A Core Calculus for Mixed-mode Secure Multi-party Computations

(extend version, with technical appendix)

Aseem Rastogi Matthew A. Hammer Michael Hicks
University of Maryland, College Park
{aseem,hammer,mwh}@cs.umd.edu

Abstract

In a Secure Multi-party Computation (SMC), mutually-distrusting parties cooperate to compute over their private data; parties only learn information that is explicitly revealed as per the agreed protocol. Standard implementations of SMC, e.g., using *garbled circuits*, can be very slow, so researchers have begun to explore what we call *mixed-mode* secure computations that combine standard SMCs with local computations, resulting in orders of magnitude speedups. Ultimately, we would like to produce a compiler that takes "ordinary code" and compiles it into mixed-mode SMC. As a step in this direction, this paper presents a core calculus that aims to capture the essence of mixed-mode SMCs, and their compilation to realizable protocols. Our approach builds on Levy's call-by-push-value calculus, and permits one to specify where a computation occurs, which computations are done locally versus securely, and also which state is carried over from one secure computation to another. We show that a well-typed program in core calculus is sound (it does not get stuck), compiles properly to a target protocol, and that the operational semantics of the source program corresponds to the operational semantics of the target protocol. We show that our framework is elegant, expressive, and practical by showing how it elucidates several recently published optimized protocols, and that it naturally supports reasoning about even more general compositions of such protocols.

1. Introduction

Secure multi-party computation (SMC) protocols [2, 6, 27] 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) [1, 12];
- the x_i are private sets (implemented as arrays) and f is the $\cap$ set-intersection operator [5, 10]; one use-case is determining only those friends, interests, etc. that individuals have in common [9];
- the x_i are bids in a second-price auction, and f determines the winning bidders [4, 21]

Of course, there are many other possibilities.

A SMC for f is typically implemented using garbled circuits [27] or homomorphic encryption [22]. Unfortunately, computing a function using these techniques can be several orders of magnitude slower than computing it directly. Therefore, a recent trend has been to transform a function f into parts that must be computed securely and parts that could be computed directly by one or more

```
let median lt x1 x2 y1 y2 = as p(A,B) do
  let b1 <- as s(A,B) do force lt !x1 !y1 in
  let x3 <- as    A     do if !b1 then !x1 else !x2 in
  let y3 <- as       B  do if !b1 then !y2 else !y1 in
  let b2 <- as s(A,B) do force lt !x3 !y3 in
  if !b2 then as s(A,B) do ret !x3
         else as s(A,B) do ret !y3
```

Figure 1. Mixed-mode secure median

of the p_i without any loss of security. We call such an approach *Mixed-Mode secure MultiParty Computation*, or M^{3}PC, since it combines local and secure computations. The upside is significant: Kerschbaum [12] has shown up to a 30× performance improvement by using a mixed-mode version of secure median, compared to the standard version.

This paper presents a core calculus that aims to capture the essence of mixed-mode multi-party computations, and their compilation to realizable protocols. Our goal is to put continuing work on M^{3}PCs on a firm footing, and lay the foundation for a trustworthy compiler that handles M^{3}PCs. We make several contributions.

First, we present a rich source language, $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, in which one can express an M^{3}PC as a single program (Section 2). Using $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, the programmer can express two essential properties of mixed-mode secure computations: which parties know which computed values, and which parties compute which computations. To make these properties more manifest, $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ builds on Levy's call-by-push-value (CBPV) calculus [15], which syntactically distinguishes values from computations. $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ has a big-step, single-threaded operational semantics, which makes programs are easier to reason about, helping participants to confidently agree on the program they will run.

As an example, Figure 1 shows secure median written in (slightly sugared) $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$.[1] In this program, Alice (principal `A`) and Bob (principal `B`) each provide two elements (`x1` and `x2` for Alice, the other two inputs for Bob). The computation begins with `as p(A,B) do` e, which can be read as "do computation e independently (in parallel) on hosts belonging to parties `A` and `B`." Within this parallel computation, the first step is to bind `b1` to the result of computation `as s(A,B) do` e_1, which can be read as "do e_1 as a joint, secure computation involving `A` and `B`." Operationally, both parties synchronize, perform the secure computation, receive its result, and then carry on. The secure computation itself compares Alice and Bob's first two elements using the comparison operator `lt`. The next step is to compute `x3`. This step is performed `as A`;

[1] This sugar consists of `let`-bound functions, boolean conditionals and access to local state (e.g., `!x1`); Figure 7 gives the desugared version.

i.e., only Alice performs this computation and binding. Likewise, the binding for y3 is done only by Bob. Using the results computed individually by Alice and Bob, b2 is computed by another secure computation, and based on the result (which both parties learn), the final result is returned to both from the party that knows it.

Our second contribution is the target language $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ (Section 4), and a translation from $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ to $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ programs. The target language's small-step operational semantics captures the key elements of a distributed secure multi-party computation protocol, e.g., whether a computation is secure or public, where it is happening, and how different parties compute and communicate. The treatment of the actual SMCs is abstract; we model this part as though the computation were being run by a trusted third party. In practice we could use garbled circuits, homomorphic encryption, or any other technique, all of which may impose restrictions on the code that can be computed (e.g., looping on private inputs may be restricted).

Third, we present a type system for both $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ and $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ programs (Section 3), and prove several useful metatheoretical results (Sections 3.3 and 4.3). We show that a well-typed program in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ is sound (it does not get stuck), is secure (parties learn only what the protocol prescribes), compiles properly to a target protocol, and that the single-threaded operational semantics of the source program corresponds to the multi-threaded operational semantics of the target protocol. As such, we can confidently reason about programs according to their simple big-step semantics; i.e., one can suppose that a program like the one in Figure 1 is executed by a single thread on a single host that respects the visibility restrictions implied by the types.

Finally, we argue that our framework is elegant, expressive, and practical by showing how it elucidates several recently published protocols, and that it naturally supports reasoning about even more general compositions of such protocols (Section 2.3). For example, the fact that our language is higher-order allows us to express median as parameterized by a comparison function. The compiler also uses a novel device we call *secure state* for safely communicating results between secure computations that involve an intermediate public computation. While existing work has looked at languages for secure computation, these languages have either not considered M^{3}PCs [3, 8, 17], are lower-level and harder to reason about [7, 21, 24, 26], and/or do not provide some important formal guarantees, e.g., that compilation is correct [24, 28]. (More comparison to related work appears in Section 6.)

We view our work as an important element of a reliable strategy for producing flexible, trustworthy, and efficient secure computations from easy-to-read source programs.

2. Specifying M^{3}PCs in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$

The language $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ specifies mixed-mode multi-party computations (M^{3}PCs) in a precise manner. We introduce the formal syntax of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$. Through several examples, we informally describe typing, operational semantics and translation of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, which we make formally precise in later sections of this paper.

2.1 Design considerations

We give an overview of our design considerations for $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$.

Mixed-mode multi-party. A M^{3}PC is a composition of sub-computations (code blocks) with the specification of *where* and *how* for each of them. In particular, a sub-computation is done by a subset of parties (*where*) in either *parallel* or *secure* mode (*how*). In parallel mode, parties run common code in parallel via local computation. In secure mode, parties run common code in a coordinated fashion via secure computation. The control flow from

Real parties	$p,\ q,\ r$	$::=$	**Alice** | **Bob** | **Chuck**
Real party sets	$P,\ Q,\ R$	$::=$	$\emptyset \mid p \mid P \uplus Q$
Psuedo-party mode	$\alpha,\ \beta,\ \gamma$	$::=$	$\mathsf{par} \mid \mathsf{sec}$
Pseudo parties	$\tilde{p},\ \tilde{q},\ \tilde{r}$	$::=$	$\alpha(P)$

Figure 2. Parties, party sets, modes, and pseudo parties.

one sub-computation to another involves inter-party inter-mode *delegation*.

Delegation. Delegation permits the composition of computations that are done by different parties in different modes. For example, a computation between some parties in parallel mode can delegate control flow to a computation between a subset of those parties in secure mode. However, M^{3}PC delegations are constrained. Secure mode computations cannot delegate, as the cryptographic techniques used to implement secure mode computations do not support delegations. Moreover, for M^{3}PC to be viewed as a single program in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, parallel mode between some parties can only delegate to a mode (parallel or secure) involving a subset of those parties.

The remaining dimension of M^{3}PCs, *what* each party can observe, is captured by local state.

Local state. In a M^{3}PC, each party has a local state containing values that the party can see. Local state of a party consists of *clear* values that the party can readily use, and *secret shares*[2] that the party needs to combine with corresponding secret shares of other parties before using them. Secret shares enable communication of secure state from one secure computation to another, through parties' local states, in a secure manner.

Whether parties involved in a computation get the final value as a clear value or secret share in their local state, depends on the mode of computation, as discussed below.

Declassification. A parallel mode computation in M^{3}PC returns a clear value to its participants. The default behavior of a secure mode computation, however, is to return secret shares of the returned value to all the participants, unless explicitly declassified.

Call-by-push-value paradigm. To conveniently express various technical issues related to the specification of M^{3}PCs, such as *where* and *how* a computation occurs, we base our calculus on Levy's *Call-By-Push-Value* (CBPV) paradigm. The principle features that distinguish CBPV from other calculi stem from the way that it distinguishes *values* from *computations*. In CBPV, the distinction between values and computations is syntactic – values and computational expressions are distinct (non-overlapping) syntactic classes. This distinction is also reflected in a term's *type*: *Value types* classify values, whereas *computation types* classify expressions.

Our calculus attaches certain M^{3}PC-specific annotations on the types of CBPV computations. When the computations are treated as values (i.e., higher-order *thunks*), the types specify *where* and *how* the computation can be *forced*. When the computations produce values, the types specify *which* parties can see the value.

This extra level of detail provides a clear and flexible way of specifying M^{3}PCs, both in a program's term syntax, as well as its types.

2.2 Program syntax

We introduce the formal syntax of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ and informally describe its typing and operational semantics.

[2] k-Secret sharing is a cryptographic technique that distributes a secret value among multiple parties so that no combination of $< k$ parties can recover the secret value, but any combination of k or more parties can [25].

$$\begin{array}{llll}
\text{Values} & v & ::= & x \mid () \mid \mathbf{inj}_i\, v \mid (v_1, v_2) \mid [e] \\
 & & & \mid l \mid s \mid \mathbf{enc}_P\, v \\[1ex]
\text{Expressions} & e & ::= & \lambda x : A.e \mid e\, v \mid \mathbf{let}\, x\, \mathbf{be}\, v\, \mathbf{in}\, e \\
 & & & \mid\ \mathbf{let}\, x {\leftarrow} e_1\, \mathbf{in}\, e_2 \quad \mid\ \mathbf{ret}_{\tilde{p}} v \\
 & & & \mid\ \mathbf{case}\, (v, x_2.e_2, x_3.e_3) \\
 & & & \mid\ \mathbf{split}\, (v, x_1.x_2.e) \ \mid\ \mathbf{force}\, v \\
 & & & \mid \mathbf{fix} f{:}C.e \mid f \mid \mathbf{load}_P(v, x.e) \\
 & & & \mid\ \mathbf{comb}_P(v, x.e) \\
 & & & \mid\ \mathbf{decode}\, (v, x.e) \\
 & & & \mid \Theta.\, \mathbf{as}\, \tilde{p}\, \mathbf{do}\, e \\[1ex]
\text{Value types} & A,\ B & ::= & \mathbf{unit} \mid A + B \mid A \times B \mid \mathsf{U}^{\delta}_{\tilde{p}} C \\
 & & & \mid \mathsf{Loc}_P A \mid \mathsf{Sh}_P A \mid \mathsf{Enc}_P A \\[1ex]
\text{Comp. types} & C,\ D & ::= & A \to C \mid \mathsf{F}_{\tilde{p}} A
\end{array}$$

Figure 3. Program syntax: values, expressions and types.

Parties and modes. Figure 2 gives the formal syntax for (real) parties (denoted by p, q, r), (real) party sets (denoted by P, Q, R), computation modes (denoted by α, β, γ), and *pseudo-parties* (denoted by $\tilde{p}$, $\tilde{q}$, $\tilde{r}$) that combine the computation mode and set of parties (*how* and *where*).

Figure 3 gives the syntax for values, expressions and their types.

Values. Values in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ consist of both standard and non-standard CBPV values. Standard value forms consist of variables (denoted x), the unit value ($()$), injections ($\mathbf{inj}_i\ v$) [3], products ((v_1, v_2)) and higher-order thunks ($[e]$). The types of these values are also standard, except that the thunk types, denoted as $\mathsf{U}^{\delta}_{\tilde{p}} C$, specify the pseudo-party that is allowed to *force* (run) the thunk ($\tilde{p}$), and whether it contains control delegations or not ($\delta = \surd$ or $\emptyset$).

Non-standard values in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ consist of store locations and encodings. Locations are either *clear* locations (l) or *share* locations (s). In a party's local state, clear locations map to clear values, and share locations map to secret shares. A location can be mapped in the local state of multiple parties, as denoted by its type: $\mathsf{Loc}_P A$ classifies clear locations, and $\mathsf{Sh}_P A$ classifies share locations.

Encoded values are denoted as $\mathbf{enc}_Q\ v$ (and typed as $\mathsf{Enc}_Q A$). An encoded value $\mathbf{enc}_Q\ v$ can be decoded by *any* party in Q to yield v. However, it is opaque to parties outside Q. Encodings enable parties to communicate values to each other without revealing them. A common use case for encodings is to model private outputs from a secure computation.

Standard CBPV expressions. Expression forms in CBPV each consist of a computational step, followed by nested sub-expression that provides the computation's local continuation. Case expressions, split expressions and force expressions eliminate values of sum, product and thunk types, respectively (viz. $A + B$, $A \times B$ and $\mathsf{U}^{\delta}_{\tilde{p}} C$ respectively).

In CBPV, function abstractions $\lambda x\ :\ A.e$ are not values but rather *computations* that compute by popping an argument from a conceptual stack and binding the argument to x in e. Dually, function applications, $e\, v$, push v on the conceptual stack and compute e. [4] The **let-be** form of binding is standard, and is restricted to only bind values (i.e., not the result of a computation); we discuss general let-binding below. Recursive computations are expressed with **fix** expressions which bind a computation variable f that may appear free within its recursive body.

[3] Throughout this paper, index i ranges over indices 1 and 2.

[4] The stack-like behavior of application and lambda gives CBPV its name (i.e., function arguments pass from caller to callee by "pushing" and "popping", at least conceptually).

$\boxed{\tilde{p} \rhd_{\mathsf{del}} \tilde{q}}$ *($\tilde{p}$ may delegate control to $\tilde{q}$.)*

$$\textsc{Del-refl}\ \frac{}{\mathsf{par}(P) \rhd_{\mathsf{del}} \mathsf{par}(P)} \qquad \textsc{Del-delegate}\ \frac{}{\mathsf{par}(P \uplus Q) \rhd_{\mathsf{del}} \alpha(P)}$$

$\boxed{\tilde{p} \rhd_{\mathsf{ret}} \tilde{q}}$ *($\tilde{p}$ may produce a return value for $\tilde{q}$)*

$$\textsc{Ret-refl}\ \frac{}{\tilde{p} \rhd_{\mathsf{ret}} \tilde{p}} \qquad \textsc{Ret-declassify}\ \frac{}{\mathsf{sec}(P) \rhd_{\mathsf{ret}} \mathsf{par}(P)}$$

Figure 4. Inter-party relations: Delegation and return.

Expressions specifically for M^{3}PCs. Return expressions in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, $\mathbf{ret}_{\tilde{p}} v$, are typed as $\mathsf{F}_{\tilde{p}} A$. They carry the $\tilde{p}$ annotation to specify the receivers of the value. For $\tilde{p} = \alpha(P)$, parties in P receive clear values if $\alpha = \mathsf{par}$ and secret shares if $\alpha = \mathsf{sec}$. Figure 4 defines a relation $\tilde{p} \rhd_{\mathsf{ret}} \tilde{q}$ to specify when it is valid for a $\tilde{p}$ computation to return a value to $\tilde{q}$. The rule RET-REFL specifies the default behavior. The rule RET-DECLASSIFY allows declassification of values from a secure mode computation.

Computation sequencing and binding (denoted $\mathbf{let}\ x {\leftarrow} e_1\ \mathbf{in}\ e_2$) is standard in CBPV, and serves to sequence sub-computations that eventually produce a value via a return expression. Unlike standard CBPV, let expressions in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ implicitly allocate and initialize (write-once) locations, and bind values with an additional layer of indirection: The let-bound variable x maps to either a clear location l or share location s in the local states of parties in P, depending on the type of e_1 being $\mathsf{F}_{\mathsf{par}(P)} A$ or $\mathsf{F}_{\mathsf{sec}(P)} A$.

We additionally add expression forms that load values from parties' stores, decode encoded values, and delegate control between party sets and computation modes. A computation can recover store values from the participating parties' stores, either directly (via **load** on a clear location), or by combining secret shares (via **comb** on a share location). In both cases, the party or parties involved annotate the term. Encodings are decoded via the **decode** form. Finally, a computation can *delegate control* to a pseudo party $\tilde{p}$ via the expression $\Theta.\ \mathbf{as}\, \tilde{p}\, \mathbf{do}\ e$. Valid delegation orderings are shown in Figure 4 and reflect the intuition discussed earlier. The multi-substitution Θ is always empty in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ programs, and thus can safely be ignored at this time; we use this structure later as a technical device for explaining distributed execution.

2.3 Example M^{3}PC programs in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$

The programming patterns found in M^{3}PCs are often subtle due to the *where* and *how* dimensions of the computations. Through several examples, we now illustrate such M^{3}PC patterns and their expression in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$.

Below, we summarize our examples and the M^{3}PC patterns that they each illustrate; following this, we explain each example in greater detail:

- Figure 5 shows an abstraction for performing oblivious transfers, a ubiquitous technique found in secure computations. This (very simple) example illustrates the notion of *private outputs*, i.e., output seen by one but not all participating parties.
- Figure 6 shows an example of using secure state to avoid recomputing common code between different rounds of secure computation.
- Figure 7 shows the median example of Section 1, restated with greater detail using the syntax of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$. In particular, this syntax makes it clear *who* loads what data, and *when* it is loaded.

```
vtype dat1
vtype dat2
ctype xfer = Loc A (dat1 * dat2) -> Loc B bool
             -> F p(A,B) (Enc B (dat1 + dat2))

xfer : U s(A,B) xfer = [
 \xsl : Loc A (dat1 * dat2). -- Alice's data pair
 \bl  : Loc B bool.          -- Bob's choice
 load A ( xsl, xs.           -- load Alice's data
 split  ( xs, x1. x2.        -- split the pair
 load B ( bl, b.             -- load Bob's choice
 case   ( b, _. ret p(A,B) enc B (inj1 x1) -- ret first
           , _. ret p(A,B) enc B (inj2 x2) -- ret second
 )))) ]
```

Figure 5. A combinator for oblivious transfer.

```
vtype dat -- abstract type of data with binary ops
ctype dat_op  = dat -> dat -> F s(A,B) dat
vtype dat_opv = U s(A,B) dat_binop

op        : dat_opv     -- a secure binary operation
x1l       : Loc B dat   -- Alice's input datum
y1l, y2l  : Loc B dat   -- Bob's input data
f         : U p(A) (dat -> F p(A) dat) -- Alice's f

let r1l <- as s(A,B) do -- Secure block 1 of 2:
  let z1l <-            -- z1 = x1 op y1
    load A ( x1l, x1.   -- .. Alice loads x1
    load B ( y1l, y1.   -- .. Bob loads y1
    force op x1 y1  ))  -- .. compute x1 op y1
  in
  let z2l <-            -- z2 = z1 op y2
    comb A,B ( z1l, z1. -- .. combine shares of z1
    load B   ( y2l, y2. -- .. Bob loads y2
    force op z1 y2 ))   -- .. compute z1 op y2
  in
  comb A,B (z2l, z2.    -- combine shares of z2
  ret p(A,B) (z1l, z2)) -- z1 is secret; z2 is not
in
let x2l <- as p(A) do ( -- Alice computes x2=f(z2)
  load A ( r1l, r1.     -- .. loads prev result
  split  ( r1, _. z2.   -- .. takes value of z2
  force f z2 )))        -- .. computes f(z2)
in
as s(A,B) do            -- Secure block 2 of 2:
 let z3l <-             -- z3 = x2 op (x1 op y1)
  load A   ( x2l, x2.   -- .. Alice loads x2
  load A   ( r1l, r1.   -- .. Alice loads r1
  split    ( r1, z1l. _. -- .. project z1l from r1
  comb A,B ( z1l, z1.   -- .. combine shares of z1
  force op x2 z1 ))))   -- .. compute x2 op z1
 in
 comb A,B ( z3l, z3.    -- combine shares of z3
 ret p(A,B) z3 )        -- return z3 to p(A,B)
```

Figure 6. Example of using secret state in M^{3}PCs.

- Figure 8 shows a recently published protocol for computing private set intersection via pair-wise comparisons [10]. The algorithm is presented by [10] as a single-threaded program, implemented as a distributed computation by concurrently-executing agents, each with their own private state. This example illustrates how such M^{3}PC patterns are quite naturally expressed as (conceptually) single-threaded algorithms in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, while having systematic and formal translation into the distributed protocols of $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ (Section 4.1).
- Figure 9 shows an example of a simple two-player guessing game, where player B tries to guess the choice of player A, who tries to choose among two possibilities in such a way that player B's guess is incorrect. Based on the outcome of each round,

```
vtype dat -- abstract type of data to compare
ctype dat_lt  = dat -> dat -> F p(A,B) bool
vtype dat_ltv = U s(A,B) dat_lt
ctype median  = dat_ltv ->
                Loc A dat -> Loc A dat ->
                Loc B dat -> Loc B dat -> F p(A,B) dat

median : U p(A,B) median = [
 \lt  : dat_ltv.    -- Data ordering (less-than)
 \x1l : Loc A dat.  -- Alice's lower data
 \x2l : Loc A dat.  -- Alice's higher data
 \y1l : Loc B dat.  -- Bob's lower data
 \y2l : Loc B dat.  -- Bob's higher data

 -- first round: compare Alice's x1 and Bob's y1
 let b1l <- as s(A,B) do (
     load A ( x1l, x1.
     load B ( y1l, y1. force lt x1 y1 )))
 in
 -- local steps: Alice and Bob discard an input:
 let x3l <- as p(A) do ( -- Alice discards x1 or x2
   load A ( b1l, b1.      -- .. based on boolean b1
   case   ( b1, _. load A ( x1l, x1. ret p(A) x2 )
              , _. load A ( x2l, x2. ret p(A) x1 ) )))
 in
 let y3l <- as p(B) do ( -- Bob discards y1 or y2
   load B ( b1l, b1.      -- .. based on boolean b1
   case   ( b1, _. load B ( y2l, y2. ret p(B) y1 )
              , _. load B ( y1l, y1. ret p(B) y2 ) )))
 in
 -- second round: compare Alice's x3 and Bob's y3:
 let b2l <- as s(A,B) do (
    load A ( x3l, x3.
    load B ( y3l, y3. force lt x3 y3 )))
 in
 -- final step: (securely) communicate the result:
 load A,B ( b2l, b2. -- Alice and Bob load b2 in parallel
 case     ( b2,      -- .. b2 determines median:
   _. as s(A,B) do load A (x3l, x3. ret p(A,B) x3)
 , _. as s(A,B) do load B (y3l, y3. ret p(A,B) y3) )) ]
```

Figure 7. Median example restated (first shown in Figure 1).

each player's strategy can *adapt* when choosing in subsequent rounds. The game continues until the winning player achieves a certain (pre-arranged) number of wins over the losing player. The participants only learn the final outcome, i.e., whether they won or lost the game. They do not learn the intermediate results, or how their strategies adapt themselves during the game.

This example illustrates a M^{3}PC where the duration of the secure computation is not known a priori, and where the content of the multi-party interaction remains unknown to the participants. Another variant of the game, which we omit for space reasons, consists of a pattern resembling that of Figure 8. In this variant, the looping control occurs locally, outside the secure context. Then, rather than keep the outcome of each round secret, one could slightly alter the *knowledge profile* [23] of the program and have the players adapt locally, outside the secure context. By giving the more secure version, we show that $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ can even express the more complex variant of the game, where the participants submit their strategies as higher-order thunks that evolve within a secure context.

Preliminaries. We show programs in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ written in a concrete syntax that is nearly identical to the formal syntax given in Figure 3. However, we allow ourselves minor liberties in this concrete syntax. These differences do not enrich the calculus in any fundamental way; rather, they simply streamline the presentation.

First, we assume a facility for declaring user-defined types (both value and computation types); these types are built from other such user-defined types, together with the standard type

```
vtype dat   -- abstract type with secure equality
eq : U s(A,B)( dat -> dat -> F p(A,B) bool )

vtype st    -- abstract state of parties A and B
ctype cmd    = st -> F p(A,B) st    -- stateful command
vtype cmdv   = U p(A,B) cmd         -- command thunk
ctype pred   = st -> F p(A,B) bool  -- stateful predicate
ctype forb   = int -> cmdv -> cmd   -- for-loop body
vtype forbv  = U p(A,B) forb        -- for-loop body thunk

-- For-loop combinator:
for : U p(A,B)(int -> int -> forbv -> cmd)

-- Accessors for Alice and Bob's private state
test_idx   : U p(A,B) (int -> pred)
save_idxs  : U p(A,B) (int -> int -> bool -> cmd)
getA       : U p(A)   (int -> F p(A) dat)
getB       : U p(B)   (int -> F p(B) dat)

-- Pairwise-comparison-based Private Set Intersection
pairwise_psi : U p(A,B) ( int -> cmd )
pairwise_psi = [ \cnt:int.
 force for 1 cnt   [ \i:int. \continue_i:cmdv.
   force for 1 cnt [ \j:int. \continue_j:cmdv. \st.
     let b1l <- test_idx j st in -- is j matched?
     load A,B ( b1l, b1. case ( b1,
       _. force continue_j st   -- yes, next j ..
     ,                          -- no, compare data:
       _. let al <- (getA i) in -- only Alice
          let bl <- (getB j) in -- only Bob
          let b2l <- as sec(A,B) do
            load A ( al, a. -- Alice loads her ith
            load B ( bl, b. -- Bob   loads his jth
            force eq a b )) -- sec. equality test
          in
          load A,B ( b2l, b2. -- Both know result..
          let st'l <- save_idxs i j b2 st in
          load A,B ( st'l, st'. -- .. update states
          case( b2,             -- .. and branch:
             _. force continue_i st' ) -- ''break''
           , _. ret st' ))             -- ''continue''
     )) ]]]
```

Figure 8. The PairwiseComparison algorithm of [10].

```
vtype playerv  = U s(A,B) player -- player is a stream
ctype player   = F s(A,B) ( bool * player'v )
vtype player'v = U s(A,B) ( bool -> playerv )

ctype gamet    = U s(A,B)( int -> game_lp )
ctype game_lp  = playerv -> playerv -> int
                 -> F p(A,B) bool

-- Secure primitives over integers:
int_incr : U s(A,B)( int ->        F s(A,B) int  )
int_decr : U s(A,B)( int ->        F s(A,B) int  )
int_absc : U s(A,B)( int -> int -> F s(A,B) bool )
int_sign : U s(A,B)( int ->        F s(A,B) bool )

game : gamev = [
 \win_diff : int.      -- Winning margin for game
 fix loop  : game_lp.  -- Game loop; iterations are rounds
 \playerA  : playerv.  -- Player A's current strategy
 \playerB  : playerv.  -- Player B's current strategy
 \diff     : int.      -- Current score (as a difference)

 -- Move each player; advance their stream
 let al <- force playerA in -- Player A advances
 let bl <- force playerB in -- Player B advances
 comb A,B ( al, a. split ( a, moveA. tellA.
 comb A,B ( bl, b. split ( b, moveB. tellB.

 -- Compare their moves; determine round winner:
 let cl <- force bool_xor moveA moveB in
 let dl <- comb A,B ( cl, c. case ( c,
  _. int_incr diff,   -- player A gets point
  _. int_decr diff )) -- player B gets point
 in
 -- Update point diff; test for end of game:
 comb A,B ( dl, diff'.
 let donel <- force int_absc diff' win_diff in
 comb A,B ( donel, done. case ( done,
  _. let winnerl <- force int_sign diff' in
     comb A,B ( winnerl, winner.
     ret p(A,B) winner ) -- reveal the winner!
  ,                      -- otherwise, continue the game:
  _. let playerA'l <- force tellA c in -- update player A
     let playerB'l <- force tellB c in -- update player B
     comb A,B ( playerA'l, playerA'.
     comb A,B ( playerB'l, playerB'.
     loop playerA' playerB' diff' )) -- next round!
 ))))))) ]
```

Figure 9. Two-player guessing game.

connectives given in Figure 3. We denote product types with `*`, as in `dat1 * dat2`. Next, while the formal syntax of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ uses **let** x **be** $(v : A)$ **in** e to bind the value constant v (of type A) to the variable x, in the examples we separately write the type annotation as `x : A`, and the value definition as `x = v`. To be abstract in a variable's value, we give its type but intentionally omit its corresponding value definition. When listing party annotations, we shorten par(**Alice** $\uplus$ **Bob**) to `p(A,B)` and similarly, we shorten sec(**Alice** $\uplus$ **Bob**) to `s(A,B)`. Finally, we use a library of common type and value definitions, including booleans (implemented as sums of units, i.e., `vtype bool = unit + unit`), integers (left abstract) and imperative control constructs.

Example: Oblivious transfer. Figure 5 gives a function for performing an *oblivious transfer*. An oblivious transfer is a fundamental secure computation consisting of two parties, Alice and Bob. Alice's inputs in the computation are two datums and Bob's input is a boolean value. Depending on Bob's input, Alice transfers one of two datums to Bob in a way that, Bob learns only the transferred datum (and not the other datum), and Alice does not learn Bob's choice. In $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, this computation consists of several steps, all of which occur in secure mode: Alice loads her two datums; Bob loads his choice; Depending on this choice, either the first or second datum is *encoded* for Bob, so that Bob learns one (and only one) of the datums, and Alice does not learn the choice.

The party annotations on the types of the computation indicate that the transfer occurs in a secure mode; this fact is witnessed by the `U s(A,B)` prefix of value `xfer`'s type. Dually, the computation type `xfer` also indicates that the final result value (an encoding for Bob only) is declassified; this fact is witnessed by the suffix `F p(A,B)` in the type `xfer`. This example illustrates how $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ exploits the CBPV approach to typing values and computations to indicate the alternation ("mixing") of *modes* across control and data flow.

Example: Secret sharing. Figure 6 shows an example of a M^{3}PC that consists of two secure rounds, where secret information in the first round is saved and reused in the second round. This example illustrates a general pattern of M^{3}PCs: Intermediate results computed in secure blocks can be saved and reused in subsequent blocks without revealing these results as clear values to any of the participating parties.

In each of the two rounds, Alice and Bob compute (abstract) binary operations over their datums. After the first round, Alice and Bob declassify the value of variable `z2`, but keep the value of variable `z1` secret, saving only its corresponding share location. Alice uses the value of `z2` to locally compute a new value (`x2`), which she provides in the second secure round. In this second round, Alice and Bob again use the value of `z1`, which they avoid recomputing by combining their secret shares from the share location `z1l`.

Example: Mixed-mode median. Figure 7 gives the (mixed mode) median example in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$. The mixed mode version of the median computation consists of an optimization that reveals `b1` and `b2` to Alice and Bob early in the computation. The parties use this declassified information to compute `x3` and `y3` locally (as opposed to computing in a secure cooperative way). The final secure computation in the optimized version just consists of secure communication (not a secure computation). This difference is witnessed by the fact that it contains no nested sub-computations, and ends immediately with a `ret` expression.

Example: Private set intersection. Figure 8 shows the recently published pair-wise-comparison algorithm for computing a private set intersection between two parties' private data items [10]. This variant of the algorithm uses two **for**-loops to structure the control flow of the mixed mode algorithm. Before the algorithm begins, each party randomly shuffles their data items, and pads their private sets to be a common size; these steps are abstracted over by the `getA` and `getB` operations provided by the two parties. In the inner body of the loops, the parties compare their data items (by accessing pairs of indices); after each comparison, the parties learn whether or not their items are "equivalent" (a comparison operation that we leave abstract). When equal, the parties avoid recomparing these items by marking their indices, and using the "continue" control operator provided by the **for**-loop to skip over redundant iterations of the inner body. This **for**-loop control structure, including its support for "continue", can be encoded directly in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ using only standard features of the calculus.

Example: Guessing game. Figure 9 gives the code for a two-player guessing game, which proceeds in rounds until one of the players wins at least `win_diff` more (total) rounds than the other player. The novelty of this game is that, though the playing strategies can adapt to the outcome of each round, the participants do not learn anything other than the final outcome of the game, and perhaps its duration. Alice and Bob play the game by each supplying a "player" value, i.e., a stream of thunks that make guesses and evolve recursively as the game proceeds. In each round, the two players each make a guess; the first player gets a point if their choice is different from the second; otherwise, the second player gets a point. The point tally is kept as a (signed) integer difference; the first (resp. second) player wins when the negative (resp. positive) difference exceeds the parameter `win_diff`.

3. Typing and operational semantics of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$

Building on the intuitions developed in the previous section, we give a precise account of the typing and operational semantics of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$.

3.1 Type system

Figure 10 gives the value and expression typing judgments, which are mutually-referential. The judgment forms consist of type environments for variables (Γ) and store locations (Σ), as well as a delegation switch (δ), with the following syntaxes:

Multi variable-typing env.	Γ	$::=$	$\varepsilon \mid \Gamma[p{:}\Upsilon]$
Local variable-typing env.	Υ	$::=$	$\varepsilon \mid \Upsilon, x : A \mid \Upsilon, f : C$
Multi store-typing env.	Σ	$::=$	$\varepsilon \mid \Sigma[p{:}\Delta]$
Local store-typing env.	Δ	$::=$	$\varepsilon \mid \Delta, l : A \mid \Delta, s : A$
Delegation switch	δ	$::=$	$\surd \mid \oslash$

Additionally, the judgments are parameterized by an ambient pseudo party ($\tilde{p}$).

The environments Σ and Γ provide a spatial dimension to the usual store and variable typing environments (denoted Δ and Υ respectively), corresponding to *where* the location or variable binding exists; more precisely, a coordinate in this spatial dimension corresponds to a party p participating in the computation. The local store-typing environments, Δ, map clear and share locations to value types A. The local variable-typing environments, Υ, map value and computation variables to their corresponding types. We denote the extension of a multi variable-typing environment Γ with the binding $x : A$ for all parties $p \in P$ as $\Gamma[P \mapsto x : A]$.

At a high level, the type system of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ ensures that programs respect the restrictions inherent to the special features of M^{3}PCs, namely, delegation, local state and encodings. Delegation also gives rise to a related notion, which we call translation independence. We explain the typing rules in Figure 10 by organizing our discussion around these contexts of concern.

Delegation. The type system enforces that control flow respects the delegation relation (Figure 4). This enforcement is most evident in its typing of delegation expressions, via rules TY-ASDOPAR and TY-ASDOSEC. In both rules, the delegating pseudo party ($\tilde{p}$) must be related to the delegation party (by the relation $\rhd_{ret}$), and the typing context is restricted (via the restriction $|\Gamma|_{\mathsf{vals}}$) to contain only value bindings, and no computation bindings (which may correspond to code for unrelated pseudo parties). Further, in the typing of thunk values (rules TY-THK and TY-DTHK) and force expressions (rule TY-FORCE), we note that the ambient pseudo party is invariant in the premise and conclusion; this invariance ensures that the delegation relation is respected even in the case of higher-order control flow (i.e., when a thunk is forced).

Translation independence. A notion related to delegation is that of translation independence, which we make precise in Section 4.3. Informally, we say that a value or expression is *translation independent* its corresponding translation in the distributed protocol is independent of the party for whom it is translated. Since it alters the ambient pseudo party, delegation gives rise to different translations, and hence must be controlled appropriately. In particular, the type system ensures that the data and code used in secure contexts is translation independent; this property is crucial, since this data and code should be consistent across all parties participating in the secure context.

To enforce this property, rules TY-ASDOPAR and TY-ASDOSEC allow delegation only when the delegation switch is $\surd$, and prohibit delegation otherwise. The rule TY-ASDOSEC prohibits delegation in secure contexts, where it types code under the delegation switch $\oslash$. Further, the rule restricts the typing context (via the restriction $[\![\Gamma]\!]$) to contain only those value bindings whose types are translation independent. A value type A is translation independent, written $[\![A]\!]$, if all occurences of the delegation switch δ appear as $\oslash$. This restriction applies specifically to thunk types, where $[\![\mathsf{U}^{\surd}_{\tilde{p}}\, C]\!]$ is undefined; the remaining cases of the predicate are congruences, and in the base case, $[\![\textbf{unit}]\!]$ always holds.

Finally, we note that secure blocks carry delayed mutli--substitutions Θ outside of their code body until that code body is ready to run. This substitution behavior is unobservable, but helps in relating the semantics of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ programs to their translated protocols, a point that we return to in Section 3.2. To type this suspended substitution, the rule TY-ASDOSEC includes a premise $[\![|\Gamma|_{\mathsf{vals}}]\!]; \Sigma \vdash \Theta$ that witnesses that this delayed mutli-substitution is well-typed.

Local state and encodings. The type system enforces that dataflow respects the restrictions inherent to clear locations, share locations and encodings. First, the values of clear locations and share locations are introduced by rules TY-LETPAR and TY-LETSEC, respectively, where the computation type indicates whether the produced value is bound as a clear or share location (rules TY-LETPAR and TY-LETSEC, respectively). Next, the values of clear locations may be loaded in secure mode if one party can provide the value (rule TY-LOADSEC); in parallel mode, all participating parties must perform the load (rule TY-LOADPAR). The values

$\boxed{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} v : A}$ *(Under Γ, Σ, δ and $\tilde{p}$, value v has value type A.)*

$$\textsc{Ty-vvar}\ \frac{\forall\, p \in P.\, \Gamma[p](x) = A}{\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} x : A} \qquad \textsc{Ty-unit}\ \frac{}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} () : \textbf{unit}} \qquad \textsc{Ty-inj}\ \frac{i \in \{1,2\} \qquad \Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} v : A_i}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} \textbf{inj}_i\, v : A_1 + A_2} \qquad \textsc{Ty-prod}\ \frac{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} v_1 : A \qquad \Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} v_2 : B}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} (v_1, v_2) : A \times B}$$

$$\textsc{Ty-ndthk}\ \frac{\Gamma; \Sigma \vdash^{\oslash}_{\tilde{p}} e : C}{\Gamma; \Sigma \vdash^{\oslash}_{\tilde{p}} [e] : \mathsf{U}^{\oslash}_{\tilde{p}} C} \qquad \textsc{Ty-dthk}\ \frac{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} e : C}{\Gamma; \Sigma \vdash^{\surd}_{\tilde{p}} [e] : \mathsf{U}^{\delta}_{\tilde{p}} C} \qquad \textsc{Ty-cloc}\ \frac{\forall\, q \in Q.\, \Sigma[q](l) = A}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} l : \mathsf{Loc}_Q A} \qquad \textsc{Ty-sloc}\ \frac{\forall\, q \in Q.\, \Sigma[q](s) = A}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} s : \mathsf{Sh}_Q A} \qquad \textsc{Ty-enc}\ \frac{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} v : A}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} \textbf{enc}_Q\, v : \mathsf{Enc}_Q A}$$

$\boxed{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} e : C}$ *(Under Γ, Σ, δ and $\tilde{p}$, expression e has computation type C.)*

$$\textsc{Ty-lam}\ \frac{\Gamma[P \Mapsto x : A]; \Sigma \vdash^{\delta}_{\alpha(P)} e : C}{\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} \lambda x : A.e : A \to C} \qquad \textsc{Ty-app}\ \frac{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} e : A \to C \qquad \Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} v : A}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} e\, v : C} \qquad \textsc{Ty-letbe}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} v : A \\ \Gamma[P \Mapsto x : A]; \Sigma \vdash^{\delta}_{\alpha(P)} e_2 : C\end{array}}{\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} \textbf{let}\, x\, \textbf{be}\, v\, \textbf{in}\, e_2 : C}$$

$$\textsc{Ty-letPar}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} e_1 : \mathsf{F}_{\mathsf{par}(Q)} A \qquad \llbracket A \rrbracket = A \\ \Gamma[Q \Mapsto x : \mathsf{Loc}_Q A]; \Sigma \vdash^{\delta}_{\tilde{p}} e_2 : C\end{array}}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} \textbf{let}\, x \leftarrow e_1\, \textbf{in}\, e_2 : C} \qquad \textsc{Ty-letSec}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} e_1 : \mathsf{F}_{\mathsf{sec}(Q)} A \qquad \llbracket A \rrbracket = A \\ \Gamma[Q \Mapsto x : \mathsf{Sh}_Q A]; \Sigma \vdash^{\delta}_{\tilde{p}} e_2 : C\end{array}}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} \textbf{let}\, x \leftarrow e_1\, \textbf{in}\, e_2 : C} \qquad \textsc{Ty-ret}\ \frac{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} v : A \qquad \tilde{p} \rhd_{\mathsf{ret}} \tilde{q}}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} \textbf{ret}_{\tilde{q}} v : \mathsf{F}_{\tilde{q}} A}$$

$$\textsc{Ty-case}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} v : A + B \\ \Gamma[P \Mapsto x_1 : A]; \Sigma \vdash^{\delta}_{\alpha(P)} e_1 : C \\ \Gamma[P \Mapsto x_2 : B]; \Sigma \vdash^{\delta}_{\alpha(P)} e_2 : C\end{array}}{\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} \textbf{case}\, (v, x_1.e_1, x_2.e_2) : C} \qquad \textsc{Ty-split}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} v : A \times B \\ \Gamma[P \Mapsto x_1 : A, x_2 : B]; \Sigma \vdash^{\delta}_{\alpha(P)} e : C\end{array}}{\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} \textbf{split}\, (v, x_1.x_2.e) : C} \qquad \textsc{Ty-force}\ \frac{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} v : \mathsf{U}^{\delta'}_{\tilde{p}} C}{\Gamma; \Sigma \vdash^{\delta}_{\tilde{p}} \textbf{force}\, v : C}$$

$$\textsc{Ty-fix}\ \frac{\Gamma[P \Mapsto f : C]; \Sigma \vdash^{\delta}_{\alpha(P)} e : C}{\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} \textbf{fix} f{:}C.e : C} \qquad \textsc{Ty-cvar}\ \frac{\forall\, p \in P.\, \Gamma[p](f) = C}{\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} f : C} \qquad \textsc{Ty-loadpar}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash^{\delta}_{\mathsf{par}(P)} v : \mathsf{Loc}_{P \uplus Q} A \\ \Gamma[P \Mapsto x : A]; \Sigma \vdash^{\delta}_{\mathsf{par}(P)} e : C\end{array}}{\Gamma; \Sigma \vdash^{\delta}_{\mathsf{par}(P)} \textbf{load}_P (v, x.e) : C}$$

$$\textsc{Ty-loadsec}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash^{\delta}_{\mathsf{par}(p)} v : \mathsf{Loc}_{p \uplus P} A \\ \Gamma[p \uplus Q \Mapsto x : A]; \Sigma \vdash^{\delta}_{\mathsf{sec}(p \uplus Q)} e : C\end{array}}{\Gamma; \Sigma \vdash^{\delta}_{\mathsf{sec}(p \uplus Q)} \textbf{load}_p (v, x.e) : C} \qquad \textsc{Ty-comb}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash^{\delta}_{\mathsf{par}(P)} v : \mathsf{Sh}_P A \\ \Gamma[P \uplus Q \Mapsto x : A]; \Sigma \vdash^{\delta}_{\mathsf{sec}(P \uplus Q)} e : C\end{array}}{\Gamma; \Sigma \vdash^{\delta}_{\mathsf{sec}(P \uplus Q)} \textbf{comb}_P (v, x.e) : C} \qquad \textsc{Ty-decpar}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash^{\delta}_{\mathsf{par}(P)} v : \mathsf{Enc}_{(P \uplus Q)} A \\ \Gamma[P \Mapsto x : A]; \Sigma \vdash^{\delta}_{\mathsf{par}(P)} e : C\end{array}}{\Gamma; \Sigma \vdash^{\delta}_{\mathsf{par}(P)} \textbf{decode}\, (v, x.e) : C}$$

$$\textsc{Ty-decsec}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash^{\delta}_{\mathsf{par}(p)} v : \mathsf{Enc}_{(p \uplus Q)} A \\ \Gamma[p \uplus P \Mapsto x : A]; \Sigma \vdash^{\delta}_{\mathsf{sec}(p \uplus P)} e : C\end{array}}{\Gamma; \Sigma \vdash^{\delta}_{\mathsf{sec}(p \uplus P)} \textbf{decode}\, (v, x.e) : C} \qquad \textsc{Ty-asdopar}\ \frac{\begin{array}{c}|\Gamma|_{\mathsf{vals}}; \Sigma \vdash^{\surd}_{\mathsf{par}(Q)} e : C \\ \tilde{p} \rhd_{\mathsf{del}} \mathsf{par}(Q)\end{array}}{\Gamma; \Sigma \vdash^{\surd}_{\tilde{p}} \varepsilon.\, \textbf{as}\, \mathsf{par}(Q)\, \textbf{do}\, e : C} \qquad \textsc{Ty-asdosec}\ \frac{\begin{array}{c}\llbracket |\Gamma|_{\mathsf{vals}} \rrbracket; \Sigma \vdash \Theta \\ \llbracket |\Gamma|_{\mathsf{vals}} \rrbracket; \Sigma \vdash^{\oslash}_{\mathsf{sec}(Q)} e : C \\ \tilde{p} \rhd_{\mathsf{del}} \mathsf{sec}(Q)\end{array}}{\Gamma; \Sigma \vdash^{\surd}_{\tilde{p}} \Theta.\, \textbf{as}\, \mathsf{sec}(Q)\, \textbf{do}\, e : C}$$

Figure 10. $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ typing relations: Value typing and expression typing.

of share locations may only be combined in secure mode, and only when all the parties holding the share are present (rule TY-COMB). Finally, encodings may be decoded in both secure and parallel modes: In secure mode, at least one party must perform the decoding (rule TY-DECSEC); in parallel mode, all parties must perform the decoding (rule TY-DECPAR).

Standard CBPV typing rules. The remainder of the $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ typing rules resemble standard CBPV typing rules, modulo the multi-party structure of the type environments. The rules TY-VVAR and TY-CVAR type a variable (value and computation resp.) when *every* ambient party types the variable. The rules TY-CLOC and TY-SLOC type clear and share locations by consulting the store typing for those parties that hold the location. Other rules that resemble standard CBPV rules consist of the introduction and elimination rules for **unit** (rule TY-UNIT introduces; there is no elimination rule), sums (rules TY-INJ and TY-CASE), products (rules TY-PROD and TY-SPLIT) and arrow types (rules TY-LAM and TY-APP). The rules TY-LETBE and TY-FIX are also standard.

3.2 Operational semantics of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$

Figure 11 gives a big-step, operational semantics for $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ by the judgment $\mu_1; e_1 \Downarrow_{\tilde{p}} \mu_2; \tilde{e}$, read as "under the store μ_1, ambient pseudo-party $\tilde{p}$ reduces e_1 to terminal expression $\tilde{e}$ and store μ_2." We define the syntax of stores and terminal expressions as follows:

$\boxed{\mu_1; e \Downarrow_{\tilde{p}} \mu_2; \tilde{e}}$ *(Under μ_1, psuedo party $\tilde{p}$ evaluates e to $\tilde{e}$ with final store μ_2)*

Ev-lam
$$\frac{}{\mu_1; \lambda x : A.e \Downarrow_{\tilde{p}} \mu_1; \lambda x : A.e}$$

Ev-app
$$\frac{\mu_1; e \Downarrow_{\alpha(P)} \mu_2; \lambda x : A.e_1 \qquad \mu_2; e_1[P \Mapsto x{:}v] \Downarrow_{\alpha(P)} \mu_3; \tilde{e}}{\mu_1; e\ v \Downarrow_{\alpha(P)} \mu_3; \tilde{e}}$$

Ev-letbe
$$\frac{\mu_1; e[P \Mapsto x{:}v] \Downarrow_{\alpha(P)} \mu_2; \tilde{e}}{\mu_1; \textbf{let}\, x\, \textbf{be}\, v\, \textbf{in}\, e \Downarrow_{\alpha(P)} \mu_2; \tilde{e}}$$

Ev-letpar
$$\frac{\mu_1; e_1 \Downarrow_{\tilde{p}} \mu_2; \textbf{ret}_{\mathsf{par}(Q)} v \qquad l = \nu_1(v, Q) \qquad \mu_2[Q \Mapsto l{:}v]; e_2[Q \Mapsto x{:}l] \Downarrow_{\alpha(P)} \mu_3; \tilde{e}}{\mu_1; \textbf{let}\, x \leftarrow e_1\, \textbf{in}\, e_2 \Downarrow_{\alpha(P)} \mu_3; \tilde{e}}$$

Ev-letsec
$$\frac{\mu_1; e_1 \Downarrow_{\tilde{p}} \mu_2; \textbf{ret}_{\mathsf{sec}(Q)} v \qquad s = \nu_{\mathsf{s}}(v, Q) \qquad \mu_2[Q \Mapsto s{:}v]; e_2[Q \Mapsto x{:}s] \Downarrow_{\alpha(P)} \mu_3; \tilde{e}}{\mu_1; \textbf{let}\, x \leftarrow e_1\, \textbf{in}\, e_2 \Downarrow_{\alpha(P)} \mu_3; \tilde{e}}$$

Ev-ret
$$\frac{}{\mu_1; \textbf{ret}_{\tilde{q}} v \Downarrow_{\tilde{p}} \mu_1; \textbf{ret}_{\tilde{q}} v}$$

Ev-case
$$\frac{\mu_1; e_i[P \Mapsto x_1{:}v] \Downarrow_{\alpha(P)} \mu_2; \tilde{e}}{\mu_1; \textbf{case}\, (\textbf{inj}_i\ v, x_1.e_1, x_2.e_2) \Downarrow_{\alpha(P)} \mu_2; \tilde{e}}$$

Ev-split
$$\frac{\mu_1; e[P \Mapsto x_1{:}v_1, x_2{:}v_2] \Downarrow_{\alpha(P)} \mu_1; \tilde{e}}{\mu_1; \textbf{split}\, ((v_1, v_2), x_1.x_2.e) \Downarrow_{\alpha(P)} \mu_2; \tilde{e}}$$

Ev-force
$$\frac{\mu_1; e \Downarrow_{\tilde{p}} \mu_2; \tilde{e}}{\mu_1; \textbf{force}\ [e] \Downarrow_{\tilde{p}} \mu_2; \tilde{e}}$$

Ev-fix
$$\frac{\mu_1; e[P \Mapsto f{:}(\textbf{fix} f{:}C.e)] \Downarrow_{\alpha(P)} \mu_2; \tilde{e}}{\mu_1; \textbf{fix} f{:}C.e \Downarrow_{\alpha(P)} \mu_2; \tilde{e}}$$

Ev-loadpar
$$\frac{\forall\, p \in P.\, \mu_1[p][l] = v \qquad \mu_1; e[P \Mapsto x{:}v] \Downarrow_{\mathsf{par}(P)} \mu_2; \tilde{e}}{\mu_1; \textbf{load}_P(l, x.e) \Downarrow_{\mathsf{par}(P)} \mu_2; \tilde{e}}$$

Ev-loadsec
$$\frac{\mu_1[p][l] = v \qquad \mu_1; e[p \uplus P \Mapsto x{:}v] \Downarrow_{\mathsf{sec}(p \uplus P)} \mu_2; \tilde{e}}{\mu_1; \textbf{load}_p(l, x.e) \Downarrow_{\mathsf{sec}(p \uplus P)} \mu_2; \tilde{e}}$$

Ev-comb
$$\frac{\forall\, p \in P.\, \mu_1[p][s] = v \qquad \mu_1; e[P \uplus Q \Mapsto x{:}v] \Downarrow_{\mathsf{sec}(P \uplus Q)} \mu_2; \tilde{e}}{\mu_1; \textbf{comb}_P(s, x.e) \Downarrow_{\mathsf{sec}(P \uplus Q)} \mu_2; \tilde{e}}$$

Ev-decpar
$$\frac{\mu_1; e[P \Mapsto x{:}v] \Downarrow_{\mathsf{par}(P)} \mu_2; \tilde{e}}{\mu_1; \textbf{decode}\, (\textbf{enc}_{P \uplus Q}\ v, x.e) \Downarrow_{\mathsf{par}(P)} \mu_2; \tilde{e}}$$

Ev-decsec
$$\frac{\mu_1; e[p \uplus P \Mapsto x{:}v] \Downarrow_{\mathsf{sec}(p \uplus P)} \mu_2; \tilde{e}}{\mu_1; \textbf{decode}\, (\textbf{enc}_{p \uplus Q}\ v, x.e) \Downarrow_{\mathsf{sec}(p \uplus P)} \mu_2; \tilde{e}}$$

Ev-asdopar
$$\frac{\mu_1; e \Downarrow_{\mathsf{par}(Q)} \mu_2; \tilde{e}}{\mu_1; \varepsilon.\, \textbf{as}\, \mathsf{par}(Q)\, \textbf{do}\, e \Downarrow_{\tilde{p}} \mu_2; \tilde{e}}$$

Ev-asdosec
$$\frac{\mu_1; e\Theta \Downarrow_{\mathsf{sec}(Q)} \mu_2; \tilde{e}}{\mu_1; \Theta.\, \textbf{as}\, \mathsf{sec}(Q)\, \textbf{do}\, e \Downarrow_{\tilde{p}} \mu_2; \tilde{e}}$$

Figure 11. $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ big-step operational semantics.

Multi store	μ	$::=$	$\varepsilon \mid \mu[p{:}\sigma]$
Local store	σ	$::=$	$\varepsilon \mid \sigma[l{:}v] \mid \sigma[s{:}v]$
Terminal expressions	$\tilde{e}$	$::=$	$\textbf{ret}_{\tilde{p}} v \mid \lambda x : A.e$

As with type environments, stores μ consist of an additional party dimension; each party's state consists of a local store, denoted σ. Evaluations end in a terminal expression, which cannot be reduced further; syntactically, terminal expressions either immediately produce a value, or are functions.

We denote the extension of a multi store μ with the mappings $l{:}v$ and $s{:}v$ for all parties $p \in P$ as $\mu[P \Mapsto l{:}v]$ and $\mu[P \Mapsto s{:}v]$, respectively. We write $[P \Mapsto x{:}v]$ to denote a multi substitution that gives each party in P the binding of variable x to value v.

The rules in Figure 11 are largely unsurprising, and follow directly from the informal discussion of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ semantics from Section 2, and the discussion of typing in Section 3.1. The primary exception concerns allocation: When allocating locations in rules Ev-letpar and Ev-letsec, we assume two injective functions, ν_1 and ν_{s}, which generate clear and share locations (respectively), for a value v and party set Q. The determinism of allocation is technically significant because it simplifies the correspondence between the single-threaded and multi-threaded semantics of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ and $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$, respectively.

Another detail concerns substitution, and more specifically, the interaction of substitution with secure blocks. The semantics of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ and $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ use a mostly standard definition of substitution, which is modified only in the case that the substitution reaches a secure block, where it is delayed and combined with the (possibly empty) substitution attached to outside the secure block. When a secure block is evaluated, this delayed substitution is applied to the secure code (rule Ev-asdosec). This delaying of substitution at the entry to secure blocks enables a closer correspondence between the semantics of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ programs and their corresponding $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ protocols.

Small-step semantics. It is straightforward to derive a small-step version of the big-step judgment given above for $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, with the form $\mu_1; e_1 \xrightarrow{\tilde{p}} \mu_2; e_2$. This judgment can be read as "under multi-store μ_1, pseudo party $\tilde{p}$ takes a single step in e_1, resulting in muti-store μ_2 and expression e_2". The distributed semantics of $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ (Section 4.2) exploits this small-step judgment on $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ programs in an auxiliary role. We relate the big-step semantics and small-step semantics using following theorem. Let $\longrightarrow_*$ denote the reflexive and transitive closure of the small-step $\longrightarrow$ relation.

Theorem 3.1 (Equivalence of big-step and small-step for $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$)**.** *$\mu_1; e_1 \Downarrow_{\tilde{p}} \mu_2; \tilde{e}$ if and only if $\mu_1; e_1 \xrightarrow{\tilde{p}}_* \mu_2; \tilde{e}$.*

3.3 Type soundness

We relate the typing and operational semantics for $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ by the following type soundess result.[5] We define consistency relation $\vDash \Sigma$ (resp. $\vDash \mu$), which says that a location is mapped to same type (resp. value) in all the local Δ (resp. σ) that it exists in. We also define $\Sigma \vDash \mu$, which is an extension of the usual store typing consistency relation to handle multiple parties. We do not give the complete relations for lack of space.

Theorem 3.2 (Type soundness of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$)**.** *Suppose that:*

[5] In this result, we consider only terminating programs. For a more general result, one can show that the small-step semantics of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ enjoys the usual type soundness properties of progress and preservation.

$\boxed{p \vdash C \rightsquigarrow e}$ *("Redactor" generation for party p and type C)*

$$\textsc{GenRed-tyF}\quad \frac{}{p \vdash \mathsf{F}_{\alpha(Q)} A \rightsquigarrow \mathbf{ret}_{\mathsf{par}(p)}\,()} \qquad \textsc{GenRed-tylam}\quad \frac{p \vdash C \rightsquigarrow e}{p \vdash A \rightarrow C \rightsquigarrow \lambda x : A.e}$$

$\boxed{p \vdash v \rightsquigarrow v'}$ *(v translates to v' for p.)*

$\boxed{p \vdash e \rightsquigarrow e'}$ *(e translates to e' for p.)*

$$\textsc{Tr-let1}\quad \frac{p \notin P \qquad y\ \texttt{fresh} \qquad p \vdash e_1 \rightsquigarrow e_1' \qquad p \vdash e_2 \rightsquigarrow e_2'}{p \vdash \mathbf{let}\, x \leftarrow (e_1 : \mathsf{F}_{\alpha(P)} A)\, \mathbf{in}\, e_2 \rightsquigarrow \mathbf{let}\, y \leftarrow e_1'\, \mathbf{in}\, e_2'}$$

$$\textsc{Tr-let2}\quad \frac{p \in P \qquad p \vdash e_1 \rightsquigarrow e_1' \qquad p \vdash e_2 \rightsquigarrow e_2'}{p \vdash \mathbf{let}\, x \leftarrow (e_1 : \mathsf{F}_{\alpha(P)} A)\, \mathbf{in}\, e_2 \rightsquigarrow \mathbf{let}\, x \leftarrow e_1'\, \mathbf{in}\, e_2'}$$

$$\textsc{Tr-ret1}\quad \frac{p \in P \qquad p \vdash v \rightsquigarrow v'}{p \vdash \mathbf{ret}_{\alpha(P)} v \rightsquigarrow \mathbf{ret}_{\tilde{p}} v'} \qquad \textsc{Tr-ret2}\quad \frac{p \notin P}{p \vdash \mathbf{ret}_{\alpha(P)} v \rightsquigarrow \mathbf{ret}_{\mathsf{par}(p)}\,()}$$

$$\textsc{Tr-asdopar}\quad \frac{p \in P \qquad p \vdash e \rightsquigarrow e'}{p \vdash \varepsilon.\,\mathbf{as}\,\mathsf{par}(p \uplus P)\,\mathbf{do}\, e \rightsquigarrow \varepsilon.\,\mathbf{as}\,\mathsf{par}(p \uplus P)\,\mathbf{do}\, e'}$$

$$\textsc{Tr-asdosec}\quad \frac{}{p \vdash \Theta.\,\mathbf{as}\,\mathsf{sec}(p \uplus P)\,\mathbf{do}\, e \rightsquigarrow [p{:}\Theta[p]].\,\mathbf{as}\,\mathsf{sec}(p \uplus P)\,\mathbf{do}\, e}$$

$$\textsc{Tr-asdoredact}\quad \frac{p \notin P \qquad p \vdash C \rightsquigarrow e'}{p \vdash \Theta.\,\mathbf{as}\,\alpha(P)\,\mathbf{do}\, e : C \rightsquigarrow e'}$$

Figure 12. Translation rules (congruence cases omitted).

- $\varepsilon; \Sigma_1 \vdash^{\delta}_{\tilde{p}} e : C$
- $\vDash \Sigma_1$, $\vDash \mu_1$, *and* $\Sigma_1 \vDash \mu_1$

then we have:

- $\mu_1; e \Downarrow_{\tilde{p}} \mu_2; \tilde{e}$
- $\vDash \mu_2$
- $\exists \Sigma_2 \supseteq \Sigma_1$ *s.t.* $\vDash \Sigma_2$, $\Sigma_2 \vDash \mu_2$, *and* $\varepsilon; \Sigma_2 \vdash^{\delta}_{\tilde{p}} \tilde{e} : C$

Proof. (Sketch): The full proof proceeds by straightforward induction on the first premise (the typing relation). We use adapted forms of standard lemmas (which we omit here for space reasons); those lemmas include straight-forward adaptations of inversion, canonical forms and substitution. □

4. Distributed protocols in $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$

In $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, programmers reason about mixed-mode multi-party secure computations as single-threaded programs. In this section, we define $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$, a distributed semantics for M^{3}PCs (Section 4.2), and give a systematic method for translating $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ programs into $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ protocols (Section 4.1). We prove that well-typed $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ programs can *always* be realized as a distributed $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ protocol with an equivalent semantics (Section 4.3).

4.1 Translation from specifications to protocols

A distributed protocol π in $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$, is a mapping from parties p to expressions e, defined for each party involved in the M^{3}PC:

$$\text{Protocol} \quad \pi \quad ::= \quad \pi_1 * \pi_2 \mid [p{:}e]$$

The protocol code translated for each party uses the same formal syntax as that of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$. In Figure 12, we give rules to translate a $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ program to a $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ protocol, for a given party involved in the corresponding M^{3}PC. The key idea underlying the transition is that of *code redaction*: If a party p does not participate in some block of code, then the translation *redacts* this code from the protocol assigned to party p.

Redacting with redactors. When redacting code, the translation encounters a small technical issue, which we solve by defining the notion of a *redactor* expression. A redactor is an expression defined by a computational type C for some party p using the judgment $p \vdash C \rightsquigarrow e$ (Figure 12). The key feature of a redactor is that it has the same *type structure* as the computation being redacted. In particular, if C is a $\mathsf{F}_{\tilde{q}} A$ type, the translation generates the expression $\mathbf{ret}_{\mathsf{par}(p)}\,()$ that has type $\mathsf{F}_{\mathsf{par}(p)}\mathbf{unit}$ (rule GenRed-tyF). If C is a function type, it generates a λ-term, that when applied, eats away the same arguments that the redacted code would have consumed (rule GenRed-tylam).

We illustrate the translation's use of redactors using a small example; consider the following program fragment:

$$\varepsilon.\,\mathbf{as}\,\mathsf{par}(\mathtt{A},\mathtt{B})\,\mathbf{do}\,\mathbf{let}\, y \leftarrow (\varepsilon.\,\mathbf{as}\,\mathsf{par}(\mathtt{B})\,\mathbf{do}\,\lambda x : A.e)\ v\ \mathbf{in}\,\mathbf{ret}_{\mathsf{par}(\mathtt{A},\mathtt{B})}\,()$$

The function $\lambda x : A.e$ appears in $\mathsf{par}(\mathtt{B})$, and hence needs to be redacted from Alice's code. However, the application of the function to v happens in $\mathsf{par}(\mathtt{A},\mathtt{B})$ context, and so needs to be preserved in Alice's code as well. The translation generates Alice's protocol as follows:

$$\varepsilon.\,\mathbf{as}\,\mathsf{par}(\mathtt{A},\mathtt{B})\,\mathbf{do}\,\mathbf{let}\, y' \leftarrow (\lambda x' : A.\mathbf{ret}_{\mathsf{par}(\mathtt{A})}\,())\ v\ \mathbf{in}\,\mathbf{ret}_{\mathsf{par}(\mathtt{A},\mathtt{B})}\,()$$

The redactor is a λ-abstraction that eats away the applied argument, ensuring that the structure of types is maintained in Alice's code.

Translation judgments. Figure 12 gives judgments $p \vdash v \rightsquigarrow v'$ and $p \vdash e \rightsquigarrow e'$, which generate translations for party p of values v and expressions e, respectively. For lack of space, we only show rules that are not congruences; we omit all rules for value translation, since they are all congruence rules. The translation is type-directed, and we assume typings for subexpressions, which can be constructed from the typing judgment for $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$.

At a high level, the structure of the translated code for a given party p is preserved, modulo those places where some code is redacted. The consequences of code redaction are most evident in rule Tr-asdoredact: For parties not participating in a delegation, the rule Tr-asdoredact replaces the delegated code block with a redactor that (eventually) produces the unit value. Since the redacted code no longer returns a useful value (but rather the unit value), rule Tr-let1 translates let-bindings of redacted code with a related let-binding for a (fresh) dummy variable y; due to the type system, this variable replacement is guaranteed to be unproblematic. The rule Tr-let2 handles the other (more straightforward) case of translating let bindings, where the let-bound variable is preserved and the sub-computations are translated accordingly.

For parties participating in a delegation, the translation proceeds by considering the mode of the delegation. In the case of secure blocks, the type system guarantees that the code is translation independent, and hence requires no further specialization. Hence rule Tr-asdosec does not recur, but only projects the suspended substitution to specialize it for the party under consideration. In the case of parallel blocks, the translation proceeds in a recursive, straightforward way.

Finally, return expressions are translated straightforwardly: If the party under consideration is participating in the return, the return value is translated recursively (rule TR-RET1); otherwise, it is redacted to the unit value (rule TR-RET2).

Translation properties. The following theorem establishes the existence of translation for a well-typed $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ program.

Theorem 4.1 (Existence of translation)**.**
If $\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} e : C$*, then* $\forall\, p \in P \,.\, p \vdash e \rightsquigarrow e_p$.

Proof. Induction on the typing derivation. □

Moreover, the protocol π for program e is simply $[p{:}e_p]$, for each party $p \in P$. As stated earlier, we show that values and computations of *translation independent types* are not altered after their translation:

Lemma 4.2 (Translation-independent values)**.**
If $\Gamma; \Sigma \vdash^{\delta}_{\alpha(P)} v : [\![A]\!]$*, then* $\forall\, p \in P \,.\, p \vdash v \rightsquigarrow v$

Lemma 4.3 (Translation-independent computations)**.**
If $\Gamma; \Sigma \vdash^{\oslash}_{\alpha(P)} e : C$*, then* $\forall\, p \in P \,.\, p \vdash e \rightsquigarrow e$

4.2 Transition semantics for $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ protocols

We give a small-step operational semantics for protocols π in Figure 13. The judgments are of the form $\mu_1; \pi_1 \longrightarrow \mu_2; \pi_2$, which are read "under multi-store μ_1, protocol π_1 steps to π_2 with multi-store μ_2".

The semantics is multi-threaded and non-deterministic: At each step in the protocol, a non-deterministically chosen party p takes the next step in its local computation. However, in secure blocks, participating parties synchronize at the beginning of the block and step together until the block terminates; afterwards, the participants resume independent steps. We note that two or more disjoint sets of parties may always interleave steps, even if the members of each set are engaged independent secure blocks. The transition rules for stepping a party p are mostly straightforward adaptations of the $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ semantics; two exceptions to this are the introduction of (share and clear) locations, and the transitions for secure blocks.

The rules TGT-LET2-PAR and TGT-LET2-SEC respectively create clear and share locations for p using injective functions ν_1 and ν_{s}, as in the semantics of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$. However, unlike in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, only the store and code for p is updated; the stores and code for other parties remains unchanged, reflecting the multi-threaded structure of the protocol semantics.

The rule TGT-ASDO-SEC1 steps inside a secure block. For a party $p \in P$, the secure block has the form $[p{:}\theta_p]$. **as** $\mathsf{sec}(P)$ **do** e, due to local substitutions, which are gathered in θ_p. The rule combines all the substitutions, and steps e using the $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ small-step semantics under the context of $\mathsf{sec}(P)$. The rule TGTSTEP-SECASELIM eliminates the secure block and returns the terminal expression to each party involved in the secure block.

4.3 Technical results for $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$

We show that the $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ semantics is confluent, and that our translation from $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ programs into $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ protocols is sound.

Confluence. Despite being non-deterministic, the semantics of $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ enjoys a confluence property, stated as follows:

Theorem 4.4 (Confluence)**.** *If* $\mu_1; \pi_1 \longrightarrow \mu_2; \pi_2$ *and* $\mu_1; \pi_1 \longrightarrow \mu_2'; \pi_2'$*, then there exists* μ_3 *and* π_3 *such that* $\mu_2; \pi_2 \longrightarrow \mu_3; \pi_3$*, and* $\mu_2'; \pi_2' \longrightarrow \mu_3; \pi_3$.

Proof. Simultaneous induction on the small-step derivations. □

The confluence result can be extended to $\longrightarrow_*$, the reflexive and transitive closure of $\longrightarrow$, using *diamond reasoning*, yielding the following corollary. Let us define $\tilde{\pi}$ as a terminal protocol, in which every party's computation has been reduced to a $\tilde{e}$.

Corollary. *If* $\mu_1; \pi_1 \longrightarrow_* \mu_2; \tilde{\pi}_2$ *and* $\mu_1; \pi_1 \longrightarrow_* \mu_3; \tilde{\pi}_3$*, then* $\mu_2 = \mu_3$ *and* $\tilde{\pi}_2 = \tilde{\pi}_3$.

The above corollary states that all possible executions of a protocol π, starting from a store μ_1, result in same terminal protocol $\tilde{\pi}$ and final store μ_2. Thus, in terms of final store and terminal protocol, the non-determinism in the $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ semantics is inconsequential.

Translations are sound. We finally relate the semantics of a source program written in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ with the semantics of its translation in $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ by following theorem.

Theorem 4.5 (Translations are sound)**.**
If $\mu_1; e_1 \xrightarrow{\alpha(P)} \mu_2; e_2$ *and* $\forall\, p \in P \,.\, p \vdash e_1 \rightsquigarrow e_{1p}$*, then*

- $\mu_1; \biguplus_{p \in P} [p \mapsto e_{1p}] \longrightarrow_* \mu_2'; \pi_2$
- $\forall\, p \in P \,.\, \mu_2'[p] \supseteq \mu_2[p]$
- $\forall\, p \in P \,.\, p \vdash e_2 \rightsquigarrow \pi_2[p]$

Proof. Induction on the $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ small-step derivation. □

The intuition for $\mu_2'[p] \supseteq \mu_2[p]$ is as follows. The party p's final store in $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ can have one extra location than $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$, corresponding to the (unit-valued) dummy variables introduced by the translation. Also, recall that $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ type system ensures that only values of translation independent types appear in the stores (rules TY-LETPAR and TY-LETSEC in Figure 10). By Lemma 4.2, the values of translation independent types are preserved across translation, and hence, the $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ semantics generates the same value v in rules TGTSTEP-LETPAR and TGTSTEP-LETSEC (Figure 13) as generated by $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ semantics. Since the store location generation functions ν_1 and ν_{s} are the same across $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ and $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$, the semantics end up generating same store locations and same values.

5. Implementing secure code

When defining both $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ and $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$, we abstract away the concrete implementation of secure code blocks from our calculus, and have chosen to focus on *policy* (i.e., specification) rather than *mechanism*. In this section, we discuss the implementation of secure blocks, and the interaction of our calculus with those mechanisms.

A secure block can be realized by techniques such as a trusted third-party (TTP), a garbled circuit (GC) and via homomorphic encryption (HE). In all cases, the participating parties have an incentive to minimize the size of computations done in secure blocks, and hence write M^{3}PCs (as opposed to monolithic secure computations). In case of a TTP, there are several factors that encourage this minimization, such as network cost (in terms of time and bandwidth) as well as other monetary costs. For cryptographic techniques such as GCs and HE, the incentive is better performance, both in terms of computation time as well as memory requirements.

Below, we focus on the realization of our calculus's cryptographic abstractions (viz., encodings and shares) in the contexts of TTPs and GCs only (similar arguments also apply to homomorphic encryption).

Trusted third parties (TTPs). A (real-life) trusted third party, when it exists, can readily implement the secure blocks found in our calculus, including their support of encodings and shares.

Encodings allow secure blocks to return private outputs (although they can used in a more general manner). In case of TTP, private output to party p can be realized using a private channel between p

$$
\begin{array}{llcl}
\textsc{Tgt-App1} & \mu_1; [p{:}e\ v] * \pi_1 & \longrightarrow & \mu_2; [p{:}e'\ v] * \pi_2 \ \text{ where } \mu_1; [p{:}e] * \pi_1 \longrightarrow \mu_2; [p{:}e'] * \pi_2 \\
\textsc{Tgt-App2} & \mu_1; [p{:}(\lambda x : A.e)\ v] * \pi_1 & \longrightarrow & \mu_1; [p{:}e[p \Mapsto x{:}v]] * \pi_1 \\
\textsc{Tgt-Letbe} & \mu_1; [p{:}\textbf{let}\, x\, \textbf{be}\, v\, \textbf{in}\, e] * \pi_1 & \longrightarrow & \mu_1; [p{:}e[p \Mapsto x{:}v]] * \pi_1 \\
\textsc{Tgt-Let1} & \mu_1; [p{:}\textbf{let}\, x{\leftarrow}e_1\, \textbf{in}\, e_2] * \pi_1 & \longrightarrow & \mu_2; [p{:}\textbf{let}\, x{\leftarrow}e_1'\, \textbf{in}\, e_2] * \pi_2 \ \text{ where } \mu_1; [p{:}e_1] * \pi_1 \longrightarrow \mu_2; [p{:}e_1'] * \pi_2 \\
\textsc{Tgt-Let2-Par} & \mu_1[p{:}\sigma]; [p{:}\textbf{let}\, x{\leftarrow}\textbf{ret}_{\mathsf{par}(P)}\, v\, \textbf{in}\, e_2] * \pi_1 & \longrightarrow & \mu_1[p{:}\sigma[l{:}v]]; [p{:}e_2[p \Mapsto x{:}l]] * \pi_1 \ \text{ where } l = \nu_1(v, P) \\
\textsc{Tgt-Let2-Sec} & \mu_1[p{:}\sigma]; [p{:}\textbf{let}\, x{\leftarrow}\textbf{ret}_{\mathsf{sec}(P)}\, v\, \textbf{in}\, e_2] * \pi_1 & \longrightarrow & \mu_1[p{:}\sigma[s{:}v]]; [p{:}e_2[p \Mapsto x{:}s]] * \pi_1 \ \text{ where } s = \nu_{\mathsf{s}}(v, P) \\
\textsc{Tgt-Case} & \mu_1; [p{:}\textbf{case}\, (\textbf{inj}_i\ v, x_1.e_1, x_2.e_2)] * \pi_1 & \longrightarrow & \mu_1; [p{:}e_i[p \Mapsto x_i{:}v]] * \pi_1 \ \text{ where } i \in \{1, 2\} \\
\textsc{Tgt-Split} & \mu_1; [p{:}\textbf{split}\, ((v_1, v_2), x_1.x_2.e)] * \pi_1 & \longrightarrow & \mu_1; [p{:}e[p \Mapsto x_1{:}v_1, x_2{:}v_2]] * \pi_1 \\
\textsc{Tgt-Force} & \mu_1; [p{:}\textbf{force}\ [e]] * \pi_1 & \longrightarrow & \mu_1; [p{:}e] * \pi_1 \\
\textsc{Tgt-Fix} & \mu_1; [p{:}\textbf{fix} f{:}C.e] * \pi_1 & \longrightarrow & \mu_1; [p{:}e[p \Mapsto f{:}\textbf{fix} f{:}C.e]] * \pi_1 \\
\textsc{Tgt-Load-Par} & \mu_1; [p{:}\textbf{load}_P(l, x.e)] * \pi_1 & \longrightarrow & \mu_1; [p{:}e[p \Mapsto x{:}\mu_1[p][l]]] * \pi_1 \\
\textsc{Tgt-decode-Par} & \mu_1; [p{:}\textbf{decode}\, (\textbf{enc}_P\ v, x.e)] * \pi_1 & \longrightarrow & \mu_1; [p{:}e[p \Mapsto x{:}v]] * \pi_1 \\
\textsc{Tgt-AsDo-Par} & \mu_1; [p{:}\textbf{decode}\, (\textbf{enc}_P\ v, x.e)] * \pi_1 & \longrightarrow & \mu_1; [p{:}e'] * \pi_1 \\
\textsc{Tgt-AsDo-Sec1} & \mu_1;\ \biguplus_{p \in P} [p \mapsto [p{:}\theta_p].\, \textbf{as}\, \mathsf{sec}(P)\, \textbf{do}\, e] * \pi_1 & \longrightarrow & \mu_2;\ \biguplus_{p \in P} [p \mapsto [p{:}\theta_p].\, \textbf{as}\, \mathsf{sec}(P)\, \textbf{do}\, e'] * \pi_1 \\
 & & & \qquad \text{where } \mu_1; e(\biguplus_{p \in P} [p \mapsto \theta_p]) \xrightarrow{\mathsf{sec}(P)} \mu_2; e' \\
\textsc{Tgt-AsDo-Sec2} & \mu_1;\ \biguplus_{p \in P} [p \mapsto [p{:}\theta_p].\, \textbf{as}\, \mathsf{sec}(P)\, \textbf{do}\, \tilde{e}] * \pi_1 & \longrightarrow & \mu_1;\ \biguplus_{p \in P} [p \mapsto \tilde{e}] * \pi_1
\end{array}
$$

Figure 13. $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ small-step semantics.

and the TTP. More general forms of encodings can be realized using encryption. A value $\textbf{enc}_Q\ v$ can be implemented as an encryption of v under some key such that only parties in Q can decrypt it.

Secret shares can be implemented using schemes such as Shamir's Secret Sharing [25]. The TTP can distribute shares among the M^3PC parties, that the parties can later combine and recover the value. Assuming that the TTP's computational model allows language constructs such as loops, conditionals, and higher-order thunks, we need not restrict the code appearing in secure blocks.

Garbled circuits (GCs). Garbled circuits implement secure blocks as a boolean and/or arithmetic circuits. For two-party GCs, one party *garbles* the circuit and other party *obliviously* evaluates it. At the conclusion of evaluation, the circuit generator sends garbling keys to the evaluator, and the evaluator sends garbled values to the generator. After this exchange (but not beforehand), both parties can recover the output. Garbled circuits can realize private outputs by not communicating shares to everyone, but instead only to those parties who are allowed to know the output. General-purpose encodings can be realized using encryption, by implementing the encryption function as additional (secure) computation within the GC.

GCs are more limited than TTPs, in that they do not admit loops and general conditionals (conditional assignments are allowed); moreover, implementing higher-order values (thunks) is non-obvious. To respect such limitations, a future compiler for our calculus can check that unsupported constructs (i.e., loops, conditionals, and thunks) are either not used inside secure blocks, or can be translated away (e.g., the compiler can attempt to unroll loops, flatten conditionals and apply thunks before such circuits are generated). However, recent research has made progress in overcoming some of these existing limitations. For example, Holzer et al. [8] present higher order computations in GCs by evaluating a GC using an interpreter, which itself is a GC.

6. Related work

Several research groups have looked at compiling high-level, straight-line programs involving multiple parties' data to secure protocols, e.g., garbled circuits suitable for execution by multiple parties. Fairplay [17] compiles a garbled circuit from a high-level imperative program. FairplayMP [3] extends this line of work to $n > 2$ parties. More recently, Holzer et al [8] developed a compiler from ANSI C to two-party secure computations. Alternatively, at the lowest level there are libraries for building garbled circuits directly, e.g., those developed by Malka [16], Huang et al [11], and Mood et al [19]. Both sets of work can act as subroutines in our system: the compilation strategies of compilers like Fairplay could be used to produce garbled circuits from our secure blocks, while we could use these libraries to optimize these circuits' ultimate implementation.

Another line of research in SMCs deals with a client-server setting, where client wants to run a function over his private input using untrusted servers (e.g. in a cloud). To protect confidentiality of his data, the client distributes secret shares of his input among the servers. The servers run *same* function, but use their own shares. Finally, they send the output shares to the client, who then recovers the clear output value. Launchbury et. al. [14] present a table-lookup based optimization for such SMC protocols, that aims at minimizing the cost incurred by expensive operations such as multiplication and network communication between servers. Mitchell et. al. [18] 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).

For our calculus, the above client-server setting can be expressed as a monolithic secure block to be run by the servers, each of which holds secret shares of client's input. As we have shown in the paper, we can express more general mixed-mode SMCs. Further, we model secure blocks implementation, by default, as TTP, and thus do not constrain programs by the execution platform. Consequently, our type system does not deal with *obliviousness* of the implementation, rather it reasons about essence of M^3PCs i.e. delegations between modes and local states of the parties. In fact, a secret shares based implementation of secure blocks in our calculus can benefit from the optimizations presented by Launchbury et al. [14].

Other groups have looked at intermediate languages for specifying mixed-mode computations. In general, these languages tend to be both less expressive, and yet lower-level, than our approach. For example, all prior languages are first-order, whereas we are higher order. Moreover, few prior systems have been formally specified and proven correct; more on this as we go.

The TASTY compiler produces secure computations that may combine homomorphic encryption and garbled circuits [7]. Its input language, TASTYL, requires explicit specification of communication between parties, as well as the means of secure computation, whereas things like communication are handled automatically (during compilation) in our approach. Kerschbaum et al. [13] explore automatic selection of mixed protocols consisting of garbled circuits and homomorphic encryption.

SMCL [21] is a language for secure computations involving a replicated client and a shared "server" which represents secure multiparty computations. Our approach is less rigid in its specification of roles: we have secure and parallel computations involving arbitrary numbers of principals, rather than all of them, or just one, as in SMCL. On the other hand, the number of principals in our approach is static; generalizing our approach to support an arbitrary number of principals (e.g., using existential quantifiers) is future work. SMCL provides a type system that aims to enforce some information flow properties modulo declassification. SMCL's successor, VIFF [26], reflects the SMCL computational model as a library/DSL in Python, but lacks type-based guarantees.

L1 [24] is an intermediate language for mixed-mode SMC, like our language. An L1 program is, by default, replicated on each party's machine, but certain blocks may be designated to run on the machines of individual clients. This is like our parallel mode, with delegations to single parties within it. Secure computations are initiated by calls to libraries from within the outermost parallel context. In general, we afford more flexibility in how computations can be decomposed into parallel and secure forms. We also hide away some details, e.g., L1 requires explicit communications (e.g., via calls to `send` and `recv` primitives) between hosts, whereas we implement this indirectly through variable binding. Interestingly, the L1 authors point out that the intermediate language provides no security or correctness guarantees. We aim to fill this gap by formalizing the language, type system, and compiler, and proving the latter is correct.

Finally, our approach bears some resemblance to Jif/Split [28], a compiler for programs written in a dialect of Java with Information Flow [20]. In Jif/Split, the programmer labels data with *confidentiality labels*, which indicate those parties who are allowed to see the data, and trust maps, which indicate which hosts are trusted by which parties. Explicit *declassifications* indicate when data may flow from one party to another. With this information, the compiler automatically partitions the program into bits that run on appropriate hosts. This process is similar to our compilation strategy, which splits up the program following the parallel and secure mode annotations. On the other hand, our notion of lock-step parallel execution does not exist in Jif/Split, nor does the notion of a joint, secure computation that synchronizes multiple parties. The Jif/Split splitting algorithm was also never formalized or proven correct, to our knowledge.

7. Conclusions

We presented a core calculus for mixed-mode, multi-party computations (M^{3}PCs). Programs in $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ correspond conceptually to single-threaded specifications, and protocols in $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ correspond to distributed protocols that realize these specifications. We proved that well-typed $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ programs do not get stuck, and always have a corresponding realization as $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$ protocols, where their semantics is preserved. We showed that our design captures patterns of M^{3}PCs found in past literature, as well as newer, higher-order patterns.

We view our calculus as a foundation for future work providing a trustworthy compilation framework for M^{3}PCs. In pursuit of this goal, we plan to build an associated theory of the security properties of $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ programs, and specifically, their *knowledge profile* [23]. In particular, we believe that our calculus is suitable for future work that reasons about security-preserving compiler optimizations, where the aim is to *automatically* decompose monolithic SMCs into mixed-mode counterparts.

References

[1] Aggarwal, G., N. Mishra, and B. Pinkas (2004). Secure computation of the k th-ranked element. In *EUROCRYPT*. Springer.

[2] Beaver, D., S. Micali, and P. Rogaway (1990). The round complexity of secure protocols. In *STOC*.

[3] Ben-David, A., N. Nisan, and B. Pinkas (2008). FairplayMP: a system for secure multi-party computation. In *CCS*.

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

[5] Freedman, M. J., K. Nissim, and B. Pinkas (2004). EfñĄcient private matching and set intersection. In *EUROCRYPT*.

[6] Goldreich, O., S. Micali, and A. Wigderson (1987). How to play ANY mental game. In *STOC*.

[7] Henecka, W., S. K ögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg (2010). Tasty: tool for automating secure two-party computations. In *CCS*.

[8] Holzer, A., M. Franz, S. Katzenbeisser, and H. Veith (2012). Secure two-party computations in ansi c. In *CCS*.

[9] Huang, Y., P. Chapman, and D. Evans (2011). Privacy-preserving applications on smartphones. In *HotSec*.

[10] Huang, Y., D. Evans, and J. Katz (2012). Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*.

[11] Huang, Y., D. Evans, J. Katz, and L. Malka (2011). Faster secure two-party computation using garbled circuits. In *In USENIX Security Symposium*.

[12] Kerschbaum, F. (2011). Automatically optimizing secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security*.

[13] Kerschbaum, F., T. Schneider, and A. Schröpfer (2013). Automatic protocol selection in secure two-party computations. In *NDSS*.

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

[15] Levy, P. (1999). Call-by-push-value: A subsuming paradigm. *Typed Lambda Calculi and Applications*, 644–644.

[16] Malka, L. (2011). Vmcrypt: modular software architecture for scalable secure computation. In *CCS*.

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

[18] Mitchell, J., R. Sharma, D. Stefan, and J. Zimmerman (2012). Information-flow control for programming on encrypted data. In *CSF*.

[19] Mood, B., L. Letaw, and K. Butler (2012). Memory-efficient garbled circuit generation for mobile devices. In *Financial Cryptography*.

[20] Myers, A. C. (1999). Jflow: practical mostly-static information flow control. In *POPL*.

[21] Nielsen, J. D. and M. I. Schwartzbach (2007). A domain-specific programming language for secure multiparty computation. In *PLAS*.

[22] Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*.

[23] Rastogi, A., P. Mardziel, M. Hammer, and M. Hicks (2013, March). Knowledge inference for optimizing secure multi-party computation.

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

[25] Shamir, A. (1979, November). How to share a secret. *Communications of the ACM 22*(11), 612–613.

[26] viff. VIFF, the virtual ideal functionality framework. `http://viff.dk/`.

[27] Yao, A. C.-C. (1986). How to generate and exchange secrets. In *FOCS*.

[28] Zdancewic, S., L. Zheng, N. Nystrom, and A. C. Myers (2002, August). Secure program partitioning. *ACM Trans. Comput. Syst. 20*(3), 283–328.

```
party p      -- abstract psuedo party
vtype st     -- abstract state type

ctype cmd    = st -> F p st    -- commands
vtype cmdt   = U p cmd         -- command thunks
vtype bool   = 1 + 1           -- booleans
ctype pred   = st -> F p bool  -- predicates
vtype predt  = U p pred        -- predicate thunks

ctype conditional  = predt -> cmdt -> cmdt -> cmd
vtype conditionalt = U p conditional

ctype whileloop  = predt -> cmdt -> cmd
vtype whileloopt = U p whileloop

ctype forbody    = int -> cmd
vtype forbodyt   = U p forbody
ctype forloop    = int -> int -> forbodyt -> cmd
vtype forloopt   = U p forloop

if : conditionalt
if = [ \pred:predt. \then:cmdt. \else:cmdt. \st:st.
  let bl = (force pred) st in
  load ( bl, b.
  case ( b, _. (force then) st
            _. (force else) st )) ]

while : whileloop
while = [ \guard:predt. \body:cmdt.
  fix loop:cmd. \st:st.
  let b1 = (force guard) in
  load ( b1, b.
  case ( b,
    _. let st'l = (force body) st in
       load (st'l, st'. loop st' )
    _. ret st )) ]

for : forloopt
for = [ \beg:int. \end:int. \body:forbodyt.
  fix loop : int -> cmd. \idx:int. \st:st.
  let bl = (force int_eq) idx end in
  load ( bl, b.
  case ( b,
    _. let st'l  = (force body) st idx in
       let idx'l = (force int_incr) idx in
       load ( st'l, st'.
       load ( idx'l, idx'.
       loop idx' st' ))
   _. ret st )) ]
```

Figure 14. We encode a WHILE-style language into our calculus: We give higher order state-passing combinators for building programs with conditionals, while-loops and for-loops.

```
ctype forbody'   = int -> cmdt -> cmd
vtype forbody't  = U p forbody'
ctype forloop'   = int -> int -> forbody't -> cmd
vtype forloop't  = U p forloop'

for' : forloop't
for' = [ \beg:int. \end:int. \body:forbody't.
  fix loop : int -> cmd. \idx:int. \st:st.
  let bl = (force int_eq) idx end in
  load ( bl, b.
  case ( b,
    _. let idx'l = (force int_incr) idx in
       load ( idx'l, idx'.
       (force body) idx [loop idx'] st )
    _. ret st )) ]
```

Figure 15. For loop in (delimited) continuation-passing style, which simulates break and continue-style control flow features commonly found in WHILE-style languages.

A. Library code for $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$

B. Full calculus

B.1 Syntactic classes

Real parties	$p,\ q,\ r$	$::=$	$\mathbf{Alice} \mid \mathbf{Bob} \mid \mathbf{Chuck}$
Real party sets	$P,\ Q,\ R$	$::=$	$\emptyset \mid p \mid P \uplus Q$
Psuedo-party mode	$\alpha,\ \beta,\ \gamma$	$::=$	$\mathsf{par} \mid \mathsf{sec}$
Pseudo parties	$\tilde{p},\ \tilde{q},\ \tilde{r}$	$::=$	$\alpha(P)$

Local substitutions	θ	$::=$	$\varepsilon \mid \theta, x : v \mid \theta, f : e$
Multi substitutions	Θ	$::=$	$\varepsilon \mid \Theta[p{:}\theta]$

Values	v	$::=$	$x \mid () \mid \mathbf{inj}_i\ v \mid (v_1, v_2) \mid [e]$
			$\mid l \mid s \mid \mathbf{enc}_P\ v$
Expressions	e	$::=$	$\lambda x : A.e \mid e\ v \mid \mathbf{let}\ x\ \mathbf{be}\ v\ \mathbf{in}\ e$
			$\mid \mathbf{let}\ x{\leftarrow}e_1\ \mathbf{in}\ e_2 \mid \mathbf{ret}_{\tilde{p}} v$
			$\mid \mathbf{case}\ (v, x_2.e_2, x_3.e_3)$
			$\mid \mathbf{split}\ (v, x_1.x_2.e) \mid \mathbf{force}\ v$
			$\mid \mathbf{fix} f{:}C.e \mid f \mid \mathbf{load}_P(v, x.e)$
			$\mid \mathbf{comb}_P(v, x.e)$
			$\mid \mathbf{decode}\ (v, x.e)$
			$\mid \Theta.\ \mathbf{as}\ \tilde{p}\ \mathbf{do}\ e$
Terminal expressions	$\tilde{e}$	$::=$	$\mathbf{ret}_{\tilde{p}} v \mid \lambda x : A.e$

Delegation switch	δ	$::=$	$\surd \mid \oslash$

Value types	$A,\ B$	$::=$	$\mathbf{unit} \mid A + B \mid A \times B \mid \mathsf{U}^{\delta}_{\tilde{p}} C$
			$\mid \mathsf{Loc}_P A \mid \mathsf{Sh}_P A \mid \mathsf{Enc}_P A$
Comp. types	$C,\ D$	$::=$	$A \to C \mid \mathsf{F}_{\tilde{p}} A$

Multi variable-typing env.	Γ	$::=$	$\varepsilon \mid \Gamma[p{:}\Upsilon]$
Local variable-typing env.	Υ	$::=$	$\varepsilon \mid \Upsilon, x : A \mid \Upsilon, f : C$
Multi store-typing env.	Σ	$::=$	$\varepsilon \mid \Sigma[p{:}\Delta]$
Local store-typing env.	Δ	$::=$	$\varepsilon \mid \Delta, l : A \mid \Delta, s : A$

Multi store	μ	$::=$	$\varepsilon \mid \mu[p{:}\sigma]$
Local store	σ	$::=$	$\varepsilon \mid \sigma[l{:}v] \mid \sigma[s{:}v]$

Protocol	π	$::=$	$\pi_1 * \pi_2 \mid [p{:}e]$

B.2 Judgments

$\boxed{\Gamma; \Sigma \vdash \Theta}$ *(Under Γ and Σ, multi substitution Θ is well-typed)*

$$\textsc{Ty-empms}\ \frac{}{\Gamma; \Sigma \vdash \varepsilon} \qquad \textsc{Ty-ms}\ \frac{\Gamma; \Sigma \vdash \Theta \qquad \Gamma; \Sigma; p \vdash \theta}{\Gamma; \Sigma \vdash \Theta[p{:}\theta]}$$

$\boxed{\Gamma; \Sigma; p \vdash \theta}$ *(Under Γ and Σ, substitution θ is well-typed)*

$$\textsc{Ty-emps}\ \frac{}{\Gamma; \Sigma; p \vdash \varepsilon} \qquad \textsc{Ty-s1}\ \frac{\begin{array}{c}\Gamma; \Sigma; p \vdash \theta \\ \Gamma; \Sigma \vdash^{\varnothing}_{\mathsf{par}(p)} v : \Gamma[p](x)\end{array}}{\Gamma; \Sigma; p \vdash \theta, x : v} \qquad \textsc{Ty-s2}\ \frac{\Gamma; \Sigma; p \vdash \theta \qquad x \notin \mathsf{dom}(\Gamma[p])}{\Gamma; \Sigma; p \vdash \theta, x : v}$$

Figure 16. Well-typed substitutions.

$\boxed{[\![\Upsilon_1]\!]}$

$$\begin{aligned}[\![\varepsilon]\!] &= \varepsilon \\ [\![\Upsilon, x : A]\!] &= [\![\Upsilon]\!] \\ [\![\Upsilon, x : A]\!] &= [\![\Upsilon]\!], x : [\![A]\!]\end{aligned}$$

$\boxed{[\![\Gamma_1]\!]}$

$$\begin{aligned}[\![\varepsilon]\!] &= \varepsilon \\ [\![\Gamma[p{:}\Upsilon]]\!] &= [\![\Gamma]\!][p{:}[\![\Upsilon]\!]]\end{aligned}$$

Figure 17. Restricting typing context (for typing secure blocks).

$\boxed{\theta_1 - x}$

$$\begin{aligned}
\varepsilon - x &= \varepsilon \\
(\theta, x : v) - x &= \theta \\
(\theta, y : v) - x &= (\theta - x), y : v
\end{aligned}$$

$\boxed{\Theta_1 - x}$

$$\begin{aligned}
\varepsilon - x &= \varepsilon \\
(\Theta[p{:}\theta]) - x &= (\Theta - x)[p{:}(\theta - x)]
\end{aligned}$$

$\boxed{v_1 \Theta}$ Applying Θ to v_1

$$\begin{aligned}
x\Theta &= \exists p.\Theta[p](x) \\
x\Theta &= x \\
()\Theta &= () \\
(\mathbf{inj}_i\; v)\Theta &= \mathbf{inj}_i\; (v\Theta) \\
(v_1, v_2)\Theta &= (v_1\Theta, v_2\Theta) \\
([e])\Theta &= [e\Theta] \\
l\Theta &= l \\
s\Theta &= s \\
(\mathbf{enc}_P\; v)\Theta &= \mathbf{enc}_P\; (v\Theta)
\end{aligned}$$

$\boxed{e_1 \Theta}$ Applying Θ to e_1

$$\begin{aligned}
(\lambda x : A.e)\Theta &= \lambda x : A.(e(\Theta - x)) \\
(e\; v)\Theta &= (e\Theta)\,(v\Theta) \\
(\mathbf{let}\, x\, \mathbf{be}\, v\, \mathbf{in}\, e)\Theta &= \mathbf{let}\, x\, \mathbf{be}\, (v\Theta)\, \mathbf{in}\, (e(\Theta - x)) \\
(\mathbf{let}\, x{\leftarrow}e_1\, \mathbf{in}\, e_2)\Theta &= \mathbf{let}\, x{\leftarrow}(e_1\Theta)\, \mathbf{in}\, (e_2(\Theta - x)) \\
(\mathbf{ret}_{\tilde{p}} v)\Theta &= \mathbf{ret}_{\tilde{p}}(v\Theta) \\
(\mathbf{case}\,(v, x_1.e_1, x_2.e_2))\Theta &= \mathbf{case}\,(v\Theta, x_1.e_1(\Theta - x_1), x_2.e_2(\Theta - x_2)) \\
(\mathbf{split}\,(v, x_1.x_2.e))\Theta &= \mathbf{split}\,(v\Theta, x_1.x_2.e(\Theta - x_1 - x_2)) \\
(\mathbf{force}\; v)\Theta &= \mathbf{force}\,(v\Theta) \\
(\mathbf{fix} f{:}C.e)\Theta &= \mathbf{fix} f{:}C.(e\Theta) \\
f\Theta &= \Theta(f) \\
(\mathbf{load}_P(v, x.e))\Theta &= \mathbf{load}_P(v\Theta, x.e(\Theta - x)) \\
(\mathbf{comb}_P(v, x.e))\Theta &= \mathbf{comb}_P(v\Theta, x.e(\Theta - x)) \\
(\mathbf{decode}\,(v, x.e))\Theta &= \mathbf{decode}\,(v\Theta, x.e(\Theta - x)) \\
(\varepsilon.\, \mathbf{as}\, \mathsf{par}(P)\, \mathbf{do}\, e)\Theta &= \varepsilon.\, \mathbf{as}\, \mathsf{par}(P)\, \mathbf{do}\, (e\Theta) \\
(\Theta_1.\, \mathbf{as}\, \mathsf{sec}(P)\, \mathbf{do}\, e)\Theta &= \biguplus_p [p \mapsto \Theta_1[p] \uplus \Theta[p]].\, \mathbf{as}\, \mathsf{sec}(P)\, \mathbf{do}\, e
\end{aligned}$$

Figure 18. Applying substitutions.

$\boxed{\vDash \mu}$ *(Consistent μ)*

$$\textsc{MultiStoreCon-one}\quad \frac{}{\vDash [p{:}\sigma]} \qquad \textsc{MultiStoreCon-mul}\quad \frac{\sigma_1 \vDash \sigma_2 \qquad \vDash \mu[p_1{:}\sigma_1] \qquad \vDash \mu[p_2{:}\sigma_2]}{\vDash \mu[p_1{:}\sigma_1][p_2{:}\sigma_2]}$$

$\boxed{\sigma_1 \vDash \sigma_2}$ *(σ_1 is consistent with σ_2)*

$$\textsc{StoresCon-commut}\quad \frac{\sigma_2 \vDash \sigma_1}{\sigma_1 \vDash \sigma_2} \qquad \textsc{StoresCon-emp}\quad \frac{}{\varepsilon \vDash \sigma} \qquad \textsc{StoresCon-EqcVal}\quad \frac{\sigma_1 \vDash \sigma_2 \qquad v_1 = v_2}{\sigma_1[l{:}v_1] \vDash \sigma_2[l{:}v_2]}$$

$$\textsc{StoresCon-EqsVal}\quad \frac{\sigma_1 \vDash \sigma_2 \qquad v_1 = v_2}{\sigma_1[s{:}v_1] \vDash \sigma_2[s{:}v_2]} \qquad \textsc{StoresCon-clnot}\quad \frac{l \notin \mathsf{dom}(\sigma_2) \qquad \sigma_1 \vDash \sigma_2}{\sigma_1[l{:}v] \vDash \sigma_2}$$

$$\textsc{StoresCon-slnot}\quad \frac{s \notin \mathsf{dom}(\sigma_2) \qquad \sigma_1 \vDash \sigma_2}{\sigma_1[s{:}v] \vDash \sigma_2}$$

$\boxed{\vDash \Gamma}$ *(Consistent Γ)*

$$\textsc{MultiTypingCon-one}\quad \frac{}{\vDash [p{:}\Upsilon]} \qquad \textsc{MultiTypingCon-mult}\quad \frac{\Upsilon_1 \vDash \Upsilon_2 \\ \vDash \Gamma[p_1{:}\Upsilon_1] \qquad \vDash \Gamma[p_2{:}\Upsilon_2]}{\vDash \Gamma[p_1{:}\Upsilon_1][p_2{:}\Upsilon_2]}$$

$\boxed{\Upsilon_1 \vDash \Upsilon_2}$ *(Υ_1 is consistent with Υ_2)*

$$\textsc{TypingCon-commut}\quad \frac{\Upsilon_2 \vDash \Upsilon_1}{\Upsilon_1 \vDash \Upsilon_2} \qquad \textsc{TypingCon-emp}\quad \frac{}{\varepsilon \vDash \Upsilon} \qquad \textsc{TypingCon-EqvTy}\quad \frac{\Upsilon_1 \vDash \Upsilon_2 \qquad A_1 = A_2}{\Upsilon_1, x : A_1 \vDash \Upsilon_2, x : A_2}$$

$$\textsc{TypingCon-EqcTy}\quad \frac{\Upsilon_1 \vDash \Upsilon_2 \qquad C_1 = C_2}{\Upsilon_1, f : C_1 \vDash \Upsilon_2, f : C_2}$$

$$\textsc{TypingCon-vnot}\quad \frac{x \notin \mathsf{dom}(\Upsilon_2) \qquad \Upsilon_1 \vDash \Upsilon_2}{\Upsilon_1, x : A \vDash \Upsilon_2} \qquad \textsc{TypingCon-cnot}\quad \frac{f \notin \mathsf{dom}(\Upsilon_2) \qquad \Upsilon_1 \vDash \Upsilon_2}{\Upsilon_1, f : C \vDash \Upsilon_2}$$

$\boxed{\vDash \Sigma}$ *(Consistent Σ)*

$$\textsc{MultiStoreTypingCon-one}\quad \frac{}{\vDash [p{:}\Delta]} \qquad \textsc{MultiStoreTypingCon-mul}\quad \frac{\Delta_1 \vDash \Delta_2 \\ \vDash \Sigma[p_1{:}\Delta_1] \qquad \vDash \Sigma[p_2{:}\Delta_2]}{\vDash \Sigma[p_1{:}\Delta_1][p_2{:}\Delta_2]}$$

$\boxed{\Delta_1 \vDash \Delta_2}$ *(Δ_1 is consistent with Δ_2)*

$$\textsc{StoresTypingCon-commut}\quad \frac{\Delta_2 \vDash \Delta_1}{\Delta_1 \vDash \Delta_2} \qquad \textsc{StoresTypingCon-emp}\quad \frac{}{\varepsilon \vDash \Delta} \qquad \textsc{StoresTypingCon-EqcTy}\quad \frac{\Delta_1 \vDash \Delta_2 \qquad A_1 = A_2}{\Delta_1, l : A_1 \vDash \Delta_2, l : A_2}$$

$$\textsc{StoresTypingCon-cnot}\quad \frac{l \notin \mathsf{dom}(\Delta_2) \qquad \Delta_1 \vDash \Delta_2}{\Delta_1, l : A \vDash \Delta_2}$$

$$\textsc{StoresTypingCon-EqsTy}\quad \frac{\Delta_1 \vDash \Delta_2 \qquad A_1 = A_2}{\Delta_1, s : A_1 \vDash \Delta_2, s : A_2} \qquad \textsc{StoresTypingCon-snot}\quad \frac{s \notin \mathsf{dom}(\Delta_2) \qquad \Delta_1 \vDash \Delta_2}{\Delta_1, s : A \vDash \Delta_2}$$

Figure 19. Consistency of environments.

$\boxed{\Sigma \vDash \mu}$ *(Σ is consistent with μ.)*

$$\textsc{Com-emp}\quad \frac{}{\varepsilon \vDash \varepsilon} \qquad\qquad \textsc{Com-p}\quad \frac{\Sigma \vDash \mu \qquad \Sigma[p{:}\Delta]; p \vdash \Delta \vDash \sigma}{\Sigma[p{:}\Delta] \vDash \mu[p{:}\sigma]}$$

$\boxed{\Sigma; p \vdash \Delta \vDash \sigma}$ *(Under Σ, for party p, Δ is consistent with σ.)*

$$\textsc{Co-emp}\quad \frac{}{\Sigma; p \vdash \varepsilon \vDash \varepsilon} \qquad\qquad \textsc{Co-l}\quad \frac{\Sigma; p \vdash \Delta \vDash \sigma \qquad \varepsilon; \Sigma \vdash^{\varnothing}_{\mathsf{par}(p)} v : A}{\Sigma; p \vdash \Delta, l : A \vDash \sigma[l{:}v]}$$

$\boxed{\Sigma_1 \subseteq \Sigma_2}$ *(Σ_1 is a subset of Σ_2.)*

$$\textsc{CtxtSub-emp}\quad \frac{}{\varepsilon \subseteq \varepsilon} \qquad\qquad \textsc{CtxtSub-p}\quad \frac{\Sigma_1 \subseteq \Sigma_2 \qquad \mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\Delta_2)}{\Sigma_1[p{:}\Delta_1] \subseteq \Sigma_2[p{:}\Delta_2]}$$

Figure 20. Consistency of store typing.

$\boxed{\mu_1; e_1 \xrightarrow{\tilde{p}} \mu_2; e_2}$ *(Under μ_1, $\tilde{p}$ takes a step to go from e_1 to e_2)*

$$\textsc{SrcStepAppen-app1}\quad \frac{\mu_1; e \xrightarrow{\tilde{p}} \mu_2; e'}{\mu_1; e\ v \xrightarrow{\tilde{p}} \mu_2; e'\ v}$$

$$\textsc{SrcStepAppen-app2}\quad \frac{}{\mu_1; (\lambda x : A.e)\ v \xrightarrow{\alpha(P)} \mu_1; e[P \Mapsto x{:}v]}$$

$$\textsc{SrcStepAppen-letbe}\quad \frac{}{\mu_1; \textbf{let}\ x\ \textbf{be}\ v\ \textbf{in}\ e \xrightarrow{\alpha(P)} \mu_1; e[P \Mapsto x{:}v]}$$

$$\textsc{SrcStepAppen-let}\quad \frac{\mu_1; e_1 \xrightarrow{\tilde{p}} \mu_2; e_1'}{\mu_1; \textbf{let}\ x {\leftarrow} e_1\ \textbf{in}\ e_2 \xrightarrow{\tilde{p}} \mu_2; \textbf{let}\ x {\leftarrow} e_1'\ \textbf{in}\ e_2}$$

$$\textsc{SrcStepAppen-letpar}\quad \frac{l = \nu_1(v, Q)}{\mu_1; \textbf{let}\ x {\leftarrow} \textbf{ret}_{\mathsf{par}(Q)}\ v\ \textbf{in}\ e_2 \xrightarrow{\tilde{p}} \mu_1[Q \Mapsto l{:}v]; e_2[Q \Mapsto x{:}l]}$$

$$\textsc{SrcStepAppen-letsec}\quad \frac{s = \nu_{\mathsf{s}}(v, Q)}{\mu_1; \textbf{let}\ x {\leftarrow} \textbf{ret}_{\mathsf{sec}(Q)}\ v\ \textbf{in}\ e_2 \xrightarrow{\tilde{p}} \mu_1[Q \Mapsto s{:}v]; e_2[Q \Mapsto x{:}s]}$$

$$\textsc{SrcStepAppen-case}\quad \frac{i \in \{1, 2\}}{\mu_1; \textbf{case}\ (\textbf{inj}_i\ v, x_1.e_1, x_2.e_2) \xrightarrow{\alpha(P)} \mu_1; e_i[P \Mapsto x_i{:}v]}$$

$$\textsc{SrcStepAppen-split}\quad \frac{}{\mu_1; \textbf{split}\ ((v_1, v_2), x_1.x_2.e) \xrightarrow{\alpha(P)} \mu_1; e[P \Mapsto x_1{:}v_1, x_2{:}v_2]}$$

$$\textsc{SrcStepAppen-force}\quad \frac{}{\mu_1; \textbf{force}\ [e] \xrightarrow{\tilde{p}} \mu_1; e}$$

$$\textsc{SrcStepAppen-fix}\quad \frac{}{\mu_1; \textbf{fix} f{:}C.e \xrightarrow{\alpha(P)} \mu_1; e[P \Mapsto f{:}\textbf{fix} f{:}C.e]}$$

$$\textsc{SrcStepAppen-loadpar}\quad \frac{\forall\, p \in P.\ \mu[p][l] = v}{\mu_1; \textbf{load}_P(l, x.e) \xrightarrow{\mathsf{par}(P)} \mu_1; e[P \Mapsto x{:}v]}$$

$$\textsc{SrcStepAppen-secload}\quad \frac{\mu_1[p][l] = v}{\mu_1; \textbf{load}_p(l, x.e) \xrightarrow{\mathsf{sec}(p \uplus P)} \mu_1; e[p \uplus P \Mapsto x{:}v]}$$

$$\textsc{SrcStepAppen-comb}\quad \frac{\forall\, p \in P.\ \mu_1[p][s] = v}{\mu_1; \textbf{comb}_P(s, x.e) \xrightarrow{\mathsf{sec}(P \uplus Q)} \mu_1; e[P \uplus Q \Mapsto x{:}v]}$$

$$\textsc{SrcStepAppen-decpar}\quad \frac{}{\mu_1; \textbf{decode}\ (\textbf{enc}_{P \uplus Q}\ v, x.e) \xrightarrow{\mathsf{par}(P)} \mu_1; e[P \Mapsto x{:}v]}$$

$$\textsc{SrcStepAppen-decsec}\quad \frac{}{\mu_1; \textbf{decode}\ (\textbf{enc}_Q\ v, x.e) \xrightarrow{\mathsf{sec}(P \uplus Q)} \mu_1; e[P \uplus Q \Mapsto x{:}v]}$$

$$\textsc{SrcStepAppen-asdopar}\quad \frac{\mu_1; e \xrightarrow{\mathsf{par}(Q)} \mu_2; e'}{\mu_1; \varepsilon.\ \textbf{as}\ \mathsf{par}(Q)\ \textbf{do}\ e \xrightarrow{\tilde{p}} \mu_2; \varepsilon.\ \textbf{as}\ \mathsf{par}(Q)\ \textbf{do}\ e'}$$

$$\textsc{SrcStepAppen-asdosec}\quad \frac{\mu_1; e\Theta \xrightarrow{\mathsf{sec}(Q)} \mu_2; e'}{\mu_1; \Theta.\ \textbf{as}\ \mathsf{sec}(Q)\ \textbf{do}\ e \xrightarrow{\tilde{p}} \mu_2; \Theta.\ \textbf{as}\ \mathsf{sec}(Q)\ \textbf{do}\ e'}$$

$$\textsc{SrcStepAppen-asdoelim}\quad \frac{}{\mu_1; \Theta.\ \textbf{as}\ \tilde{q}\ \textbf{do}\ \tilde{e} \xrightarrow{\tilde{p}} \mu_1; \tilde{e}}$$

Figure 21. $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ small-step semantics.

$\boxed{\mu_1; e_1 \xrightarrow{\tilde{p}}_* \mu_2; e_2}$ *(Under μ_1, $\tilde{p}$ takes multiple steps)*

SrcStepClosure-z
$$\frac{}{\mu_1; e_1 \xrightarrow{\tilde{p}}_* \mu_1; e_1}$$

SrcStepClosure-nz
$$\frac{\mu_1; e_1 \xrightarrow{\tilde{p}} \mu_2; e_2 \qquad \mu_2; e_2 \xrightarrow{\tilde{p}}_* \mu_3; e_3}{\mu_1; e_1 \xrightarrow{\tilde{p}}_* \mu_3; e_3}$$

Figure 22. $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ small-step closure.

$\boxed{p \vdash C \rightsquigarrow e'}$ *(Redaction generation for party p and type C)*

GenRedA-tyf
$$\frac{}{p \vdash \mathsf{F}_{\alpha(Q)} A \rightsquigarrow \mathbf{ret}_{\mathsf{par}(p)}\, ()}$$

GenRedA-tylam
$$\frac{p \vdash C \rightsquigarrow e' : C'}{p \vdash A \rightarrow C \rightsquigarrow \lambda x : A . e'}$$

$\boxed{p \vdash v \rightsquigarrow v'}$ *(v translates to v' for p.)*

TrA-vvar
$$\frac{}{p \vdash x \rightsquigarrow x}$$

TrA-unit
$$\frac{}{p \vdash () \rightsquigarrow ()}$$

TrA-inj
$$\frac{p \vdash v \rightsquigarrow v'}{p \vdash \mathbf{inj}_i\, v \rightsquigarrow \mathbf{inj}_i\, v'}$$

TrA-pair
$$\frac{p \vdash v_1 \rightsquigarrow v_1' \qquad p \vdash v_2 \rightsquigarrow v_2'}{p \vdash (v_1, v_2) \rightsquigarrow (v_1', v_2')}$$

TrA-thk
$$\frac{p \vdash e \rightsquigarrow e'}{p \vdash [e] \rightsquigarrow [e']}$$

TrA-cloc
$$\frac{}{p \vdash l \rightsquigarrow l}$$

TrA-sloc
$$\frac{}{p \vdash s \rightsquigarrow s}$$

TrA-enc
$$\frac{p \vdash v \rightsquigarrow v'}{p \vdash \mathbf{enc}_P\, v \rightsquigarrow \mathbf{enc}_P\, v'}$$

$\boxed{p \vdash e \rightsquigarrow e'}$ *(e translates to e' for p.)*

TrA-lam
$$\frac{p \vdash e \rightsquigarrow e'}{p \vdash \lambda x : A . e \rightsquigarrow \lambda x : A . e'}$$

TrA-app
$$\frac{p \vdash e \rightsquigarrow e' \qquad p \vdash v \rightsquigarrow v'}{p \vdash e\, v \rightsquigarrow e'\, v'}$$

TrA-letbe
$$\frac{p \vdash v \rightsquigarrow v' \qquad p \vdash e \rightsquigarrow e'}{p \vdash \mathbf{let}\, x\, \mathbf{be}\, v\, \mathbf{in}\, e \rightsquigarrow \mathbf{let}\, x\, \mathbf{be}\, v'\, \mathbf{in}\, e'}$$

TrA-let1
$$\frac{\begin{array}{c} p \notin P \qquad y\ \mathtt{fresh} \\ p \vdash e_1 \rightsquigarrow e_1' \qquad p \vdash e_2 \rightsquigarrow e_2' \end{array}}{p \vdash \mathbf{let}\, x \leftarrow (e_1 : \mathsf{F}_{\alpha(P)} A)\, \mathbf{in}\, e_2 \rightsquigarrow \mathbf{let}\, y \leftarrow e_1'\, \mathbf{in}\, e_2'}$$

TrA-let2
$$\frac{\begin{array}{c} p \in P \\ p \vdash e_1 \rightsquigarrow e_1' \qquad p \vdash e_2 \rightsquigarrow e_2' \end{array}}{p \vdash \mathbf{let}\, x \leftarrow (e_1 : \mathsf{F}_{\alpha(P)} A)\, \mathbf{in}\, e_2 \rightsquigarrow \mathbf{let}\, x \leftarrow e_1'\, \mathbf{in}\, e_2'}$$

TrA-ret1
$$\frac{p \in P \qquad p \vdash v \rightsquigarrow v'}{p \vdash \mathbf{ret}_{\alpha(P)} v \rightsquigarrow \mathbf{ret}_{\tilde{p}} v'}$$

TrA-ret2
$$\frac{p \notin P}{p \vdash \mathbf{ret}_{\alpha(P)} v \rightsquigarrow \mathbf{ret}_{\mathsf{par}(p)}\, ()}$$

TrA-case
$$\frac{\begin{array}{c} p \vdash v \rightsquigarrow v' \\ p \vdash e_1 \rightsquigarrow e_1' \qquad p \vdash e_2 \rightsquigarrow e_2' \end{array}}{p \vdash \mathbf{case}\, (v, x_1.e_1, x_2.e_2) \rightsquigarrow \mathbf{case}\, (v', x_1.e_1', x_2.e_2')}$$

TrA-split
$$\frac{p \vdash v \rightsquigarrow v' \qquad p \vdash e \rightsquigarrow e'}{p \vdash \mathbf{split}\, (v, x_1.x_2.e) \rightsquigarrow \mathbf{split}\, (v', x_1.x_2.e')}$$

TrA-force
$$\frac{p \vdash v \rightsquigarrow v'}{p \vdash \mathbf{force}\, v \rightsquigarrow \mathbf{force}\, v'}$$

TrA-fix
$$\frac{p \vdash e \rightsquigarrow e'}{p \vdash \mathbf{fix} f{:}C.e \rightsquigarrow \mathbf{fix} f{:}C.e'}$$

TrA-cvar
$$\frac{}{p \vdash f \rightsquigarrow f}$$

TrA-load
$$\frac{p \vdash v \rightsquigarrow v' \qquad p \vdash e \rightsquigarrow e'}{p \vdash \mathbf{load}_P(v, x.e) \rightsquigarrow \mathbf{load}_P(v', x.e')}$$

TrA-comb
$$\frac{p \vdash v \rightsquigarrow v' \qquad p \vdash e \rightsquigarrow e'}{p \vdash \mathbf{comb}_P(v, x.e) \rightsquigarrow \mathbf{comb}_P(v', x.e')}$$

TrA-decode
$$\frac{p \vdash v \rightsquigarrow v' \qquad p \vdash e \rightsquigarrow e'}{p \vdash \mathbf{decode}\, (v, x.e) \rightsquigarrow \mathbf{decode}\, (v', x.e')}$$

TrA-asdopar
$$\frac{p \in P \qquad p \vdash e \rightsquigarrow e'}{p \vdash \varepsilon.\, \mathbf{as}\, \mathsf{par}(p \uplus P)\, \mathbf{do}\, e \rightsquigarrow \varepsilon.\, \mathbf{as}\, \mathsf{par}(p \uplus P)\, \mathbf{do}\, e'}$$

TrA-asdosec
$$\frac{}{p \vdash \Theta.\, \mathbf{as}\, \mathsf{sec}(p \uplus P)\, \mathbf{do}\, e \rightsquigarrow [p{:}\Theta[p]].\, \mathbf{as}\, \mathsf{sec}(p \uplus P)\, \mathbf{do}\, e}$$

TrA-asdoredact
$$\frac{p \notin P \qquad p \vdash C \rightsquigarrow e'}{p \vdash \Theta.\, \mathbf{as}\, \alpha(P)\, \mathbf{do}\, e : C \rightsquigarrow e'}$$

Figure 23. Translation from $\lambda^{\mathsf{src}}_{\mathsf{M3PC}}$ to $\lambda^{\mathsf{tgt}}_{\mathsf{M3PC}}$.