essentially becomes quadratic, and the setup cost for each secure computation takes over.

We note that our WYSTERIA evaluation resulted in a similar performance profile for PSI (Section 2.7).

Joint median. We program unoptimized and optimized versions of the two-party joint median (Section 2.7). The programs take two distinct, sorted inputs from alice, x_1 and x_2, and two distinct, sorted inputs from bob, y_1 and y_2 and return the median of all four. In the unoptimized version, the whole computation takes place as a monolithic secure computation, whereas the optimized version breaks the computation, revealing some intermediate results, and off-loading some parts to the local hosts (much like PSI).

For both the versions, we prove functional correctness:

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

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

Card dealing. We have implemented an MPC-based card dealing application in WYS*. Such an application can play the role of the dealer in a game of online poker, thereby eliminating the need to trust the game portal for card dealing. The application relies on WYS*'s support for *secret shares* [46]. Using secret shares, the participating parties can share a value in a way that none of the parties can observe the actual value individually

(each party's share consists of some random-looking bytes), but they can recover the value by combining their shares in a secure block.

In the application, the parties maintain a list of secret shares of already dealt cards (the number of already dealt cards is public information). To deal a new card, each party first generates a random number locally. The parties then perform a secure computation to compute the sum of their random numbers modulo 52, let's call it n. The output of the secure block is secret shares of n. Before declaring n as the newly dealt card, the parties needs to ensure that the card n has not already been dealt. To do so, they iterate over the list of secret shares of already dealt cards, and for each element of the list, check that it is different from n. The check is performed in a secure block that simply combines the shares of n, combines the shares of the list element, and checks the equality of the two values. If n is different from all the previously dealt cards, it is declared to be the new card, else the parties repeat the protocol by again generating a fresh random number each.

Wys* exports the following API for secret shares:

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

val v_of_sh: #a:Type → sh:Sh a → Ghost a
val ps_of_sh: #a:Type → sh:Sh a → Ghost prins

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

Type `Sh a` types the shares of values of type `a`. Our implementation currently supports shares of `int` values only; the `can_sh` predicate enforces this restriction on the source

programs. Extend secret shares support for other types (such as pairs) should be straightforward. Functions v_of_sh and ps_of_sh are marked Ghost, meaning that they can only be used in specifications for reasoning purposes. In the concrete code, shares are created and combined using the mk_sh and comb_sh functions. Together, the specifications of these functions enforce that the shares are created and combined by the same set of parties (through ps_of_sh), and that comb_sh recovers the original value (through v_of_sh). The WYS* interpreter transparently handles the low-level details of extracting shares from the GMW implementation of Choi et al. (mk_sh), and reconstituting the shares back (comb_sh).

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

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

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

Other applications and secure server. We have implemented some more applications in WYS*, including a geo-location sharing application. At the moment, we have only run these applications using a *secure server* backend. In this backend, as_sec works by literally sending code and inputs to a separate server that implements the ST semantics directly.

The server returns the result with a cryptographic proof of correctness to each party (we have verified the use of cryptography using a technique similar to [56]). We conjecture that such a server could be useful for a trusted hardware based deployment scenario.

3.6 Concluding remarks

In this chapter we have presented Wys*, a Verified, Domain-specific Integrated Language Extension hosted in F*. Wys* advances Wysteria on three fronts: (a) It enables the programmers to formally verify the correctness and security properties of their MPC programs using F*'s expressive type-and-effect system, (b) It provides a partially verified toolchain to run the MPC programs, thereby significantly reducing the risk of security-critical bugs in the toolchain, and (c) It enables MPC programs to use standard language constructs and libraries directly from F*, thereby enhancing the usability and scalability of the language. Wys* also simplifies the Wysteria language, making it more standard.

Wys* also has certain limitations as compared to Wysteria. In particular, the Wys* API does not perform checks that are prerequisite for successful circuit compilation at runtime (such as no private functions as input). Currently, such errors manifest in Wys* at runtime.

We presented an implementation of Wys* with two backends for secure computation. One that is based on the circuit library of Choi et al. [11], and one that uses a trusted server. We presented several examples that we have implemented using Wys*.

In the next chapter, we present static analyses that help optimize a monolithic MPC into a mixed-mode MPC with same privacy characteristics.

Chapter 4: Knowledge Inference for Optimizing Secure Multi-party Computations

In the previous chapters, we have seen that decomposing a monolithic secure multi-party computation into a mixed-mode secure multi-party computation sometimes provides big performance gains. We experimentally demonstrated such gains specifically for the private set intersection and joint median computation in Section 2.7. Indeed, other researchers have observed similar results [5, 7].

While Wysteria and Wys* provide language abstractions that enable programmers to program and formally verify the mixed-mode MPCs, in this chapter we consider the problem of optimizing a monolithic MPC into a mixed-mode MPC, providing similar privacy guarantees as the monolithic version.

A key observation underlying such optimizations is that while MPC protocols typically only reveal the final output to each party, a party may be able to infer the results of intermediate computations given the final output, their inputs, and the function being computed. When such inference is possible, the inferable intermediate results need not be cryptographically concealed. Revealing inferable results does not change the *knowledge profile* of the protocol: If the party will eventually know the intermediate result (e.g., given

the final output), then revealing it earlier does not change what is known to whom.[1] Decomposing monolithic protocols into smaller protocols that explicitly reveal intermediate results can significantly improve their performance.

Within the context of MPCs, we formally define notions of knowledge (of a program variable y to a party p), and the problems of knowledge inference and constructive knowledge inference. A solution to the knowledge inference problem states which parties can learn which additional variables, if any, from a cooperative run of the unoptimized protocol. We call a knowledge inference solution constructive if, in addition to correctly asserting that a party p knows a variable y, the solution also gives an evidence of party p's knowledge of y in the form of a program that computes y from p's private data and the final output.

We present algorithms that solve these problems, prove their soundness, and characterize the conditions under which they are also complete (Section 4.2). We implement our solutions and evaluate them experimentally (Section 4.4). While previous work has explored techniques for optimizing MPCs using knowledge inference [5], ours is the first to formalize the problem and its solutions, in addition to experimentally measure their performance. We begin with an overview of the problem as applied to the joint median computation MPC.

[1] As mentioned in Section 1.2, we assume the semi-honest threat model.

4.1 Overview

Consider the joint median computation between two parties Alice and Bob.[2]

```
1  (* assume a1 < a2, b1 < b2, distinct (a1, a2, b1, b2) *)
2
3  let median a1 a2 b1 b2 =
4    let x1 = a1 ≤ b1 in
5    let a3 = if x1 then a2 else a1 in
6    let b3 = if x1 then b1 else b2 in
7    let x2 = a3 ≤ b3 in
8    let m = if x2 then a3 else m = b3 in
9    m
```

Let a1 and a2 be Alice's inputs and b1 and b2 be Bob's. We also assume that these numbers are distinct, with a1 $<$ a2 and b1 $<$ b2. At the end of the computation, both parties share the joint median output m.

In the unoptimized version of secure computation for this example, the whole program is computed as a single secure computation. However, one can show that, with the knowledge of a1, a2, and m, Alice can always infer the values of x1 and x2, no matter what Bob's input values are [4]. Similarly, Bob can also infer the values of x1 and x2 from the knowledge of b1, b2, and m. Therefore, declassifying values of x1 and x2 *explicitly* to Alice and Bob during the computation would not compromise privacy, since they can infer them anyway, and it turns out that doing so it enables the following, more efficient MPC protocol:

Alice and Bob compute a1 $\leq$ b1 using secure computation and share the output x1 (line 4). Alice *locally* computes a3 (line 5). Bob locally computes b3 (line 6). Alice and

[2]While we show the example using a functional language syntax, our implementation operates on C++ programs.

Bob compute a3 $\leq$ b3 using secure computation and share the output x2 (line 7). If x2 is true, Alice sends a3 to Bob as the final median, else if x2 is false, Bob sends Alice b3 as the final median (line 8 and 9).

Thus, in the optimized version, only the two comparisons, need to be done securely. Moreover, Alice and Bob do not learn anything more than they did in the unoptimized version. For median computation on a joint set with 64 elements, Kerschbaum [5] shows 30x performance improvement using this optimization. We have also seen similar gains in WYSTERIA evaluation (Section 2.7).

More generally, in our setting, party p knows the (deterministic) program (call it S), his own input set I, and his output O.[3] We say party p can *infer* the value of local variable $y \in S$ if there exists a function F such that $y = F(I, O)$ in all runs of S. Another way of putting it is that no matter the values of p's inputs or those of other participants of the MPC, p can always compute y given knowledge of only his inputs and the final result. Our goal to find all those variables in S that p can infer. We can do this by either showing merely that the required function F exists, without saying what it is, or we can produce F directly, thus constituting a constructive proof. In this paper we present approaches to both tasks.

4.1.1 Knowledge inference

To show that an intermediate variable can be expressed as a function of one party's inputs and outputs, we can attempt to prove that given any pair of runs of S that agree on the

[3] Some MPCs may have different outputs for different parties; in the median example, there is a single output O = m known to both parties.

$$\varphi_1 \stackrel{\text{def}}{=} \mathsf{a1} < \mathsf{b1} \wedge \mathsf{x1} = \mathsf{true} \wedge \mathsf{a3} = \mathsf{a2} \wedge \mathsf{b3} = \mathsf{b1} \wedge \mathsf{a3} < \mathsf{b3} \wedge \mathsf{x2} = \mathsf{true} \wedge \mathsf{m} = \mathsf{a3} \wedge \phi_{pre}$$
$$\varphi_2 \stackrel{\text{def}}{=} \mathsf{a1} < \mathsf{b1} \wedge \mathsf{x1} = \mathsf{true} \wedge \mathsf{a3} = \mathsf{a2} \wedge \mathsf{b3} = \mathsf{b1} \ \wedge \mathsf{a3} \geq \mathsf{b3} \wedge \mathsf{x2} = \mathsf{false} \wedge \mathsf{m} = \mathsf{b3} \wedge \phi_{pre}$$
$$\varphi_3 \stackrel{\text{def}}{=} \mathsf{a1} \geq \mathsf{b1} \wedge \mathsf{x1} = \mathsf{false} \wedge \mathsf{a3} = \mathsf{a1} \wedge \mathsf{b3} = \mathsf{b2} \ \wedge \mathsf{a3} < \mathsf{b3} \wedge \mathsf{x2} = \mathsf{true} \wedge \mathsf{m} = \mathsf{a3} \wedge \phi_{pre}$$
$$\varphi_4 \stackrel{\text{def}}{=} \mathsf{a1} \geq \mathsf{b1} \wedge \mathsf{x1} = \mathsf{false} \wedge \mathsf{a3} = \mathsf{a1} \wedge \mathsf{b3} = \mathsf{b2} \ \wedge \mathsf{a3} \geq \mathsf{b3} \wedge \mathsf{x2} = \mathsf{false} \wedge \mathsf{m} = \mathsf{b3} \wedge \phi_{pre}$$

Figure 4.1: Path conditions for secure median

valuations of variables in I and O (but may not agree on the input and output variables of other parties), the valuations of y on those two runs must also agree. In other words, y can be determined uniquely from I and O, and thus a function F exists such that $F(I, O) = y$. We can construct such a proof in two steps.

First we use a program analysis to produce a formula ϕ_{post} that soundly approximates the final state of the program S (that is, the final values of all program variables) for all possible program runs. So that the meaning of a variable y mentioned in ϕ_{post} is unambiguous, we assume that a variable is assigned at most once during a program run.

One program analysis we might use to produce ϕ_{post} is symbolic execution [58]. Each feasible program path is characterized by a *path condition* φ_i, which is a set of predicates relating the program variables. The path conditions can be combined to provide a complete description of the program's behavior: $\phi_{post} \stackrel{\text{def}}{=} \bigvee_i \varphi_i$. For the median program above, there are four possible paths, having the path conditions given in Figure 4.1.

Consider the first path condition φ_1. Conceptually, it describes the program path in which `then` branch of both conditionals (lines 5 and 8) is taken. The remaining three paths constitute the other three possible branching combinations. Note that each path also requires ϕ_{pre}. This formula defines the publicly-known constraints on all inputs; in the case of the median program we have $\phi_{pre} \stackrel{\text{def}}{=} \mathsf{a1} < \mathsf{a2} \wedge \mathsf{b1} < \mathsf{b2} \wedge \mathsf{a1} \neq \mathsf{b1} \wedge \mathsf{a1} \neq \mathsf{b2} \wedge \mathsf{a2} \neq$

$\mathsf{b1} \wedge \mathsf{a2} \neq \mathsf{b2}$.

The next step is to prove that any two runs of the program S that agree on variables known to p will also agree on the value of y. This statement is a 2-safety property [59], and we can prove it using a technique called self-composition [60]. The idea is to reduce this two-run condition on program S to a condition on a single run of a *self-composed program* S_c, which is the sequential composition of S with itself, with the second copy of S's variables renamed, e.g., so that x is renamed to x'. Given the formula ϕ^{sc}_{post} for this self-composed program, we can ask whether, under the assumption that the normal and primed versions of p-visible variables are equal, that the normal and primed version of y is also equal.

As an example, Figure 4.2 shows self composition of the median function. We write median' for the function median but with the local variables renamed to x1', x2', ... The self-composed program effectively runs median twice, on two separate spaces of variables. We can express the question of knowledge inference as a question on the relationship between the two copies of the variables. Namely, Alice can infer x1 if and only if for every feasible final state of the composed program, when the two copies of a1,a2,m agree on their values then the copies of x1 agree on their value. More formally we need to check the validity of the following formula for any feasible final state.

$$\phi^{sc}_{post} \wedge (\mathsf{a1} = \mathsf{a1'} \wedge \mathsf{a2} = \mathsf{a2'} \wedge \mathsf{m} = \mathsf{m'}) \Rightarrow (\mathsf{x1} = \mathsf{x1'})$$

Here, the formula ϕ^{sc}_{post} will involve sixteen path conditions (self-composition squares

```
(* a1 < a2, b1 < b2, distinct (a1, a2, b1, b2) *)
m = median a1 a2 b1 b2;

(* a1' < a2', b1' < b2', distinct (a1', a2', b1', b2') *)
m' = median' a1' a2' b1' b2'
```

Figure 4.2: Median computation composed with itself.

the number of paths). For example, among them will be:

$$\varphi_1^{sc} \stackrel{\text{def}}{=} \mathsf{a1} < \mathsf{b1} \wedge \mathsf{x1} = \mathsf{true} \wedge \mathsf{a3} = \mathsf{a2} \wedge \mathsf{b3} = \mathsf{b1} \ \wedge \mathsf{a3} < \mathsf{b3} \wedge \mathsf{x2} = \mathsf{true} \wedge \mathsf{m} = \mathsf{a3} \wedge \phi_{pre} \wedge$$
$$\mathsf{a1'} < \mathsf{b1'} \wedge \mathsf{x1'} = \mathsf{true} \wedge \mathsf{a3'} = \mathsf{a2'} \wedge \mathsf{b3'} = \mathsf{b1'} \ \wedge \mathsf{a3'} < \mathsf{b3'} \wedge \mathsf{x2'} = \mathsf{true} \wedge \mathsf{m'} = \mathsf{a3'}$$
$$\wedge \phi'_{pre}$$

The formula φ_1^{sc} is actually the conjunction of φ_1 with a version of φ_1 that has all its variables renamed to the primed versions. We can think of the entire post condition $\phi_{post}^{sc} = \varphi_1^{sc} \vee ... \vee \varphi_{16}^{sc}$ as the conjunction of the post condition ϕ_{post} with its primed version.

Being a quantifier-free formula in the theory of integer linear arithmetic, the final formula poses no problem for an SMT solver such as Z3 [40], which can indeed verify its validity. Additionally, the same can be said for Alice's knowledge of x2 and a3, and Bob's knowledge of x1, x2 and b3.

The knowledge inference question bears a close resemblance to deciding the property of *delimited release* [51]. We explore this connection in more detail in Section 4.3.

4.1.2 Constructive Knowledge Inference

The technique just described can establish that there exists some function F such that $y = F(I, O)$, where y is an intermediate variable in S, and I and O are variables

known to party p. However, this technique cannot say what F actually is. To construct F we can leverage ideas from template-based program verification.

Program verification generally aims at inferring invariants in a program that are strong enough to verify some assertions of interest. Template-based program verification [38, 39] requires programmers to specify the structure of these invariants in the form of a template. The algorithm then generates verification constraints for the assertions, to be solved by an SMT solver (e.g. Z3 [40]). A solution to the constraints yields a valid proof of the correctness of the assertions as well as a solution to the template unknowns. Gulwani et al. [38] present constraint-based verification techniques over the abstraction of linear arithmetic and Srivastava et al. [39] present these techniques over the abstraction of predicates.

To infer p's knowledge of a variable y, our algorithm tries to infer a formula ϕ s.t. (a) at the end of the program $y = \phi$, and (b) ϕ only mentions the input and output variables known to p. If we add an assertion $y = \phi$ at the end of the program, and provide a template structure for ϕ (limited to formulae over variables known to p) this becomes a template-driven verification problem where the assertion and the invariant are the same. A successful verification of the assertion $y = \phi$ establishes p's knowledge of y, and also the solution for ϕ yields a formula for y in terms of input and output variables of p.

The template structure for this problem is defined as follows. Suppose y is a boolean variable. Then the template for ϕ requires it to be in *disjunctive normal form* (DNF) such that there are exactly d disjuncts each consisting of c conjuncts, with each conjunct drawn from a set of predicates Q. This set contains predicates over linear expressions involving I and O. In the median example, for Alice, one choice of Q is $\{v_1 \odot v_2 \mid v_1, v_2 \in$

$\{\mathsf{a1}, \mathsf{a2}, \mathsf{m}\}, \odot \in \{>, \geq, <, \leq, =, \neq\}\}$. For Bob, similar Q would be $\{v_1 \odot v_2 \mid v_1, v_2 \in \{\mathsf{b1}, \mathsf{b2}, \mathsf{m}\}, \odot \in \{>, \geq, <, \leq, =, \neq\}\}$.

Our algorithm searches for a ϕ conforming to the prescribed template. For example, if $(c = 2, d = 2)$, then the search space for ϕ is all the boolean formulae $(q_1 \wedge q_2) \vee (q_3 \wedge q_4), q_1, q_2, q_3, q_4 \in Q$. We denote this search space for formulae as $\mathsf{DNF}(c, d, Q)$. A naive search algorithm would make $O(|Q|^{cd})$ queries to the SMT solver, one for every possible formula in $\mathsf{DNF}(c, d, Q)$. This algorithm is complete in the sense that if there exists a solution for ϕ in $\mathsf{DNF}(c, d, Q)$ then the naive search algorithm finds it. Our algorithm, on the other hand, makes $O(|Q|^c + |Q|^d)$ queries to the SMT solver, and still guarantees completeness, provided the existence of solution in $\mathsf{DNF}(c, d, Q)$.

Consider variable $\mathsf{x1}$ from the median example. With Q_{Alice} and Q_{Bob} as above, and $(c = 1, d = 1)$, we are able to establish knowledge of $\mathsf{x1}$ for both Alice and Bob as: $\mathsf{x1} = \mathsf{m} > \mathsf{a1}$ for Alice, and $\mathsf{x1} = \mathsf{m} \leq \mathsf{b1}$ for Bob. Interestingly, $(c = 1, d = 1)$ is insufficient to discover invariants describing Alice's and Bob's knowledge of $\mathsf{x2}$. With $(c = 1, d = 2)$, we are able to establish Alice's knowledge of $\mathsf{x2}$ as $\mathsf{x2} = (\mathsf{m} = \mathsf{a1} \vee \mathsf{m} = \mathsf{a2})$. And with $(c = 2, d = 1)$, we are able to establish Bob's knowledge of $\mathsf{x2}$ as $\mathsf{x2} = (\mathsf{m} \neq \mathsf{b1} \wedge \mathsf{m} \neq \mathsf{b2})$. In general, starting with $(c = 1, d = 1)$, we can increment (c, d) in steps until either we find a solution or we leave x as being unknown to p.

We can also infer formulae for integer variables. In this case we use a different template structure, and leverage ideas from Gulwani et al. [38]. We discuss the algorithm further in the next section.

Constructive knowledge inference problem is closely related to the problem of inferring output function in *required release* [61], a connection we explore more in Section 4.3.

$$
\begin{array}{rrcl}
\text{Value} & v & ::= & n \mid \mathsf{true} \mid \mathsf{false} \\
\text{Exprn./Formula} & e, \phi & ::= & v \mid x \mid e_1 \odot e_2 \\
\text{Binary operator} & \odot & \in & \{\leq, \geq, >, <, =, \neq\} \cup \{\wedge, \vee, \neg, \Rightarrow\} \cup \{+, -\} \\
\text{Statement} & S & ::= & x := e \mid S_1; S_2 \mid \mathsf{skip} \mid \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2
\end{array}
$$

Figure 4.3: Syntax.

4.2 Formal development

In this section, we formally describe our knowledge inference algorithm. We first give the language syntax, operational semantics, and formal definition of *knowledge* in Section 4.2.1. We then present inference in Section 4.2.2, and constructive inference in Section 4.2.3.

4.2.1 Language Syntax

Let parties $p_1, \ldots, p_n$ want to compute a secure computation S whose syntax is given in Figure 4.3. The language is standard aside from the omission of a looping construct; this makes sense in our setting since most MPC methods forbid dynamic looping (rather, they require a static loop unrolling). Our methods support loops as well, but we elide them nevertheless to keep the formalization simpler. We also assume, for simplicity, that each program path is in single assignment form, i.e. in an execution of a program, every variable is assigned at most once.

The semantics of computations is given in Figure 4.4. The judgments have the form $\langle \sigma, S \rangle \Downarrow \sigma'$, meaning, statement S executed in state σ results in new state σ'. States σ are

$$\text{(E-VAR)}\ \frac{}{\langle \sigma, x\rangle \Downarrow \sigma[x]} \qquad \text{(E-VAL)}\ \frac{}{\langle \sigma, v\rangle \Downarrow v}$$

$$\text{(E-BINOP)}\ \frac{\langle \sigma, e_1\rangle \Downarrow v_1 \quad \langle \sigma, e_2\rangle \Downarrow v_2}{\langle \sigma, e_1 \odot e_2\rangle \Downarrow v_1 \odot v_2} \qquad \text{(E-ASSIGN)}\ \frac{x \notin \mathsf{dom}(\sigma) \qquad \langle \sigma, e\rangle \Downarrow v}{\langle \sigma, x := e\rangle \Downarrow \sigma[x \mapsto v]}$$

$$\text{(E-SEQ)}\ \frac{\langle \sigma, S_1\rangle \Downarrow \sigma' \quad \langle \sigma', S_2\rangle \Downarrow \sigma''}{\langle \sigma, S_1; S_2\rangle \Downarrow \sigma''} \qquad \text{(E-IFTRUE)}\ \frac{\langle \sigma, e\rangle \Downarrow \mathsf{true} \quad \langle \sigma, S_1\rangle \Downarrow \sigma'}{\langle \sigma, \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \Downarrow \sigma'}$$

$$\text{(E-IFFALSE)}\ \frac{\langle \sigma, e\rangle \Downarrow \mathsf{false} \quad \langle \sigma, S_2\rangle \Downarrow \sigma'}{\langle \sigma, \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \Downarrow \sigma'}$$

$$\text{(E-SKIP)}\ \frac{}{\langle \sigma, \mathsf{skip}\rangle \Downarrow \sigma} \qquad (\phi\text{-VALID})\ \frac{\langle \sigma, \phi\rangle \Downarrow \mathsf{true}}{\sigma \models \phi}$$

Figure 4.4: Semantics.

maps from variables x to values v; we write $\sigma[x]$ to look up x in σ, and we write $\sigma[x \mapsto v]$ to define a map identical to σ except that x maps to v. The figure also defines an auxiliary judgment for expressions having the form $\langle \sigma, e\rangle \Downarrow v$, meaning, expression e evaluated in state σ results in value v. The rules are standard, with one exception. The rule (E-ASSIGN) checks that $x \notin \mathsf{dom}(\sigma)$ to enforce single assignment form for the current program path. When an expression is viewed as a formula ϕ, we write $\sigma \models \phi$ to mean that in σ the formula ϕ evaluates to true. We also write predicate to mean a boolean valued formula.

Let V be a set of variables. We define two states as being *equivalent* on a set of variables as follows:

Definition 9 (Equivalence of States)**.** Two states, σ_1 and σ_2, are equivalent on a set of variables V, written as $\sigma_1 \overset{V}{\equiv} \sigma_2$, iff $\forall x \in V, \sigma_1[x] = \sigma_2[x]$.

Let ϕ_{pre} denote the precondition for a secure computation program S. It represents the assumptions that S makes about parties' inputs. In the median example, $\phi_{pre} = \texttt{a1} <$

$\mathsf{a2} \wedge \mathsf{b1} < \mathsf{b2} \wedge \mathsf{a1} \neq \mathsf{b1} \wedge \mathsf{a1} \neq \mathsf{b2} \wedge \mathsf{a2} \neq \mathsf{b1} \wedge \mathsf{a2} \neq \mathsf{b2}$. We are interested in executions $\langle \sigma, S \rangle \Downarrow \sigma'$ when $\sigma \models \phi_{pre}$.

We now define *knowledge* of a variable y to a party p in S, written as $\mathfrak{K}(S, p, y)$. Informally, y is known to p, if, whenever two final states of S are equivalent on the set of input and output variables of p, they are also equivalent on $\{y\}$.

Definition 10. [Knowledge of a Variable] Let S be a secure computation program with precondition ϕ_{pre}. For a party p in the computation, let I be the set of input variables of p, and O be the set of output variables of p. Then, a variable y in S is *known* to p, written as $\mathfrak{K}(S, p, y)$, if for all initial states σ_1, σ_2 s.t. $\sigma_1 \models \phi_{pre}$, $\sigma_2 \models \phi_{pre}$, and $\sigma_1 \overset{I}{\equiv} \sigma_2$, whenever $\langle \sigma_1, S \rangle \Downarrow \sigma_1'$ and $\langle \sigma_2, S \rangle \Downarrow \sigma_2'$ s.t. $\sigma_1' \overset{O}{\equiv} \sigma_2'$, we have $\sigma_1' \overset{\{y\}}{\equiv} \sigma_2'$.

The definition models the 2-safety property discussed in the Section 4.1. It says that the value of y can be uniquely determined from the knowledge of input and output variables of p, independent of the inputs of other parties in the computation. We should be careful when defining knowledge for extensions of the language in Figure 4.3. For example, although an RSA modulus N uniquely identifies the two primes p and q in the information-theoretic sense, it is computationally hard to infer p and q from N. We now give the formal description of knowledge inference algorithm.

4.2.2 Knowledge Inference

The problem of knowledge inference is as follows. For a secure computation program S, we want to know whether a party p *knows* a program variable y according to Definition 10. We present our knowledge inference algorithm in Figure 4.7, but before

$$
\begin{array}{lcl}
\varsigma(\mathsf{skip}, \phi) & = & \phi \\
\varsigma(\mathsf{x} := \mathsf{e}, \phi) & = & \phi \wedge (x = e) \\
\varsigma(\mathsf{S}_1; \mathsf{S}_2, \phi) & = & \varsigma(S_2, \varsigma(S_1, \phi)) \\
\varsigma(\mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, \phi) & = & (e \wedge \varsigma(S_1, \phi)) \ \vee (\neg e \wedge \varsigma(S_2, \phi))
\end{array}
$$

Figure 4.5: Postcondition of a predicate ϕ w.r.t. statement S.

that we give some auxiliary definitions.

Definition 11 (Validity of a Predicate)**.** A predicate ϕ is valid at the end of a program S with precondition ϕ_{pre}, if $\forall\sigma$ s.t. $\sigma \models \phi_{pre}$, $\langle \sigma, S \rangle \Downarrow \sigma'$, we have $\sigma' \models \phi$.

We define the postcondition of a predicate ϕ w.r.t. statement S, written as $\varsigma(S, \phi)$, in Figure 4.5. The following theorem states the properties of $\varsigma(S, \phi)$.

Theorem 12 (Soundness and Completeness of Postcondition)**.** *For a program S with precondition ϕ_{pre}, $\varsigma(S, \phi_{pre})$ is valid at the end of program S (Soundness). Moreover, for any other predicate ϕ s.t. ϕ is valid at the end of S, $\varsigma(S, \phi_{pre}) \Rightarrow \phi$ (Completeness).*

Proof. Soundness – Structural induction on S. Completeness – Structural induction on S and using following lemma for each case. Let $\sigma \models \varsigma(S, \phi_{pre})$. Then, $\exists\sigma'$ s.t. $\sigma' \models \phi_{pre}$, $\langle \sigma', S \rangle \Downarrow \sigma$, and $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) - \mathsf{Def}(S)$, where $\mathsf{Def}(S)$ is the set of variables defined by S. □

The theorem depends on the program paths being in single assignment form. Specifically, the postcondition rule for assignment statement assumes that x does not occur in ϕ.

We now define a variable renaming translation on predicates. The idea is to replace every variable in the predicate with a *copy* of the variable. Let θ be a mapping from

$$\text{(T-VAR1)}\quad \frac{x \in \theta}{\langle \theta, x \rangle \rightsquigarrow \langle \theta, \theta[x] \rangle} \qquad \text{(T-VAR2)}\quad \frac{x \notin \mathsf{dom}(\theta) \quad x' \text{ is fresh}}{\langle \theta, x \rangle \rightsquigarrow \langle \theta[x \mapsto x'], x' \rangle}$$

$$\text{(T-BINOP)}\quad \frac{\langle \theta, \phi_1 \rangle \rightsquigarrow \langle \theta', \phi_1' \rangle \quad \langle \theta', \phi_2 \rangle \rightsquigarrow \langle \theta'', \phi_2' \rangle}{\langle \theta, \phi_1 \odot \phi_2 \rangle \rightsquigarrow \langle \theta'', \phi_1' \odot \phi_2' \rangle}$$

$$\text{(T-VAL)}\quad \frac{}{\langle \theta, v \rangle \rightsquigarrow \langle \theta, v \rangle}$$

Figure 4.6: Variable renaming translation for a predicate.

```
1  InferKnowledge(S, φ_pre)
2    for each party p
3      let I be the set of p's input variables.
4      let O be the set of p's output variables.
5      φ_post := ς(S, φ_pre);
6      ⟨ϵ, φ_post⟩ ⇝ ⟨θ, φ'_post⟩.
7      φ_k := ⋀_{x∈I∪O} (x = θ[x]);
8      for each program variable y
9        φ := (φ_post ∧ φ'_post ∧ φ_k) ⇒ (y = θ[y]);
10       if (⊢_alg φ)
11         output y is known to p.
12       else
13         output y is not known to p.
```

Figure 4.7: Knowledge inference algorithm

variables to variables. The translation judgment is shown in Figure 4.6. We define similar translation judgments for statements and states and refer to them in the theorem proofs later on, however we do not show them here for lack of space.

Our algorithm is parameterized by an SMT solver (e.g. Z3 [40], STP [62]), that we denote as `alg`. We use `alg` to determine whether a given predicate is a tautology (always true). We write $\vdash_{\texttt{alg}} \phi$ as the query to `alg` for predicate ϕ.

The knowledge inference algorithm is shown in Figure 4.7. It takes as input the secure computation program S and its precondition ϕ_{pre}. For each party p and program

variable y, it outputs whether p knows y or not.

The algorithm first computes the postcondition of ϕ_{pre} w.r.t. S (ϕ_{post}). It then performs variable translation on ϕ_{post} to generate ϕ'_{post}. Essentially ϕ_{post} and ϕ'_{post} model two *different* runs of the program. ϕ_k then asserts that $\forall x \in I \cup O$, x has same value across these two runs. Under these assumptions, if a program variable y also has same value across these two runs, the variable y is known to p.

The soundness theorem of our algorithm is as follows:

Theorem 13 (Soundness of Knowledge Inference)**.** *Let S be a secure computation program with the precondition ϕ_{pre}. If $\texttt{InferKnowledge}(S, \phi_{pre})$ outputs variable y is known to party p, then $\mathfrak{K}(S, p, y)$.*

Proof. We want to prove that for two states σ and σ' s.t. $\sigma \overset{I \cup O}{\equiv} \sigma'$, $\sigma[y] = \sigma[y']$ (see Definition 10). If we translate σ according to θ to yield σ' ($\mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma') = \epsilon$), then we can see that $\sigma \cup \sigma' \models \phi_{post}$, $\sigma \cup \sigma' \models \phi'_{post}$ (soundness of postcondition), and $\sigma \cup \sigma' \models \phi_k$ (using above equivalence). Thus, it follows from line 9 in Figure 4.7 that $\sigma \cup \sigma' \models (y = \theta[y])$. □

Moreover, we can also state a completeness theorem.

Theorem 14 (Completeness of Knowledge Inference)**.** *Let S be a secure computation program with precondition ϕ_{pre}. For a program variable y and party p, if $\mathfrak{K}(S, p, y)$, then $\texttt{InferKnowledge}(S, \phi_{pre})$ outputs variable y is known to party p.*

Proof. For a program S, let S' be the translation of S. Then, we can see that $\phi_{post} \wedge \phi'_{post} \wedge \phi_k = \varsigma(S; S', \phi_{pre} \wedge \phi'_{pre} \wedge \phi_k)$. By completeness of postcondition, if $y = \theta[y]$ is valid at the end of $S; S'$, line 9 in Figure 4.7 must be true. □

```
1  ConstructKnowledgeB (S, φ_pre)
2    for each party p
3      construct the predicate set Q.
4      for each boolean program variable y
5        c = 1; d = 1;
6        do
7          φ := CFormula(y, c, d, Q);
8          increment (c, d) in lockstep.
9        while (φ is failure and c < c_max, d < d_max);
10       if (φ = failure)
11         output y is not known to p.
12       else
13         output y is known to p by φ.
```

Figure 4.8: Constructive knowledge inference for boolean variables

4.2.3 Constructive Knowledge Inference

The knowledge inference algorithm from Figure 4.7 establishes whether p knows y or not, however it does not give a formula for y in terms of p's input and output variables. In this section, we present constructive knowledge inference algorithms, that output such a formula.

Constructive knowledge inference for boolean variables. Define the verification condition of a predicate ϕ w.r.t. a statement S with precondition ϕ_{pre}, $\mathtt{VC}(S, \phi_{pre}, \phi)$, as $\varsigma(S, \phi_{pre}) \Rightarrow \phi$. Then, if $\vdash_{\mathtt{alg}} \mathtt{VC}(S, \phi_{pre}, \phi)$, the predicate ϕ is valid at the end of S.

Recall that to construct knowledge of variable y for a party p, we want to infer a formula ϕ, s.t. at the end of the program $y = \phi$ holds. For boolean variables, the search space for ϕ is $\mathsf{DNF}(c, d, Q)$, where Q is a set of predicates constructed from input and output variables of p.

The algorithm for boolean variables is shown in Figure 4.8. For each party p, it first

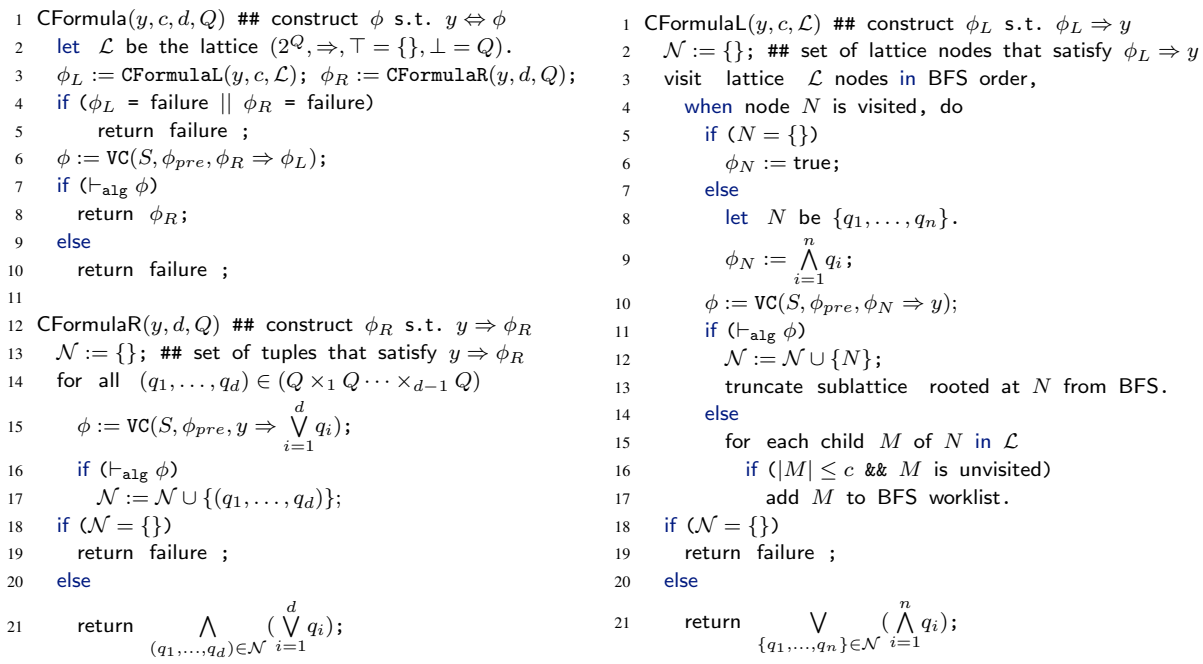

Figure 4.9: Routines `CFormula`, `CFormulaL`, and `CFormulaR`

constructs a predicate set Q. As mentioned earlier, this can either be provided as input by the programmer, or it can be mined from the expressions appearing in the program. For each boolean program variable y, starting with $(c = 1, d = 1)$ and incrementing (c, d) in lockstep until (c_{max}, d_{max}), it tries to find ϕ. It uses an auxiliary routine `CFormula`, defined in Figure 4.9.

Figure 4.9 consists of the subroutine `CFormula` and two other subroutines, that it invokes, `CFormulaL` and `CFormulaR`. We divide the problem of constructing ϕ into sub-problems of constructing ϕ_L and ϕ_R s.t. (a) ϕ_L and ϕ_R consist only of predicates from Q, and (b) at the end of the program, $\phi_L \Rightarrow y$, $y \Rightarrow \phi_R$, and $\phi_R \Rightarrow \phi_L$ hold. Then, we have $\phi = \phi_R$. `CFormulaL` constructs ϕ_L and `CFormulaR` constructs ϕ_R.

Construction of ϕ_L. To construct ϕ_L (`CFormulaL` in Figure 4.9), we perform breadth first search on the lattice of subsets of Q ordered by implication (i.e. $M \sqsubseteq N, M, N \in 2^Q$,

iff $\vdash_{\texttt{alg}} (\bigwedge_{q \in M} p) \Rightarrow (\bigwedge_{q' \in N} q'))$ with $\top = \{\}$ and $\bot = Q$, and collect all nodes of the lattice that form a *solution* to $\phi_L \Rightarrow y$. A node N in the lattice is a solution to $\phi_L \Rightarrow y$ if $\vdash_{\texttt{alg}} \texttt{VC}(S, \phi_{pre}, (\bigwedge_{q \in N} q) \Rightarrow y)$. When we find a node N that is a solution, we delete the subtree rooted at N from the lattice, since any node in the subtree is a "weaker" solution than N (i.e. for any node M in the subtree under N, we have $\vdash_{\texttt{alg}} (\bigwedge_{q \in M} q) \Rightarrow (\bigwedge_{q' \in N} q')$, and since $\vdash_{\texttt{alg}} (\bigwedge_{q' \in N} q') \Rightarrow y$, we already have $\vdash_{\texttt{alg}} (\bigwedge_{q \in M} q) \Rightarrow y$). Moreover, we also prune any subtree rooted at a node (including the node itself) whose size is greater than c (since the current search space is $\mathsf{DNF}(c, d, Q)$, we need not consider lattice nodes with more than c elements). Let $\mathcal{N}$ be the set of lattice nodes that are found as solutions. We assign $\phi_L = \bigvee_{N \in \mathcal{N}} (\bigwedge_{q \in N} q)$. If $\mathcal{N} = \{\}$, the algorithm fails to infer p's knowledge of y (under input values of c and d). Construction of ϕ_L makes $O(|Q|^c)$ queries to the SMT solver.

Construction of ϕ_R. To construct ϕ_R (`CFormulaR` in Figure 4.9), we consider all possible $(q_1, \ldots, q_d) \in (Q \times_1 Q \cdots \times_{d-1} Q)$, and collect all such tuples that form a *solution* to $y \Rightarrow \phi_R$. $(q_1, \ldots, q_d)$ is a *solution* to $y \Rightarrow \phi_R$ if $\vdash_{\texttt{pre}} \texttt{VC}(S, \phi_{pre}, y \Rightarrow \bigvee_{i=1}^{d} q_i)$. Let $\mathcal{N}$ be the set of such solutions. Then, we assign $\phi_R = \bigwedge_{(q_1, \ldots, q_d) \in \mathcal{N}} (\bigvee_{i=1}^{d} q_i)$. If $\mathcal{N} = \{\}$, the algorithm fails to infer P's knowledge of y (under input values of c and d). Construction of ϕ_R makes $O(|Q|^d)$ queries to the SMT solver.

Construction of ϕ. We now check that $\phi_R \Rightarrow \phi_L$ is valid at the end of the program using the formulae for ϕ_L and ϕ_R constructed above (`CFormula` in Figure 4.9). If it is, y is known to p using the formula ϕ_R, otherwise our algorithm returns y is not known to p (under input values of c and d).

Constructive knowledge inference for integer variables. For integer variables in the

```
1  ConstructKnowledgeI(S, φ_pre)
2    for each party p
3      let {x_i}^{i∈1...n} be input and output variables of p.
4      for each integer program variable y
5        let a_i, i ∈ 1...n be n integer unknowns.
6        φ := −y + Σ_{i=1}^{n} a_i x_i ≥ 0 ∧ y + Σ_{i=1}^{n} − a_i x_i ≥ 0;
7        verify φ at the end of S.
8        if verification fails
9          output y is not known to p.
10       else
11         output y is known to p by Σ_{i=1}^{n} a_i x_i.
```

Figure 4.10: Constructive knowledge inference for integer variables

program S, constructive knowledge inference algorithm is shown in Figure 4.10. To verify ϕ on line 6, we use the algorithm given by Gulwani et al. [38]. Their algorithm uses Farka's lemma to convert ϕ into SAT solver constraints, the solution of which returns a solution for the template unknowns a_i s.t. ϕ holds true at the end of S, and thus, $y = \sum_{i=1}^{n} a_i x_i$.

We state soundness theorems for constructive knowledge inference algorithms as follows:

Theorem 15 (Soundness of Constructive Knowledge Inference)**.** *Let S be a secure computation program with precondition ϕ_{pre}. If `ConstructKnowledgeB`(S, ϕ_{pre}) (Figure 4.8) and `ConstructKnowledgeI`(S, ϕ_{pre}) (Figure 4.10) output variable y is known to party p, then $\mathfrak{K}(S, p, y)$.*

Proof. If the algorithms infer $y = \phi$ for a party p, then $y = \phi$ is valid at the end of S. Moreover, since only variables in ϕ are variables from $I \cup O$ (input and output variables of p), p knows y by Definition 10. □

Moreover, constructive knowledge inference algorithms are also complete, provided

a solution exists in the template form they consider.

Theorem 16 (Completeness of Constructive Knowledge Inference for Boolean Variables)**.** *Let S be a secure computation program. For a party p, let Q be a set of predicates, where the only variables appearing in each predicate in Q are input and output variables of p. Let y be a boolean program variable in S. If $\exists \phi$ s.t. $x = \phi$ at the end of S, and ϕ is in $\mathsf{DNF}(c, d, Q)$ form, for some values of (c, d), then `CFormula`(y, c, d, Q) returns a solution (and not `failure`).*

Proof. We give an outline for $(c = 2, d = 2)$, the proof for general case follows similarly. Let $y = \phi$ at the end of S s.t. ϕ is in $\mathsf{DNF}(c, d, Q)$ form. Then, for some $q_1, q_2, q_3, q_4 \in Q$, $y = (q_1 \wedge q_2) \vee (q_3 \wedge q_4)$, equivalently, $y = (q_1 \vee q_3) \wedge (q_1 \vee q_4) \wedge (q_2 \vee q_3) \wedge (q_2 \vee q_4)$. Since `CFormulaR` considers all elements in $Q \times Q$, it would construct $\phi_R = (q_1 \vee q_3) \wedge (q_1 \vee q_4) \wedge (q_2 \vee q_3) \wedge (q_2 \vee q_4) \wedge \phi'$, for some ϕ' (possibly just true). On the other hand, since `CFormulaL` considers all lattice nodes up to size c, it would construct $\phi_L = (q_1 \wedge q_2) \vee (q_3 \wedge q_4) \vee \phi''$ for some ϕ'' (possibly just false). We can see that $\phi_R \Rightarrow \phi_L$, and hence `CFormula` returns ϕ_R. □

The following theorem of completeness for integer variables follows from the completeness of the algorithm by Gulwani et al. [38][4].

Theorem 17 (Completeness of Constructive Knowledge Inference for Integer Variables)**.** *Let S be a secure computation program. Let y be an integer variable in S. For a party p, let $\{x_i\}^{i \in 1 \ldots n}$ be the set of input and output variables of p. If $\exists a_i, i \in 1 \ldots n$ s.t. $y = \sum_{i=1}^{n} a_i x_i$*

[4] Similar to the restriction in [38], the theorem holds if checking the invariant $y = \phi$ does not require integral reasoning.

at the end of S, then `ConstructKnowledgeI`(S, ϕ_{pre}) *(Figure 4.10) outputs y is known to p.*

Proof. Follows from the completeness of [38]. □

4.3 Discussion

This section considers some aspects of our approach, including the possible use of type-based information flow analysis for knowledge inference, and the application of knowledge inference to allowing MPC computations with loops.

Applying information flow analysis. In the limit, we can use a grossly over-approximating language-based information flow analysis [63] for knowledge inference. Following the formulation relating knowledge inference to delimited release given above, we can label each of party p's input variables as L and all other input variables as H, restricting valid flows in the program as $L \sqsubseteq H$ as usual, while explicitly declassifying the final output when it is returned. Then we can do type inference [64, 65] to determine whether any unlabeled, local variables can safely be given label L, and if so then we know these can be determined solely from knowledge of p's inputs.

Such a type-based analysis is less precise than the semantic analysis we have given to this point. It cannot, for example, infer the knowledge of `x1` and `x2` in the median example. As soon as it sees `x1` = `a1` $\leq$ `b1` in the median example, it assumes that there is information flow from both `a1` and `b1` to `x1`, and hence, neither Alice nor Bob can determine `x1` alone.

However, it is far less expensive than a semantic analysis, and there are some useful examples where such an analysis is enough to establish knowledge facts. Consider the

```
1  ## variables  with suffix  A are  Alice's  inputs,
2  ## with  suffix  B are  Bob's.  yd  is  known to both.
3  int  lot_size  (int  fvA,  int  cA,  int  hvA, int  fbB,  int  hbB, int  yd)
4
5    int  a,  b,  c,  d,  e,  f,  g,  h,  i;
6
7    a = 2 * yd;
8    b = a * fvA;
9    c = yd /  cA;
10   d = c * hvA;
11
12   e = 2 * yd;
13   f  = e * fbB;
14
15   g = f  + b;
16   h = hbB + d;
17   i  = g /  h;
18
19   return  sqrt(i);  ## integer  square  root
```

Figure 4.11: Joint economic lot size example

joint economic lot size computation example from Kerschbaum [5], shown in Figure 4.11. The program computes an order quantity (or lot size) between a buyer (Bob) and vendor (Alice). The buyer's private inputs include the holding cost per item (hbB) and the fixed ordering costs per order (fbB). The vendor's private inputs include the holding cost per item (hvA), the fixed setup costs per order (fvA), and the capacity (cA). Both parties know the yearly demand of the buyer (yd). For vendor Alice, if we label yd, fvA, cA, hvA as L, fbB, hbB as H, and do type inference in an information flow type system, it can infer that a, b, c, d can have label L and are thus known to Alice. Similarly, it can infer that e, f are known to Bob. Using these knowledge facts, the MPC protocol can be optimized to compute lines 7-10 locally on Alice's host, and lines 12-13 locally on Bob's host, leaving only lines 15-17, and 19 to be computed securely.

Adding loops to the programs. MPC programs do not admit loop constructs because in many cases the execution of a loop, specifically the number of times it iterates, can potentially reveal information about parties' input values beyond what is revealed by the output. However, if we can prove that using their own input and output variables, all parties in the secure computation can infer the number of loop iterations, we can allow MPC programs to have loops in them, without compromising security. For example, for a loop .. i = 0; while(i $<$ n) { ... ++i;} .. , if n is already known to all the parties in the computation, they can infer the number of loop iterations, and hence running this loop in MPC does not compromise security.

Constructive knowledge inference can be useful in this situation. In particular, we can use it to infer loop invariants in terms of known variables for a party, and if we can do so for all the parties, we can admit the loop in MPC.

4.4 Experiments

In this section, we present an experimental evaluation of our approach. We provide performance measurements for our algorithms on several example programs.

4.4.1 Implementation

We present evaluation of three implementations of our algorithms – two for the knowledge inference algorithm from Figure 4.7 that handle linear and non-linear arithmetic respectively, and one for the constructive knowledge inference algorithm from Fig-

ures 4.8 and 4.9. [5]

Convex polyhedra based implementation. We have implemented the knowledge inference algorithm from Figure 4.7 using the polyhedra powerset domain as implemented in Parma Polyhedra Library (PPL, v0.11.2) [66]. This approach represents the program postcondition, ϕ_{post}, as a set of convex polyhedra (each of which is a conjunction of linear inequalities), interpreted over real-valued variables. We use polyhedra in the implementation to avoid reasoning about integers as much as possible. To verify the validity of ϕ (line 9 in Figure 4.7), we check if the negation of ϕ has an integer solution. This corresponds to checking, for every polyhedron/disjunct φ in $\phi_{post} \wedge \phi'_{post} \wedge \phi_k$, that the formulae $\varphi \wedge (y > y')$ and $\varphi \wedge (y < y')$ define convex regions with no real points (quick check) and no integer points (slower check). If so, ϕ is valid. This implementation only handles programs that use linear arithmetic.

Bitvectors based implementation. Our second implementation of the algorithm from Figure 4.7 uses a bitvector representation of program variables via the Simple Theorem Prover [62] (STP, revision 1671). This implementation handles non-linear arithmetic. It represents formulae (postcondition, ϕ) using logical and arithmetic expressions over fixed-width bit vectors. The validity of ϕ is checked using STP. In addition, STP allows us to construct formulas that relate individual bits of the integer variables, which means we can construct for every $1 \leq i \leq \mathbf{x}$ (for bit width $\mathbf{x}$) the formula $\phi_{post} \wedge \phi'_{post} \wedge \phi_k \Rightarrow (y_i = y'_i)$ where y_i designates bit i of variable y. Checking validity of such formulas lets us conclude that parties can potentially infer individual bits, even if they cannot infer whole variables.

[5] The implementation and evaluation of the non-constructive knowledge inference algorithm was done by Piotr Mardziel, who was also a co-author on the related PLAS 2013 publication [33].

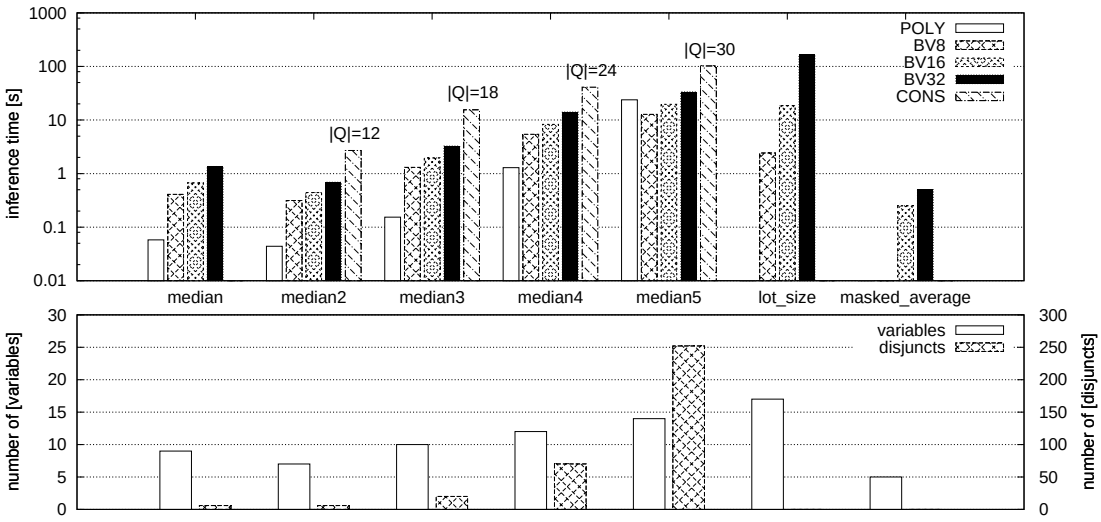

Figure 4.12: Results

Constructive algorithm for boolean variables. We have implemented the constructive knowledge inference algorithm for boolean variables (Figure 4.8 and Figure 4.9) using the LLVM compiler infrastructure [67]. We use the Z3 SMT solver [40] for the validity queries.

4.4.2 Results

We have conducted the experiments on a Mac Pro with two 2.26 GHz quad-core Xeon processors, 16 GB RAM, and running OS X v10.8. The results are in Figure 4.12.

The top chart shows time taken (in log-scale) by our three implementations, **POLY** (convex polyhedra based), **BVx** (bit-vector based, for **x** as 8, 16, and 32), and **CONS** (constructive algorithm), on several example programs (discussed later). We evaluated **BVx** on all programs, whereas other implementations only on the (linear) median examples. In all programs, we try to infer all variables for both the parties. Additionally, in the case of

BVx, we also try to infer every intermediate bit.

The bottom chart provides some characteristics of the test cases that contribute to the running times above: the total number of variables in the test cases, and for the linear programs, the number of convex disjuncts in program postcondition (see **POLY** implementation description).

Median example. We consider the joint median computation for 2, 3, 4, and 5 inputs per party. Unsurprisingly, the time taken by non-constructive implementations increases with the number of inputs. **POLY** is especially susceptible to the large number of disjuncts in the program postcondition (due to the large number of paths), taking around 24 seconds for analyzing `median5`, up from as little as 0.044 seconds for analyzing `median2`.

For **CONS**, we consider the set Q for Alice as $\{\mathsf{m} \odot a_i\}$ and for Bob as $\{\mathsf{m} \odot b_i\}$, where $\odot \in \{<, \leq, >, \geq, =, \neq\}$, and a_i and b_i range over inputs of Alice and Bob respectively. We used $(c = 2, d = 2)$ for all input sizes. It is able to infer knowledge of all comparisons for all the median programs. However, as the number of candidate predicates ($|Q|$) increases, the algorithm takes more time. For median with 4 inputs, for example, $|Q_{\text{Alice}}| = |Q_{\text{Bob}}| = 24$, and it takes $\sim$41 seconds to infer all the variables for both parties, as compared to $\sim$2 seconds in the case of 2 inputs per party and $|Q_{\text{Alice}}| = |Q_{\text{Bob}}| = 12$.

We note that at present, our implementation does not aggressively optimize the use of the SMT solver (like caching query responses etc.) that can potentially bring down the inference time since there are lots of redundant validity queries. Moreover, the **CONS** implementation computes ς for every to-be-inferred variable, something that can be optimized as well.

Lot size example. The joint computation of economic lot size in Figure 4.11 is a

non-linear arithmetic example. As described in Section 4.3, information flow analysis infers that Alice knows `a, b, c, d` and Bob knows `e, f`. Using **BVx**, for **x** as 8, 16, and 32, we infer the same conclusions. In addition, various bits of some other variables are inferred. For example, Alice knows bit 1 of `f` and `g`, while Bob knows bit 1 of `b` and `g`. These are due to the multiplications by 2 on lines 9 and 15, resulting in null bits of lowest order. The performance of **BVx** for this test case naturally decreases as **x** is increased. For **x** as 8, the analysis takes around 2.4 seconds, while for **x** as 32, it takes 165 seconds. Note that a significant portion of this additional time is spent checking a much larger number of bits for partial inference (when complete variables cannot be inferred).

```
1  ## assume 0 ≤ a, b < 0x0fff
2  int masked_average (int a, int b)
3    int sum = a + b;
4    int avg = sum / 2;
5    return (avg & 0xfff0 );
```

Figure 4.13: Masked average example

Masked average example. Our final example serves to better demonstrate the inference of bits of variables that cannot be inferred completely. Inference on the scale of bits lets us determine bit-width requirements of a circuit implementing some computation, as well as determine which bits can be revealed ahead of time due to the output of the computation. Consider a `masked_average` function in Figure 4.13. The function outputs the high-order bits of the average of two 16 (or 32) bit inputs, which are assumed to be 12 bits big. **BVx** implementations for **x** as 16 and 32, analyzes this function in 0.25 and 0.50 seconds respectively.

If we only consider the input assumptions (view the program as outputting nothing),

then both parties can infer the null values of sum at bits 14-16 and of avg at bits 13-16 . Additionally, since the function returns all but the lower 4 bits of the average, knowledge inference lets us conclude that bits 6-13 of sum and bits 5-12 of avg can be inferred given the output. An optimized circuit for this function would (a) reduce the size of sum and avg to 13 and 12 bits respectively, and (b) reveal bits 6-13 of sum after computing it, so that the final division circuit can be performed with just 5 bits.

4.5 Concluding remarks

This chapter has presented algorithms for knowledge inference in MPC programs, that can be used to optimize a monolithic MPC program into a mixed-mode MPC program, with formal guarantee that the mixed-mode version does not reveal more than the monolithic version. Such an optimization can result in big performance gains in some cases, as we have demonstrated in previous chapters for the PSI program and the joint median computation program.

We formalized the notion of knowledge in an MPC program, and the problem of knowledge inference that asks whether a party knows a variable in the MPC program. We formulated a related constructive knowledge inference problem, that in addition demands a witness of a party's knowledge of a variable. We then presented algorithms for solving the knowledge inference and constructive knowledge inference problems, formalized them, and proved their soundness and (conditional) completeness. We also presented an implementation and empirical evaluation of our algorithms.

Chapter 5: Related Work

5.1 Circuit libraries

VMCrypt [12] and FastGC [13] are frameworks for building and executing Yao's garbled circuits [1] directly. They employ several techniques, such as free XOR gates [68], pipelined execution, and so on, that improve the running time and memory requirements of the garbled circuits protocol.

Mood et al. [14] port the garbled circuits protocol execution to mobile phones. They control the memory requirements by using a novel intermediate language, called PAL. Circuits are generated from PAL by using a database of pre-generated circuits matching instructions to their circuit representations. They port the Fairplay compiler [15] to Android, and generate circuits that were previously infeasible to create in the mobile environment. Huang et al. [69] also demonstrate MPC applications, such as common contacts, running on the smartphones. They use custom, hand-optimized circuits for this purpose.

While these frameworks have presented generic techniques that make Yao's protocol practical even for larger circuits, it is arguably hard to program large MPC application directly using these libraries. As such, they can be easily used as a backend for Wysteria and Wys*.

5.2 High-level DSLs for MPC

Fairplay [15] was the first *high-level* MPC DSL. It 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. [16] developed a compiler for programs with similarly specified entry points, but written in (a subset of) ANSI C. The TASTY compiler produces secure computations that may combine homomorphic encryption and garbled circuits [70]. Its input language, TASTYL, requires explicit specification of communication between parties, as well as the means of secure computation. More recently, OblivC [71] is an extension to C for two-party MPC that annotates variables and conditionals with an obliv qualifier to identify private inputs; these programs are compiled by source-to-source translation. All these languages are however limited to two-parties, have limited support for mixed-mode, and do not have a formalized specification.

Laud et al. [21] present a DSL for programming low-level MPC protocols related to efficient integer and floating-point operations. They formalize their language, type system, and semantics. The DSLs that we have presented in this dissertation are meant for higher-level applications, and it should be possible to use the techniques from Laud et al. in the backend for such applications.

FairplayMP [18] supports $n > 2$ parties. Its programs are similar to those of Fair-Play, but now the entry point can contain many player arguments, including arrays of players, where the size of the array is statically known. The wire bundles in Wysteria and Wys* 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 and Wys*, these subsets can themselves be treated as variable.

5.2.1 Support for mixed-mode computations

L1 [34] 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).

Kreuter et al. [17] present PCF, a circuit format language for expressing mixed-mode secure computations for two parties. They implement an interpreter to load and execute PCF programs. Their interpreter uses online circuit compression and lazy gate generation for more efficient circuit compilation. However, compared to our DSLs, it is (by design) very low level, in that it lacks abstractions for supporting multiple parties, as well as a formal semantics.

SMCL [35] is a language for secure computations involving a replicated client and a shared "server" which represents secure multiparty computations. Our DSLs are less rigid in their 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 [72], reflects the SMCL computational model as a library/DSL in Python.

Liu et al. [73] define a typed intermediate language for mixed-mode SMC protocols. They support stateful secure computations using an ORAM-based [74] backend. They build a compiler that aims to provide instruction-trace obliviousness [75] and memory-trace obliviousness [76] properties. Their source programs do not have computation mode annotations. Instead the programs contain simple annotations identifying the private data of the parties, while the compiler performs an information-flow based analysis to assign code blocks to either local computations (Par blocks) or secure computations (Sec blocks).

However, such an information-flow based analysis is often inadequate in identifying the optimal placement of secure blocks. For example, their type system would not be able to identify that in the joint median computation example, only the comparisons need to be done securely. In terms of language design, their language is simplistic as compared to Wysteria, e.g. it lacks function abstractions and is limited to two parties only. Another implication of their design choice is that the programmer has to explicitly put declassify annotations for the secure block outputs.

On the other hand, in Wysteria, the programmer has to provide the computation mode annotations and the secure blocks outputs are implicitly declassified. Arguably, Wysteria provides the programmers more flexibility in terms of secure blocks placement, allowing them to express any well-typed code structure. In future, we would like to enhance Wysteria with computation mode inference based on the analyses developed in

Chapter 4. Wysteria also does not have an ORAM backend and does not support stateful secure computations (the type system prohibits state manipulation in secure blocks). If we were to provide oblivious array accesses without ORAM, the complexity for each access would be $O(\log n)$, which would clearly not scale. But Wysteria does provide the instruction-trace obliviousness property – the circuit compiler hoists the then and else branches of an if conditional in a secure block, using a multiplexer circuit to select the branch.

5.2.2 DSLs for cloud-based MPC

Another line of research in MPCs 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 the 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. [77] present a table-lookup based optimization for such MPC protocols, that aims at minimizing the cost incurred by expensive operations such as multiplication and network communication between servers. Mitchell et al. [78] 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 dissertation, we can express more general mixed-mode MPCs.

5.2.3 Other MPC languages

Kerschbaum et al. [79] explore automatic selection of mixed protocols consisting of garbled circuits and homomorphic encryption. They provide an extended cost model for mixed protocols, and present two algorithms for automatically selecting mixed protocols with (near-) optimal performance based on this model. Jif/Split enables writing multi-party computations in Java as (conceptually) single-threaded programs [80]. 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 does not have support for *collaborative* computations, such as two parties looping in-parallel performing a secure computation every loop iteration. It depends on actual trusted third parties, and lacks language abstractions and run-time techniques for employing secure computations without a trusted third party. On the other hand, Jif/Split automatically partitions the program, while in our DSLs, the programmer has to provide computation mode annotations.

5.3 Crypto DSLs

Similar to WYSTERIA, researchers have proposed other DSLs for making cryptography more accessible to programmers. $\lambda\bullet$ is a DSL for generic authenticated data-structures (ADS) [81]. ADS aim to minimize the amount of client storage by allowing the client to

store a hash digest of their data, while the actual data is stored on a server. A *prover* and *verifier* protocol between client and server guarantees that the client can query the data, with formal guarantees that the server sends the correct result. A $\lambda\bullet$ source program, like WYSTERIA, is for the most part a typical functional language program except for certain types designated as *authenticated types*, and coercions *auth* and *unauth*. The compiler then compiles a function f that uses authenticated types to two variations f_P and f_V for prover and verifier respectively, while the type system guarantees the correctness and security of the generated code. This is similar to WYSTERIA where the source program contains computation mode annotations, and the interpreter interprets them accordingly.

ZQL [82] is a query language for expressing simple computations over private data in scenarios such as smart meters. Given a ZQL source program, the compiler produces code to perform the computation, and verify the correctness of its result. Behind the scenes, it generates zero-knowledge protocols that guarantee both the privacy of the data and correctness of the computation result. Pinocchio [83] is a framework for efficiently verifying the results of outsourced (such as to the cloud) computations. It provides a compiler that converts C code into verifiable arithmetic circuits, and tools for running the actual verification protocol. As with WYSTERIA, these languages make cryptographic protocols accessible to the application programmers with no cryptography expertise.

5.4 Verification of source MPC programs

None of the DSLs that we mentioned above provides formal reasoning about the correctness and the security properties of the source MPC programs, or a verified toolchain.

While the verification of the underlying crypto protocols has received some attention [84], the verification of the MPC source programs has remained largely unexplored. The only previous work that we know of is Backes et al. [85] who devise an applied pi-calculus [86] based abstraction for MPC, and use it for formal verification. For an auction protocol that computes the min function, their abstraction comprises about 1400 lines of code. In contrast, Wys* permits direct verification of higher-level MPC source programs and provides a verified toolchain.

5.5 DSL implementation strategies

Similar to Wys*, other researchers have also embedded DSLs in verification-oriented host languages. Chlipala [87] presents design and implementation of an extensible macro-system in Coq [88], intended for verified low-level programming. In his system, every macro, in addition to a compilation rule to the Bedrock IL, comes with a proof rule, and the system requires the macro designers to prove that the macro definition satisfies the proof rule. The macros can then be assembled to build applications that can be verified by modular composition of the macro proof rules. The semantics of the BedRock IL is trusted, but small enough to be audited manually.

Similar to the Bedrock IL, Wys* defines the deep embedding of Wysteria AST in F* and formalizes its semantics. However, instead of taking the macro approach, we have taken the API approach, with modifications to the F* compiler for producing those ASTs from the F* source. Also, the API specifications (pre- and post-conditions) are trusted in Wys*. As we mentioned in Section 3.3.3, verifying these is future work. Wys*

additionally has a notion of *multi-party* with distributed target semantics, while Bedrock IL has a straightforward compilation to the assembly code. Wys* API (and formal semantics) also provides a notion of *observable traces* that can be used to verify the security properties of the MPC programs.

Wys*'s language-integrated syntax bears relation to the approach taken in LINQ [89], which embeds a query language in normal C# programs, and implements these programs by extracting the query syntax tree and passing it to a custom-provider that implements a particular backend.

5.6 Knowledge inference for MPC

Huang et al. [13, 69] identify the need for knowledge based MPC optimizations, while leaving the automatic generation of optimized MPC protocols to future work. Kerschbaum [5] solves the knowledge inference problem using a custom program analysis based on epistemic modal logic inference rules. He shows that his approach works on the median example (Section 4.1), and the lot size computation example (Figure 4.11). Our work in Chapter 4 can be viewed as a generalization and improvement of his approach, making several advances. First, we formally define the notion of knowledge in SMC, and the problem of knowledge inference. In addition, we also formally define and solve the related constructive knowledge inference problem. Second, we prove our algorithms are sound and (relatively) complete. Moreover, our algorithms are built on top of the SMT solvers, thus leveraging recent advances in SMT solving techniques. Indeed, we present experimental measurements to characterize the performance of our algorithms, while he

does not.

5.6.1 Self-composition and noninterference

This section considers some aspects of our approach to knowledge inference, including the relationship of knowledge inference to the property of *delimited release* [51], the relationship of constructive knowledge inference to *required release* [61], and the effect of using a different program analysis to determine a program's final states.

Relating knowledge inference to noninterference. As mentioned in Section 4.1.1, the knowledge inference problem bears some resemblance to the problem of proving noninterference, as evidenced by the similarity of our use of self-composition with its previous use in proving noninterference [60]. More precisely, knowledge inference is closely related Sabelfeld and Myers' *delimited release* [51] property. Next we define delimited release, and then show how a method for proving a program satisfies delimited release can be applied to knowledge inference.

In the setting of normal delimited release, we suppose there exists a *security labeling* Γ, which maps each program variable in S to one of two security labels, L (low) and H (high). We say that memories σ_1 and σ_2 are *low-equivalent*, written $\sigma_1 \sim_\Gamma \sigma_2$, if $\sigma_1(x) = \sigma_2(x)$ for all variables x such that $\Gamma(x) = L$. We also suppose that the program S may contain expressions `declassify(`e`)`, which signal that e's security label should be considered L, even if its contents may otherwise suggest its label should be H. (In an MPC, we can think of the output as being declassified; e.g., in the median example, we would change the last line to be `return declassify(m)`.) We say that S enjoys

delimited release with respect to Γ *iff* for all memories $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ such that if $\sigma_1 \sim_\Gamma \sigma_2$, and $\langle S, \sigma_1 \rangle \Downarrow \sigma_1'$ and $\langle S, \sigma_2 \rangle \Downarrow \sigma_2'$ where $\langle \sigma_1', e_i \rangle \Downarrow v \Leftrightarrow \langle \sigma_2', e_i \rangle \Downarrow v$ for some v for all declassification expressions $e_i \in S$, then $\sigma_1' \sim_\Gamma \sigma_2'$. In short, all pairs of program evaluations that agree on the results of declassified expressions e_i should also agree on other low-visible outputs. Satisfying this condition means that nothing is leaked via low outputs beyond what the declassification expressions already reveal.

We can describe knowledge inference for p in terms of delimited release. Let Γ_p map p-visible variables to L and all remaining variables to H. The set of declassification expressions is the set of output variables (e.g., `m` in the median example). Now, to see whether local variable y can be inferred by p, we simply label y with L and see whether S still satisfies delimited release. If so, revealing y to p provides no additional information.

The self-composition algorithm described in Section 4.1.1 is basically checking delimited release. For example, consider the condition presented for the median example:

$$\phi_{post}^{sc} \wedge (\mathsf{a1} = \mathsf{a1'} \wedge \mathsf{a2} = \mathsf{a2'} \wedge \mathsf{m} = \mathsf{m'}) \Rightarrow (\mathsf{x1} = \mathsf{x1'})$$

The ϕ_{post}^{sc} part captures the semantics of the two executions. The next two equalities are establishing $\sigma_1 \sim_\Gamma \sigma_2$, since they require Alice's two input variables to be equal. The third equality establishes the equality of the declassified output variable `m`. The final equality `x1 = x1'` establishes that $\sigma_1' \sim_\Gamma \sigma_2'$ (where the other low-security variables are known to be equal by virtue of them appearing to the left of the implication, and the program respecting single-assignment semantics).

Constructive knowledge inference is related to *required release* [61]. In this setting, a program S satisfies required release of an input expression e to user p using output

expression F if p can evaluate F (i.e. F only uses variables visible to p) and F evaluates to the same value as e, i.e. for all final states σ of S, $\langle \sigma, e \rangle \Downarrow v \Leftrightarrow \langle |\sigma|_p, F \rangle \Downarrow v$ where $|\sigma|_p$ denotes the state visible to p. The problem of constructive knowledge inference then is to infer the function F for a party p and program variable y such that the program S satisfies required release of y to party p using F.

As far as we are aware, we are the first to observe that knowledge inference can be reduced to the question of deciding delimited release [51], and we are the first to show how to decide this property using self-composition. Moreover, in the form of constructive knowledge inference, we are the first to propose inference algorithms for inferring the output function to decide the problem of required release [61].

Alternatives to $\varsigma(S, \phi)$. The role of $\varsigma(S, \phi)$ (Figure 4.5) is to provide a sound approximation of final states of executing the program S starting from an initial state that satisfies ϕ. We can use other program analyses to get such an approximation. In Section 4.1.1 we used symbolic execution for this purpose; for our language (Figure 4.3), which lacks loops, symbolic execution generates equivalent formula as ς.

While $\varsigma(S, \phi)$ as defined in Figure 4.5 provides a complete approximation of final program states (Theorem 12), for large programs the formula can become prohibitively large. In such cases, we can always trade completeness of the approximation, and use abstract interpretation [90] to provide a sound approximation. With such analyses, our knowledge inference algorithms are still sound, in that if they output y is known to p then $\mathfrak{K}(S, p, y)$, but they lose completeness.

Using information flow. Inferring local variables known to p via information flow analysis, as described earlier, is similar to the splitting algorithm employed by Jif/Split [65],

which partitions a program to run on multiple hosts. As mentioned before, Jif/Split does not employ MPCs, but rather relies on trusted third parties, and employs a simple syntactic algorithm incapable of inferring deeper relationships, e.g., it would not be able to deduce that the parties can infer the two booleans in the median example

5.6.2 Template based program verification

Our constructive knowledge inference algorithms (Figure 4.8, Figure 4.9, and Figure 4.10) are inspired by template driven program verification techniques [38, 39]. However, our algorithms take advantage of features specific to our problem. Our templates, instead of having arbitrary structure, have restricted form of $\phi_L \Rightarrow y \wedge y \Rightarrow \phi_R$. For *negative* variables (i.e., variables on the left side of an implication), independent of c and d, we never have to consider more than one lattice, since we always have only one template variable on the left of implication. Second, as mentioned in the inference of ϕ_L earlier, in addition to pruning the subtree of a solution node, we also prune subtrees whose root node has size greater than c. Finally, we infer ϕ_L independent of ϕ_R, i.e. solve $\phi_L \Rightarrow y$ separately from $y \Rightarrow \phi_R$, which is different from [39], where *negative* variables are inferred for every permutation of *positive* variables (variables on the right side of an implication). Again, the simple structure of our templates enables us to do so.

Chapter 6: Looking back and going forward

MPC eliminates the need for a trusted third party and replaces it with a cryptographic protocol. It provides a general-purpose solution to address growing data privacy concerns – users do not have to trust a third-party with their data, instead they can retain control of their data while using MPC for computing useful information with other users. Cryptographic protocols for evaluating arbitrary functions as MPC have existed since the 1980s [1]. More recent developments have enabled the use of MPC in real-world applications [9, 10, 91].

6.1 Looking back

We began this dissertation with the vision of rich, practical, and provably secure MPC applications. To attain this goal, the advances in the MPC cryptographic protocols are necessary, but not sufficient – we also need to make MPC accessible to the application programmers in an easy-to-use manner. This dissertation has applied techniques from programming language design and program verification to do so.

We have followed a three pronged strategy. Firstly, we have designed high-level, modular abstractions for MPC that enjoy a conceptual single-threaded interpretation. We have embedded these abstractions inside a general purpose, verification-oriented program-

ming language, so that the application programmers can build upon them while proving useful correctness and security properties about their programs. Secondly, we have formally verified parts of our MPC toolchain, thereby considerably reducing the risk of security-critical bugs that can compromise the privacy of parties' data. Finally, we have designed algorithms to optimize monolithic MPC programs to mixed-mode MPC programs, with formal guarantees that the mixed-mode versions do not reveal more information about the parties' data than their monolithic counterparts.

6.1.1 Wysteria

Wysteria is the first mixed-mode MPC DSL that provides a formalized type system and a conceptual single-threaded semantics, with theorems that prove the type system to be sensible, and establish the correspondence between the single-threaded and the actual distributed semantics. Wysteria also provides high-level support for secret shares (secure state), while the Wysteria interpreter transparently handles the low-level cryptography and message passing details.

We have experimentally demonstrated that Wysteria makes significant advances towards our goal of practical and rich secure computations. We have built several novel MPC applications using Wysteria, including, for the first time, a card dealing application. Crucially, this application relies on multiple rounds of secure computations, that manipulate the deck of cards in the form of secure state. Wysteria's easy-to-use, high-level abstractions simplify the programming task, reducing the risks of programmer mistakes, while the Wysteria interpreter handles cryptography behind-the-scenes. Wysteria's

conceptual, single-threaded semantics also simplifies the reasoning about the programs.

6.1.2 WYS*

WYS* is the first language to provide formal verification capabilities for the mixed-mode MPC programs. It is also the first DSL to provide a partially verified MPC toolchain. WYS* enhances the WYSTERIA semantics with *observable traces*, with which the programmers can state and verify security properties of their programs. WYS* also enhances the usability of WYSTERIA – WYS* programs can directly use the standard language constructs and libraries from the host language F*.

WYS* makes advances towards more reliable and trustworthy MPCs. Using WYS*, we have shown that tricky MPC programs that use multiple rounds of secure computations and message exchanges, such as PSI and the joint median computation, can be formally verified for correctness and security. At the same time, we have used F*'s extraction facilities to extract an executable, verified interpreter for WYS* programs, that we have shown to be practical and usable.

6.1.3 Knowledge inference

Unlike previous approaches for knowledge inference for MPC programs, we have taken a more formal approach. We have formally defined the notion of knowledge, the knowledge inference problem, and a novel constructive notion of knowledge inference. We have then developed sound and (relatively) complete algorithms to solve these problems. We have also implemented and empirically evaluates these algorithms, showing them to

be practical.

6.2 Going forward

This dissertation has made important contributions towards the goal of rich, practical, and trustworthy MPC applications. Yet, we still have a way to go. Below we discuss limitations of our work and several avenues of future work.

Inferring computation mode annotations. Wysteria requires the programmer to annotate computation modes, i.e. the programmer needs to decide which parts of the code are executed locally and which parts are executed securely (the knowledge inference analyses that we developed in Chapter 4 can guide this decision). This can, sometimes, be onerous for the programmer. SCVM [73], on the other hand, only requires the programmers to annotate private data of the parties, while the compiler performs a lightweight, information-flow based analysis to allocate code blocks to local or secure computations. However, the information-flow based analysis can miss several opportunities for optimizations. For example, for the joint median computation example, their analysis would fail to identify that only the two comparisons need to be done securely. It would be an interesting future work to integrate our knowledge inference analysis directly into the compiler for inferring computation modes.

Support for stateful secure blocks. Our current Wysteria development lacks the support of stateful secure computations, the Wysteria type system forbids updating state inside secure blocks. Recent work [73, 92] has used ORAM-based backends for supporting stateful MPCs. One of the key properties that such frameworks aim for is that of

memory trace obliviousness [76]. Extending Wysteria with an ORAM backend would enable writing, for example, big-data applications. Interestingly, we can perhaps provide the memory traces in the Wys* API (adding to the observation traces), and the programmers would be able to prove that their programs enjoy the memory trace obliviousness property, rather than rely on a one-size-fits-all type system.

Supporting the malicious adversary threat model. Our work assumes a semi-honest adversary, i.e. the parties are honest (they follow the prescribed protocol) but curious (they want to learn about other parties' data). However, in some settings this model may not be applicable. One difficulty in extending mixed-mode MPCs to the malicious setting is that parties would need to be able to verify the local computations of other parties (secure computation protocols for malicious settings are already known [93, 94]). One possibility is to leverage recent advances in verifiable computation [83], where with every local computation, a party generates a proof about the computation that other parties can verify. The practicality of such an approach could be an issue.

Eliminating trust from the API specification and the source semantics. As we noted previously, the Wys* single-threaded semantics that we have formalized in F*, is a trusted model of the embedded Wysteria API in F*. Although the API and the semantics are quite small and can be easily audited, there is still a chance of bugs. Formally connecting the two would require us to formalize the F*'s monadic type system, and connect the Wys* source semantics to the official F* semantics [37]. Our circuit library is also unverified. While verifying the circuit compiler that converts Wys* ASTs to boolean circuits seems straightforward, verifying the GMW protocol is an open and interesting problem. Previous work on verifying Yao's garbled circuits protocol [84] provides a starting point.

Security policies for MPC. Wys* makes *observation traces* accessible to programmers. As we have seen, these traces can be used to reason about the security of the MPC programs. In particular, we have seen the use of these traces in verifying the security of the mixed-mode versions of the median and PSI programs. However, users might want to verify other types of security policies for their MPC programs. For example, Mardziel et al. [95] present abstract interpretation-based analysis for quantifying adversary knowledge during an MPC. As they demonstrate, a party can use such an analysis to decide whether to participate in an MPC depending on if the participation would lead to unacceptable knowledge increase of the other parties. As another example, for the card dealing application, we might want to verify that the dealt cards are from a randomized deck (verifying freshness of the newly dealt card should be easier). In short, future work might investigate what other security policies users want to enforce for their MPC programs, and whether those policies can be formalized in our Wys* framework. Since such policies typically involve probabilistic reasoning, we would probably use the probabilistic relational version of F* [52].

Cloud computing. In addition to joint computations over private data, MPC also opens up new opportunities for cloud computations [78, 96]. Untrusted cloud servers can *obliviously* compute a function on a client's private data using secret shares-based MPC protocol (such as GMW) or homomorphic encryption. For better performance, the computations could involve both secret and public data, and secure and local computations (mixed-mode). We can express and verify such computations easily in Wysteria. One way would be to write the MPC program as a multi-party program between n cloud servers, with some secret-share inputs provided by the client. Moreover, the Wys* interpreter pro-

vides a generic way to run such computations on the cloud servers. However, to realize practical cloud applications, we would need to enhance the WYS* toolchain with support for state (so that applications such as those related to the databases can be programmed), and a way to bootstrap the computation using client data's input shares. To handle the malicious cloud servers, we would need enhancements similar to those for the malicious model (discussed above).

Building rich MPC applications. WYS* enables the programmers to write the MPC specific code *only* in WYS*, while the rest of the application code can be written in the host language F*. WYS* provides a Foreign Function Interface (FFI) to seamlessly integrate the MPC code and the rest of the application code. We have, for example, used standard datatypes such as lists, tuples, and options, and I/O libraries from F* in our example WYS* applications. However, we leave it for future work to build a more complete, rich MPC application using WYS*.

Some example applications that we could target are the email client and web browser plug-in based applications that we discussed at the beginning of Chapter 1. One way to go forward is to enhance F* with a JavaScript backend (previous F* version used to have it), and extract the WYS* interpreter and source programs to JavaScript. Building these applications might also require us to enhance the WYS* FFI interface to handle higher-order data, it is currently first-order. We can use standard techniques of function wrappers for this purpose [57]. Such JavaScript applications can then be easily executed using a web-browser, or a JavaScript engine such as node.js [97]. To run native C/C++ based implementations of the cryptographic protocols (such as GMW), we can look into the Native Client technology [98].

One potential concern with JavaScript-based MPC could be the presence of malicious scripts (such as ads) on the same page. Since all the scripts on a webpage run in the same security sandbox, such malicious scripts can easily access the private data in the MPC programs. To protect the MPC code and data from the malicious JavaScript context, we can use recently developed type-based sandboxing techniques [99, 100].

Appendix A: Formal definitions for WYSTERIA

In this appendix, we present several formal definitions for WYSTERIA that were elided from Chapter 2.

$\boxed{\Gamma \vdash v : \tau}$ *(Value typing with no mode)*

$$\textsc{TN-VAR}\ \frac{x : \tau \in \Gamma \qquad \Gamma \vdash \tau}{\Gamma \vdash x : \tau} \qquad \textsc{TN-UNIT}\ \frac{}{\Gamma \vdash () : \mathbf{unit}} \qquad \textsc{TN-INJ}\ \frac{\Gamma \vdash v : \tau_i \qquad j \in \{1,2\} \qquad \tau_j\ \texttt{IsFlat} \qquad \Gamma \vdash \tau_j}{\Gamma \vdash \mathbf{inj}_i\ v : \tau_1 + \tau_2}$$

$$\textsc{TN-PROD}\ \frac{\Gamma \vdash v_i : \tau_i}{\Gamma \vdash (v_1, v_2) : \tau_1 \times \tau_2} \qquad \textsc{TN-PRINC}\ \frac{}{\Gamma \vdash p : \mathbf{ps}\ (\nu = \{p\})} \qquad \textsc{TN-PSONE}\ \frac{\Gamma \vdash w : \mathbf{ps}\ (\mathbf{singl}(\nu))}{\Gamma \vdash \{w\} : \mathbf{ps}\ (\nu = \{w\})}$$

$$\textsc{TN-PSUNION}\ \frac{\Gamma \vdash w_i : \mathbf{ps}\ \phi_i}{\Gamma \vdash w_1 \cup w_2 : \mathbf{ps}\ (\nu = w_1 \cup w_2)} \qquad \textsc{TN-PSVAR}\ \frac{\Gamma \vdash x : \mathbf{ps}\ \phi}{\Gamma \vdash x : \mathbf{ps}\ (\nu = x)}$$

$$\textsc{TN-SUB}\ \frac{\Gamma \vdash v : \tau' \qquad \Gamma \vdash \tau' <: \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash v : \tau}$$

Figure A.1: Value typing with no mode

$\boxed{\Gamma \vdash M \rhd \epsilon}$ *(Effects delegation)*

$$\textsc{EFFDEL-EMPTY}\ \frac{}{\Gamma \vdash M \rhd \cdot} \qquad \textsc{EFFDEL-MODE}\ \frac{\Gamma \vdash M \rhd \epsilon \qquad \Gamma \vdash M \rhd N}{\Gamma \vdash M \rhd \epsilon, N}$$

Figure A.2: Effects delegation

$\boxed{\tau \texttt{ IsFO}}$ *(First order types)*

F-UNIT
$$\frac{}{\textbf{unit} \texttt{ IsFO}}$$

F-SUM
$$\frac{\tau_1 \texttt{ IsFO} \qquad \tau_2 \texttt{ IsFO}}{\tau_1 + \tau_2 \texttt{ IsFO}}$$

F-PROD
$$\frac{\tau_1 \texttt{ IsFO} \qquad \tau_2 \texttt{ IsFO}}{\tau_1 \times \tau_2 \texttt{ IsFO}}$$

F-PRINCS
$$\frac{}{\textbf{ps } \phi \texttt{ IsFO}}$$

F-WIRE
$$\frac{\tau \texttt{ IsFO}}{\textbf{W } w\ \tau \texttt{ IsFO}}$$

F-ARRAY
$$\frac{\tau \texttt{ IsFO}}{\textbf{Array } \tau \texttt{ IsFO}}$$

F-SHARE
$$\frac{\tau \texttt{ IsFO}}{\textbf{Sh } w\ \tau \texttt{ IsFO}}$$

$\boxed{\tau \texttt{ IsSecIn}}$ *(Valid input types for secure blocks)*

SIN-FO
$$\frac{\tau \texttt{ IsFO}}{\tau \texttt{ IsSecIn}}$$

SIN-ARROW
$$\frac{\tau_i \texttt{ IsSecIn}}{x{:}\tau_1 \xrightarrow{\epsilon} \tau_2 \texttt{ IsSecIn}}$$

$\boxed{\tau \texttt{ IsFlat}}$ *(Wire and Share free types)*

W-UNIT
$$\frac{}{\textbf{unit} \texttt{ IsFlat}}$$

W-SUM
$$\frac{\tau_1 \texttt{ IsFlat} \qquad \tau_2 \texttt{ IsFlat}}{\tau_1 + \tau_2 \texttt{ IsFlat}}$$

W-PROD
$$\frac{\tau_1 \texttt{ IsFlat} \qquad \tau_2 \texttt{ IsFlat}}{\tau_1 \times \tau_2 \texttt{ IsFlat}}$$

W-ARR
$$\frac{\tau_1 \texttt{ IsFlat} \qquad \tau_2 \texttt{ IsFlat}}{x{:}\tau_1 \xrightarrow{\epsilon} \tau_2 \texttt{ IsFlat}}$$

W-PRINCS
$$\frac{}{\textbf{ps } \phi \texttt{ IsFlat}}$$

W-ARRAY
$$\frac{\tau \texttt{ IsFlat}}{\textbf{Array } \tau \texttt{ IsFlat}}$$

Figure A.3: Auxiliary judgements used in the type system

$\boxed{\Gamma \vdash \epsilon}$ *(Well formed effect)*

$$\textsc{WFEFF-EMPTY}\ \frac{}{\Gamma \vdash \cdot} \qquad \textsc{WFEFF-MODE}\ \frac{\Gamma \vdash \epsilon \qquad \Gamma \vdash M}{\Gamma \vdash \epsilon, M}$$

$\boxed{\Gamma \vdash \phi}$ *(Well formed refinement)*

$$\textsc{WFREF-TRUE}\ \frac{}{\Gamma \vdash \mathsf{true}} \qquad \textsc{WFREF-SINGL}\ \frac{}{\Gamma \vdash \mathbf{singl}(\nu)} \qquad \textsc{WFREF-SUB}\ \frac{\Gamma \vdash w : \mathbf{ps}\ \phi}{\Gamma \vdash \nu \subseteq w} \qquad \textsc{WFREF-EQ}\ \frac{\Gamma \vdash w : \mathbf{ps}\ \phi}{\Gamma \vdash \nu = w} \qquad \textsc{WFREF-CONJ}\ \frac{\Gamma \vdash \phi_i}{\Gamma \vdash \phi_1 \wedge \phi_2}$$

$\boxed{\Gamma \vdash \tau}$ *(Well formed type)*

$$\textsc{WF-UNIT}\ \frac{}{\Gamma \vdash \mathbf{unit}} \qquad \textsc{WF-SUM}\ \frac{\Gamma \vdash \tau_i \qquad \tau_i\ \mathtt{IsFlat}}{\Gamma \vdash \tau_1 + \tau_2} \qquad \textsc{WF-PROD}\ \frac{\Gamma \vdash \tau_i}{\Gamma \vdash \tau_1 \times \tau_2} \qquad \textsc{WF-PRINC}\ \frac{\Gamma \vdash \phi}{\Gamma \vdash \mathbf{ps}\ \phi} \qquad \textsc{WF-ARROW}\ \frac{\Gamma \vdash \tau_1 \qquad \Gamma, x : \tau_1 \vdash \epsilon}{\Gamma, x : \tau_1 \vdash \tau_2}{\Gamma \vdash x{:}\tau_1 \xrightarrow{\epsilon} \tau_2}$$

$$\textsc{WF-WIRE}\ \frac{\Gamma \vdash w : \mathbf{ps}\ \phi \qquad \Gamma \vdash \tau \qquad \tau\ \mathtt{IsFlat}}{\Gamma \vdash \mathbf{W}\ w\ \tau} \qquad \textsc{WF-ARRAY}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathbf{Array}\ \tau} \qquad \textsc{WF-SHARE}\ \frac{\Gamma \vdash w : \mathbf{ps}\ \phi \qquad \Gamma \vdash \tau \qquad \tau\ \mathtt{IsFO} \qquad \tau\ \mathtt{IsFlat}}{\Gamma \vdash \mathbf{Sh}\ w\ \tau}$$

$\boxed{\Gamma \vdash M}$ *(Well formed place)*

$$\textsc{WFPL-TOP}\ \frac{}{\Gamma \vdash \top} \qquad \textsc{WFPL-OTHER}\ \frac{\Gamma \vdash w : \mathbf{ps}\ \phi}{\Gamma \vdash m(w)}$$

Figure A.4: Well-formedness judgements

$\boxed{\psi[\![v_1]\!]^\Gamma_M = v_2}$ *(Closing value)*

VL-VAR1

$$\frac{x \mapsto_N v \in \psi \qquad \Gamma \vdash N \rhd M}{\psi[\![x]\!]^\Gamma_M = v}$$

VL-VAR2

$$\frac{x \mapsto v \in \psi}{\psi[\![x]\!]^\Gamma_M = v}$$

VL-VAR3

$$\frac{x \notin \mathsf{dom}(\psi)}{\psi[\![x]\!]^\Gamma_M = x}$$

VL-UNIT

$$\frac{}{\psi[\![()]\!]^\Gamma_M = ()}$$

VL-PROD

$$\frac{}{\psi[\![(v_1, v_2)]\!]^\Gamma_M = (\psi[\![v_1]\!]^\Gamma_M, \psi[\![v_2]\!]^\Gamma_M)}$$

VL-INJ

$$\frac{}{\psi[\![\mathbf{inj}_i\ v]\!]^\Gamma_M = \mathbf{inj}_i\ (\psi[\![v]\!]^\Gamma_M)}$$

VL-PRINC

$$\frac{}{\psi[\![p]\!]^\Gamma_M = p}$$

VL-SINGL

$$\frac{}{\psi[\![\{v\}]\!]^\Gamma_M = \{\psi[\![v]\!]^\Gamma_M\}}$$

VL-UNION

$$\frac{}{\psi[\![v_1 \cup v_2]\!]^\Gamma_M = \psi[\![v_1]\!]^\Gamma_M \cup \psi[\![v_2]\!]^\Gamma_M}$$

$\boxed{\psi[\![v_1]\!] = v_2}$ *(Closing value)*

VLT-VAR1

$$\frac{x \mapsto v \in \psi}{\psi[\![x]\!] = v}$$

VLT-VAR2

$$\frac{x \notin \mathsf{dom}(\psi)}{\psi[\![x]\!] = x}$$

VLT-UNIT

$$\frac{}{\psi[\![()]\!] = ()}$$

VLT-PROD

$$\frac{}{\psi[\![(v_1, v_2)]\!] = (\psi[\![v_1]\!], \psi[\![v_2]\!])}$$

VLT-INJ

$$\frac{}{\psi[\![\mathbf{inj}_i\ v]\!] = \mathbf{inj}_i\ (\psi[\![v]\!])}$$

VLT-PRINC

$$\frac{}{\psi[\![p]\!] = p}$$

VLT-SINGL

$$\frac{}{\psi[\![\{v\}]\!] = \{\psi[\![v]\!]\}}$$

VLT-UNION

$$\frac{}{\psi[\![(v_1 \cup v_2)]\!] = (\psi[\![v_1]\!]) \cup (\psi[\![v_2]\!])}$$

$\boxed{\psi[\![\phi_1]\!] = \phi_2}$ *(Closing refinement)*

RL-TRUE

$$\frac{}{\psi[\![\mathsf{true}]\!] = \mathsf{true}}$$

RL-SINGL

$$\frac{}{\psi[\![\mathbf{singl}(\nu)]\!] = \mathbf{singl}(\nu)}$$

RL-SUB

$$\frac{}{\psi[\![\nu \subseteq w]\!] = \nu \subseteq \psi[\![w]\!]}$$

RL-EQ

$$\frac{}{\psi[\![\nu = w]\!] = \nu = \psi[\![w]\!]}$$

RL-CONJ

$$\frac{}{\psi[\![(\phi_1 \wedge \phi_2)]\!] = (\psi[\![\phi_1]\!]) \wedge (\psi[\![\phi_2]\!])}$$

Figure A.5: Environment closing judgements

$\boxed{\psi[\![\tau_1]\!] = \tau_2}$ *(Closing type)*

TL-UNIT

$$\psi[\![\mathbf{unit}]\!] = \mathbf{unit}$$

TL-SUM

$$\psi[\![(\tau_1 + \tau_2)]\!] = (\psi[\![\tau_1]\!]) + (\psi[\![\tau_2]\!])$$

TL-PROD

$$\psi[\![(\tau_1 \times \tau_2)]\!] = (\psi[\![\tau_1]\!]) \times (\psi[\![\tau_2]\!])$$

TL-PRINCS

$$\psi[\![(\mathbf{ps}\ \phi)]\!] = \mathbf{ps}\ \psi[\![\phi]\!]$$

TL-WIRE

$$\psi[\![(\mathbf{W}\ w\ \tau)]\!] = \mathbf{W}\ (\psi[\![w]\!])\ \psi[\![\tau]\!]$$

TL-ARRAY

$$\psi[\![\mathbf{Array}\ \tau]\!] = \mathbf{Array}\ \psi[\![\tau]\!]$$

TL-SHARE

$$\psi[\![\mathbf{Sh}\ w\ \tau]\!] = \mathbf{Sh}\ \psi[\![w]\!]\ \psi[\![\tau]\!]$$

TL-ARROW

$$\psi[\![x{:}\tau_1 \xrightarrow{\epsilon} \tau_2]\!] = x{:}\psi[\![\tau_1]\!] \overset{\psi[\![\epsilon]\!]}{\rightarrow} \psi[\![\tau_2]\!]$$

$\boxed{\psi[\![M_1]\!] = M_2}$ *(Closing place)*

PL-TOP

$$\psi[\![\top]\!] = \top$$

PL-PC

$$\psi[\![m(w)]\!] = m(\psi[\![w]\!])$$

$\boxed{\psi[\![\epsilon_1]\!] = \epsilon_2}$ *(Closing effect)*

EL-EMPTY

$$\psi[\![\cdot]\!] = \cdot$$

EL-PL

$$\psi[\![N]\!] = \psi[\![N]\!]$$

EL-SEQ

$$\psi[\![(\epsilon_1, \epsilon_2)]\!] = (\psi[\![\epsilon_1]\!]), (\psi[\![\epsilon_2]\!])$$

$\boxed{\psi[\![\Gamma_1]\!] = \Gamma_2}$ *(Closing type environment)*

TEL-EMP

$$\psi[\![.]\!] = .$$

TEL-BND1

$$\frac{\psi[\![\Gamma]\!] = \Gamma'}{\psi[\![\Gamma, x : \tau]\!] = \Gamma', x : \psi[\![\tau]\!]}$$

TEL-BND2

$$\frac{\psi[\![\Gamma]\!] = \Gamma'}{\psi[\![\Gamma, x :_M \tau]\!] = \Gamma', x :_{\psi[\![M]\!]} \psi[\![\tau]\!]}$$

Figure A.6: Environment closing judgements

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

SLICEENV-EMP

$$\frac{}{\mathsf{slice}_p(\cdot) \rightsquigarrow \cdot}$$

SLICEENV-BIND1

$$\frac{p \in w \qquad \mathsf{slice}_p(\psi) \rightsquigarrow \psi'}{\mathsf{slice}_p(\psi\{x \mapsto_{\mathsf{p}(w)} v\}) \rightsquigarrow \psi'\{x \mapsto_{\mathsf{p}(\{p\})} \mathsf{slice}_p(v)\}}$$

SLICEENV-BIND2

$$\frac{p \notin w \qquad \mathsf{slice}_p(\psi) \rightsquigarrow \psi'}{\mathsf{slice}_p(\psi\{x \mapsto_{\mathsf{p}(w)} v\}) \rightsquigarrow \psi'}$$

SLICEENV-BIND3

$$\frac{}{\mathsf{slice}_p(\psi\{x \mapsto v\}) \rightsquigarrow \psi'\{x \mapsto \mathsf{slice}_p(v)\}}$$

SLICEENV-BIND4

$$\frac{\mathsf{slice}_p(\psi) \rightsquigarrow \psi'}{\mathsf{slice}_p(\psi\{x \mapsto_{\mathsf{s}(w)} v\}) \rightsquigarrow \psi'}$$

$\boxed{\mathsf{slice}_p(\kappa) \rightsquigarrow \kappa'}$ *(**Stack slicing**: "Stack κ sliced for p is κ'. ")*

SLICESTK-PAR1

$$\frac{\mathsf{slice}_p(\kappa) \rightsquigarrow \kappa' \qquad \mathsf{slice}_p(\psi) \rightsquigarrow \psi'}{\mathsf{slice}_p(\kappa :: \langle \mathsf{p}(\{p\} \cup w); \psi; x.e\rangle) \rightsquigarrow \kappa' :: \langle \psi'; x.e\rangle}$$

SLICESTK-PAR2

$$\frac{\mathsf{slice}_p(\kappa) \rightsquigarrow \kappa' \qquad \mathsf{slice}_p(\psi) \rightsquigarrow \psi'}{\mathsf{slice}_p(\kappa :: \langle \psi; x.e\rangle) \rightsquigarrow \kappa' :: \langle \psi'; x.e\rangle}$$

$\boxed{\mathsf{slice}_p(\sigma) \rightsquigarrow \sigma'}$ *(**Store slicing**: "Store σ sliced for p is σ'. ")*

SLICESTR-PAR1

$$\frac{p \in w \qquad \mathsf{slice}_p(\sigma) \rightsquigarrow \sigma' \qquad \mathsf{slice}_p(v_i) = v_i'}{\mathsf{slice}_p(\sigma\{\ell :_{\mathsf{p}(w)} v_1, .., v_k\}) \rightsquigarrow \sigma'\{\ell :_{\mathsf{p}(\{p\})} v_1', .., v_k'\}}$$

SLICESTR-PAR2

$$\frac{p \notin w \qquad \mathsf{slice}_p(\sigma) \rightsquigarrow \sigma'}{\mathsf{slice}_p(\sigma\{\ell :_{\mathsf{p}(w)} v_1, .., v_k\}) \rightsquigarrow \sigma'}$$

Figure A.7: Slicing judgements

$\boxed{\mathsf{slice}_w(\mathcal{C}) \rightsquigarrow \pi}$ *(**Configuration slicing**: "Configuration $\mathcal{C}$ sliced for w is π. ")*

SLICECFG-EMP

$$\mathsf{slice}_\cdot(\mathcal{C}) \rightsquigarrow \varepsilon$$

SLICECFG-UNION

$$\frac{\mathsf{slice}_{w_1}(\mathcal{C}) \rightsquigarrow \pi_1 \qquad \mathsf{slice}_{w_2}(\mathcal{C}) \rightsquigarrow \pi_2}{\mathsf{slice}_{w_1 \cup w_2}(\mathcal{C}) \rightsquigarrow \pi_1 \cdot \pi_2}$$

SLICECFG-PAR

$$\frac{p \in w \qquad \mathsf{slice}_p(\sigma) \rightsquigarrow \sigma' \qquad \mathsf{slice}_p(\kappa) \rightsquigarrow \kappa' \qquad \mathsf{slice}_p(\psi) \rightsquigarrow \psi'}{\mathsf{slice}_{\{p\}}(\mathsf{p}(w)\{\sigma; \kappa; \psi; e\}) \rightsquigarrow p\,\{\sigma'; \kappa'; \psi'; e\}}$$

SLICECFG-ABS1

$$\frac{p \notin w \qquad \mathsf{slice}_{\{p\}}(m_1(w_1)\{\sigma; \kappa; \psi_2; e_1\}) \rightsquigarrow p\,\{\sigma'; \kappa'; \psi'; e'\} \qquad \psi_2 = \psi_1\{x \mapsto_{m_1(w)} \bigcirc\}}{\mathsf{slice}_{\{p\}}(m(w)\{\sigma; \kappa :: \langle m_1(w_1); \psi_1; x.e_1\rangle; \psi; e\}) \rightsquigarrow p\,\{\sigma'; \kappa'; \psi'; e'\}}$$

SLICECFG-ABS2

$$\frac{p \notin w \qquad \mathsf{slice}_{\{p\}}(m(w)\{\sigma; \kappa; \psi_2; e_1\}) \rightsquigarrow p\,\{\sigma'; \kappa'; \psi'; e'\} \qquad \psi_2 = \psi_1\{x \mapsto \bigcirc\}}{\mathsf{slice}_{\{p\}}(m(w)\{\sigma; \kappa :: \langle \psi_1; x.e_1\rangle; \psi; e\}) \rightsquigarrow p\,\{\sigma'; \kappa'; \psi'; e'\}}$$

SLICECFG-SEC

$$\frac{\pi = \mathsf{s}\binom{w}{w}\left\{\underset{w}{\circ}\sigma_w; \kappa'; \underset{w}{\circ}\psi_w; e\right\} \cdot p\,\{\sigma'; \kappa_1; \cdot; \mathbf{wait}\} \qquad p \in w \qquad \mathsf{slice}_p(\kappa :: \langle \mathsf{p}(w); \psi_1; x.e_1\rangle) \rightsquigarrow \kappa_1 \qquad \mathsf{slice}_p(\sigma) \rightsquigarrow \sigma' \qquad \kappa' \mathsf{smallest}}{\mathsf{slice}_{\{p\}}(\mathsf{s}(w)\{\sigma; \kappa :: \langle \mathsf{p}(w); \psi_1; x.e_1\rangle :: \kappa'; \psi; e\}) \rightsquigarrow \pi}$$

Figure A.8: Configuration slicing judgements

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

SLICEVAL-UNIT

$$\frac{}{\mathsf{slice}_p(()) \rightsquigarrow ()}$$

SLICEVAL-INJ

$$\frac{\mathsf{slice}_p(v) \rightsquigarrow v'}{\mathsf{slice}_p(\mathbf{inj}_i\ v) \rightsquigarrow \mathbf{inj}_i\ v'}$$

SLICEVAL-PROD

$$\frac{\mathsf{slice}_p(v_i) \rightsquigarrow v_i'}{\mathsf{slice}_p((v_1, v_2)) \rightsquigarrow (v_1', v_2')}$$

SLICEVAL-PS

$$\frac{}{\mathsf{slice}_p((w_1 \cup w_2)) \rightsquigarrow (w_1 \cup w_2)}$$

SLICEVAL-WIRE

$$\frac{}{\mathsf{slice}_p((\{p : v\} +\!\!+ v_1)) \rightsquigarrow \{p : v\}}$$

SLICEVAL-WIREABS

$$\frac{p \notin \mathsf{dom}((v_1 +\!\!+ v_2))}{\mathsf{slice}_p((v_1 +\!\!+ v_2)) \rightsquigarrow \cdot}$$

SLICEVAL-LOC

$$\frac{}{\mathsf{slice}_p(\ell) \rightsquigarrow \ell}$$

SLICEVAL-CLOS

$$\frac{}{\mathsf{slice}_p(\mathbf{clos}\,(\psi; \lambda x.e)) \rightsquigarrow \mathbf{clos}\,(\psi'; \lambda x.e)}$$

SLICEVAL-FIXCLOS

$$\frac{}{\mathsf{slice}_p(\mathbf{clos}\,(\psi; \mathbf{fix}\,x.\lambda y.e)) \rightsquigarrow \mathbf{clos}\,(\psi'; \mathbf{fix}\,x.\lambda y.e)}$$

SLICEVAL-SH

$$\frac{\mathsf{slice}_p(v) \rightsquigarrow v'}{\mathsf{slice}_p(\mathbf{sh}\ w\ v) \rightsquigarrow \mathbf{sh}\ w\ v'}$$

$\boxed{\psi_1 \circ \psi_2 \rightsquigarrow \psi_3}$ ***(Environment composing**: "Environment ψ_1 composed with ψ_2 is ψ_3. ")*

COMPENV-EMP

$$\frac{}{\cdot \circ \psi \rightsquigarrow \psi}$$

COMPENV-BIND1

$$\frac{\psi_1 \circ \psi_2 \rightsquigarrow \psi}{\psi_1\{x \mapsto_{\mathsf{p}(\{p\})}\ v_1\} \circ \psi_2\{x \mapsto_{\mathsf{p}(w)}\ v_2\} \rightsquigarrow \psi\{x \mapsto_{\mathsf{p}(\{p\} \cup w)}\ (v_1 \circ v_2)\}}$$

COMPENV-BIND2

$$\frac{\psi_1 \circ \psi_2 \rightsquigarrow \psi}{\psi_1\{x \mapsto_{\mathsf{s}(w)}\ v_1\} \circ \psi_2\{x \mapsto_{\mathsf{s}(w)}\ v_2\} \rightsquigarrow \psi\{x \mapsto_{\mathsf{s}(w)}\ (v_1 \circ v_2)\}}$$

COMPENV-BIND3

$$\frac{\psi_1 \circ \psi_2 \rightsquigarrow \psi}{\psi_1\{x \mapsto v_1\} \circ \psi_2\{x \mapsto v_2\} \rightsquigarrow \psi\{x \mapsto v_1 \circ v_2\}}$$

Figure A.9: Slicing and composing judgments

$\boxed{v_1 \circ v_2 \rightsquigarrow v_3}$ *(**Value composing**: "Value v_1 composed with v_2 is v_3. ")*

compVal-Unit

$$\frac{}{() \circ () \rightsquigarrow ()}$$

compVal-Inj

$$\frac{v_1 \circ v_2 \rightsquigarrow v'}{\mathbf{inj}_i\ v_1 \circ \mathbf{inj}_i\ v_2 \rightsquigarrow \mathbf{inj}_i\ v'}$$

compVal-Prod

$$\frac{v_i \circ v_i' \rightsquigarrow v_i''}{(v_1, v_2) \circ (v_1', v_2') \rightsquigarrow (v_1'', v_2'')}$$

compVal-Ps

$$\frac{}{(w_1 \cup w_2) \circ (w_1 \cup w_2) \rightsquigarrow (w_1 \cup w_2)}$$

compVal-Wire

$$\frac{}{\{p : v\} \circ v_1 \rightsquigarrow \{p : v\} +\!\!+ v}$$

compVal-Loc

$$\frac{}{\ell \circ \ell \rightsquigarrow \ell}$$

compVal-Clos

$$\frac{\psi_1 \circ \psi_2 \rightsquigarrow \psi}{\mathbf{clos}\,(\psi_1; \lambda x.e) \circ \mathbf{clos}\,(\psi_2; \lambda x.e) \rightsquigarrow \mathbf{clos}\,(\psi; \lambda x.e)}$$

compVal-FixClos

$$\frac{\psi_1 \circ \psi_2 \rightsquigarrow \psi}{\mathbf{clos}\,(\psi_1; \mathbf{fix}\,x.\lambda y.e) \circ \mathbf{clos}\,(\psi_2; \mathbf{fix}\,x.\lambda y.e) \rightsquigarrow \mathbf{clos}\,(\psi; \mathbf{fix}\,x.\lambda y.e)}$$

compVal-Sh

$$\frac{v_1 \circ v_2 \rightsquigarrow v}{\mathbf{sh}\ w\ v_1 \circ \mathbf{sh}\ w\ v_2 \rightsquigarrow \mathbf{sh}\ w\ v}$$

$\boxed{\sigma_1 \circ \sigma_2 \rightsquigarrow \sigma_3}$ *(**Store composing**: "Store σ_1 composed with σ_2 is σ_3. ")*

compStr-Emp

$$\frac{}{\cdot \circ \cdot \rightsquigarrow \cdot}$$

compStr-Par

$$\frac{\sigma_1 \circ \sigma_2 \rightsquigarrow \sigma}{\sigma_1\{\ell :_{\mathsf{p}(\{p\})}\ v_1, .., v_k\} \circ \sigma_2\{\ell :_{\mathsf{p}(w)}\ v_1', .., v_k'\} \rightsquigarrow \sigma\{\ell :_{\mathsf{p}(\{p\}\cup w)}\ v_1 \circ v_1', .., v_k \circ v_k'\}}$$

Figure A.10: Value and store composing judgements

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

T-SINGLWIRE

$$\frac{M = m(w_1) \quad m = \mathsf{s} \Rightarrow N = \mathsf{s}(w_1) \quad m = \mathsf{p} \Rightarrow N = \mathsf{p}(\{p\}) \quad . \vdash_N v : \tau \quad . \vdash \tau \quad \tau \ \texttt{IsFlat}}{\Gamma' \vdash_M \{p : v\} : \mathbf{W}\,\{p\}\,\tau}$$

T-PS-EMP

$$\frac{}{\Gamma \vdash_M \cdot : \mathbf{ps}\ (\nu = \cdot)}$$

T-EMP

$$\frac{. \vdash \tau}{\Gamma \vdash_M \cdot : \mathbf{W}\ \cdot\ \tau}$$

T-WIRECAT

$$\frac{. \vdash_M v_1 : \mathbf{W}\,w_1\,\tau \quad . \vdash_M v_2 : \mathbf{W}\,w_2\,\tau}{\Gamma' \vdash_M v_1 +\!\!+ v_2 : \mathbf{W}\,(w_1 \cup w_2)\,\tau}$$

T-LOC

$$\frac{\Sigma(\ell) = \tau \quad . \vdash \tau}{\Gamma' \vdash_M \ell : \mathbf{Array}\,\tau}$$

T-SH

$$\frac{. \vdash_M v : \tau \quad . \vdash \tau \quad \tau \ \texttt{IsFO} \quad \tau \ \texttt{IsFlat}}{\Gamma' \vdash_M \mathbf{sh}\ w\ v : \mathbf{Sh}\ w\,\tau}$$

T-CLOS

$$\frac{\Sigma \vdash \psi \rightsquigarrow \Gamma \quad \Gamma, x : \tau_1 \vdash_M e : \tau_2; \epsilon \quad . \vdash (x{:}\tau_1 \xrightarrow{\epsilon} \tau_2)}{\Gamma' \vdash_M \mathbf{clos}\,(\psi; \lambda x.e) : x{:}\tau_1 \xrightarrow{\epsilon} \tau_2}$$

T-FIXCLOS

$$\frac{\Sigma \vdash \psi \rightsquigarrow \Gamma \quad \Gamma, f : y{:}\tau_1 \xrightarrow{\epsilon} \tau_2 \vdash_M \lambda x.e : y{:}\tau_1 \xrightarrow{\epsilon} \tau_2; \cdot \quad . \vdash (y{:}\tau_1 \xrightarrow{\epsilon} \tau_2)}{\Gamma' \vdash_M \mathbf{clos}\,(\psi; \mathbf{fix}\,f.\lambda x.e) : y{:}\tau_1 \xrightarrow{\epsilon} \tau_2}$$

$\boxed{\Gamma \vdash_M e : \tau; \epsilon}$ *(Runtime expression typing)*

T-SECBLK

$$\frac{\Gamma \vdash w : \mathbf{ps}\ (\nu = w') \quad \Gamma \vdash_{\mathsf{s}(w)} e : \tau; \epsilon}{\Gamma \vdash_{m(w')} \mathbf{secure}_w(e) : \tau; \epsilon}$$

Figure A.11: Runtime value and expression typing

$\boxed{\Sigma \vdash \sigma\ \mathbf{wf}}$ *(Store typing)*

TSTORE-EMP

$$\frac{}{\Sigma \vdash \cdot\ \mathbf{wf}}$$

TSTORE-LOC

$$\frac{\Sigma \vdash \sigma\ \mathbf{wf} \qquad \Sigma; . \vdash_M v_i : \tau}{\Sigma, \ell :_M \tau \vdash \sigma\{\ell :_M v_1, .., v_k\}\ \mathbf{wf}}$$

$\boxed{\Sigma \vdash_M \kappa : \tau_1 \hookrightarrow \tau_2}$ *(Stack typing)*

TSTK-EMP

$$\frac{. \vdash \tau \qquad w\ \texttt{is all parties}}{\Sigma \vdash_{\mathsf{p}} (w)\cdot : \tau \hookrightarrow \tau}$$

TSTK-FRAME1

$$\frac{\begin{array}{c} M = \_(w) \qquad N = m(\_) \\ \Sigma \vdash \psi \rightsquigarrow \Gamma \qquad \Gamma \vdash \tau_1 \\ \Sigma; \Gamma, x :_{m(w)} \tau_1 \vdash_N e : \tau_2; \epsilon \\ \Sigma \vdash_N \kappa : \psi[\![\tau_2]\!] \hookrightarrow \tau_3 \end{array}}{\Sigma \vdash_M \kappa :: \langle N; \psi; x.e\rangle : \psi[\![\tau_1]\!] \hookrightarrow \tau_3}$$

TSTK-FRAME2

$$\frac{\begin{array}{c} \Sigma \vdash \psi \rightsquigarrow \Gamma \qquad \Gamma \vdash \tau_1 \\ \Sigma; \Gamma, x : \tau_1 \vdash_M e : \tau_2; \epsilon \\ \Sigma \vdash_M \kappa : \psi[\![\tau_2]\!] \hookrightarrow \tau_3 \end{array}}{\Sigma \vdash_M \kappa :: \langle \psi; x.e\rangle : \psi[\![\tau_1]\!] \hookrightarrow \tau_3}$$

$\boxed{\Sigma \vdash \psi \rightsquigarrow \Gamma}$ *(Environment typing)*

TENV-EMP

$$\frac{}{\Sigma \vdash \cdot \rightsquigarrow .}$$

TENV-MAPP

$$\frac{\Sigma \vdash \psi \rightsquigarrow \Gamma \qquad \Sigma; . \vdash_M v : \psi[\![\tau]\!]}{\Sigma \vdash \psi\{x \mapsto_M v\} \rightsquigarrow \Gamma, x :_M \tau}$$

TENV-MAPP2

$$\frac{\Sigma \vdash \psi \rightsquigarrow \Gamma \qquad \Sigma; . \vdash v : \psi[\![\tau]\!]}{\Sigma \vdash \psi\{x \mapsto v\} \rightsquigarrow \Gamma, x : \tau}$$

TENV-MUNK

$$\frac{\Sigma \vdash \psi \rightsquigarrow \Gamma \qquad \Gamma \vdash \tau}{\Sigma \vdash \psi\{x \mapsto_M \bigcirc\} \rightsquigarrow \Gamma, x :_M \tau}$$

TENV-MUNK2

$$\frac{\Sigma \vdash \psi \rightsquigarrow \Gamma \qquad \Gamma \vdash \tau}{\Sigma \vdash \psi\{x \mapsto \bigcirc\} \rightsquigarrow \Gamma, x : \tau}$$

Figure A.12: Typing for store, stack, and environment

$\boxed{\Sigma \vdash \mathcal{C} : \tau}$ *(Configuration typing)*

TCONFIG-CONFIG

$$\frac{\Sigma \vdash \sigma\ \mathbf{wf} \qquad . \vdash M \qquad \Sigma \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau_2 \qquad \Sigma \vdash \psi \rightsquigarrow \Gamma \qquad \Sigma; \Gamma \vdash_M e : \tau_1; \epsilon}{\Sigma \vdash M\{\sigma; \kappa; \psi; e\} : \tau_2}$$

$\boxed{\mathcal{C}\ \mathbf{halted}}$ *(Configuration halt states)*

HALTED-ANSWER

$$\frac{w \texttt{ is all parties}}{\mathsf{p}(w)\{\sigma; \cdot; \psi; v\}\ \mathbf{halted}}$$

HALTED-ERROR

$$\frac{}{M\{\sigma; \kappa; \psi; \mathsf{error}\}\ \mathbf{halted}}$$

$\boxed{\mathcal{C}\ \mathbf{st}}$ *(Stack structure invariants)*

STOK-SEC

$$\frac{\kappa = \kappa' :: \langle \mathsf{p}(w); \psi'; x.e_2 \rangle :: \kappa_1}{\mathsf{s}(w)\{\sigma; \kappa; \psi; e\}\ \mathbf{st}}$$

STOK-SECE

$$\frac{\kappa = \kappa' :: \langle \mathsf{p}(w); \psi'; x.e_2 \rangle}{\mathsf{p}(w)\{\sigma; \kappa; \psi; \mathbf{secure}_{w'}(e)\}\ \mathbf{st}}$$

STOK-PAR1

$$\frac{}{\mathsf{p}(w)\{\sigma; \cdot; \psi; e\}\ \mathbf{st}}$$

STOK-PAR2

$$\frac{w \subseteq w' \qquad \mathsf{p}(w)\{\sigma; \kappa; \psi; e\}\ \mathbf{st}}{\mathsf{p}(w)\{\sigma; \kappa :: \langle \mathsf{p}(w'); \psi'; x.e' \rangle; \psi; e\}\ \mathbf{st}}$$

STOK-PAR3

$$\frac{\mathsf{p}(w)\{\sigma; \kappa; \psi; e\}\ \mathbf{st}}{\mathsf{p}(w)\{\sigma; \kappa :: \langle \psi'; x.e' \rangle; \psi; e\}\ \mathbf{st}}$$

Figure A.13: Runtime configuration typing

Appendix B: WYSTERIA Proofs

We first present several auxiliary lemmas. Main theorems are proved towards the end. This section is best read electronically, as it has several hyperlinks to aid navigation (such as for skipping long proofs, etc.).

Lemma 18 (Weakening of type environment).
Let $x \notin \mathsf{dom}(\Gamma)$ *and* $\Gamma_1 = \Gamma, x :_{M_1} \tau'$.

1. *If* $\Gamma \vdash_M v : \tau$, *then* $\Gamma_1 \vdash_M v : \tau$.
2. *If* $\Gamma \vdash v : \tau$, *then* $\Gamma_1 \vdash v : \tau$.
3. *If* $\Gamma \vdash \tau$, *then* $\Gamma_1 \vdash \tau$.
4. *If* $\Gamma \vdash \phi$, *then* $\Gamma_1 \vdash \phi$.
5. *If* $\Gamma \vdash \tau_1 <: \tau_2$, *then* $\Gamma_1 \vdash \tau_1 <: \tau_2$.
6. *If* $\Gamma \vdash N$, *then* $\Gamma_1 \vdash N$.
7. *If* $\Gamma \vdash M \rhd N$, *then* $\Gamma_1 \vdash M \rhd N$.
8. *If* $\Gamma \vdash \epsilon$, *then* $\Gamma_1 \vdash \epsilon$.

Proof. *(Skip)* By simultaneous induction.
(1.) Induction on derivation of $\Gamma \vdash_M v : \tau$, case analysis on the last rule used.
Rule T-VAR. We have,
(a) $v = y$
(b) $y :_M \tau \in \Gamma \vee y : \tau \in \Gamma$
(c) $\Gamma \vdash \tau$
((b) means y is different from x)
From (b) we have,
(d) $y :_M \tau \in \Gamma_1 \vee y : \tau \in \Gamma_1$
Use I.H. (3.) on (c) to get,
(e) $\Gamma_1 \vdash \tau$
With (d) and (e), use rule T-VAR to derive $\Gamma_1 \vdash_M v : \tau$.

Rule T-UNIT. Use rule T-UNIT with Γ_1.

Rule T-INJ. We have,

(a) $\Gamma \vdash_M v : \tau_i$

(b) τ_j `IsFlat`

(c) $\Gamma \vdash \tau_j$

Use I.H. (1.) on (a), I.H. (3.) on (c), and with (b) use rule T-INJ.

Rule T-PROD. Use I.H. (1.) on premises.

Rule T-PRINC. Use rule T-PRINC with Γ_1.

Rule T-PSONE. Use I.H. (1.) on rule premise, and then use rule T-PSONE.

Rule T-PSUNION. Use I.H. (1.) on rule premises, and then use rule T-PSUNION.

Rule T-PSVAR. Use I.H. (1.) on rule premise, and then rule T-PSVAR (note that $v = y$, different from x).

Rule T-MSUB. Use of I.H. (6.), (1.), and (7.), and then use rule T-MSUB.

Rule T-SUB. use of I.H. (1.), (5.), and (3.), and then use rule T-SUB.

Rule T-SINGLWIRE. given premises, use rule T-SINGLWIRE with Γ_1.

Rule T-WIRECAT. given premises, use rule T-WIRECAT with Γ_1.

Rule T-LOC. given premises, use rule T-LOC with Γ_1.

Rule T-SH. given premises, use rule T-SH with Γ_1.

Rule T-CLOS. given premises, use rule T-CLOS with Γ_1.

Rule T-FIXCLOS. given premises, use rule T-FIXCLOS with Γ_1.

(2.) Similar to proof above.

(3.) Induction on derivation of $\Gamma \vdash \tau$, case analysis on the last rule used.

Rule WF-UNIT. use rule WF-UNIT with Γ_1.

Rule WF-SUM. use I.H. (3.) on premises, and then use rule WF-SUM.

Rule WF-PROD. use I.H. (3.) on premises, and then use rule WF-PROD.

Rule WF-PRINC. use I.H. (4.) on rule premise, and then use rule WF-PRINC.

Rule WF-ARROW. we have,

(a) $\tau = y{:}\tau_1 \xrightarrow{\epsilon} \tau_2$

(b) $\Gamma \vdash \tau_1$

(c) $\Gamma, y : \tau_1 \vdash \epsilon$

(d) $\Gamma, y : \tau_1 \vdash \tau_2$

Use I.H. (3.) on (b), I.H. (8.) on (c), and I.H. (3.) on (d) to get

(e) $\Gamma_1 \vdash \tau_1$

(f) $\Gamma, y : \tau_1, x :_{M_1} \tau' \vdash \epsilon$

(g) $\Gamma, y : \tau_1, x :_{M_1} \tau' \vdash \tau_2$

Use type environment permutation lemma on (f) and (g) and then use rule WF-ARROW with (e) to get the derivation.

Rule WF-WIRE. use I.H. (2.) and (3.), and then use rule WF-WIRE.

Rule WF-ARRAY. use I.H. (3.), and then use rule WF-ARRAY.

Rule WF-SHARE. use I.H. (2.) and (3.), and then use rule WF-SHARE.

(4.) Induction on derivation of $\Gamma \vdash \phi$, case analysis on the last rule used.

Rule WFREF-TRUE. use rule WFREF-TRUE with Γ_1.

Rule WFREF-SINGL. use rule WFREF-SINGL with Γ_1.

Rule WFREF-SUB. use I.H. (2.) on the premise, and then use rule WFREF-SUB.

Rule WFREF-EQ. use I.H. (2.) on the premise, and then use rule WFREF-EQ.

Rule WFREF-CONJ. use I.H. (4.) on the premises, and then use rule WFREF-CONJ.
(5.) Induction on derivation of $\Gamma \vdash \tau_1 <: \tau_2$, case analysis on the last rule used.
Rule S-REFL. use rule S-REFL with Γ_1.
Rule S-TRANS. use I.H. (5.) on premises, and then use rule S-TRANS.
Rule S-SUM. use I.H. (5.) on premises, and then use rule S-SUM.
Rule S-PROD. use I.H. (5.) on premises, and then use rule S-PROD.
Rule S-PRINCS. we have,
(a) $[\![\Gamma]\!] \vDash \phi_1 \Rightarrow \phi_2$
We can derive
(b) $[\![\Gamma_1]\!] \vDash \phi_1 \Rightarrow \phi_2$
Use rule S-PRINCS again.
Rule S-WIRE. use I.H. (2.) and (5.) on premises, and then use rule S-WIRE.
Rule S-ARRAY. use I.H. (5.) on premises, and then use rule S-ARRAY.
Rule S-SHARE. use I.H. (2.) and (5.) on premises, and then use rule S-SHARE.
Rule S-ARROW. use I.H. (5.) on premises, permutation lemma, and then use rule S-ARROW.
(6.) Induction on derivation of $\Gamma \vdash N$, case analysis on the last rule used.
Rule WFPL-TOP. use rule WFPL-TOP with Γ_1.
Rule WFPL-OTHER. use I.H. (2.) on premise, and then use rule WFPL-OTHER.
(7.) Induction on derivation of $\Gamma \vdash M \rhd N$, case analysis on the last rule used.
Rule D-REFL. use I.H. (2.) on premise, and then use rule D-REFL.
Rule D-TOP. use I.H. (2.) on premise, and then use rule D-TOP.
Rule D-PAR. use I.H. (2.) on premise, and then use rule D-PAR.
Rule D-SEC. use I.H. (2.) on premise, and then use rule D-SEC.
(8.) Induction on derivation of $\Gamma \vdash \epsilon$, case analysis on the last rule used.
Rule WFEFF-EMPTY. use rule WFEFF-EMPTY with Γ_1.
Rule WFEFF-MODE. use I.H. (8.) and I.H. (6.) on premises, and then use rule WFEFF-MODE.

□

Lemma 19 (Weakening of type environment).
Let $x \notin \mathsf{dom}(\Gamma)$ *and* $\Gamma_1 = \Gamma, x : \tau'$.

1. *If* $\Gamma \vdash_M v : \tau$, *then* $\Gamma_1 \vdash_M v : \tau$.
2. *If* $\Gamma \vdash v : \tau$, *then* $\Gamma_1 \vdash v : \tau$.
3. *If* $\Gamma \vdash \tau$, *then* $\Gamma_1 \vdash \tau$.
4. *If* $\Gamma \vdash \phi$, *then* $\Gamma_1 \vdash \phi$.
5. *If* $\Gamma \vdash \tau_1 <: \tau_2$, *then* $\Gamma_1 \vdash \tau_1 <: \tau_2$.
6. *If* $\Gamma \vdash N$, *then* $\Gamma_1 \vdash N$.
7. *If* $\Gamma \vdash M \rhd N$, *then* $\Gamma_1 \vdash M \rhd N$.
8. *If* $\Gamma \vdash \epsilon$, *then* $\Gamma_1 \vdash \epsilon$.

Proof. Similar to the proof of Lemma 18. □

Lemma 20 (Weakening of type environment under subtyping).
Let $\Gamma \vdash \tau' <: \tau$ *and* $\Gamma \vdash \tau'$. *Let* $\Gamma_1 = \Gamma, x :_M \tau$. *For* $\Gamma_2 = \Gamma, x :_M \tau'$,

1. *If* $\Gamma_1 \vdash_N v : \tau$, *then* $\Gamma_2 \vdash_N v : \tau$.
2. *If* $\Gamma_1 \vdash v : \tau$, *then* $\Gamma_2 \vdash v : \tau$.
3. *If* $\Gamma_1 \vdash \tau_1 <: \tau_2$, *then* $\Gamma_2 \vdash \tau_1 <: \tau_2$.
4. *If* $\Gamma_1 \vdash \tau$, *then* $\Gamma_2 \vdash \tau$.
5. *If* $\Gamma_1 \vdash \phi$, *then* $\Gamma_2 \vdash \phi$.
6. *If* $\Gamma_1 \vdash N \rhd M_1$, *then* $\Gamma_2 \vdash N \rhd M_1$.
7. *If* $\Gamma_1 \vdash M_1$, *then* $\Gamma_2 \vdash M_1$.
8. *If* $\Gamma_1 \vdash \epsilon$, *then* $\Gamma_2 \vdash \epsilon$.
9. *If* $\Gamma_1 \vdash_N e : \tau; \epsilon$, *then* $\Gamma_2 \vdash_N e : \tau; \epsilon$.

Proof. *(Skip)*
By simultaneous induction.
(1.) Induction on derivation of $\Gamma_1 \vdash_N v : \tau$, case analysis on the last rule used.
Rule T-VAR. We have,
(a) $v = y$
(b) $y :_N \tau \in \Gamma_1 \vee y : \tau \in \Gamma_1$
(b') $\Gamma_1 \vdash \tau$
We have two cases now,
(i) $y = x$
This means,
(c) $M = N$
With lemma premise $\Gamma \vdash \tau'$, use Lemma 18 to get,
(d) $\Gamma_1 \vdash \tau'$
Use I.H. (4.) on (d) to get,
(e) $\Gamma_2 \vdash \tau'$
With (d), use rule T-VAR on Γ_2 to get,
(f) $\Gamma_2 \vdash_M x : \tau'$
With $\Gamma \vdash \tau' <: \tau$, use Lemma 18 to get
(g) $\Gamma_2 \vdash \tau' <: \tau$
Use I.H. (4.) on (b') to get,
(h) $\Gamma_2 \vdash \tau$ $(M = N)$
With (f), (g) and (h), use rule T-SUB.
(ii) y is different from x, in which case rule T-VAR still holds on Γ_2 (with use of I.H. (4.)).
Rule T-UNIT. Use rule T-UNIT with Γ_2.

Rule T-INJ. Use I.H. (1.) and (4.) on premises, and then use rule T-INJ.

Rule T-PROD. Use I.H. (1.) on premises, and then use rule T-PROD.

Rule T-PRINC. Use rule T-PRINC with Γ_2.

Rules T-PSONE, T-PSUNION, *and* T-PSVAR. Use I.H. (1.) on premises, and then use respective rule again.

Rule T-MSUB. We have,

(a) $v = y$

(b) $\Gamma_1 \vdash M_1$

(c) $\Gamma_1 \vdash_{M_1} y : \tau$

(d) $\Gamma_1 \vdash M_1 \rhd N$

Use I.H. (7.) on (b),

(f) $\Gamma_2 \vdash M_1$

Use I.H. (1.) on (c)

(g) $\Gamma_2 \vdash_{M_1} y : \tau$

Use I.H. (6.) on (d)

(h) $\Gamma_2 \vdash M_1 \rhd N$

With (f), (g), (h), use rule T-MSUB (the fact about τ `IsSecIn` carries)

Rule T-SUB. Use I.H. (1.), (3.), and (4.) on premises, and then use rule T-SUB.

Runtime value typing rules are similar to proof of Lemma 18 (they don't depend on Γ).

(2.) Induction on derivation of $\Gamma \vdash v : \tau$, case analysis on the last rule used.

Rule TN-VAR. We have,

(a) $v = y$

(b) $y : \tau \in \Gamma_1$

(c) $\Gamma_1 \vdash \tau$

From (b) it follows that y is different from x, and so,

(d) $y : \tau \in \Gamma_2$

Use I.H. (4.) on (c) and with (b), use rule TN-VAR.

Other cases are similar to (1.).

(3.) Induction on derivation of $\Gamma_1 \vdash \tau_1 <: \tau_2$, case analysis on the last rule used.

Rule S-REFL. Use rule S-REFL with Γ_2.

Rules S-TRANS, S-SUM, *and* S-PROD. Use I.H. (3.) on premises, and the use respective rule.

Rule S-PRINCS. We have,

(a) $[\![\Gamma_1]\!] \vDash \phi_1 \Rightarrow \phi_2$

We need to prove $[\![\Gamma_2]\!] \vDash \phi_1 \Rightarrow \phi_2$. Informally, only principal types in Γ matter when deciding logical implications. And, a more precise type in the typing environment means stronger assumption.

Rule S-WIRE. Use I.H. (2.) and I.H. (3.) on premises, and then use rule S-WIRE.

Rule S-ARRAY. Use I.H. (3.) on premises, and then use rule S-ARRAY.

Rule S-SHARE. Use I.H. (2.) and I.H. (3.) on premises, and then use rule S-SHARE.

Rule S-ARROW. Use I.H. (3.) on premises with permutation lemma for second premise, and then use rule S-ARROW.

(4.) Induction on derivation of $\Gamma_1 \vdash \tau$, case analysis on the last rule used.

Rule WF-UNIT. Use rule WF-UNIT with Γ_2.

Rule WF-SUM. Use I.H. (4.) on premises, and then use rule WF-SUM.
Rule WF-PROD. Use I.H. (4.) on premises, and then use rule WF-PROD.
Rule WF-PRINC. Use I.H. (5.) on premise, and then use rule WF-PRINC.
Rule WF-ARROW. Use I.H. (4.) and (8.) with permutation lemma on typing environment, and then use rule WF-ARROW.
Rule WF-WIRE. Use I.H. (2.) and (3.) on premises, and then ise rule WF-WIRE.
Rule WF-ARRAY. Use I.H. (4.) on premise, and then use rule WF-ARRAY.
Rule WF-SHARE. Use I.H. (2.) and (4.) on premises, and then use rule WF-SHARE.
(5.) Induction on derivation of $\Gamma_1 \vdash \phi$, case analysis on the last rule used.
Rule WFREF-TRUE. Use rule WFREF-TRUE with Γ_2.
Rule WFREF-SINGL. Use rule WFREF-SINGL with Γ_2.
Rules WFREF-SUB *and* WFREF-EQ. Use I.H. on premises, and then use respective rule.
Rule WFREF-CONJ. Use I.H. on premises, and then use rule WFREF-CONJ.
(6.) Induction on derivation of $\Gamma_1 \vdash N \rhd M_1$, case analysis on the last rule used.
Rule D-REFL. Use I.H. on premise, and then use rule D-REFL.
Rule D-TOP. Use I.H. on premise, and then use rule D-TOP.
Rules D-PAR *and* D-SEC. Similar use of I.H. and then respective rule.
(7.) Induction on derivation of $\Gamma_1 \vdash M_1$, case analysis on the last rule used.
Rule WFPL-TOP. Use rule WFPL-TOP with Γ_2.
Rule WFPL-OTHER. Use I.H. on premise, and then use rule WFPL-OTHER.
(8.) Induction on derivation of $\Gamma_1 \vdash \epsilon$, case analysis on the last rule used.
Rule WFEFF-EMPTY. Use rule WFEFF-EMPTY with Γ_2.
Rule WFEFF-MODE. Use I.H. on premises, and then use rule WFEFF-MODE.
(9.) Proof by induction on derivation of $\Gamma_1 \vdash_N e : \tau; \epsilon$.

□

Lemma 21 (Can derive self equality in refinements).
If $\Gamma \vdash v$: ***ps*** ϕ, *then* $\Gamma \vdash v$: ***ps*** $(\nu = v)$.

Proof. Structural induction on v.
$v = x$. We have,
(a) $\Gamma \vdash x$: **ps** ϕ
With (a) use rule TN-PSVAR to get $\Gamma \vdash x$: **ps** $(\nu = x)$.
$v = p$. Use rule TN-PRINC.
$v = \{w\}$. Use rule TN-PSONE.
$v = w_1 \cup w_2$. Use rule TN-PSUNION.
No other form of v is possible.

□

Lemma 22 (Can derive self subset refinements).
If $\Gamma \vdash v$: ***ps*** ϕ, *then* $\Gamma \vdash v$: ***ps*** $(\nu \subseteq v)$.

Proof. Using Lemma 21, we have
(a) $\Gamma \vdash v$: **ps** $(\nu = v)$
Also,

(b) $[\![\Gamma]\!] \vDash (\nu = v) \Rightarrow (\nu \subseteq v)$
With (b) use rule S-PRINCS to get
(c) $\Gamma \vdash \mathbf{ps}\ (\nu = v) <: \mathbf{ps}\ (\nu \subseteq v)$
With lemma premise $\Gamma \vdash v : \mathbf{ps}\ \phi$, we can use rule WFREF-SUB and rule WF-PRINC to get,
(d) $\Gamma \vdash \mathbf{ps}\ (\nu \subseteq v)$
With (a), (c), and (d), use rule T-SUB.

□

Lemma 23 (Transitivity of Delegation).
If $\Gamma \vdash M \rhd M_1$, *and* $\Gamma \vdash M_1 \rhd M_2$, *then* $\Gamma \vdash M \rhd M_2$.

Lemma 24 (Secure place can only delegate to self).
If $\Gamma \vdash \mathsf{s}(w_1) \rhd m(w_2)$, *then*

1. $m = \mathsf{s}$
2. $\Gamma \vdash w_2 : \textbf{\textit{ps}}\ (\nu = w_1)$
3. $\Gamma \vdash \mathsf{p}(w_1) \rhd m(w_2)$

Proof. $\Gamma \vdash \mathsf{s}(w_1) \rhd m(w_2)$ can only be derived using rule D-REFL, which immediately gives us (1.) and (2.). For (3.), use rule D-SEC and then Lemma 23.

□

Lemma 25 (Delegation implies well-formedness).
If $\Gamma \vdash M$, $\Gamma \vdash M \rhd N$, *then* $\Gamma \vdash N$.

Proof. Proof by induction on derivation of $\Gamma \vdash M \rhd N$, case analysis on the last rule used.
Rule D-REFL. We have,
(a) $M = m(w_1)$
(b) $N = m(w_2)$
(c) $\Gamma \vdash w_2 : \mathbf{ps}\ (\nu = w_1)$
With (c), use rule WFPL-OTHER on $m(w_2)$.
Rule D-TOP. We have,
(a) $M = \top$
(b) $N = m(w)$
(c) $\Gamma \vdash_\top w : \mathbf{ps}\ \phi$
With (c), use rule WFPL-OTHER to get $\Gamma \vdash m(w)$.
Rules D-PAR *and* D-SEC. Similar to rule D-TOP.

□

Lemma 26 (Effect delegation implies well-formedness).
If $\Gamma \vdash M$, $\Gamma \vdash M \rhd \epsilon$, *then* $\Gamma \vdash \epsilon$.

Proof. Straightforward extension of Lemma 25. □

Lemma 27 (Typing results in well-formed types).
Let $\Gamma \vdash M$.

1. *If* $\Gamma \vdash_M v : \tau$, *then* $\Gamma \vdash \tau$.

2. *If* $\Gamma \vdash v : \tau$, *then* $\Gamma \vdash \tau$.

3. *If* $\Gamma \vdash_M e : \tau; \epsilon$, *then* $\Gamma \vdash \tau$ *and* $\Gamma \vdash M \rhd \epsilon$.

Proof. *(Skip)*

Proof by induction on derivation of $\Gamma \vdash_M v : \tau$, case analysis on the last rule used.

Rule T-VAR. Follows from rule premise.

Rule T-UNIT. Use rule WF-UNIT.

Rule T-INJ. With rule premises, use rule WF-SUM.

Rule T-PROD. Use I.H. on rule premises, and then rule WF-PROD.

Rule T-PRINC. We have

(a) $\Gamma \vdash_M p : \mathbf{ps}\ (\nu = \{p\})$

With (a), use rule WFREF-EQ and rule WF-PRINC.

Rules T-PSONE, T-PSUNION, *and* T-PSVAR. Use rule premise in rule WFREF-EQ and rule WF-PRINC.

Rule T-MSUB. We have,

(a) $\Gamma \vdash_N x : \tau$

(b) $\Gamma \vdash N \rhd M$

Use I.H. on (a) to get $\Gamma \vdash \tau$

Rule T-SUB. Follows from rule premise.

Rule T-SINGLWIRE. We have,

(a) $v = \{p : v'\}$

(c) $\tau = \mathbf{W}\ \{p\}\ \tau$

(d) $. \vdash \tau$

(d') τ `IsFlat`

Use rule TN-PRINC to get,

(e) $. \vdash p : \mathbf{ps}\ (\nu = \{p\})$

With (e), use rule T-PSONE to get,

(f) $. \vdash \{p\} : \mathbf{ps}\ (\nu = \{p\})$

Use weakening on (d) and (f), and then with (d') use rule WF-WIRE.

Rule T-WIRECAT. We have,

(a) $v = v_1 +\!\!+ v_2$

(b) $\tau = \mathbf{W}\ (w_1 \cup w_2)\ \tau$

(c) $. \vdash_M v_1 : \mathbf{W}\ w_1\ \tau$

(d) $. \vdash_M v_2 : \mathbf{W}\ w_2\ \tau$

Use I.H. on (c) and (d) to get,

(e) $. \vdash \mathbf{W}\ w_1\ \tau$

(f) $. \vdash \mathbf{W}\ w_2\ \tau$

Invert rule WF-WIRE on (e) and (f) to get,

(g) $. \vdash \tau$

(h) $. \vdash w_i : \mathbf{ps}\ \phi_i$

Use weakening on (g) and (h) (with Γ), and then use rule WF-WIRE.

Rule T-LOC. Follows from the premise.

Rule T-SH. We have,

(a) $. \vdash M$
(b) $M = \_(w)$
(c) $. \vdash \tau$
With (a) and (b), invert rule WFPL-OTHER to get,
(d) $. \vdash w : \mathbf{ps}\ \phi$
Use weakening on (c) and (d), and then use rule WF-SH.
Rules T-CLOS *and* T-FIXCLOS. Follows from the premises (with weakening).
(2.) Induction on derivation of $\Gamma \vdash_M e : \tau; \epsilon$, case analysis on the last rule used.
Rule T-FST. We have,
(a) $e = \mathbf{fst}\,(v)$
(b) $\epsilon = \cdot$
(c) $\Gamma \vdash_M v : \tau_1 \times \tau_2$ (rule premise)
(d) $\tau = \tau_1$
With (c), use I.H. to get
(e) $\Gamma \vdash \tau_1 \times \tau_2$
With (e), invert rule rule WF-PROD to get,
(f) $\Gamma \vdash \tau_1$
With (f), and rule EFFDEL-EMPTY, we have the proof.
Rule T-SND. Similar to rule T-Fst.
Rule T-CASE. Follows from the premises.
Rule T-APP.
Rules T-LET1, T-LET2, *and* T-FIX. Follows directly from rule premises.
Rule T-ARRAY. We have,
(a) $e = \mathbf{array}(v_1, v_2)$
(b) $\tau = \mathbf{Array}\,\tau_2$
(c) $\Gamma \vdash_M v_2 : \tau_2$ (rule premise)
With (c), use Lemma 27, to get
(d) $\Gamma \vdash \tau_2$
With (d), use rule WF-ARRAY to get $\Gamma \vdash \mathbf{Array}\,\tau_2$. $\Gamma \vdash M \rhd \cdot$ follows from rule EFFDEL-EMPTY.
Rule T-SELECT. Use inversion on $\Gamma \vdash \mathbf{Array}\,\tau$ (from rule premise and I.H.).
Rule T-UPDATE. Use rule WF-UNIT and rule EFFDEL-EMPTY.
Rule T-WIRE. We have,
(a) $\Gamma \vdash w_1 : \mathbf{ps}\ (\nu \subseteq w_2)$
(b) $\Gamma \vdash_N v : \tau$
Use I.H. on (b) and then with (a) use rule WF-WIRE.
Rule T-WPROJ. Use I.H. on premise $\Gamma \vdash_{m(w_1)} v : \mathbf{W}\ w_2\ \tau$, and then invert rule WF-WIRE.
Rule T-WIREUN. Use I.H. on premises and then rule WF-WIRE.
Rule T-WFOLD. Follows from rule premise $\Gamma \vdash_M v_2 : \tau_2$.
Rule T-WAPP. We have,
(a) $e = \mathbf{wapp}_w(v_1, v_2)$
(b) $\tau = \mathbf{W}\ w\ \tau_2$
(c) $M = \mathsf{p}(\_)$
(d) $\Gamma \vdash_M v_1 : \mathbf{W}\ w\ \tau_1$

(e) $\Gamma \vdash_M v_2 : \mathbf{W}\ w\ (\tau_1 \xrightarrow{\cdot} \tau_2)$
With (d) and (e), use I.H., to get,
(f) $\Gamma \vdash \mathbf{W}\ w\ (\tau_1 \xrightarrow{\cdot} \tau_2)$
(g) $\Gamma \vdash w : \mathbf{ps}\ \phi$
Inverting rule WF-WIRE on (f),
(h) $\Gamma \vdash \tau_1 \xrightarrow{\cdot} \tau_2$
(i) $\tau_1 \xrightarrow{\cdot} \tau_2$ `IsFlat`
Inverting rule WF-ARROW on (h) to get
(j) $\Gamma \vdash \tau_2$
Inverting rule W-ARR on (i),
(k) τ_2 `IsFlat`
With (g), (j), (k), use rule WF-WIRE to get $\Gamma \vdash \mathbf{W}\ w\ \tau_2$.
Rule T-WAPS. We have,
(a) $e = \mathbf{waps}_w(v_1, v_2)$
(b) $M = \mathtt{s}(\_)$
(c) τ_2 `IsFlat`
(d) $\Gamma \vdash_M v_1 : \mathbf{W}\ w\ \tau_1$
(e) $\Gamma \vdash_M \lambda x.e : \tau_1 \xrightarrow{\cdot} \tau_2; \cdot$
Invert rule WF-WIRE on (d) to get,
(f) $\Gamma \vdash w : \mathbf{ps}\ \phi$
Invert rule WF-ARR on (e) to get,
(g) $\Gamma \vdash \tau_2$
With (b), (f), (g), (c), use rule WF-WIRE, to get $\Gamma \vdash \mathbf{W}\ w\ \tau_2$.
Rule W-COPY. Use I.H. on premise.
Rule T-MAKESH. Invert rule WFPL-OTHER with lemma premise $\Gamma \vdash M$, use I.H. on $\Gamma \vdash_M v : \tau$, and then use rule WF-SH.
Rule T-COMBSH. Invert rule WF-SHARE on rule premise.

□

Lemma 28 (Subtyping inversion).

1. *If* $\Gamma \vdash \tau <: \textbf{\textit{unit}}$, *then* $\tau = \textbf{\textit{unit}}$.
2. *If* $\Gamma \vdash \tau <: \tau_1 \times \tau_2$, *then* $\tau = \tau_3 \times \tau_4$ *s.t.* $\Gamma \vdash \tau_3 <: \tau_1$ *and* $\Gamma \vdash \tau_4 <: \tau_2$.
3. *If* $\Gamma \vdash \tau <: \tau_1 + \tau_2$, *then* $\tau = \tau_3 + \tau_4$ *s.t.* $\Gamma \vdash \tau_3 <: \tau_1$ *and* $\Gamma \vdash \tau_4 <: \tau_2$.
4. *If* $\Gamma \vdash \tau <: \textbf{\textit{ps}}\ \phi$, *then* $\tau = \textbf{\textit{ps}}\ \phi_2$ *s.t.* $[\![\Gamma]\!] \vDash \phi_2 \Rightarrow \phi$.
5. *If* $\Gamma \vdash \tau <: \textbf{\textit{W}}\ w_2\ \tau_2$ *and* $\Gamma \vdash w_2 : \textbf{\textit{ps}}\ \phi$, *then* $\tau = \textbf{\textit{W}}\ w_1\ \tau_1$ *s.t.* $\Gamma \vdash w_2 : \textbf{\textit{ps}}\ (\nu \subseteq w_1)$ *and* $\Gamma \vdash \tau_1 <: \tau_2$.
6. *If* $\Gamma \vdash \tau <: \textbf{\textit{Array}}\ \tau_2$, *then* $\tau = \textbf{\textit{Array}}\ \tau_1$ *s.t.* $\Gamma \vdash \tau_1 <: \tau_2$ *and* $\Gamma \vdash \tau_2 <: \tau_1$.
7. *If* $\Gamma \vdash \textbf{\textit{Sh}}\ w_2\ \tau_2$ *and* $\Gamma \vdash w_2 : \textbf{\textit{ps}}\ \phi$, *then* $\tau = \textbf{\textit{Sh}}\ w_1\ \tau_1$ *s.t.* $\Gamma \vdash w_1 : \textbf{\textit{ps}}\ (\nu = w_2)$, $\Gamma \vdash \tau_1 <: \tau_2$, *and* $\Gamma \vdash \tau_2 <: \tau_1$.

8. *If* $\Gamma \vdash \tau <: x{:}\tau_1 \xrightarrow{\epsilon} \tau_2$, *then* $\tau = x{:}\tau_3 \xrightarrow{\epsilon} \tau_4$ *s.t.* $\Gamma \vdash \tau_1 <: \tau_3$ *and* $\Gamma, x : \tau_1 \vdash \tau_4 <: \tau_2$.

Proof. *(Skip)*

(1.) Only possible last rules in derivation of $\Gamma \vdash \tau <:$ **unit** are rule S-REFL (immediate) and rule S-TRANS (Use I.H. twice)

(2.) Induction on derivation of $\Gamma \vdash \tau <: \tau_1 \times \tau_2$, case analysis on the last rule used.

Rule S-REFL. We get $\tau = \tau_1 \times \tau_2$, use rule S-REFL on τ_1 and τ_2 to complete the proof.

Rule S-TRANS. We have,

(a) $\Gamma \vdash \tau <: \tau'$

(b) $\Gamma \vdash \tau' <: \tau_1 \times \tau_2$

Use I.H. on (b) to get,

(c) $\tau' = \tau_1' \times \tau_2'$

(d) $\Gamma \vdash \tau_1' <: \tau_1$

(e) $\Gamma \vdash \tau_2' <: \tau_2$

Substitute τ' from (c) in (a), and then use I.H. on (a) to get,

(f) $\tau = \tau_3 \times \tau_4$

(g) $\Gamma \vdash \tau_3 <: \tau_1'$

(h) $\Gamma \vdash \tau_4 <: \tau_2'$

Use rule S-TRANS on (g) and (d), and then (h) and (e), with (f) this completes the proof.

Rule S-PROD. Read from the rule.

(3.) Similar to (2.)

(4.) Induction on derivation of $\Gamma \vdash \tau <:$ **ps** ϕ, case analysis on the last rule used.

Rule S-REFL. $\tau =$ **ps** ϕ, and $[\![\Gamma]\!] \vDash \phi \Rightarrow \phi$ is trivially true.

Rule S-TRANS. We have,

(a) $\Gamma \vdash \tau <: \tau_1$

(b) $\Gamma \vdash \tau_1 <:$ **ps** ϕ

Use I.H. on (b), we get

(c) $\tau_1 =$ **ps** ϕ_1

(d) $[\![\Gamma]\!] \vDash \phi_1 \Rightarrow \phi$

Substitute (c) in (a) to get

(e) $\Gamma \vdash \tau <:$ **ps** ϕ_1

Use I.H. on (e), we get

(f) $\tau =$ **ps** ϕ_2

(g) $[\![\Gamma]\!] \vDash \phi_2 \Rightarrow \phi_1$

From (f) and transitivity of implication on (g) and (d), we have the proof.

Rule S-PRINCS. Read from the rule.

(5.) Proof by induction on derivation of $\Gamma \vdash \tau <:$ **W** $w_2\, \tau_2$, case analysis on the last rule used.

Rule S-REFL. We have $\tau =$ **W** $w_2\, \tau_2$, thus $\tau_1 = \tau_2$ and $w_1 = w_2$. To prove $\Gamma \vdash \tau_1 <: \tau_2$, use rule S-REFL. To prove $\Gamma \vdash_M w_2 :$ **ps** $(\nu \subseteq w_2)$, use Lemma 22 (we have $\Gamma \vdash w_2 :$ **ps** ϕ from premise.)

Rule S-TRANS. We have,

(a) $\Gamma \vdash \tau <: \tau'$

(b) $\Gamma \vdash \tau' <: \mathbf{W}\ w_2\ \tau_2$
Using I.H. (6.) on (b) to get,
(c) $\tau' = \mathbf{W}\ w_3\ \tau_3$
(d) $\Gamma \vdash w_2 : \mathbf{ps}\ (\nu \subseteq w_3)$
(e) $\Gamma \vdash \tau_3 <: \tau_2$
Using Lemma 27 on (d), we get
(f) $\Gamma \vdash \mathbf{ps}\ (\nu \subseteq w_3)$
Inverting rule WF-PRINC on (f),
(g) $\Gamma \vdash \nu \subseteq w_3$
Inverting rule WF-SUB, we get
(h) $\Gamma \vdash w_3 : \mathbf{ps}\ \phi$
Use I.H. on (a) (now that we have (h)) (substitute τ' from (c))
(i) $\tau = \mathbf{W}\ w_1\ \tau_1$
(j) $\Gamma \vdash w_3 : \mathbf{ps}\ (\nu \subseteq w_1)$
(k) $\Gamma \vdash \tau_3 <: \tau_1$
From (j), we can use rule S-PRINCS to derive:
(l) $\Gamma \vdash \mathbf{ps}\ (\nu \subseteq w_3) <: \mathbf{ps}\ (\nu \subseteq w_1)$
From (j), use Lemma 27 and inversions on rule WF-PRINC and rule WF-SUB to get
(m) $\Gamma \vdash w_1 : \mathbf{ps}\ \phi$
With (d), (l), and (m), use rule T-SUB to derive
(n) $\Gamma \vdash w_2 : \mathbf{ps}\ (\nu \subseteq w_1)$
With (e) and (k), use rule S-TRANS to complete the proof.
Rule S-WIRE. Read from the rule.
(6.) Straightforward using I.H.
(7.) Similar to (5.)
(8.) Interesting case is rule S-TRANS. We have,
(a) $\Gamma \vdash \tau <: \tau'$
(b) $\Gamma \vdash \tau' <: x{:}\tau_1 \xrightarrow{\epsilon} \tau_2$
Using I.H. (9.) on (b),
(c) $\tau' = x{:}\tau_1' \xrightarrow{\epsilon} \tau_2'$
(d) $\Gamma \vdash \tau_1 <: \tau_1'$
(e) $\Gamma, x : \tau_1 \vdash \tau_2' <: \tau_2$
Using I.H. on (a) now (with (c))
(f) $\tau = x{:}\tau_3 \xrightarrow{\epsilon} \tau_4$
(g) $\Gamma \vdash \tau_1' <: \tau_3$
(h) $\Gamma, x : \tau_1' \vdash \tau_4 <: \tau_2'$
With (d), (h), use Lemma 20, to get
(i) $\Gamma, x : \tau_1 \vdash \tau_4 <: \tau_2'$
Use rule S-TRANS on (d) and (g), and then on (i) and (e) to complete the proof.

□

Lemma 29 (Canonical forms).

1. *If* $. \vdash_M v : \textbf{\textit{unit}}$, *then* $v = ()$.
2. *If* $. \vdash_M v : \tau_1 \times \tau_2$, *then* $v = (v_1, v_2)$ *s.t.* $. \vdash_M v_1 : \tau_1$ *and* $. \vdash_M v_2 : \tau_2$.

3. *If* $. \vdash_M v : \tau_1 + \tau_2$, *then* $v = \textbf{\textit{inj}}_i\ v'$ *s.t.* $. \vdash_M v' : \tau_i$.

4. *If* $. \vdash_M v : \textbf{\textit{ps}}\ \phi$, *then* $v = w_1 \cup w_2$ *s.t.* $[\![.]\!] \vDash \phi[w_1 \cup w_2/\nu]$.

5. *If* $. \vdash_{\mathsf{p}(w')} v : \textbf{\textit{W}}\ w\ \tau$, *then* $v = v_1 +\!\!+ v_2$ *s.t.* $w \subseteq \mathsf{dom}(v_1 +\!\!+ v_2)$ *and for all* $p \in \mathsf{dom}(v_1 +\!\!+ v_2)$, $. \vdash_{\mathsf{p}(\{p\})} v[p] : \tau$.

6. *If* $. \vdash_{\mathsf{s}(w_1)} v : \textbf{\textit{W}}\ w\ \tau$, *then* $v = v_1 +\!\!+ v_2$ *s.t.* $w \subseteq \mathsf{dom}(v_1 +\!\!+ v_2)$ *and for all* $p \in \mathsf{dom}(v_1 +\!\!+ v_2)$, $. \vdash_{\mathsf{s}(w_1)} v[p] : \tau$.

7. *If* $. \vdash_M v : \textbf{\textit{Array}}\ \tau$, *then* $v = \ell$ *s.t.* $\Sigma(\ell) = \tau_1$, $. \vdash \tau_1 <: \tau$, *and* $. \vdash \tau <: \tau_1$.

8. *If* $. \vdash_M v : \textbf{\textit{Sh}}\ w\ \tau$, *then* $v = \textbf{\textit{sh}}\ w\ v'$ *s.t.* $. \vdash_M v' : \tau$.

9. *If* $. \vdash_M v : x{:}\tau_1 \xrightarrow{\epsilon} \tau_2$, *then either* $v = \textbf{\textit{clos}}\ (\psi; \lambda x.e)$ *s.t.* $\Sigma \vdash \psi \rightsquigarrow \Gamma$ *and* $\Gamma, x : \tau_1 \vdash_M e : \tau_2; \epsilon$, *or* $v = \textbf{\textit{clos}}\ (\psi; \textbf{\textit{fix}}\ f.\lambda x.e)$ *s.t.* $\Sigma \vdash \psi \rightsquigarrow \Gamma$ *and* $\Gamma, f : y{:}\tau_1 \xrightarrow{\epsilon} \tau_2 \vdash_M \lambda x.e : y{:}\tau_1 \xrightarrow{\epsilon} \tau_2; \cdot$.

10. *If* $. \vdash N$, *then* $N = \top$ *or* $N = m(w_1 \cup w_2)$.

Proof. *(Skip)* By simultaneous induction.

(1.) Proof by induction on derivation of $. \vdash_M v : \textbf{unit}$, case analysis on the last rule used.

Rule T-UNIT. Follows from the rule.

Rule T-SUB. We have,

(a) $. \vdash_M v : \tau$

(b) $. \vdash \tau <: \textbf{unit}$

With (b), use Lemma 28 to get

(c) $\tau = \textbf{unit}$

Use I.H. on (a).

(2.) Proof by induction on derivation of $. \vdash_M v : \tau_1 \times \tau_2$, case analysis on the last rule used.

Rule T-PROD. We have,

(a) $v = (v_1, v_2)$

(b) $. \vdash_M v_i : \tau_i$

Proof follows.

Rule T-SUB. We have,

(a) $. \vdash_M v : \tau$

(b) $. \vdash \tau <: \tau_1 \times \tau_2$

(c) $. \vdash \tau_1 \times \tau_2$

With (b), use Lemma 28 to get,

(d) $\tau = \tau_3 \times \tau_4$

(e) $. \vdash \tau_3 <: \tau_1$

(f) $. \vdash \tau_4 <: \tau_2$

Use I.H. on (a) (subtituting from (d)) to get,

(g) $v = (v_1, v_2)$

(h) $. \vdash v_1 : \tau_3$

(i) $. \vdash v_2 : \tau_4$
Inverting rule WF-PROD on (c),
(k) $. \vdash \tau_1$
(l) $. \vdash \tau_2$
Use rule T-SUB on (h), (e) and (k), and then (i), (f), and (l) to get rest of the proof.
(3.) Similar to rule T-PROD.
(4.) Induction on derivation of $. \vdash_M v : \mathbf{ps}\ \phi$, case analysis on the last rule used.
Rule T-PRINC. We have,
(a) $v = p$
(b) $\phi = (\nu = \{p\})$
Choose $w_1 = \{p\}$, $w_2 = \cdot$, and $[\![.]\!] \vDash (\nu = \{p\})[\{p\}/\nu]$.
Rule T-PSONE. We have,
(a) $v = \{w\}$
(b) $\phi = (\nu = \{w\})$
Choose $w_1 = \{w\}$, $w_2 = \cdot$, and then similar to rule T-PRINC.
Rule T-PSUNION. Follows similarly.
Rule T-SUB. We have,
(a) $. \vdash_M v : \tau$
(b) $. \vdash \tau <: \mathbf{ps}\ \phi$
With (b), use Lemma 28 to get,
(c) $\tau = \mathbf{ps}\ \phi_1$
(d) $[\![.]\!] \vDash \phi_1 \Rightarrow \phi$
Use I.H. on (a) (substituting from (c)),
(e) $v = w_1 \cup w_2$
(f) $\phi_1[v/\nu]$
With (e), (d), and (f), we have the proof.
(5.) Induction on derivation of $. \vdash_{\mathsf{p}(w')} v : \mathbf{W}\ w\ \tau$, case analysis on the last rule used.
Rule T-SINGLWIRE. We have,
(a) $v = \{p : v'\}$
(b) $w = \{p\}$
Choose $v_1 = \{p : v'\}$, $v_2 = \cdot$, clearly $w \subseteq \mathsf{dom}(v_1 \cup v_2)$. We need to show $. \vdash_{\mathsf{p}(\{p\})} v' : \tau$, it follows from premise of the rule.
Rule T-WIRECAT. We have,
(a) $v = v_1 +\!\!+ v_2$
(b) $w = w_1 \cup w_2$
(c) $. \vdash_{\mathsf{p}(w')} v_1 : \mathbf{W}\ w_1\ \tau$
(d) $. \vdash_{\mathsf{p}(w')} v_2 : \mathbf{W}\ w_2\ \tau$
Use I.H. on (c) to get,
(e) $w_1 \subseteq \mathsf{dom}(v_1)$
(f) for all $p \in \mathsf{dom}(v_1)$, $. \vdash_{\mathsf{p}(\{p\})} v_1[p] : \tau$
Use I.H. on (d) to get,
(g) $w_2 \subseteq \mathsf{dom}(v_2)$
(h) for all $p \in \mathsf{dom}(v_2)$, $. \vdash_{\mathsf{p}(\{p\})} v_2[p] : \tau$
From (e) and (g), we get

(g) $w_1 \cup w_2 \subseteq \mathsf{dom}(v_1 +\!\!+ v_2)$
From (f) and (h), we get
(i) for all $p \in \mathsf{dom}(v_1 +\!\!+ v_2)$, $. \vdash_{\mathsf{p}(\{p\})} (v_1 +\!\!+ v_2)[p] : \tau$
From (a), (g), and (i), we have the proof.
Rule T-SUB. We have,
(a) $. \vdash_{\mathsf{p}(w')} v : \tau'$
(b) $. \vdash \tau' <: \mathbf{W}\ w\ \tau$
(c) $. \vdash \mathbf{W}\ w\ \tau$
Invert rule WF-WIRE on (c) to get,
(c') $. \vdash \tau$
(d) $. \vdash w : \mathbf{ps}\ \phi$
With (b) and (d) use Lemma 28 to get,
(e) $\tau' = \mathbf{W}\ w_1\ \tau_1$
(f) $. \vdash w : \mathbf{ps}\ (\nu \subseteq w_1)$
(g) $. \vdash \tau_1 <: \tau$
Use I.H. (5.) on (a) (substituting τ from (c)) to get,
(h) $v = v_1 +\!\!+ v_2$
(i) $w_1 \subseteq \mathsf{dom}(v_1 +\!\!+ v_2)$
(j) for all $p \in \mathsf{dom}(v_1 +\!\!+ v_2)$, $. \vdash_{\mathsf{p}(\{p\})} v[p] : \tau_1$
Use I.H. (f) to get,
(k) $[\![.]\!] \vDash (\nu \subseteq w_1)[w/\nu]$
From (i) and (k) we get,
(l) $w \subseteq \mathsf{dom}(v_1 +\!\!+ v_2)$
Use rule T-SUB with (j), (g) and (c'), we have the proof.
(6.) Induction on derivation of $. \vdash_{\mathsf{s}(w_1)} v : \mathbf{W}\ w\ \tau$, case analysis on the last rule.
Rule T-SINGLWIRE. We have,
(a) $v = \{p : v'\}$
(b) $w = \{p\}$
(c) $. \vdash_{\mathsf{s}(w_1)} v' : \tau$
Choose $v_1 = \{p : v'\}$, $v_2 = \cdot$. We have $w \subseteq \mathsf{dom}(v_1 +\!\!+ v_2)$, and (c) completes rest of the proof.
Rule T-WIRECAT. We have,
(a) $v = v_1 +\!\!+ v_2$
(b) $. \vdash_{\mathsf{s}(w_1)} v_1 : \mathbf{W}\ w_1'\ \tau$
(c) $. \vdash_{\mathsf{s}(w_1)} v_2 : \mathbf{W}\ w_2'\ \tau$
(d) $w = w_1 \cup w_2$
Use I.H. (6.) on (b), and on (c) to get,
(e) $w_1' \subseteq \mathsf{dom}(v_1)$
(f) $w_2' \subseteq \mathsf{dom}(v_2)$
(g) for all $p \in \mathsf{dom}(v_1)$, $. \vdash_{\mathsf{s}(w_1)} v_1[p] : \tau$.
(h) for all $p \in \mathsf{dom}(v_2)$, $. \vdash_{\mathsf{s}(w_1)} v_2[p] : \tau$.
Using (d), (e), and (f), we get,
(i) $w \subseteq \mathsf{dom}(v_1 +\!\!+ v_2)$
(g), and (h) complete rest of the proof.
Rule T-SUB. We have,

(a) $. \vdash_{\mathsf{s}(w_1)} v : \tau''$
(b) $. \vdash \tau'' <: \mathbf{W}\ w\ \tau$
(c) $. \vdash \mathbf{W}\ w\ \tau$
With (b), use Lemma 28 to get,
(d) $\tau'' = \mathbf{W}\ w'\ \tau'$
(e) $. \vdash w : \mathbf{ps}\ (\nu \subseteq w')$
(f) $. \vdash \tau' <: \tau$
Use I.H. (6.) on (a) (substituting (d) in (a)),
(g) $v = v_1 +\!\!+ v_2$
(h) $w' \subseteq \mathsf{dom}(v_1 +\!\!+ v_2)$
(i) for all $p \in \mathsf{dom}(v_1 +\!\!+ v_2)$, $. \vdash_{\mathsf{s}(w_1)} v[p] : \tau'$.
Use I.H. on (e) to get,
(j) $[\![.]\!] \vDash (\nu \subseteq w')[w/\nu]$
Use (h) and (j) to get,
(k) $w \subseteq \mathsf{dom}(v_1 +\!\!+ v_2)$
Invert rule WF-WIRE on (c) to get,
(l) $. \vdash \tau$
With (i), (f), and (l), use rule T-SUB to complete rest of the proof.
(7.) Induction on derivation of $. \vdash_M v : \mathbf{Array}\ \tau$, case analysis on the last rule.
Rule T-LOC. Read from the rule, use rule S-REFL.
Rule T-SUB. We have,
(a) $. \vdash_M v : \tau'$
(b) $. \vdash \tau' <: \mathbf{Array}\ \tau$
(c) $. \vdash \mathbf{Array}\ \tau$
With (b) use Lemma 28 to get,
(d) $\tau' = \mathbf{Array}\ \tau''$
(e) $. \vdash \tau'' <: \tau$
(f) $. \vdash \tau <: \tau''$
Use I.H. (7.) on (a) (substituting τ' from (d)),
(g) $v = \ell$
(h) $\Sigma(\ell) = \tau_1$
(i) $. \vdash \tau_1 <: \tau''$
(j) $. \vdash \tau'' <: \tau_1$
Use rule S-TRANS on (i) and (e), and then (f) and (j) to complete the proof.
(8.) Induction on derivation of $. \vdash_M v : \mathbf{Sh}\ w\ \tau$, case analysis on the last rule used.
Rule T-SH. Read from the rule.
Rule T-SUB. We have,
(a) $. \vdash_M v : \tau'$
(b) $. \vdash \tau' <: \mathbf{Sh}\ w\ \tau$
(c) $. \vdash \mathbf{Sh}\ w\ \tau$
With (b), use Lemma 28 to get,
(d) $\tau' = \mathbf{Sh}\ w_2\ \tau_2$
(e) $. \vdash w_2 : \mathbf{ps}\ (\nu = w)$
(f) $. \vdash \tau_2 <: \tau$
(g) $. \vdash \tau <: \tau_2$

Use I.H. (8.) on (a) (substituting τ' from (d)),
(h) $v = \mathbf{sh}\ w\ v'$
(j) $. \vdash_M v' : \tau_2$
Invert rule WF-SH on (c) to get,
(l) $. \vdash \tau$
With (j), (f), and (l), use rule S-TRANS to complete rest of the proof.
(9.) Induction on derivation of $. \vdash_M v : x{:}\tau_1 \xrightarrow{\epsilon} \tau_2$, case analysis on the last rule used.
Rule T-CLOS. Read from the rule.
Rule T-FIXCLOS. Read from the rule.
Rule T-SUB. We have,
(a) $. \vdash_M v : \tau'$
(b) $. \vdash \tau' <: x{:}\tau_1 \xrightarrow{\epsilon} \tau_2$
(c) $. \vdash (x{:}\tau_1 \xrightarrow{\epsilon} \tau_2)$
With (b), use Lemma 28 to get,
(d) $\tau' = x{:}\tau_1' \xrightarrow{\epsilon} \tau_2'$
(e) $. \vdash \tau_1 <: \tau_1'$
(f) $., x : \tau_1 \vdash \tau_2' <: \tau_2$
Use I.H. on (a) (substituting τ' from (d)), we have 2 cases,
(i) $v = \mathbf{clos}\ (\psi; \lambda x.e)$
(g) $\Sigma \vdash \psi \rightsquigarrow \Gamma$
(h) $\Gamma, x : \tau_1' \vdash_M e : \tau_2'; \epsilon$
Use weakening and permutation of type environment on (f) and (e) to get,
(i) $\Gamma, x : \tau_1 \vdash \tau_2' <: \tau_2$
(j) $\Gamma \vdash \tau_1 <: \tau_1'$
With (h) and (j), use Lemma 20 to get (well-formedness of τ_1 follows from Lemma 27 applied on typing of v).
(k) $\Gamma, x : \tau_1 \vdash_M e : \tau_2'; \epsilon$
With (k) and (i), use rule T-SUBE to complete the proof.
Second case is,
(ii) $v = \mathbf{clos}\ (\psi; \mathbf{fix}\, f.\lambda x.e)$
(g) $\Sigma \vdash \psi \rightsquigarrow \Gamma$
(h) $\Gamma, f : y{:}\tau_1' \xrightarrow{\epsilon} \tau_2' \vdash_M \lambda x.e : y{:}\tau_1' \xrightarrow{\epsilon} \tau_2'; \cdot$.
Use weakening of type environment on (b) to get,
(i) $\Gamma \vdash y{:}\tau_1' \xrightarrow{\epsilon} \tau_2' <: x{:}\tau_1 \xrightarrow{\epsilon} \tau_2$
With (h) and (i), use Lemma 20, and then rule T-SUBE to complete the proof (similar to above).
(10.) Induction on derivation of $. \vdash N$, case analysis on the last rule used.
Rule WFPL-GLOBAL. Follows.
Rule WFPL-PC. Use I.H. on premise.

□

Lemma 30 (Delegation among closed places).
Let w be a closed principal set. Let $. \vdash m(w) \rhd N$. Then, $N = m_1(w_1)$ s.t. one of the following holds:

1. $m_1 = m$ *and* $w_1 = w$.

2. $m = \mathsf{p}$, $m_1 = \mathsf{p}$, *and* $w_1 \subseteq w$.

3. $m = \mathsf{p}$, $m_1 = \mathsf{s}$, *and* $w_1 = w$.

Proof. Induction on derivation of $. \vdash m(w) \rhd N$, case analysis on the last rule used, using Lemma 29.

□

Lemma 31 (Environment lookup same across delegation).
If $\Gamma \vdash M \rhd N$, *then* $\psi[\![x]\!]^{\Gamma}_{N} = \psi[\![x]\!]^{\Gamma}_{M}$.

Proof. Case analysis on $\psi[\![x]\!]^{\Gamma}_{M}$.
Rule VL-VAR1. We have,
(a) $\psi[\![x]\!]^{\Gamma}_{M} = v$
(b) $x \mapsto_{M_1} v \in \psi$
(c) $\Gamma \vdash M_1 \rhd M$
With (c) use Lemma 23 to get,
(d) $\Gamma \vdash M_1 \rhd N$
With (b) and (d), we get $\psi[\![x]\!]^{\Gamma}_{N} = v$
Rule VL-VAR2. Use rule VL-VAR2 again with N.
Rule VL-VAR3. Use rule VL-VAR3 again with N.

□

Lemma 32 (Well-formedness of runtime environment).
Let $\Sigma \vdash \psi \rightsquigarrow \Gamma$ *and* $\Gamma, \Gamma' \vdash M$. *Also, let* $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \phi$.

1. *If* $\Gamma, \Gamma' \vdash_M v : \tau$, *then* $\psi[\![\Gamma']\!] \vdash_{\psi[\![M]\!]} \psi[\![v]\!]^{\psi[\![\Gamma']\!]}_{\psi[\![M]\!]} : \psi[\![\tau]\!]$.

2. *If* $\Gamma, \Gamma' \vdash v : \tau$, *then* $\psi[\![\Gamma']\!] \vdash \psi[\![v]\!] : \psi[\![\tau]\!]$.

3. *If* $\Gamma, \Gamma' \vdash \tau_1 <: \tau_2$, *then* $\psi[\![\Gamma']\!] \vdash \psi[\![\tau_1]\!] <: \psi[\![\tau_2]\!]$.

4. *If* $\Gamma, \Gamma' \vdash \tau$, *then* $\psi[\![\Gamma']\!] \vdash \psi[\![\tau]\!]$.

5. *If* $\Gamma, \Gamma' \vdash \phi$, *then* $\psi[\![\Gamma']\!] \vdash \psi[\![\phi]\!]$.

6. *If* $\Gamma, \Gamma' \vdash N$, *then* $\psi[\![\Gamma']\!] \vdash \psi[\![N]\!]$.

7. *If* $\Gamma, \Gamma' \vdash \epsilon$, *then* $\psi[\![\Gamma']\!] \vdash \psi[\![\epsilon]\!]$.

8. *If* $\Gamma, \Gamma' \vdash N_1 \rhd N_2$, *then* $\psi[\![\Gamma']\!] \vdash \psi[\![N_1]\!] \rhd \psi[\![N_2]\!]$.

Proof. *(Skip)* By simultaneous induction.
(1.) By induction on derivation of $\Gamma, \Gamma' \vdash_M v : \tau$, case analysis on the last rule used.
Rule T-VAR. We have two cases here,
(a) $x :_M \tau \in \Gamma, \Gamma'$

(b) $\Gamma, \Gamma' \vdash \tau$
If $x :_M \tau \in \Gamma$, then using rule TENV-MAPP,
(c) $x \mapsto_M v \in \psi$
(d) $. \vdash_M v : \tau$
(e) M is closed.
Since M is closed,
(f) $\psi[\![M]\!] = M$
(g) $\psi[\![\Gamma']\!] \vdash M \rhd M$
Use Lemma 27 on (e),
(h) $. \vdash \tau$
Since τ is closed,
(i) $\psi[\![\tau]\!] = \tau$
With (c) and (g) use rule VL-VAR1 to get,
(j) $\psi[\![x]\!]_M^{\psi[\![\Gamma']\!]} = v$
We want to prove,
(k) $\psi[\![\Gamma']\!] \vdash_{\psi[\![M]\!]} \psi[\![x]\!]_{\psi[\![M]\!]}^{\psi[\![\Gamma']\!]} : \psi[\![\tau]\!]$.
i.e. (substitute using (f)),
(l) $\psi[\![\Gamma']\!] \vdash_M \psi[\![x]\!]_M^{\psi[\![\Gamma']\!]} : \psi[\![\tau]\!]$.
i.e. (substitute using (j) and (i)),
(m) $\psi[\![\Gamma']\!] \vdash_M v : \tau$.
which is true by weakening (d).
If $x :_M \tau \in \Gamma'$, then
(c) $x :_{\psi[\![M]\!]} \psi[\![\tau]\!] \in \psi[\![\Gamma']\!]$
Using I.H. on (b), we get,
(d) $\psi[\![\Gamma']\!] \vdash \psi[\![\tau]\!]$
Since $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \phi$, we have
(e) $x \notin \mathsf{dom}(\psi)$
And so,
(f) $\psi[\![x]\!]_{\psi[\![M]\!]}^{\psi[\![\Gamma']\!]} = x$
We want to prove,
(g) $\psi[\![\Gamma']\!] \vdash_{\psi[\![M]\!]} \psi[\![x]\!]_{\psi[\![M]\!]}^{\psi[\![\Gamma']\!]} : \psi[\![\tau]\!]$
Substitute (f), we want
(h) $\psi[\![\Gamma']\!] \vdash_{\psi[\![M]\!]} x : \psi[\![\tau]\!]$
Derive using rule T-VAR on (c) and (d).
The second case is,
(a) $x : \tau \in \Gamma, \Gamma'$
(b) $\Gamma \vdash \tau$
Split on two cases Γ and Γ' as above.
Rule T-UNIT. We have,
(a) $v = ()$
(b) $\tau =$ **unit**
Using rule VL-UNIT and rule TL-UNIT,
(c) $\psi[\![()]\!]_{\psi[\![M]\!]}^{\psi[\![\Gamma']\!]} = ()$
(d) $\psi[\![\textbf{unit}]\!] =$ **unit**

So, now we need to prove $\psi[\![\Gamma']\!] \vdash_{\psi[\![M]\!]} () : \mathbf{unit}$.
Follows from rule T-UNIT.
Rules T-INJ *and* T-PROD. Use I.H. on premises and then correponding typing rule.
Rule T-PRINC. Similar to rule T-UNIT.
Rules T-PSONE, T-PSUNION, *and* T-PSVAR. Use I.H. on premises and then corresponding typing rule.
Rule T-MSUB. We have,
(a) $\Gamma, \Gamma' \vdash N$
(b) $\Gamma, \Gamma' \vdash N \rhd M$
(c) $\Gamma, \Gamma' \vdash_N x : \tau$
Use I.H. on (a) and (c) to get,
(d) $\psi[\![\Gamma']\!] \vdash_{\psi[\![N]\!]} \psi[\![x]\!]^{\psi[\![\Gamma']\!]}_{\psi[\![N]\!]} : \psi[\![\tau]\!]$.
Use I.H. on (b),
(e) $\psi[\![\Gamma']\!] \vdash \psi[\![N]\!] \rhd \psi[\![M]\!]$
Use I.H. on (a),
(f) $\psi[\![\Gamma']\!] \vdash \psi[\![N]\!]$
With (d), (e), and (f), use rule T-MSUB to get,
(g) $\psi[\![\Gamma']\!] \vdash_{\psi[\![M]\!]} \psi[\![x]\!]^{\psi[\![\Gamma']\!]}_{\psi[\![N]\!]} : \psi[\![\tau]\!]$
With (e), use Lemma 31 to get,
(h) $\psi[\![x]\!]^{\psi[\![\Gamma']\!]}_{\psi[\![M]\!]} = \psi[\![x]\!]^{\psi[\![\Gamma']\!]}_{\psi[\![M]\!]}$
Substitute in (g) to get the proof.
(2.) Similar to proof of (1.)
(3.) Induction on derivation of $\Gamma, \Gamma' \vdash \tau_1 <: \tau_2$, case analysis on the last rule.
Rule S-REFL. Use Rule S-REFL on $\psi[\![\tau]\!]$
Rule S-TRANS. Use I.H. on premises, and then rule S-TRANS.
Rule S-SUM. Use I.H. on premises, and then rule S-SUM.
Rule S-PROD. Use I.H. on premises, and then rule S-PROD.
Rule S-PRINCS.
Rules S-WIRE, S-ARRAY, *and* S-SHARE. Use I.H. on premises and then corresponding rule.
Rule S-ARROW. We have,
(a) $\Gamma, \Gamma' \vdash \tau_1' <: \tau_1$
(b) $\Gamma, \Gamma', x : \tau_1' \vdash \tau_2 <: \tau_2'$
Use I.H. on (a),
(c) $\psi[\![\Gamma']\!] \vdash \psi[\![\tau_1']\!] <: \psi[\![\tau_1]\!]$
Use I.H. on (b),
(d) $\psi[\![\Gamma']\!], x : \psi[\![\tau_1']\!] \vdash \psi[\![\tau_2]\!] <: \psi[\![\tau_2']\!]$
With (c) and (d), use rule S-ARROW.
(4.) By induction on derivation of $\Gamma, \Gamma' \vdash \tau$, case analysis on the last rule used.
Rule WF-UNIT. Since $\psi[\![\mathbf{unit}]\!] = \mathbf{unit}$, use rule WF-UNIT.
Rule WF-SUM. Use I.H. on the premises, and then use rule WF-SUM.
Rule WF-PROD. Use I.H. on the premises, and then use rule WF-PROD.
Rule WF-PRINC. Use I.H. on the premise, and then use rule WF-PRINC.
Rule WF-ARROW. We have,

(a) $\tau = x{:}\tau_1 \xrightarrow{\epsilon} \tau_2$
(b) $\Gamma, \Gamma' \vdash \tau_1$
(c) $\Gamma, \Gamma', x : \tau_1 \vdash \epsilon$
(d) $\Gamma, \Gamma', x : \tau_1 \vdash \tau_2$
Using I.H. on (b), (c), and (d), we get,
(e) $\psi[\![\Gamma']\!] \vdash \psi[\![\tau_1]\!]$
(f) $\psi[\![\Gamma']\!], x : \psi[\![\tau_1]\!] \vdash \psi[\![\epsilon]\!]$
(g) $\psi[\![\Gamma']\!], x : \psi[\![\tau_1]\!] \vdash \psi[\![\tau_2]\!]$
Use rule WF-ARROW on (e), (f), and (g).
Rule WF-WIRE. Use I.H. on the premises, and then use rule WF-WIRE.
Rule WF-ARRAY. Use I.H. on the premise, and then use rule WF-ARRAY.
Rule WF-SHARE. Similar to rule WF-WIRE.
(5.), (6.), (7.), (8.): Induction on respective derivations.

□

Lemma 33 (Well-formedness of runtime environment).
Let $\Sigma \vdash \psi \rightsquigarrow \Gamma$ *and* $\Gamma \vdash M$.

1. *If* $\Gamma \vdash_M v : \tau$, *then* $. \vdash_{\psi[\![M]\!]} \psi[\![v]\!]_{\psi[\![M]\!]} : \psi[\![\tau]\!]$.
2. *If* $\Gamma \vdash v : \tau$, *then* $. \vdash \psi[\![v]\!] : \psi[\![\tau]\!]$.
3. *If* $\Gamma \vdash \tau_1 <: \tau_2$, *then* $. \vdash \psi[\![\tau_1]\!] <: \psi[\![\tau_2]\!]$.
4. *If* $\Gamma \vdash \tau$, *then* $. \vdash \psi[\![\tau]\!]$.
5. *If* $\Gamma \vdash \phi$, *then* $. \vdash \psi[\![\phi]\!]$.
6. *If* $\Gamma \vdash N$, *then* $. \vdash \psi[\![N]\!]$.
7. *If* $\Gamma \vdash \epsilon$, *then* $. \vdash \psi[\![\epsilon]\!]$.
8. *If* $\Gamma \vdash N_1 \rhd N_2$, *then* $. \vdash \psi[\![N_1]\!] \rhd \psi[\![N_2]\!]$.

Proof. Corollary of Lemma 32 with $\Gamma' = .$ □

Lemma 34 (Subset of environment).
If $\psi_1 \subseteq \psi_2$, $\Sigma_1 \vdash \psi_1 \rightsquigarrow \Gamma_1$, $\Sigma_1 \vdash \psi_2 \rightsquigarrow \Gamma_2$, $\Gamma_1 \vdash \tau$, *and* $\Gamma_2 \vdash \tau$, *then* $\psi_1[\![\tau]\!] = \psi_2[\![\tau]\!]$.

Lemma 35 (Value and Environment Slicing Always Exists).
For all v *and* ψ, $\mathsf{slice}_p(v) \rightsquigarrow v'$ *and* $\mathsf{slice}_p(\psi) \rightsquigarrow \psi'$.

Proof. Structural induction on v and ψ.

□

Lemma 36 (Going up the stack for slicing retains config well-typedness).

1. *If* $\Sigma \vdash m(w)\{\sigma; \kappa :: \langle m_1(w_1); \psi_1; x.e_1 \rangle ; \psi; e\} : \tau$, *then*
$\Sigma \vdash m_1(w_1)\{\sigma; \kappa; \psi_1\{x \mapsto_{m_1(w)} \bigcirc\}; e_1\} : \tau$

2. *If* $\Sigma \vdash m(w)\{\sigma; \kappa :: \langle \psi_1; x.e_1 \rangle ; \psi; e\} : \tau$, *then* $\Sigma \vdash m(w)\{\sigma; \kappa; \psi_1\{x \mapsto \bigcirc\}; e_1\} : \tau$

Lemma 37 (Existence of Slice).
If $\Sigma \vdash \mathcal{C} : \tau$ *and* $\mathcal{C}$ ***st****, then* $\mathsf{slice}_w(\mathcal{C}) \rightsquigarrow \pi$.

Proof. Let $\mathcal{C} = M\{\sigma; \kappa; \psi; e\}$. We consider $\mathsf{slice}_{\{p\}}(\mathcal{C})$. If $M = \mathsf{p}(w)$ s.t. $p \in w$, then use SLICECFG-PAR with rules STOK-PAR1, STOK-PAR2, and STOK-PAR3. If $M = \mathsf{s}(w)$ s.t. $p \in w$, then use SLICECFG-SEC with rule STOK-SEC(we have $\mathcal{C}$ **st**. When $p \notin w$, stack cannot be empty (empty stack is well typed only in a context when all parties are present, rule TSTK-EMP). Then, use SLICECFG-ABS1 or SLICECFG-ABS2 with Lemma 36 for well-typedness of inductive configurations.

□

Lemma 38 (Lookup in sliced environment).

1. *If* $\psi[\![v]\!]_{\mathsf{p}(w)} = v'$, $p \in w$, $\mathsf{slice}_p(\psi) \rightsquigarrow \psi'$, *and* $\mathsf{slice}_p(v') \rightsquigarrow v''$, *then* $\psi'[\![v]\!]_{\mathsf{p}(\{p\})} = v''$.

2. *If* $\psi[\![v]\!]_{\mathsf{s}(w)} = v'$, $\forall\ p \in w$ $\mathsf{slice}_p(\psi) \rightsquigarrow \psi_p$, $\mathsf{slice}_p(v') \rightsquigarrow v_p$, *then* $\underset{p}{\circ}\psi_p[\![v]\!]_{\mathsf{s}(w)} = \underset{p}{\circ}v_p$.

Lemma 39 (Unique local transitions).
If $p\ \{\sigma; \kappa; \psi; e\} \longrightarrow p\ \{\sigma'; \kappa'; \psi'; e'\}$, *then there exists no other rule by which* $p\ \{\sigma; \kappa; \psi; e\}$ *can step (and* σ', κ', ψ' *are unique).*

Proof. Proof sketch: By structural induction on e, and verifying that every syntactic form corresponds to one semantics rule. Moreover the unique rule is also algorithmic: the input configuration uniquely determines the output configuration, including the rule STPL-ARRAY, where the fresh location is chosen by the function $\mathsf{next}_M(\sigma)$.

□

Theorem 40 (Progress).
If $\Sigma \vdash \mathcal{C} : \tau$, *then either* $\mathcal{C}$ ***halted*** *or* $\mathcal{C} \longrightarrow \mathcal{C}'$. *Moreover if* $\mathcal{C}$ ***st****, then* $\mathcal{C}'$ ***st****.*

Proof. *(Skip)* We have $\mathcal{C} = M\{\sigma; \kappa; \psi; e\}$, $\Sigma \vdash \sigma$ **wf**, $. \vdash M$, $\Sigma \vdash_M \kappa : \tau_1 \hookrightarrow \tau_2$, $\Sigma \vdash \psi \rightsquigarrow \Gamma$, and $\Sigma; \Gamma \vdash_M e : \tau_1; \epsilon$. We proceed by induction on derivation of $\Sigma; \Gamma \vdash_M e : \tau_1; \epsilon$, case analysis on the last rule used.

Since $. \vdash M$, $\psi[\![M]\!] = M$.

Rule T-FST. We have,

(a) $e = \mathbf{fst}\,(v)$

(b) $\Gamma \vdash_M v : \tau_1 \times \tau_2$

With (b), use Lemma 33 to get,

(c) $. \vdash_M \psi[\![v]\!]_M : (\psi[\![\tau_1]\!]) \times (\psi[\![\tau_2]\!])$

With (c), use Lemma 29 to get,

(d) $\psi[\![v]\!]_M = (v_1, v_2)$

$\mathcal{C}$ can now take step using STPC-LOCAL and STPL-FST to $M\{\sigma; \kappa; \psi; v_1\}$.

Rule T-SND. Similar to rule T-FST.

Rule T-CASE. We have,
(a) $e = \textbf{case}\,(v, x_1.e_2, x_2.e_2)$
(b) $\Gamma \vdash_M v : \tau_1 + \tau_2$
(c) $\Gamma, x_i : \tau_i \vdash_M e_i : \tau; \epsilon_i$
(d) $\Gamma \vdash \tau$
(e) $\Gamma \vdash M \rhd \epsilon_i$
With (b), use Lemma 33 to get,
(d) $. \vdash_M \psi[\![v]\!]_M : (\psi[\![\tau_1]\!]) + (\psi[\![\tau_2]\!])$
With (d), use Lemma 29 to get,
(e) $\psi[\![v]\!]_M = \textbf{inj}_i\ v'$
$\mathcal{C}$ can now take a step using STPC-LOCAL and STPL-CASE to $M\{\sigma; \kappa; \psi\{x_i \mapsto v'\}; e_i\}$.
Rule T-LAM. We have,
(a) $e = \lambda x.e$
$\mathcal{C}$ can take a step using STPC-LOCAL and STPL-LAMBDA.
Rule T-APP. We have,
(a) $e = v_1\ v_2$
(b) $\Gamma \vdash_M v_1 : x{:}\tau_1 \xrightarrow{\epsilon} \tau_2$
(c) $\Gamma \vdash v_2 : \tau_1$
With (b) and (c) use Lemma 33 to get,
(d) $. \vdash_M \psi[\![v_1]\!]_M : x{:}\psi[\![\tau_1]\!] \xrightarrow{\psi[\![\epsilon]\!]} \psi[\![\tau_2]\!]$
(e) $. \vdash \psi[\![v_2]\!] : \psi[\![\tau_1]\!]$
With (d) use Lemma 29 to get,
Case 1: $\psi[\![v_1]\!]_M = \textbf{clos}\,(\psi; \lambda x.e)$
$\mathcal{C}$ can take a step using STPC-LOCAL and STPL-APPLY.
Case 1: $\psi[\![v_1]\!]_M = \textbf{clos}\,(\psi; \textbf{fix}\,x.\lambda y.e)$
$\mathcal{C}$ can take a step using STPC-LOCAL and STPL-FIXAPPLY.
Rule T-LET1. We have,
(a) $e = \textbf{let}\ x \overset{N}{=} e_1\ \textbf{in}\ e_2$
(b) $M = m(\_)$
(c) $N = \_(w)$
(d) $\Gamma \vdash M \rhd N$
(e) $\Gamma \vdash_N e_1 : \tau_1; \epsilon_1$
(f) $\Gamma, x :_{m(w)} \tau_1 \vdash_M e_2 : \tau_2; \epsilon_2$
(g) $\Gamma \vdash \tau_2$
(h) $\Gamma \vdash M \rhd \epsilon_2$
With (d), use Lemma 25 to get,
(i) $\Gamma \vdash N$
With (i), use Lemma 33 to get,
(j) $. \vdash \psi[\![N]\!]$
With (d), use Lemma 33 to get,
(k) $. \vdash M \rhd \psi[\![N]\!]$
With (k), use Lemma 30 to get,
Either
(l) $M = \mathsf{p}(w)$, $N = \mathsf{p}(w_1)$, and $w_1 \subseteq w$

In this case, $\mathcal{C}$ can take a step using STPC-DELPAR to $\mathsf{p}(w_1)\{\sigma; \kappa :: \langle \mathsf{p}(w); \psi; x.e_2 \rangle ; \psi; e_1\}$
Or
(l) $M = \mathsf{p}(w)$ and $N = \mathsf{s}(w)$.
In this case, $\mathcal{C}$ can take a step using STPC-DELSSEC to
$\mathsf{p}(w)\{\sigma; \kappa :: \langle \mathsf{p}(w); \psi; x.e_2 \rangle ; \psi; \mathbf{secure}_w(e_1)\}$
Or
(l) $M = m(w)$ and $N = m(w)$.
Depending on m, one of the above applies.
Rule T-LET2. We have,
(a) $e = \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$
(b) $\Gamma \vdash_M e_1 : \tau_1; \epsilon_1$
(c) $\Gamma, x : \tau_1 \vdash_M e_2 : \tau_2; \epsilon_2$
(d) $\Gamma \vdash \tau_2$
(e) $\Gamma \vdash M \rhd \epsilon_2$
$\mathcal{C}$ can take a step using STPC-LOCAL to $M\{\sigma; \kappa :: \langle \psi; x.e_2 \rangle ; \psi; e_1\}$.
Rule T-FIX. $\mathcal{C}$ can take a step using STPC-LOCAL and STPL-FIX.
Rule T-ARRAY. We have,
(a) $e = \mathbf{array}(v_1, v_2)$
(b) $\Gamma \vdash_M v_1 : \mathbf{nat}$
(c) $\Gamma \vdash_M v_2 : \tau$
With (b) and (c), use Lemma 33 to get,
(d) $. \vdash_M \psi[\![v_1]\!]_M : \mathbf{nat}$
(e) $. \vdash_M \psi[\![v_2]\!]_M : (\psi[\![\tau]\!])$
$\mathcal{C}$ can take a step using STPC-LOCAL and STPL-ARRAY to $M\{\sigma\{\ell :_M \{\psi[\![\bar{v_2}]\!]_M\}_k\}; \kappa; \psi; \ell\}$.
Rule T-SELECT. We have,
(a) $e = \mathbf{select}(v_1, v_2)$
(b) $\Gamma \vdash_M v_1 : \mathbf{Array}\, \tau$
(c) $\Gamma \vdash_M v_2 : \mathbf{nat}$
With (b) and (c), use Lemma 33 to get,
(d) $. \vdash_M \psi[\![v_1]\!]_M : \mathbf{Array}\, \psi[\![\tau]\!]$
(e) $. \vdash_M \psi[\![v_2]\!]_M : \mathbf{nat}$
With (d) use Lemma 29 to get,
(f) $\psi[\![v_1]\!]_M = \ell$
(g) $\Sigma(\ell) = \psi[\![\tau]\!]$
Since $\Sigma \vdash \sigma$ **wf**, $\ell \in \mathsf{dom}(\sigma)$
$\mathcal{C}$ can take a step using STPC-LOCAL with STPL-SELECT or STPL-SEL-ERR.
Rule T-UPDATE. Similar to rule T-SELECT.
Rule T-WIRE. We have,
(a) $e = \mathbf{wire}_{w_1}(v)$
(b) $M = m(w_2)$
(c) $\Gamma \vdash w_1 : \mathbf{ps}\ (\nu \subseteq w_2)$
(d) $m = \mathsf{s} \Rightarrow N = M$ and $m = \mathsf{p} \Rightarrow N = \mathsf{p}(w_1)$
(e) $\Gamma \vdash_N v : \tau$
With (c), use Lemma 33 to get,
(f) $. \vdash \psi[\![w_1]\!] : \mathbf{ps}\ (\nu \subseteq w_2)$

Case 1: $m = \mathsf{s} \Rightarrow N = \mathsf{s}(w_2)$
With (e), use Lemma 33 to get,
(g) . $\vdash_M \psi[\![v]\!]_M : \psi[\![\tau]\!]$
$\mathcal{C}$ can now take step using STPC-LOCAL and STPL-WIRE to $M\{\sigma; \kappa; \psi; \{\!|(\psi[\![v]\!]_N)|\!\}^{\mathsf{wires}}_{\psi[\![w_1]\!]}\}$.
Case 2: $m = \mathsf{p} \Rightarrow N = \mathsf{p}(w_1)$
With (e), use Lemma 33 to get,
(g) . $\vdash_{\mathsf{p}(\psi[\![w_1]\!])} \psi[\![v]\!]_{\mathsf{p}(\psi[\![w_1]\!])} : (\psi[\![\tau]\!])$
$\mathcal{C}$ can now take step using STPC-LOCAL and STPL-WIRE to $M\{\sigma; \kappa; \psi; \{\!|(\psi[\![v]\!]_N)|\!\}^{\mathsf{wires}}_{\psi[\![w_1]\!]}\}$.
Rule T-WPROJ. We have,
(a) $e = v[w_2]$
(b) $M = m(w_1)$
(c) $m = \mathsf{p} \Rightarrow \phi = \nu = w_1$ and $m = \mathsf{s} \Rightarrow \phi = \nu \subseteq w_1$
(d) $\Gamma \vdash_M v : \mathbf{W}\ w_2\ \tau$
(e) $\Gamma \vdash w_2 : \mathbf{ps}\ (\phi \wedge \mathbf{singl}(\nu))$
Case 1: $m = \mathsf{p} \Rightarrow \phi = \nu = w_1$
With (e), use Lemma 33 to get,
(f) . $\vdash (\psi[\![w_2]\!]) : \mathbf{ps}\ ((\psi[\![(\nu = w_1)]\!]) \wedge (\mathbf{singl}(\nu)))$
With (f), use Lemma 29 to get,
(g) $\psi[\![w_2]\!] = p$, $w_1 = p$, and $M = \mathsf{p}(\{p\})$
With (d), use Lemma 33 to get,
(h) . $\vdash_M \psi[\![v]\!]_M : \mathbf{W}\ \psi[\![w_2]\!]\ \psi[\![\tau]\!]$
With (h), use Lemma 29 to get,
(i) $\psi[\![v]\!]_M = v_1 +\!\!+ v_2$ and $p \in \mathsf{dom}(v_1 +\!\!+ v_2)$.
$\mathcal{C}$ can now take a step using STPC-LOCAL and STPL-PARPROJ.
Case 2: $m = \mathsf{s} \Rightarrow \phi = \nu \subseteq w_1$
With (e), use Lemma 33 to get,
(f) . $\vdash (\psi[\![w_2]\!]) : \mathbf{ps}\ ((\psi[\![(\nu \subseteq w_1)]\!]) \wedge (\mathbf{singl}(\nu)))$
With (f), use Lemma 29 to get,
(g) $\psi[\![w_2]\!] = p$, $w_1 = p \cup w''$, and $M = \mathsf{s}(\{p \cup w''\})$
With (d), use Lemma 33 to get,
(h) . $\vdash_M \psi[\![v]\!]_M : \mathbf{W}\ \psi[\![w_2]\!]\ \psi[\![\tau]\!]$
With (h), use Lemma 29 to get,
(i) $\psi[\![v]\!]_M = v_1 +\!\!+ v_2$ and $p \in \mathsf{dom}(v_1 +\!\!+ v_2)$.
$\mathcal{C}$ can now take a step using STPC-LOCAL and STPL-PARPROJ.
Rule T-WIREUN. We have,
(a) $e = v_1 +\!\!+ v_2$
(b) $\Gamma \vdash_M v_1 : \mathbf{W}\ w_1\ \tau$
(c) $\Gamma \vdash_M v_2 : \mathbf{W}\ w_2\ \tau$
With (b) and (c) use Lemma 33, and then Lemma 29, and then $\mathcal{C}$ can take a step using STPC-LOCAL and STPC-WIREUN.
Rule T-WFOLD. We have,
(a) $e = \mathbf{wfold}_w(v_1, v_2, v_3)$
(b) $M = \mathsf{s}(\_)$
(c) $\phi = \nu \subseteq w \wedge \mathbf{singl}(\nu)$

(d) $\Gamma \vdash_M v_1 : \mathbf{W}\ w\ \tau$
(e) $\Gamma \vdash_M v_2 : \tau_2$
(f) $\Gamma \vdash_M v_3 : \tau_2 \dot{\to} \mathbf{ps}\ \phi \dot{\to} \tau \dot{\to} \tau_2$
With (d) use Lemma 27 to get,
(g) $\Gamma \vdash \mathbf{W}\ w\ \tau$
With (g) invert rule WF-WIRE to get,
(h) $\Gamma \vdash w : \mathbf{ps}\ \phi$
With (h) use Lemma 33 to get,
(i) $. \vdash \psi[\![w]\!] : \mathbf{ps}\ (\psi[\![\phi]\!])$
With (i) use Lemma 29 to get,
(j) $\psi[\![w]\!] = w_1 \cup w_2$
$\mathcal{C}$ can take a step using STPC-LOCAL and either STPL-WFOLD1 or STPL-WFOLD2.
Rule T-WAPP. We have,
(a) $e = \mathbf{wapp}_w(v_1, v_2)$
(b) $M = \mathsf{p}(\_)$
(c) $\Gamma \vdash_M v_1 : \mathbf{W}\ w\ \tau_1$
(d) $\Gamma \vdash_M v_2 : \mathbf{W}\ w\ (\tau_1 \dot{\to} \tau_2)$
Similar to rule T-WFOLD now.
Rule T-WAPS. Similart to rule T-WFOLD.
Rule T-WCOPY. $\mathcal{C}$ can take a step using STPC-LOCAL with STPL-WCOPY.
Rule T-MAKESH. We have,
(a) $e = \mathbf{makesh}(v)$
(b) $M = \mathsf{s}(w)$
(c) $\Gamma \vdash_M v : \tau$
With (c) use Lemma 33 we get,
(d $. \vdash_M \psi[\![v]\!]_M : \psi[\![\tau]\!]$
$\mathcal{C}$ can take a step using STPC-LOCAL with STPL-MAKESH.
Rule T-COMBSH. We have,
(a) $e = \mathbf{combsh}(v)$
(b) $M = \mathsf{s}(w)$ (therefore w is closed)
(c) $\Gamma \vdash_M v : \mathbf{Sh}\ w\ \tau$
With (c) use Lemma 33 we get,
(d) $. \vdash_M \psi[\![v]\!]_M : \mathbf{Sh}\ w\ \psi[\![\tau]\!]$
With (d) use Lemma 29 we get,
(e) $\psi[\![v]\!]_M = \mathbf{sh}\ w\ v'$
$\mathcal{C}$ can take a step using STPC-LOCAL with STPL-COMBSH.
Rule T-SECBLK. We have,
(a) $e = \mathbf{secure}_w(e)$
(b) $M = m(w')$
(c) $\Gamma \vdash w : \mathbf{ps}\ (\nu = w')$
Use Lemma 33 and then Lemma 29 on (c). $\mathcal{C}$ can take a step using STPC-SECENTER.
Rule T-VALUE. We consider case when
(a) $e = v$
(b) $\Gamma \vdash_M v : \tau$
If κ is empty then by rule HALTED-ANSWER $\mathcal{C}$ is empty.

If κ is not empty, then depending on top frame, $\mathcal{C}$ can take a step using STPC-POPSTK1 or STPC-POPSTK2.

(After applying Lemma 33 and then Lemma 29 on (b).)

□

Theorem 41 (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. $\Sigma_2 \vdash \mathcal{C}_2 : \tau$.*

Proof. *(Skip)* Case analysis on $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$.

STPL-CASE. We have,
(a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; \textbf{case}\,(v, x_1.e_1, x_2.e_2)\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi\{x_i \mapsto v'\}; e_i\}$
(c) $\psi[\![v]\!]_M = \textbf{inj}_i\ v'$
(d) $\Sigma_1 \vdash \sigma$ **wf**
(e) $. \vdash M$
(f) $\Sigma_1 \vdash_M \kappa : \tau_1 \hookrightarrow \tau$
(g) $\Sigma_1 \vdash \psi \rightsquigarrow \Gamma$
(h) $\Sigma_1; \Gamma \vdash_M \textbf{case}\,(v, x_1.e_1, x_2.e_2) : \tau_1; \epsilon$
(i) $\Gamma \vdash M \rhd \epsilon$
Inverting rule T-CASE on (h), we get,
(j) $\Gamma \vdash_M v : \tau_1' + \tau_2'$
(k) $\Gamma, x_i : \tau_i' \vdash_M e_i : \tau_1; \epsilon_i$
(l) $\Gamma \vdash \tau$
(m') $\epsilon = \epsilon, \epsilon_1, \epsilon_2$
(m) $\Gamma \vdash M \rhd \epsilon_i$
With (j) use Lemma 33 to get,
(n) $. \vdash_M \psi[\![v]\!]_M : (\psi[\![\tau_1']\!]) + (\psi[\![\tau_2']\!])$
Substitute (c) in (n) to get,
(o) $. \vdash_M \textbf{inj}_i\ v' : (\psi[\![\tau_1']\!]) + (\psi[\![\tau_2']\!])$
Inverting rule T-INJ on (o) we get,
(p) $. \vdash_M v' : \psi[\![\tau_i']\!]$
With (p) and (g), use rule TENV-MAPP2 to get,
(q) $\Sigma_1 \vdash \psi\{x_i \mapsto v'\} \rightsquigarrow \Gamma, x_i : \tau_i'$
Choose $\Sigma_2 = \Sigma_1$
With (d), (e), (f), (q), (k), (m), use rule TCONFIG-CONFIG to get
$\Sigma_2 \vdash M\{\sigma; \kappa; \psi\{x_i \mapsto v_i\}; e_i\} : \tau$
STPL-FST. We have, (a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; \textbf{fst}\,(v)\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi; v_1\}$
(c) $\psi[\![v]\!]_M = (v_1, v_2)$
(d) $\Sigma_1 \vdash \sigma$ **wf**
(e) $. \vdash M$
(f) $\Sigma_1 \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
(g) $\Sigma_1 \vdash \psi \rightsquigarrow \Gamma$
(h) $\Sigma_1; \Gamma \vdash_M \textbf{fst}\,(v) : \tau_1; \cdot$
(i) $\Gamma \vdash M \rhd \epsilon$
Invert rule T-FST on (h) to get,

(j) $\Gamma \vdash_M v : \tau_1 \times \tau_2$

With (j) and (c) use Lemma 33 to get,

(k) $. \vdash_M v_1 : \psi[\![\tau_1]\!]$

Choose $\Sigma_2 = \Sigma_1$

With (d), (e), (f), (g), (k) (after weakening), we get,

$\Sigma_2 \vdash M\{\sigma; \kappa; \psi; v_1\} : \tau$

STPL-SND. Similar to STPL-FST.

STPL-LAMBDA. There is no change in the stack andenvironment, and the closure has same type as lambda.

STPL-APPLY. We have, (a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi_1; v_1\ v_2\}$

(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi_2\{x \mapsto v'\}; e\}$

(c) $\psi_1[\![v_1]\!]_M = \mathbf{clos}\ (\psi_2; \lambda x.e)$

(d) $\psi_1[\![v_2]\!]_M = v'$

(e) $\Sigma \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$

(f) $\Gamma \vdash_M v_1\ v_2 : \tau_1; \epsilon$

(g) $\Gamma \vdash M \rhd \epsilon$

Invert rule T-APP on (f) to get,

(h) $\Gamma \vdash_M v_1 : \tau_2 \xrightarrow{\epsilon_1} \tau_1$

(i) $\Gamma \vdash_M v_2 : \tau_2$

(j) $\Gamma \vdash M \rhd \epsilon_1[v_2/x]$

(k) $\epsilon = \epsilon_1[v_2/x]$

With (h) and (c) use Lemma 33 to get,

(l) $. \vdash_M \mathbf{clos}\ (\psi_2; \lambda x.e) : (\psi[\![\tau_2]\!]) \overset{(\psi[\![\epsilon_1]\!])}{\rightarrow} (\psi[\![\tau_1]\!])$

Invert rule T-CLOS on (l) to get,

(m) $\Sigma_1 \vdash \psi_2 \rightsquigarrow \Gamma_2$

(n) $\Gamma_2, x : \psi[\![\tau_2]\!] \vdash_M e : \psi[\![\tau_1]\!]; \psi[\![\epsilon_1]\!]$

With (i) and (d) use Lemma 33 to get,

(o) $. \vdash_M v' : \psi[\![\tau_2]\!]$

i.e.

(p) $. \vdash_M v' : \psi_2[\![(\psi[\![\tau_2]\!])]\!]$

From (m) and (o), use rule TENV-MAPP2 to derive,

(q) $\Sigma_1 \vdash \psi_2\{x \mapsto v'\} \rightsquigarrow \Gamma_2, x : \psi[\![\tau_2]\!]$

(e) can also be written as,

(r) $\Sigma \vdash_M \kappa : \psi_2[\![(\psi[\![\tau_1]\!])]\!] \hookrightarrow \tau$

With (q), (n), (r), we get $\Sigma_1 \vdash M\{\sigma; \kappa; \psi_2\{x \mapsto v'\}; e\} : \tau$ (effect delegation comes from Lemma 33 on (j).)

STPL-FIX. Similar to STPL-LAMBDA.

STPL-FIXAPPLY. Similar to STPL-APPLY.

STPL-ARRAY. Standard proof, choose Σ_2 as Σ_1 with new ℓ. Other array cases are also standard.

STPL-MAKESH. We have, (a) $\mathcal{C}_1 = \mathsf{s}(w)\{\sigma; \kappa; \psi; \mathbf{makesh}(v)\}$

(b) $\mathcal{C}_2 = \mathsf{s}(w)\{\sigma; \kappa; \psi; \mathbf{sh}\ w\ v'\}$

(c) $\psi[\![v]\!]_{\mathsf{s}(w)} = v'$

(d) $\Sigma_1 \vdash_{\mathsf{s}} (w)\kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau_2$

(e) $\Sigma_1 \vdash \psi \rightsquigarrow \Gamma$
(f) $\Gamma \vdash_{\mathsf{s}(w)} \mathbf{makesh}(v) : \tau_1; \epsilon$
Invert rule T-MAKESH on (f) to get,
(g) $\tau_1 = \mathbf{Sh}\ w\ \tau_2$
(g') $\Gamma \vdash_{\mathsf{s}(w)} v : \tau_2$
Use Lemma 33 on (g') and (c) to get,
(h) $. \vdash_{\mathsf{s}(w)} v' : \psi[\![\tau_2]\!]$
Use rule T-SH on (h) to get,
(i) $. \vdash_{\mathsf{s}(w)} \mathbf{sh}\ w\ v' : \mathbf{Sh}\ w\ \psi[\![\tau_2]\!]$
Observe that $\psi[\![(\mathbf{Sh}\ w\ \psi[\![\tau_2]\!])]\!] = \psi[\![(\mathbf{Sh}\ w\ \tau_2)]\!]$
Therefore, stack typing (d) still holds, and new configuration is well-typed.
STPL-COMBSH. Similar to STPL-MAKESH.
STPL-WIRE. We have, (a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; \mathbf{wire}_w(v)\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi; \{\!|v'|\!\}^{\mathsf{wires}}_{w'}\}$
(c) $\psi[\![w]\!] = w'$
(d) $\psi[\![v]\!]_N = v'$
(e) $M = m(w_1)$
(f) $m = \mathsf{p} \Rightarrow N = \mathsf{p}(w')$ and $m = \mathsf{s} \Rightarrow N = M$
(g) $\Sigma_1 \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
(e) $\Sigma_1 \vdash \psi \rightsquigarrow \Gamma$
(f) $\Gamma \vdash_M \mathbf{wire}_w(v) : \mathbf{W}\ w\ \tau_2; \cdot$
(g) $\tau_1 = \mathbf{W}\ w\ \tau_2$
Invert rule T-WIRE on (f) to get,
(h) $\Gamma \vdash w : \mathbf{ps}\ (\nu \subseteq w_1)$
(i) $m = \mathsf{p} \Rightarrow N_1 = \mathsf{p}(w)$ and $m = \mathsf{s} \Rightarrow N_1 = M$
(j) $\Gamma \vdash_{N_1} v : \tau_2$
Case 1: $m = \mathsf{p}$
Use Lemma 33 on (j) to get,
(k) $. \vdash_{\mathsf{p}(w')} v' : \psi[\![\tau_2]\!]$
Using rule T-SINGLWIRE and rule T-WIRECAT, we can derive:
(l) $. \vdash_{\mathsf{p}(w_1)} \{\!|v'|\!\}^{\mathsf{wires}}_{w'} : \mathbf{W}\ w'\ (\psi[\![\tau_2]\!])$
Observe that $\psi[\![\mathbf{W}\ w'\ (\psi[\![\tau_2]\!])]\!] = \psi[\![\tau_1]\!]$.
Hence, stack typing (g) still holds, therefore $\mathcal{C}_2$ is well-typed.
Case 2: $m = \mathsf{s}$
Use Lemma 33 on (j) to get,
(k) $. \vdash_M v' : \psi[\![\tau_2]\!]$
Using rule T-SINGLWIRE and rule T-WIRECAT, we can derive:
Observe that $\psi[\![\mathbf{W}\ w'\ (\psi[\![\tau_2]\!])]\!] = \psi[\![\tau_1]\!]$.
Hence, stack typing (g) still holds, therefore $\mathcal{C}_2$ is well-typed.
(l) $. \vdash_M \{\!|v'|\!\}^{\mathsf{wires}}_{w'} : \mathbf{W}\ w'\ (\psi[\![\tau_2]\!])$
STPL-PARPROJ. We have, (a') $M = \mathsf{p}(\{p\})$
(a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; v_1[v_2]\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi; v'\}$
(c) $\psi[\![v_1]\!]_M = \{p : v'\} +\!\!+ w'$
(d) $\psi[\![v_2]\!]_M = p$

(e) $\Sigma_1 \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
(f) $\Gamma \vdash_M v_1[v_2] : \tau_1; \epsilon$
Inverting rule T-WPROJ with $M = \mathsf{p}(\_)$ we get,
(g) $\Gamma \vdash_M v_1 : \mathbf{W}\, v_2\, \tau_1$
(h) $\Gamma \vdash v_2 : \mathbf{ps}\ (\nu = \{p\} \wedge \mathbf{singl}(\nu))$
Applying Lemma 33 on (h) we get,
(i) $\psi[\![v_2]\!] = \{p\}$
Applying Lemma 33 on (g) we get,
(j) $. \vdash_M \{p : v'\} +\!\!+ w' : \mathbf{W}\, \{p\}\, \psi[\![\tau_1]\!]$
Inverting rule T-WIRECAT and rule T-SINGLWIRE on (j) we get,
(k) $. \vdash_M v' : \psi[\![\tau_1]\!]$
Observe that $\psi[\![(\psi[\![\tau_1]\!])]\!] = \psi[\![\tau_1]\!]$
Hence, stack typing (e) still holds and $\mathcal{C}_2$ is well-typed.
STPL-SECPROJ. We have, (a') $M = \mathsf{s}(\{p\} \cup w)$
(a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; v_1[v_2]\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi; v'\}$
(c) $\psi[\![v_1]\!]_M = \{p : v'\} +\!\!+ w'$
(d) $\psi[\![v_2]\!]_M = p$
(e) $\Sigma_1 \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
(f) $\Gamma \vdash_M v_1[v_2] : \tau_1; \epsilon$
Similar to STPL-PARPROJ.
STPL-WIREUN. We have, (a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; v_1 +\!\!+ v_2\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi; v_1' +\!\!+ v_2'\}$
(c) $\psi[\![v_1]\!]_M = v_1'$
(d) $\psi[\![v_2]\!]_M = v_2'$
(e) $\Sigma_1 \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
(f) $\Gamma \vdash_M v_1 +\!\!+ v_2 : \tau_1; \epsilon$
Inverting rule T-WIREUN on (f), we get,
(g) $\tau_1 = \mathbf{W}\, (w_1 \cup w_2)\, \tau_2$
(h) $\Gamma \vdash_M v_1 : \mathbf{W}\, w_1\, \tau_2$
(i) $\Gamma \vdash_M v_2 : \mathbf{W}\, w_2\, \tau_2$
Applying Lemma 33 on (h) and (i), we get,
(j) $. \vdash_M v_1' : \mathbf{W}\, \psi[\![w_1]\!]\, \psi[\![\tau_2]\!]$
(k) $. \vdash_M v_2' : \mathbf{W}\, \psi[\![w_2]\!]\, \psi[\![\tau_2]\!]$
Using rule T-WIRECAT with (j) and (k) we get,
(l) $. \vdash_M v_1' +\!\!+ v_2' : \mathbf{W}\, (\psi[\![w_1]\!]) \cup (\psi[\![w_2]\!])\, \psi[\![\tau_2]\!]$
Proof now follows by showing that stack typing holds.
STPL-WAPP1. We have, (a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; \mathbf{wapp}_w(v_1, v_2)\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi; \cdot\}$
(c) $\psi[\![w]\!] = \cdot$
(d) $\Gamma \vdash_M \mathbf{wapp}_w(v_1, v_2) : \mathbf{W}\, w\, \tau_2; \cdot$
(e) $\Sigma_1 \vdash_M \kappa : \psi[\![(\mathbf{W}\, w\, \tau_2)]\!] \hookrightarrow \tau$
Rewriting (e),
(f) $\Sigma_1 \vdash_M \kappa : \mathbf{W}\ \cdot\ (\psi[\![\tau_2]\!]) \hookrightarrow \tau$
We have,

(g) $\Gamma \vdash_M \cdot : \mathbf{W}\ \cdot\ (\psi[\![\tau_2]\!])$
Also,
(h) $\psi[\![(\mathbf{W}\ \cdot\ (\psi[\![\tau_2]\!]))]\!] = \mathbf{W}\ \cdot\ (\psi[\![\tau_2]\!])$
With (h) and (f), we can derive stack typing in $\mathcal{C}_2$.
STPL-WAPP2. We have, (a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; \mathbf{wapp}_w(v_1, v_2)\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi; e\}$
(b') $M = \mathsf{p}((\{p\} \cup w') \cup w_1)$
(c) $\psi[\![w]\!] = \{p\} \cup w'$
(d) $\psi[\![v_1]\!]_M = v_1'$
(e) $\psi[\![v_2]\!]_M = v_2'$
(f) $e = \mathbf{let}\ z_1 \overset{\mathsf{p}(\{p\})}{=} \mathbf{let}\ z_2 = v_1'[p]\ \mathbf{in}\ \mathbf{let}\ z_3 = v_2'[p]\ \mathbf{in}\ z_2\ z_3\ \mathbf{in}$
$\mathbf{let}\ z_4 = \mathbf{wapp}_{w'}(v_1', v_2')\ \mathbf{in}\ (\mathbf{wire}_{\{p\}}(z_1 +\!\!+ z_4))$
(g) $\Gamma \vdash_M \mathbf{wapp}_w(v_1, v_2) : \mathbf{W}\ w\ \tau_2; \cdot$
(h) $\Gamma \vdash_M v_1 : \mathbf{W}\ w\ \tau_1$
(i) $\Gamma \vdash_M v_2 : \tau_1 \dot{\to} \tau_2$
(j) $\Sigma_1 \vdash_M \kappa : \psi[\![(\mathbf{W}\ w\ \tau_2)]\!] \hookrightarrow \tau$
Using Lemma 33 on (h) and (i) we get,
(k) $. \vdash_M v_1' : \mathbf{W}\ \{p\} \cup w'\ \psi[\![\tau_1]\!]$
(l) $. \vdash_M v_2' : \mathbf{W}\ (\{p\} \cup w')\ ((\psi[\![\tau_1]\!]) \dot{\to} (\psi[\![\tau_2]\!]))$
We now consider typing of e from (f).
Using rule T-WPROJ we get,
(m) $. \vdash_{\mathsf{p}(\{p\})} v_1'[p] : \psi[\![\tau_1]\!]; \cdot$
(n) $. \vdash_{\mathsf{p}(\{p\})} v_2'[p] : ((\psi[\![\tau_1]\!]) \dot{\to} (\psi[\![\tau_2]\!])); \cdot$
Using rule T-APP we get,
(o) $. \vdash_{\mathsf{p}(\{p\})} z_2\ z_3 : \psi[\![\tau_2]\!]; \cdot$
Using rule T-WIRE we get,
(p) $. \vdash_M \mathbf{wire}_{\{p\}}(z_1) : \mathbf{W}\ \{p\}\ \psi[\![\tau_2]\!]; \cdot$
Using rule T-WAPP we get,
(q) $. \vdash_M \mathbf{wapp}_{w'}(v_1', v_2') : \mathbf{W}\ w'\ \psi[\![\tau_2]\!]; \cdot$
Using rule T-WIREUN we get,
(r) $. \vdash_M (\mathbf{wire}_{\{p\}}(z_1 +\!\!+ z_4)) : \mathbf{W}\ (\{p\} \cup w')\ \psi[\![\tau_2]\!]; \cdot$
We also have,
(s) $\psi[\![(\mathbf{W}\ (\{p\} \cup w')\ \psi[\![\tau_2]\!])]\!] = \psi[\![(\mathbf{W}\ w\ \tau_2)]\!]$
and hence stack typing from (j) holds for $\mathcal{C}_2$ as well.
STPL-WAPS1. Similar to STPL-WAPP1.
STPL-WAPS2. We have,
(a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; \mathbf{waps}_w(v_1, v_2)\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi; e\}$
(b') $e = \mathbf{let}\ z_1 = v_1'[p]\ \mathbf{in}\ \mathbf{let}\ z_2 = v_2'\ z_1\ \mathbf{in}\ \mathbf{let}\ z_3 = \mathbf{waps}_{w'}(v_1', v_2')\ \mathbf{in}\ (\mathbf{wire}_{\{p\}}(z_2 +\!\!+ z_3))$
(c) $M = \mathsf{s}((\{p\} \cup w') \cup w_1)$
(d) $\psi[\![w]\!] = \{p\} \cup w'$
(e) $\psi[\![v_1]\!]_M = v_1'$
(f) $\psi[\![v_2]\!]_M = v_2'$

(g) $\Gamma \vdash_M \mathbf{waps}_w(v_1, v_2) : \mathbf{W}\ w\ \tau_2; \cdot$
(h) $\Sigma_1 \vdash_M \kappa : \psi[\![(\mathbf{W}\ w\ \tau_2)]\!] \hookrightarrow \tau$
Inverting rule T-WAPS on (g) we get,
(i) $\Gamma \vdash_M v_1 : \mathbf{W}\ w\ \tau_1$
(j) $\Gamma \vdash_M v_2 : \tau_1 \dot{\to} \tau_2$
Using Lemma 33 on (i) and (j),
(k) $. \vdash_M v_1' : \mathbf{W}\ (\{p\} \cup w')\ (\psi[\![\tau_1]\!])$
(l) $. \vdash_M v_2' : (\psi[\![\tau_1]\!]) \dot{\to} (\psi[\![\tau_2]\!])$
We now consider typing of e from (b')
Using rule T-WPROJ we get,
(m) $. \vdash_M v_1'[p] : \psi[\![\tau_1]\!]; \cdot$
Using rule T-APP we get,
(n) $. \vdash_M v_2'\ z_1 : \psi[\![\tau_2]\!]; \cdot$
Using rule T-WIRE we get,
(o) $. \vdash_M \mathbf{wire}_{\{p\}}(z_2) : \mathbf{W}\ \{p\}\ (\psi[\![\tau_2]\!]); \cdot$
Using rule T-WAPS we get,
(p) $. \vdash_M \mathbf{waps}_{w'}(v_1', v_2') : \mathbf{W}\ w'\ (\psi[\![\tau_2]\!]); \cdot$
Using rule T-WIREUN we get,
(q) $. \vdash_M (\mathbf{wire}_{\{p\}}(z_2 +\!\!+ z_3)) : \mathbf{W}\ (\{p\} \cup w')\ (\psi[\![\tau_2]\!]); \cdot$
Also,
(r) $\psi[\![(\mathbf{W}\ (\{p\} \cup w')\ (\psi[\![\tau_2]\!]))]\!] = \psi[\![(\mathbf{W}\ w\ \tau_2)]\!]$
and hence stack typing (h) holds for $\mathcal{C}_2$ also.
STPL-WFOLD1. Immediate from typing of **wfold** andLemma 33.
STPL-WFOLD2. We have, (a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; \mathbf{wfold}_w(v_1, v_2, v_3)\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi; e\}$
(c) $\psi[\![w]\!] = \{p\} \cup w'$
(d) $M = \mathsf{s}((\{p\} \cup w') \cup w_1)$
(e) $\psi[\![v_1]\!]_M = v_1'$
(f) $\psi[\![v_2]\!]_M = v_2'$
(g) $\psi[\![v_3]\!]_M = v_3'$
(h) $e = \mathbf{let}\ z_1 = v_1'[p]\ \mathbf{in}\ \mathbf{let}\ z_2 = v_3'\ v_2'\ p\ z_1\ \mathbf{in}\ \mathbf{wfold}_{w'}(v_1', z_2, v_3')$
(i) $\Gamma \vdash_M \mathbf{wfold}_w(v_1, v_2, v_3) : \tau_2; \cdot$
(j) $\Sigma_1 \vdash_M \kappa : \psi[\![\tau_2]\!] \hookrightarrow \tau$
Inverting rule T-WFOLD on (i) we get,
(k) $\Gamma \vdash_M v_1 : \mathbf{W}\ w\ \tau$
(l) $\Gamma \vdash_M v_2 : \tau_2$
(m) $\Gamma \vdash_M v_3 : \tau_2 \dot{\to} \mathbf{ps}\ (\nu \subseteq w \wedge \mathbf{singl}(\nu)) \dot{\to} \tau \dot{\to} \tau_2$
Using Lemma 33 on (k), (l), and (m), we get,
(n) $. \vdash_M v_1' : \mathbf{W}\ (\{p\} \cup w')\ (\psi[\![\tau]\!])$
(o) $. \vdash_M v_2' : \psi[\![\tau_2]\!]$
(p) $. \vdash_M v_3' : (\psi[\![\tau_2]\!]) \dot{\to} \mathbf{ps}\ (\nu \subseteq (\{p\} \cup w') \wedge \mathbf{singl}(\nu)) \dot{\to} (\psi[\![\tau]\!]) \dot{\to} (\psi[\![\tau_2]\!])$
We now consider typing of e from (h).
Using rule T-WPROJ, we get,
(q) $. \vdash_M v_1'[p] : \psi[\![\tau]\!]; \cdot$

Using rule T-APP we get,
(r) $. \vdash_M v_3'\ v_2'\ p\ z_1 : \psi[\![\tau_2]\!]; \cdot$
Using rule T-WFOLD we get,
(s) $. \vdash_M \mathbf{wfold}_{w'}(v_1', z_2, v_3') : \psi[\![\tau_2]\!]; \cdot$
And,
(t) $\psi[\![(\psi[\![\tau_2]\!])]\!] = \psi[\![\tau_2]\!]$
and hence stack typing (j) remains valid for $\mathcal{C}_2$.
STPC-LET. We have, (a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa :: \langle \psi; x.e_2 \rangle ; \psi; e_1\}$
(c) $\Sigma_1 \vdash \sigma\ \mathbf{wf}$
(d) $. \vdash M$
(e) $\Sigma_1 \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
(f) $\Sigma_1 \vdash \psi \rightsquigarrow \Gamma$
(g) $\Sigma_1; \Gamma \vdash_M \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_1; \epsilon$
Invert rule T-LET on (g) to get,
(h) $\Gamma \vdash_M e_1 : \tau_1'; \epsilon_1$
(i) $\Gamma, x : \tau_1' \vdash_M e_2 : \tau_1; \epsilon_2$
(j) $\epsilon = \epsilon_1, \epsilon_2$
To prove $\Sigma_2 \vdash M\{\sigma; \kappa :: \langle \psi; x.e_2 \rangle ; \psi; e_1\} : \tau$,
we need to prove
(e) $\Sigma_2 \vdash_M \kappa :: \langle \psi; x.e_2 \rangle : \psi[\![\tau_1']\!] \hookrightarrow \tau$
i.e. (from rule rule TSTK-FRAME2) we need to prove,
(f) $\Gamma, x : \tau_1' \vdash_M e : \tau_1; \epsilon_2$
and
(g) $\Sigma_2 \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
Choose $\Sigma_2 = \Sigma_1$, then (f) is same as (i) and (g) is same as (e)
Thus (e) holds.
With (c), (d), (e), (f), (h) (effect delegation follows from Lemma 27 on (h)), we have
$\Sigma_2 \vdash M\{\sigma; \kappa :: \langle \psi; x.e_2 \rangle ; \psi; e_1\} : \tau$
STPC-DELPAR. We have, (a') $M = \mathsf{p}(w_1 \cup w_2)$
(a) $\mathcal{C}_1 = M\{\sigma; \kappa; \psi; \mathbf{let}\ x \overset{\mathsf{p}(w')}{=} e_1\ \mathbf{in}\ e_2\}$
(b) $\mathcal{C}_2 = \mathsf{p}(w_2)\{\sigma; \kappa :: \langle M; \psi; x.e_2 \rangle ; \psi; e_1\}$
(c) $\psi[\![w']\!] = w_2$
(d) $\Sigma_1 \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
(e) $\Gamma \vdash_M \mathbf{let}\ x \overset{\mathsf{p}(w')}{=} e_1\ \mathbf{in}\ e_2 : \tau_1; \epsilon$
(f) $\Gamma \vdash M \rhd \epsilon$
Inverting rule T-LET1 on (e) we get,
(g) $\Gamma \vdash_{\mathsf{p}(w_2)} e_1 : \tau_2; \epsilon_1$
(h) $\Gamma, x :_{\mathsf{p}(w_2)} \tau_2 \vdash_M e_2 : \tau_1; \epsilon_2$
(i) $\epsilon = \mathsf{p}(w_2), \epsilon_2$
(j) $\Gamma \vdash \mathsf{p}(w_1 \cup w_2) \rhd \epsilon_2$
To prove $\mathcal{C}_2$ is well-typed, we need to prove:
(k) $\Gamma \vdash_{\mathsf{p}(w_2)} e_1 : \tau_2; \epsilon_1$
(l) $\Sigma_1 \vdash_{\mathsf{p}} (w_2)\kappa :: \langle M; \psi; x.e_2 \rangle : \psi[\![\tau_2]\!] \hookrightarrow \tau$

(k) follows from (g)
To prove (l), we need to prove:
(m') $\Gamma \vdash \tau_2$ (follows from Lemma 27 on (g).
(m) $\Gamma, x :_{\mathsf{p}(w_2)} \tau_2 \vdash_M e_2 : \tau_1; \epsilon$
(n) $\Sigma_1 \vdash_M \kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
(m) follows from (h), (n) follows from (d).
STPC-DELSSEC. We have, (a) $\mathcal{C}_1 = m(w)\{\sigma; \kappa; \psi; \textbf{let } x \overset{\mathsf{s}(w')}{=} e_1 \textbf{ in } e_2\}$
(b) $\mathcal{C}_2 = m(w)\{\sigma; \kappa :: \langle m(w); \psi; x.e_2\rangle ; \psi; \textbf{secure}_{w'}(e_1)\}$
(c) $\Sigma_1 \vdash_m (w)\kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
(d) $\Gamma \vdash_{m(w)} \textbf{let } x \overset{\mathsf{s}(w')}{=} e_1 \textbf{ in } e_2 : \tau_1; \epsilon$
(d') $\Gamma \vdash m(w) \rhd \mathsf{s}(w')$
Inverting rule T-LET1 on (d) we get,
(e) $\Gamma \vdash_{\mathsf{s}(w')} e_1 : \tau_2; \epsilon_1$
(f) $\Gamma, x :_{m(w')} \tau_2 \vdash_{m(w)} e_2 : \tau_1; \epsilon_2$
(g) $\epsilon = \mathsf{s}(w'), \epsilon_2$
Inverting rule D-REFL or rule D-SEC on (d') we get,
(h) $\Gamma \vdash w' : \textbf{ps}\ (\nu = w)$
Using rule T-SECBLK on (h) and (e) we get,
(i) $\Gamma \vdash_{m(w)} \textbf{secure}_{w'}(e) : \tau_2; \epsilon_1$
To prove $\mathcal{C}_2$ is well-typed, we need to prove,
(j) $\Sigma_1 \vdash_m (w)\kappa :: \langle m(w); \psi; x.e_2\rangle : \psi[\![\tau_2]\!] \hookrightarrow \tau$
i.e.
(l) $\Gamma, x :_{m(w)} \tau_2 \vdash_{m(w)} e_2 : \tau_1; \epsilon_2$
and
(m) $\Sigma_1 \vdash_m (w)\kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
(l) follows from (f) and (h), (m) follows from (c).
STPC-SECENTER. We have, (a) $\mathcal{C}_1 = m(w)\{\sigma; \kappa; \psi; \textbf{secure}_{w'}(e)\}$
(b) $\mathcal{C}_2 = \mathsf{s}(w)\{\sigma; \kappa; \psi; e\}$
(c) $\psi[\![w']\!] = w$
(d) $\Gamma \vdash_{m(w)} \textbf{secure}_{w'}(e) : \tau_1; \epsilon$
(e) $\Sigma_1 \vdash_m (w)\kappa : \psi[\![\tau_1]\!] \hookrightarrow \tau$
Inverting rule T-SECBLK on (d) we get,
(f) $\Gamma \vdash w' : \textbf{ps}\ (\nu = w)$
(g) $\Gamma \vdash_{\mathsf{s}(w')} e : \tau_1; \epsilon$
Proof now follows from (g), (f), and (e).
STPC-POPSTK1. We have, (a) $\mathcal{C}_1 = N\{\sigma; \kappa :: \langle M; \psi_1; x.e\rangle ; \psi_2; v\}$
(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi_1\{x \mapsto_{m(w)} v'\}; e\}$
(c) $N = \_(w)$
(d) $M = m(\_)$
(e) $\psi_2[\![v]\!]_N = v'$
(f) $\Sigma_1 \vdash_N \kappa :: \langle M; \psi_1; x.e\rangle : \psi_2[\![\tau_1]\!] \hookrightarrow \tau$
(g) $\Gamma_2 \vdash_N v : \tau_1$
Inverting rule TSTK-FRAME1 on (f) we get,
(h') $\Gamma_1 \vdash \tau_1$

(h) $\Gamma_1, x :_{m(w)} \tau_1 \vdash_M e : \tau_2; \epsilon$
(i) $\Sigma_1 \vdash_M \kappa : \psi_2[\![\tau_2]\!] \hookrightarrow \tau$
Using Lemma 33 on (g) to get,
(j) $. \vdash_N v' : \psi_2[\![\tau_1]\!]$
Using Lemma 34,
(j') $. \vdash_N v' : \psi_1[\![\tau_1]\!]$
Use rule TENV-MAPP2 with (j') to get,
(k) $\Sigma_1 \vdash \psi_1\{x \mapsto_{m(w)} v'\} \rightsquigarrow \Gamma_1, x :_{m(w)} \tau_1$
With (k), (h), and (i), we have the proof.
STPC-POPSTK2. Similar to STPC-POPSTK1.

□

Theorem 42 (Sound simulation).
*Let $\Sigma \vdash \mathcal{C} : \tau$, $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, $\mathcal{C}$ **st**, and $\mathsf{slice}_w(\mathcal{C}_1) \rightsquigarrow \pi_1$, where w is the set of all parties. Then, there exists π_2 s.t. $\pi_1 \longrightarrow^* \pi_2$ and $\mathsf{slice}_w(\mathcal{C}_2) \longrightarrow \pi_2$.*

Proof. Case analysis on $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$.
STPC-DELPAR. We have,
(a) $\mathcal{C}_1 = \mathsf{p}(w_1 \cup w_2)\{\sigma; \kappa; \psi; \textbf{let } x \overset{\mathsf{p}(w')}{=} e_1 \textbf{ in } e_2\}$
(b) $\mathcal{C}_2 = \mathsf{p}(w_2)\{\sigma; \kappa :: \langle \mathsf{p}(w_1 \cup w_2); \psi; x.e_2\rangle ; \psi; e_1\}$
(c) $\psi[\![w']\!] = w_2$
(d) $\mathsf{slice}_p(\mathcal{C}_1) \rightsquigarrow p\left\{\sigma'; \kappa'; \psi'; \textbf{let } x \overset{\mathsf{p}(w')}{=} e_1 \textbf{ in } e_2\right\}$, where $\mathsf{slice}_p(\sigma) \rightsquigarrow \sigma'$, $\mathsf{slice}_p(\kappa) \rightsquigarrow \kappa'$, and $\mathsf{slice}_p(\psi) \rightsquigarrow \psi'$ when $p \in w_1 \cup w_2$
(e) $\mathsf{slice}_p(\mathcal{C}_1) \rightsquigarrow \mathsf{slice}_p(m(w)\{\sigma; \kappa'; \psi'\{x \mapsto_{m(w_1 \cup w_2)} \bigcirc\}; e'\})$ when $\kappa = \kappa' :: \langle m(w); \psi'; x.e'\rangle$ or $\mathsf{slice}_p(\mathcal{C}_1) \rightsquigarrow \mathsf{slice}_p(\mathsf{p}(w_1 \cup w_2)\{\sigma; \kappa'; \psi'\{x \mapsto \bigcirc\}; e'\})$ when $\kappa = \kappa' :: \langle \psi'; x.e'\rangle$
Consider $p \in w_1 \cup w_2$. By Lemma 38,
(f) $\psi'[\![w]\!] = w_2$
Case 1. $p \in w_2$
Then it can take step using STPP-PRESENT to $p\ \{\sigma'; \kappa' :: \langle \mathsf{p}(\{p\}); \psi'; x.e_2\rangle ; \psi'; e_1\}$ which is slice of $\mathcal{C}_2$.
Case 2. $\{p\}$ not in w_2
Then it takes step using STPP-ABSENT to $p\ \{\sigma'; \kappa'; \psi'; e_2\}$ which is slice of $\mathcal{C}_2$ using SLICECFG-ABS1.
Consider $\{p\}$ not in $w_1 \cup w_2$. These parties do not take a step, and their slice remains same via SLICECFG-ABS1.
STPC-DELSSEC. In the protocol only secure agent takes a step per rule STPP-SECSTEP and STPC-DELSSEC. All other parties remain as is.
STPC-SECENTER. We have,
(a) $\mathcal{C}_1 = \mathsf{p}(w)\{\sigma; \kappa; \psi; \textbf{secure}_{w'}(e)\}$
(b) $\mathcal{C}_2 = \mathsf{s}(w)\{\sigma; \kappa; \psi; e\}$
(c) $\psi[\![w']\!] = w$
(d) $\mathsf{slice}_p(\mathcal{C}_1) \rightsquigarrow p\ \{\sigma'; \kappa'; \psi'; \textbf{secure}_{w'}(e)\}$ when $p \in w$
(e) For $\{p\}$ not in w slice is by SLICECFG-ABS1 or SLICECFG-ABS2.

For parties not in w, they do not take any step and easy to see that their slice holds in $\mathcal{C}_2$ as well.

For parties in w, we first note that $\kappa = \kappa_1 :: \langle \mathsf{p}(w); \psi'; x.e' \rangle$ (rule STOK-SECE).

Their slice in $\mathcal{C}_2$ is $p\,\{\sigma'; \kappa_1; \cdot; \textbf{wait}\}$, where $\mathsf{slice}_p(\kappa) \rightsquigarrow \kappa_1$.

The execution goes as: STPP-BEGIN, followed by STPP-SECENTER for each $p \in w$.

STPC-POPSTK1. We have,

(a) $\mathcal{C}_1 = N\{\sigma; \kappa :: \langle M; \psi_1; x.e \rangle; \psi_2; v\}$

(b) $\mathcal{C}_2 = M\{\sigma; \kappa; \psi_1\{x \mapsto_{m(w)} \psi[\![v]\!]_N\}; e\}$

(c) $M = m(\_)$

(d) $N = \_(w)$

If $p \in w$, slicing in $\mathcal{C}_2$ follows easily (since parties in N must be there in M, parties remain same or grow up the stack).

If $\{p\}$ not in w.

Depending on whether $\{p\}$ in M or not, we can prove the slicing relation on $\mathcal{C}_2$ (if $\{p\}$ in M but not in N, it cannot be the case that either M or N is secure).

STPC-LET, STPC-LOCAL, STPC-DELPSEC. Similar to STPC-DELPAR. In protocol, parties in w take same step, while others do not.

□

Theorem 43 (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 $\pi_3 \longrightarrow \pi_4$.

Proof. Proof sketch: From Lemma 39, if same agent (a party or secure agent) takes step in $\pi_1 \longrightarrow \pi_2$ and $\pi_1 \longrightarrow \pi_3$, then $\pi_2 = \pi_3$.

If different agents take step, then they can take corresponding steps in π_2 and π_3 to reach π_4.

A complete formal proof can be derived using case analysis on $\pi_1 \longrightarrow \pi_2$ and $\pi_1 \longrightarrow \pi_3$.

□

Bibliography

[1] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society.

[2] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

[3] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. ACM.

[4] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure computation of the k th-ranked element. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 40–55, 2004.

[5] Florian Kerschbaum. Automatically optimizing secure computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 703–714, New York, NY, USA, 2011. ACM.

[6] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 1–19, 2004.

[7] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[8] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt

Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security*, pages 325–343, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] Dan Bogdanov, Marko JÃţemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multiparty computation. In Rainer BÃűhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security*, volume 8975 of *Lecture Notes in Computer Science*, pages 227–234. Springer Berlin Heidelberg, 2015.

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

[11] SeungGeol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In Orr Dunkelman, editor, *Topics in CryptologyâĂŞCT-RSA 2012*, volume 7178 of *Lecture Notes in Computer Science*, pages 416–432. Springer Berlin Heidelberg, 2012.

[12] Lior Malka. Vmcrypt: Modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 715–724, New York, NY, USA, 2011. ACM.

[13] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[14] Benjamin Mood, Lara Letaw, and Kevin R. B. Butler. Memory-efficient garbled circuit generation for mobile devices. In *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*, pages 254–268, 2012.

[15] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[16] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 772–783, 2012.

[17] Benjamin Kreuter, Abhi Shelat, Benjamin Mood, and Kevin R. B. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Proceedings*

of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013, pages 321–336, 2013.

[18] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 257–266, 2008.

[19] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 62–72, New York, NY, USA, 2005. ACM.

[20] Piotr Mardziel, Michael Hicks, Jonathan Katz, Matthew Hammer, Aseem Rastogi, and Mudhakar Srivatsa. Knowledge inference for optimizing and enforcing secure computations. In *Proceedings of the Annual Meeting of the US/UK International Technology Alliance*, 2013. This short paper consists of coherent excerpts from several prior papers.

[21] Peeter Laud and Jaak Randmets. A domain-specific language for low-level secure multiparty computation protocols. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1492–1503, New York, NY, USA, 2015. ACM.

[22] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark Challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'05, pages 50–65, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: On the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM.

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[25] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 2009.

[26] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security & Privacy (Oakland)*, pages 445–462, 2013.

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

[28] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 99–110, New York, NY, USA, 2010. ACM.

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[30] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[31] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX âĂŞ Advanced Computing Systems Association, October 2014.

[32] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 655–670, 2014.

[33] Aseem Rastogi, Piotr Mardziel, Michael Hicks, and Matthew A. Hammer. Knowledge inference for optimizing secure multi-party computation. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '13, pages 3–14, New York, NY, USA, 2013. ACM.

[34] Axel Schröpfer, Florian Kerschbaum, and Günter Müller. L1 - an intermediate language for mixed-protocol secure computation. In *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2011, Munich, Germany, 18-22 July 2011*, pages 298–307, 2011.

[35] Janus Dam Nielsen and Michael I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, pages 21–30, New York, NY, USA, 2007. ACM.

[36] Aseem Rastogi, Nikhil Swamy, and Michael Hicks. Wys*: A verified language extension for secure multi-party computations, Nov. 2015.

[37] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[38] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 281–292, New York, NY, USA, 2008. ACM.

[39] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 223–234, New York, NY, USA, 2009. ACM.

[40] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[41] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[42] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.

[43] Hongwei Xi. Applied type system (extended abstract. In *In post-workshop Proceedings of TYPES 2003*, pages 394–408, 2004.

[44] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[45] Matthew Flatt, Robert Bruce Findler, and John Clements. Gui: Racket graphics toolkit. Technical Report PLT-TR-2010-3, PLT Design Inc., 2010. http://racket-lang.org/tr3/.

[46] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[47] Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, July 2009.

[48] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.

[49] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 14–25, New York, NY, USA, 2004. ACM.

[50] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

[51] Andrei Sabelfeld and AndrewC. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer Berlin Heidelberg, 2004.

[52] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 193–205, New York, NY, USA, 2014. ACM.

[53] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[54] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, 103(2):235–271, 1992.

[55] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[56] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 341–350, New York, NY, USA, 2011. ACM.

[57] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Program.*, 22(3):197–230, 1994.

[58] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[59] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[60] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, pages 100–114, 2004.

[61] Stephen Chong. Required information release. *Journal of Computer Security*, 20(6):637–676, 2012.

[62] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[63] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[64] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.

[65] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 1–14, New York, NY, USA, 2001. ACM.

[66] PPL: Parma polyhedral library. www.cs.unipr.it/ppl.

[67] LLVM. http://llvm.org.

[68] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 486–498, Berlin, Heidelberg, 2008. Springer-Verlag.

[69] Yan Huang, Peter Chapman, and David Evans. Privacy-preserving applications on smartphones. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security*, HotSec'11, pages 4–4, Berkeley, CA, USA, 2011. USENIX Association.

[70] Wilko Henecka, Stefan K ögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: Tool for automating secure two-party computations. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 451–462, New York, NY, USA, 2010. ACM.

[71] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. Unpublished, 2015. http://oblivc.org/downloads/oblivc.pdf.

[72] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09*, Irvine, pages 160–179, Berlin, Heidelberg, 2009. Springer-Verlag.

[73] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient ram-model secure computation. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 623–638, Washington, DC, USA, 2014. IEEE Computer Society.

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

[75] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology*, ICISC'05, pages 156–168, Berlin, Heidelberg, 2006. Springer-Verlag.

[76] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 51–65, Washington, DC, USA, 2013. IEEE Computer Society.

[77] John Launchbury, Iavor S. Diatchki, Thomas DuBuisson, and Andy Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 189–200, New York, NY, USA, 2012. ACM.

[78] John C. Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. Information-flow control for programming on encrypted data. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 45–60, 2012.

[79] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[80] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, August 2002.

[81] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 411–423, New York, NY, USA, 2014. ACM.

[82] Cédric Fournet, Markulf Kohlweiss, George Danezis, and Zhengqin Luo. Zql: A compiler for privacy-preserving data processing. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 163–178, Berkeley, CA, USA, 2013. USENIX Association.

[83] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 238–252, Washington, DC, USA, 2013. IEEE Computer Society.

[84] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Guillaume Davy, François Dupressoir, Benjamin Grégoire, and Pierre-Yves Strub. Verified implementations for secure and verifiable computation. *IACR Cryptology ePrint Archive*, 2014:456, 2014.

[85] Michael Backes, Matteo Maffei, and Esfandiar Mohammadi. Computationally sound abstraction and verification of secure multi-party computations. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*, pages 352–363, 2010.

[86] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 104–115, New York, NY, USA, 2001. ACM.

[87] Adam Chlipala. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 391–402, New York, NY, USA, 2013. ACM.

[88] The Coq development team. *The Coq proof assistant*.

[89] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[90] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 77–94, New York, NY, USA, 1977. ACM.

[91] Liina Kamm. *Privacy-preserving statistical analysis using secure multi-party computation*. PhD thesis, University of Tartu, 2015.

[92] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376, 2015.

[93] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology*, EUROCRYPT '07, pages 52–78, Berlin, Heidelberg, 2007. Springer-Verlag.

[94] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 681–700, 2012.

[95] Piotr Mardziel, Michael Hicks, Jonathan Katz, and Mudhakar Srivatsa. Knowledge-oriented secure multiparty computation. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, PLAS '12, pages 2:1–2:12, New York, NY, USA, 2012. ACM.

[96] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 192–206, Berlin, Heidelberg, 2008. Springer-Verlag.

[97] Node.js. https://nodejs.org/en/.

[98] Native client. https://developer.chrome.com/native-client.

[99] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 425–437, New York, NY, USA, 2014. ACM.

[100] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180, New York, NY, USA, 2015. ACM.