Domain-Polymorphic Programming of Privacy-Preserving Applications *

Dan Bogdanov Peeter Laud Jaak Randmets
Cybernetica AS
{dan.bogdanov,peeter.laud,jaak.randmets}@cyber.ee

April 22, 2014

Abstract

Secure Multiparty Computation (SMC) is seen as one of the main enablers for secure outsourcing of computation. Currently, there are many different SMC techniques (garbled circuits, secret sharing, homomorphic encryption, etc.) and none of them is clearly superior to others in terms of efficiency, security guarantees, ease of implementation, etc. For maximum efficiency, and for obeying the trust policies, a privacy-preserving application may wish to use several different SMC techniques for different operations it performs. A straightforward implementation of this application may result in a program that (i) contains a lot of duplicated code, differing only in the used SMC technique; (ii) is difficult to maintain, if policies or SMC implementations change; and (iii) is difficult to reuse in similar applications using different SMC techniques.

In this paper, we propose a programming language with associated compilation techniques for simple orchestration of multiple SMC techniques and multiple *protection domains*. It is a simple imperative language with function calls where the types of data items are annotated with protection domains and where the function declarations may be *domain-polymorphic*. This allows most of the program code working with private data to be written in a SMC-technique-agnostic manner. It also allows rapid deployment of new SMC techniques and implementations in existing applications. We have implemented the compiler for the language, integrated it with an existing SMC framework, and are currently using it for new privacy-preserving applications.

1 Introduction

Secure multiparty computation (SMC) is a cryptographic method for n different parties to evaluate a function

$$(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$$

so that each party $\mathcal{P}_i$ provides the input x_i and learns the output y_i such that no party can learn the inputs or outputs of another party. The first protocols for secure multiparty computation

*This research was, in part, funded by the Defense Advanced Research Projects Agency (DARPA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. Distribution Statement A (Approved for Public Release, Distribution Unlimited). This research has also been supported by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and by the Estonian Research Council through grant IUT27-1.

were proposed by Yao [34]. The techniques have been developed since then and several practical implementations of programmable secure computation have been created [21, 5, 15, 11].

When developing an application making use of SMC techniques (or other, possibly non-cryptographic secure computation techniques), we may want to use more than one technique simultaneously, and/or we may want to defer the choice of particular SMC techniques to a later stage of development. The main reason for this is efficiency — different operations may be fastest using different techniques, even when considering the costs of translating between data representations [19]. Confidentiality policies may compound this issue, stating that different pieces of data must be treated with techniques providing protection against different kinds of adversaries (passive vs. active; the size of coalitions it's able to form). In this case, we may use faster techniques for data needing less protection. Also, in our quest for speed, we may want to try out and profile different SMC techniques; this should be possible without rewriting the application.

We have responded to the wish to simultaneously use multiple SMC techniques by developing a secure computation runtime which is modular and makes the integration of new techniques easy. The application programmer, in order to make full use of the capabilities of the runtime, needs a language to express the functionality of the application and the possible choices of SMC techniques, without being forced to commit to particular techniques too early. The goal of this paper is to present such a programming language, describe its compilation and integration with the runtime. The design of the language has been somewhat inspired by Jif [23]. Several design choices have also been affected by our unique practical experience in developing SMC applications.

Our contribution. In this paper we present an implementation of the secure programming model. Our main contributions include the introduction of protection domains as an abstraction for a set of SMC protocols and the design of a programming language (an extension of SecreC [17, 31]) that uses protection domains as its foundation. The notion of protection domains is elaborated on in Sect. 2 of this paper. The language is a simple, strongly typed imperative language where each variable and piece of data carries its protection domain as part of its type information. Importantly, our type system supports protection domain polymorphism. This allows the actual choices of SMC techniques to be done after the implementation of the subroutines of the application. This also allows the development of libraries of common privacy-preserving functionalities that can be used with many SMC techniques. To obtain speed-ups from the existence of particularly fast protocols for some common functionality with some SMC technique, our language also supports overloading (full and partial) of polymorphic functions. The language and its type system are described in Sect. 3.

We have developed a compiler for our programming language that translates programs that use protection domains to bytecode executables that run on a secure computation runtime that enforces the protection domain restrictions. In this paper (Sect. 4 and Sect. 5), we present the translation of our language to a monomorphic intermediate language; its further translation to bytecode is standard.

Related work. Several languages for programming secure computation systems have been proposed [21, 25, 15, 32, 22]. However, these language do not provide a clear separation of data with different policies on the type system level. Furthermore, some of them are fixed to a single secure computation paradigm.

Our design has been influenced by the Decentralized Label Model (DLM) [24] and its implemen-

tation in Jif [23]. A label in DLM is closely related to our notion of a protection domain and label polymorphism appeared in Jif. The polymorphism is even more fine-grained in information flow analyses for programs in ML [27] or Haskell [20]. Our choice of following the imperative paradigm was influenced by its ubiquity in cryptographic literature.

2 Protection Domains

We start by defining the protection domains and their kinds.

Definition A protection domain kind (PDK) is a set of data representations, algorithms and protocols for storing and computing on protected data.

Definition A protection domain (PD) is a set of data that is protected with the same resources and for which there is a well- defined set of algorithms and protocols for computing on that data while keeping the protection.

Each protection domain belongs to a certain protection domain kind and each protection domain kind can have several protection domains.

A typical example of a PDK is secret sharing, with implementations for sharing, reconstruction, and arithmetic operations on shared values. A PD in this PDK would specify the actual parties doing the secret sharing, and the number of cooperating parties for reconstruction. Another example of a PDK is a fully homomorphic encryption scheme with operations for encryption, decryption, as well as for addition and multiplication of encrypted values. Here different keys correspond to different PD-s. Non-cryptographic methods for implementing PDK-s may involve trusted hardware or virtualization. In general, a PDK has to provide

1. A list of data types used in the PDK.
2. For each data type in a PDK we need:
   (a) a *classification* and *declassification* functions to convert between public and protected representation of values,
   (b) protocols or functions that perform operations on protected values

The functions performing secure operations should be universally composable so they can be combined into programs [7].

Constructing a useful PDK from secure multiparty computation is non-trivial, as many such schemes only support a few secure operations. It may be possible to construct more complex secure computation operations by composing simpler ones, but dedicated protocols have been shown to be more efficient in practice. We have identified several protocol suites that are suitable for implementation as protection domain runtimes with some work, including, but not limited to [2, 5, 10].

Our programming model includes a special *public* protection domain that does not apply any protective measures. The public protection domain is useful for working with values like public constants that do not have to be hidden.

Our programming language will allow the application developer to define protection domain kinds and instantiate these kinds as individual protection domains. Each PDK (including the

public one) supports a range of data types and operations on those data types; the operations are accessed through system calls whose implementation (as a cryptographic algorithm or protocol) is beyond the scope for application programmer. The data type of each value used in the program is annotated with its protection domain. In order to write generic code, our programming language makes use of polymorphism.

3 Polymorphic Language

We will now describe the programming language for expressing SMC applications. In this paper, we concentrate on the details related to protection domains. We skip the features that are also necessary for the ease of use of the language (e.g. the module system, arrays and arithmetic operators), but are orthogonal to the semantics and type system of the protection domains, and are also more straightforward to implement.

3.1 Abstract Syntax

The polymorphic language program consists of a sequence of declarations followed by a statement s. Every declaration is either a protection domain kind declaration, protection domain declaration, or a function declaration.

$$P \quad ::= (\texttt{pdk}\ k\texttt{;} \mid \texttt{pd}\ d\ \texttt{:}\ k\texttt{;} \mid F)^*\ s \tag{1}$$

Functions are optionally quantified over some protection domains and their body consists of a statement. If the function body is missing the function acts as a system call declaration.

$$\begin{array}{lcl} F & ::= & [\forall\ \alpha_1\texttt{,}\ \ldots\ \texttt{,}\alpha_n]\ f\texttt{(}\overline{x_i : d_i\ t_i}\texttt{)}\ \texttt{:}\ d\,t\ [s] \\ \alpha & ::= & d \mid d\ \texttt{:}\ k \end{array} \tag{2}$$

Data types, denoted with t, include integers `int`, booleans `bool`, and vectors of integers `int[]` and booleans `bool[]`. Kind variables are denoted with k, protection domain variables with d, function and system call names with f, and variables with x. A special protection domain name `public`, and the kind it belongs to `Public`, are reserved and considered predefined. Public data types `public` t are often written t^+ for conciseness.

Language statements and expressions are fairly standard for a WHILE-language. We also provide the ability to declare variables of given (domain and data) type in some scope. Expressions are limited to variables, literal constants and function or system call invocations.

$$\begin{array}{lcl} s & ::= & \texttt{skip} \mid s_1\ \texttt{;}\ s_2 \mid x\ \texttt{=}\ e \mid \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\ & \mid & \texttt{while}\ e\ \texttt{do}\ s \mid \texttt{return}\ e \mid \{x : d\,t\texttt{;}\ s\} \\ e & ::= & x \mid c_t \mid f\texttt{(}e_1\texttt{,}\ \ldots\ \texttt{,}e_n\texttt{)} \end{array}$$

3.2 Examples

The first example demonstrates that the selection of protection domains that a given algorithm is invoked on can greatly affect its performance. Consider the task of computing the Hamming distance between two vectors. We can implement this by comparing the input vectors point-wise in domain `D` and summing the result in a different protection domain `DT`. The user of this algorithm is free to select both protection domains provided that they support the used operations.

```
∀ D, DT. hammingDist (x: D int[], y: D int[]): DT int
{
    eq : D bool[] = x != y; // pointwise
    ns : DT int[] = boolToInt(reclassify(eq));
    return sum(ns);
}
```

Listing 1: Hamming distance

We will explore the performance of the previous algorithm by looking at two different protection domain kinds, both available on our runtime system. The first is additive secret sharing (among three parties) and the second is XOR secret sharing where data is bit-wise additively shared (also among three parties). Assume that integers are 64-bit wide and both protection domains are run on the same set of physical machines. In this particular case the algorithm performs best if the input is XOR shared and output is additively shared. In this case only 8 communication rounds are required for doing the computation. If both input and output are additively shared then the computation takes 10 communication rounds [6].

As the second example consider the task of sorting a vector of integers in privacy-preserving manner. The sorting method is polymorphic over the protection domain — the only restriction is that the given protection domain kind supports basic arithmetic and comparison. The generic sorting function operates by constructing a sorting network and obliviously performing compare-and-swap on pairs provided by the network.

```
∀ D. sort(src : D int[]) : D int[] {
  i : public int;
  a, b, c : D int;
  alength : public int = length(src);
  sn : public int[] = sortNetwork (alength);
  for (i=0; i < length(sn)-1; i = i + 2) {
    a = src[sn[i+0]]; b = src[sn[i+1]];
    c = isLessThan(a, b);
    src[sn[i+0]] = c*a + (1 - c)*b;
    src[sn[i+1]] = c*b + (1 - c)*a;
  }
  return src;
}
```

Listing 2: Generic sort

However, we can do better if we have more information about the given protection domain kind. In particular, if a protection domain kind `K` provides a fast method to shuffle vectors, an efficient method for sorting can be implemented. Comparison results of shuffled vector can be declassified and control flow of the program can depend on the declassified results. Sorting can be overloaded for this special case and when sorting a vector the overload resolution mechanism selects the appropriate implementation.

```
∀ D:K. sort(src : D int[]) : D int[] {
  dest : D int[] = shuffle(src);
  // Sort dest vector using public comparisons:
  // declassify(isLessThan(dest[i], dest[j]))
```

```
  return dest;
}
```

Listing 3: Specialized sort

The examples above do not precisely match the abstract syntax of the language. For readability we have used syntactic sugar and operators, such as arithmetic, that we have not defined syntactically. However, all of those operations can be modeled through system calls and both of the for-loops can be straightforwardly implemented using a while-loop.

3.3 Static Semantics

A polymorphic language program must satisfy a number of conditions in order to be considered valid, to have a (dynamic) semantics, and to be compilable. The main verification pass, which we state as a type system, makes sure that for all function calls in the program, there is a function declaration that matches it. A final pass ensures that all variables are initialized before being used and all (finite) execution paths through function bodies (except the main function) end with a `return` statement. Here we only consider the type checking phase.

In order to state typing judgments for various parts of the program, we will give unique labels to all function declarations. Let $\mathcal{L}$ be the set of labels of function declarations. Let us introduce the following notation:

- For a program P, let $\mathsf{pdk}(P)$ and $\mathsf{pd}(P)$ be the sets of protection domain kinds and protection domains declared in P. For $d \in \mathsf{pd}(P)$, let $\mathit{kind}(d) \in \mathsf{pdk}(P)$ be the kind of the protection domain d.
- For a function declaration f^ℓ in program P, let $\mathsf{arg}_P(\ell)$ be the list of declarations of its formal arguments, and $\mathsf{ret}_P(\ell)$ be the return type.
- For a program P and a function name f let

  $$\mathit{impl}_P(f; d_1\, t_1, \ldots, d_n\, t_n \to d) \in \mathcal{L} \cup \{\bot\}$$

  denote the label of a function declaration in P that has the name f and is the best match among all those declarations with the name f, and with the type matching the arguments with protection domains $d_1, \ldots, d_n \in \mathsf{pd}(P)$ and data types $t_1, \ldots, t_n$, and output with the protection domain d. If there is no such function declaration, or if there are multiple equally good matches, then let the value of $\mathit{impl}_P(\ldots)$ be $\bot$.
- For a label $\ell \in \mathcal{L}$ let $\delta(\ell)$ be the set of protection domains quantified in the declaration of the function with label ℓ.
- Let $\mathsf{body}_P(\ell)$ be the body of the function declared at the label $\ell \in \mathcal{L}$. If ℓ refers to a system call let $\mathsf{body}_P(\ell)$ be $\bot$. For a mapping $\theta : \delta(\ell) \to \mathsf{pd}(P)$, let $\mathsf{body}^\theta_P(\ell)$ be the body of the function declared at label ℓ, where all protection domains $d \in \delta(\ell)$ have been syntactically replaced with protection domains $d\theta$. We define $\mathsf{arg}^\theta_P(\ell)$, and $\mathsf{ret}^\theta_P(\ell)$ in identical manner by syntactically replacing domains d in $\delta(\ell)$ by $d\theta$.
- For a function declaration F labeled ℓ and protection domains $d'_0, \ldots, d'_n$ let $\theta = \mathit{unif}_\ell(d'_0, \ldots, d'_n)$ be a mapping from $\delta(\ell)$ to protection domains declared by the program, such that $d'_i = d_i\theta$ for every quantified d_i.

$$\frac{}{\Delta;P;\bot\,\bot;\emptyset\vdash \mathit{body}(P)}\quad \frac{}{\Delta;P;\mathsf{ret}^{\theta}_{P}(\ell);\mathsf{arg}^{\theta}_{P}(\ell)\vdash \mathsf{body}^{\theta}_{P}(\ell)}\quad \frac{}{\Delta;P;\theta\vdash f^{\ell}}\ \mathsf{body}^{\theta}_{P}(\ell)=\bot\quad \frac{}{\Delta;P;\theta\vdash f^{\ell}}\ d\in\mathsf{pd}(P)\quad \Delta;P;d_0\,t_0;(\Gamma,x:d\,t)\vdash s\quad \frac{}{\Delta;P;d_0\,t_0;\Gamma\vdash\{x:d\,t;\ s\}}$$

$$\frac{}{\Delta;P;d\,t;\Gamma\vdash \texttt{skip}}\quad \frac{}{\Delta;P;d\,t;\Gamma\vdash s_1\quad \Delta;P;d\,t;\Gamma\vdash s_2}\quad \frac{}{\Delta;P;d\,t;\Gamma\vdash s_1\ ;\ s_2}\quad \frac{}{\Delta;P;\Gamma\vdash e:\texttt{bool}^{+}\quad \Delta;P;d\,t;\Gamma\vdash s_1\quad \Delta;P;d\,t;\Gamma\vdash s_2}\quad \frac{}{\Delta;P;d\,t;\Gamma\vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

$$\frac{}{\Delta;P;\Gamma\vdash e:\texttt{bool}^{+}\quad \Delta;P;d\,t;\Gamma\vdash s}\quad \frac{}{\Delta;P;d\,t;\Gamma\vdash \texttt{while}\ e\ \texttt{do}\ s}\quad \frac{}{\Delta;P;\Gamma\vdash e:d\,t}\quad \frac{}{\Delta;P;d\,t;\Gamma\vdash \texttt{return}\ e}\quad \frac{}{(x:d\,t)\in\Gamma\quad \Delta;P;\Gamma\vdash e:d\,t}\quad \frac{}{\Delta;P;d_0\,t_0;\Gamma\vdash x\ \texttt{=}\ e}$$

$$\frac{}{\Delta;P;\Gamma\vdash c_t:t^{+}}\quad \frac{}{(x:d\,t)\in\Gamma}\quad \frac{}{\begin{array}{c}\Delta;P;\Gamma\vdash e_1:d_1\,t_1\quad\ldots\quad\Delta;P;\Gamma\vdash e_n:d_n\,t_n\\ f^{\ell}=\mathit{impl}_P(f;d_1\,t_1,\ldots,d_n\,t_n\to d)\quad(\ell,\theta)=\Delta(f,d,d_1,\ldots,d_n)\\ \theta=\mathit{unif}_{\ell}(d,d_1,\ldots,d_n)\quad\Delta;P;\theta\vdash f^{\ell}\quad d\,t=\mathsf{ret}^{\theta}_{P}(\ell)\end{array}}\quad \frac{}{\Delta;P;\Gamma\vdash f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}:d\,t}$$

Figure 1: Typing judgments of the polymorphic language

- Let $\mathit{body}(P)$ denote the body of a program P.

We have not specified how the function impl_P, for finding the concrete location of the function, given the function name and types of the arguments, is implemented. There are multiple possible implementations, for instance, we could return the first match or we could select a match that, in some sense, best fits the template parameters. Regardless of the implementation the type checking judgements remain the same. In our actual implementation, we select the function declaration that (i) quantifies over the least number of polymorphic protection domains, and (ii) out of those, has the strictest kind annotations. If multiple such function definitions match then a type error is raised.

Let the Δ be a function that maps function name f, return domain d, and domains of parameters $d_1,\ldots,d_n$ to pairs consisting of the location $\ell\in\mathcal{L}$, and domain substitution $\theta\in\mathsf{pd}(P)$. Intuitively Δ is instantiation context that stores function instantsiations to concrete domains. At times we will consider Δ as a set. In the type system, the following kinds of judgments will be considered.

- $\Delta\vdash P$ means that the program P is well-typed in instantiation context Δ.
- $\Delta;P;\theta\vdash f^{\ell}$ means that the function f with label ℓ in the program P is well-typed if protection domains in $\delta(\ell)$ are bound to the protection domains given by θ.
- $\Delta;P;d\,t;x_1:d_1\,t_1,\ldots,x_n:d_n\,t_n\vdash s$ means that in the program P, in a function returning a value with data type t in protection domain d, the statement s is well-typed if the free variables x_i have the protection domains d_i and the data types t_i. Shorthand Γ is often used to represent the product of types $x_1:d_1\,t_1,\ldots,x_n:d_n\,t_n$, and we write $\mathcal{G}$ as shorthand for $\Delta;P;d\,t;\Gamma$.
- $\Delta;P;\Gamma\vdash e:d\,t$ means that the expression e, in the given context, has data type t in protection domain d. We write $\mathcal{H}$ as a shorthand for $\Delta;P;\Gamma$.

The typing rules are presented in the Fig. 1. There are two noteworthy points. First, our typing rules are not to be interpreted inductively, but co-inductively. Intuitively, we consider a typing judgment to be valid if it belongs to a set of typing judgments where all elements can be

justified using these rules and the judgments in this set. There exists the largest such set; it is obtained by starting from the set of all possible typing judgments and iteratively deleting from it all judgments that cannot be justified.

The second point is that these rules constitute the definition of typability, but are not intended to describe an actual type-checking algorithm. The actual algorithm deployed in our compiler has similarities to the instantiation of templates during the compilation of C++ programs.

Similarly to existing languages for programming secure computation, we let the control flow of the program to only depend on public values. This reflects the common consideration of the costs of hiding the control flow as too high to perform automatically (it would involve the execution of both branches, and obliviously selecting the result of one of them). If needed, the programmer has to explicitly encode the branching on private values (or it could be provided as syntactic sugar).

For a well-typed program the instantiation context that the program type checks under is not unique. This is because the type checking rules do not restrict the instantiation context in any way but only require that some specific set of instantiations occur. Redundant instantiations can easily be removed from the context, but there doesn't always have to be an unique smallest instantiation context.

Consider a function $\forall$`d.f():d int` polymorphic in the return type and $\forall$`d.g(x:d int):int` polymorphic in the argument return type. According to our type checking rules the composition of these functions `g(f())` is well-typed, but the concrete protection domain can be picked arbitrarily. In practice we want to reject programs that lead to this kind of ambiguous behavior. We say that the program P is *unambiguously typed* if there exists an instantiation context Δ such that $\Delta \vdash P$ and for any Δ' if $\Delta' \vdash P$ then $\Delta \subseteq \Delta'$. One may notice that if the program doesn't declare any protection domains then the program with the expression `g(f())` is unambiguously typed. We can reject this kind of situation if for every protection domain kind we add countably infinite set of dummy protection domains that may not be used in the program. From now if we talk about well-typed program we always mean unambiguously typed program.

3.4 Dynamic Semantics

We will in the following define a small-step operational semantics of the polymorphic language. To define the small-step transition rules of the language we extend the abstract syntax of expressions with all possible values that variables can take: $e ::= \ldots \mid d\,v$.

Values carry the protection domain and are pairs consisting of a protection domain d, and some representation $v \in \mathbf{Val}$ (for simplicity of treatment, we conflate all data types to a single set **Val** here) which is left abstract. Public values `public` v are denoted as v^+ for conciseness. A special bottom value $\bot \in \mathbf{Val}$ is used to denote uninitialized or undefined values.

In order to specify the evaluation order we define statement evaluation context $\mathcal{S}$ and expression evaluation contexts $\mathcal{E}$ using a simple grammar in Fig. 2. We choose to evaluate expressions from left to right in a deterministic order. For every evaluation context we define a mapping from expressions to either statements or expressions respectively. The definitions of the functions $\mathcal{S}[-] : e \to s$, and $\mathcal{E}[-] : e \to e$ have been omitted, but are easily defined by structural recursion.

The semantics is a set of triples that we denote $C^\circ \xrightarrow{\kappa} C^\bullet$. Here C° is a *program configuration* (defined below), $C^\bullet$ is the configuration after making a single step in the program, and κ is the *action* performed during that step. If the step was not an invocation of a system call, then the action is empty (denoted either with τ or by simply omitting it). If the step consisted of making a system call with name f, arguments $v_1, \ldots, v_n$ in protection domains $d_1, \ldots, d_n$, returning a value

$$\begin{array}{rcl} \mathcal{S} & ::= & \mathcal{S}\ \texttt{;}\ s_2 \mid x\ \texttt{=}\ \mathcal{E} \mid \texttt{return}\ \mathcal{E} \mid \texttt{if}\ \mathcal{E}\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\ \mathcal{E} & ::= & [] \mid f\,(d_1\,v_1\texttt{,}\ \ldots\texttt{,}\ \mathcal{E}\texttt{,}\ e_i\texttt{,}\ \ldots\texttt{,}\ e_n) \end{array}$$

Figure 2: Evaluation context

v in the protection domain d then we denote the action with $\kappa \equiv v = f(d, d_1, v_1, \ldots, d_n, v_n)$. System calls can not choose the protection domain the value is returned in. The only source of system calls in the polymorphic language are function calls that refer to functions with omitted body.

A *configuration* $C = c_1 : \ldots : c_n : c_{n+1}$ consists of a list of *stack frames*, where every frame c_i, other than the last, is a triple consisting of a statement evaluation context $\mathcal{S}$, return domain d, and local environment γ. A configuration always consists of at least one stack frame. The statement evaluation context $\mathcal{S}$ acts as a return position, and if the current procedure returns we plug the returned value into the context $\mathcal{S}$ of the previous stack frame and continue evaluating the resulting statement. Currently active stack frame c_{n+1} is a pair consisting of return domain and local environment.

The program configuration C° is either a program $\langle P \rangle$ or a pair of configuration and statement $\langle C, s \rangle$. The target configuration $C^\bullet$ of transition may be a regular program configuration C° or a configuration C if the evaluation halts.

With such set-up, the definition of the triples $C^\circ \xrightarrow{\kappa} C^\bullet$ is quite straightforward. First we define the expression evaluation rules in the form $\gamma \vdash \langle e \rangle \Rightarrow \langle d\,v \rangle$, stating that the expression e evaluates to the value v in protection domain d, if the values and protection domains of variables occurring in e are given by γ. The transition rules for the statements are in the form $\Delta \vdash \langle C, s \rangle \xrightarrow{\kappa} C^\bullet$, where s is the statement still left to execute. The instantiation context Δ has the same meaning as in Sec. 3.3; we assume that the program P is unambiguity typed and the context Δ is the unique smallest one. The context Δ is needed to dispatch function calls dynamically based on the possible protection domains the function call resides in and the protection domains of the arguments. The expression evaluation and transition rules are defined in Appendix A.

3.5 Trace Semantics

Trace is a sequence of program states that are connected by actions. Individual states of trace are hidden and denoted with $\bullet$ as we are only concerned about which sequences of actions a program may perform. Traces may be empty, finite or infinite. For example, configuration in final state has trace $\bullet$, but a infinite trace $\bullet \to \bullet \ldots$ corresponds to configuration which performs no actions and loops indefinitely. A set of all finite traces $\mathcal{T}^*$, infinite traces $\mathcal{T}^\omega$, and arbitrary traces $\mathcal{T}^\infty$ are defined over the set of labels $\mathcal{A}$ corresponding to actions (i.e. system calls) a program may perform.

Trace semantics of a well-typed program $\Delta \vdash P$ is defined by a set of sequences of labels $[\![P]\!] \subseteq \mathcal{T}^\infty$ by collecting all finite and infinite traces starting with the program configuration $\langle P \rangle$. All finite traces have to stop in some configuration C as we assume that the body of the program does not contain a return statement.

$$\begin{array}{rcl} [\![P]\!] & = & \{\bullet \xrightarrow{\kappa_1} \bullet \ldots \bullet \xrightarrow{\kappa_n} \bullet \mid \Delta \vdash \langle P \rangle \xrightarrow{\kappa_1} \ldots \xrightarrow{\kappa_n} C\} \\ & \cup & \{\bullet \xrightarrow{\kappa_1} \bullet \ldots \mid \Delta \vdash \langle P \rangle \xrightarrow{\kappa_1} \ldots\} \end{array} \tag{3}$$

A *symbolic trace* T summarizes a set of traces. A symbolic trace is a rooted, (potentially)

infinite and infinitely branching tree, where each path, starting from the root vertex, corresponds to a trace potentially generated by a program P. Formally, for each type t, let $\mathcal{N}_t$ be a countable set of *names*, with $\mathcal{N}_t \cap \mathcal{N}_{t'} = \emptyset$ if $t \neq t'$. A *symbolic action* $\mathbf{A}$ is either empty or of the form $\mathsf{v} = f(d, d_1, \mathsf{v}_1, \ldots, d_n, \mathsf{v}_n)$, where $\mathsf{v} \in \mathcal{N}_t$ and $\mathsf{v}_i \in \mathcal{N}_{t_i} \cup \{\bot\}$, such that the system call f exists for protection domains $d, d_1, \ldots, d_n$, argument types $t_1, \ldots, t_n$ and return type t. Each node of a symbolic trace T is labeled with a symbolic action $\mathbf{A}$. The number of descendants of non-leaf nodes u of T depends on the symbolic action $\mathbf{A}_u \equiv \mathsf{v} = f(d, \ldots)$ labeling it. If d is not `public`, (or if $\mathbf{A}_u$ is empty) then u has one descendant. If d is `public`, then the descendants of u are in one-to-one relationship with the values of the type t, where $\mathsf{v} \in \mathcal{N}_t$. Finally, for each path from the root of T downwards, any name v occurring in some symbolic action on this path may only occur in the argument position of some symbolic action after it has occurred in the value position before.

Let $g = (g_t)$ be a family of mappings (indexed by the types t) from $\mathcal{N}_t$ to the values of the type t. The application $g(\mathsf{T})$ gives us a concrete trace where each name $\mathsf{v} \in \mathcal{N}_t$ occurring in some symbolic action is replaced with $g_t(\mathsf{v})$, and where for all nodes u of T labeled with $\mathbf{A}_u \equiv \mathsf{v} = f(\mathtt{public}, \ldots)$, the subtree corresponding to $g(\mathsf{v})$ is selected. Let $[\![\mathsf{T}]\!]$ be the set of traces $\{g(\mathsf{T})\}$ for all possible families of mappings g.

The semantics of a program P defines a unique (up to α-conversion) symbolic trace $[\![P]\!]$ which can be either defined similarly to Sec. 3.4 or recovered from the trace semantics of P. It is easy to see that $[\![[\![P]\!]]\!] = [\![P]\!]$. It is also easy to see that $[\![\cdot]\!]$ is an injective (up to α-conversion) mapping, thus there is essentially a single symbolic trace corresponding to the trace semantics $[\![P]\!]$.

3.6 Execution

Our runtime environment securely implements (in the sense of universal composability [7]) an *arithmetic black box (ABB)* [9] with multiple protection domains. Recall that an arithmetic black box for n parties is an ideal functionality $\mathcal{F}_{\mathsf{ABB}}$ that allows the parties to perform secure arithmetic with values stored in the ABB. The parties refer to the stored values through handles. To perform an operation, sufficiently many parties (the exact number depending on the security model of the protocols implementing the ABB) must instruct the ABB to perform that operation, specifying the handles of the arguments, as well as giving a name to the handle to the result. A value in the ABB can be declassified if sufficiently many parties decide to give the respective instruction to the ABB, in this case all parties will learn that value. Values can be input to the ABB either by the parties themselves, or the ABB can be instructed to obtain a value from the outside environment. Such values can later be accessed through handles and used in computations.

In our case, the ABB $\mathcal{F}_{\mathsf{ABB}}$ stores for each type t and protection domain d the mapping $\mathsf{st}_{t,d}$ from the handles defined for this type and domain to the values in this type. It is natural to consider the domain of $\mathsf{st}_{t,d}$ to be a finite subset of $\mathcal{N}_t$. The ABB accepts instructions of the form $\mathsf{v} = f(d, d_1, \mathsf{v}_1, \ldots, d_k, \mathsf{v}_k)$, where the system call f has been defined for the protection domains $d, d_1, \ldots, d_k$ and the types $t, t_1, \ldots, t_k$, where $\mathsf{v} \in \mathcal{N}_t$ and $\mathsf{v}_i \in \mathcal{N}_{t_i} \cup \{\bot\}$. The ABB *implements* the system calls: when receiving the instruction $\mathsf{v} = f(d, d_1, \mathsf{v}_1, \ldots, d_k, \mathsf{v}_k)$, it will pick a value corresponding to the handle v. This value may depend on the values assigned to $\mathsf{v}_1, \ldots, \mathsf{v}_k$, as well as on the responses $\mathcal{F}_{\mathsf{ABB}}$ receives from the environment, and the random choices of $\mathcal{F}_{\mathsf{ABB}}$. Hence the ABB provides a refinement for the non-determinism present in the dynamic semantics of the programming language.

Using an ABB for n parties, a program P is executed as follows. There are n computing parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, the ABB $\mathcal{F}_{\mathsf{ABB}}$ and the outer environment $\mathcal{Z}$ (that may e.g. provide inputs to

the computation), as well as the adversary $\mathcal{A}_\mathcal{S}$ that may e.g. corrupt certain parties and perform other "permitted" attacks on $\mathcal{F}_{\mathsf{ABB}}$. All parties $\mathcal{P}_i$ give instructions to $\mathcal{F}_{\mathsf{ABB}}$ according to the symbolic trace $[\![P]\!]$. If the instruction produces a value in protection domain `public`, the parties subsequently ask the ABB to declassify this value. In this way, they select one of the traces from $[\![P]\!]$. In effect, this execution defines a probability distribution over the set of traces $[\![P]\!]$. We denote this distribution by $\mathcal{D}[\mathcal{F}_{\mathsf{ABB}}, \mathcal{Z}, \mathcal{A}_\mathcal{S}](P)$.

A number of protocol sets π_{ABB} implementing an ABB have been proposed: garbled circuits [34] making use of encryption and oblivious transfer [30]; protocols that perform arithmetic operations with values in secret-shared form [33, 14, 12]; homomorphic encryption [26, 8], including fully homomorphic encryption (FHE) [13]; private BDDs [16], etc. Different protocol sets may have a different set of operations that can be executed by the parties using the ABB. They all are *at least as secure as* $\mathcal{F}_{\mathsf{ABB}}$ [7]: for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ from a class of adversaries (this class determines the number of tolerated corruptions of parties, as well as their kind — either active or passive), there exists an adversary $\mathcal{A}_\mathcal{S}$, such that the views of $\mathcal{Z}$ in the systems $\mathcal{Z}\|\pi_{\mathsf{ABB}}\|\mathcal{A}$ and $\mathcal{Z}\|\mathcal{F}_{\mathsf{ABB}}\|\mathcal{A}_\mathcal{S}$ are indistinguishable. These ABB implementations correspond to ABBs with single protection domain. The combination of such ABB implementations, together with extra functionality for converting data between protection domains [28] gives us an ABB implementation with multiple protection domains.

We can now state the theorem on the correctness and security of the execution of the program P using the real implementations of protocols, compared to the specification. The proof of the theorem is trivial, showing that the universal composability framework is wholly adequate to relating abstract and concrete executions of protocols, allowing us to concentrate on specifications of what should be executed.

Theorem 3.1 *Let π_{ABB} be a secure implementation of $\mathcal{F}_{\mathsf{ABB}}$. Then for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ from the class of adversaries admitted by π_{ABB}, there exists an adversary $\mathcal{A}_\mathcal{S}$, such that*

Correctness *the distributions of traces $\mathcal{D}[\mathcal{F}_{\mathsf{ABB}}, \mathcal{Z}, \mathcal{A}_\mathcal{S}](P)$ and $\mathcal{D}[\pi_{\mathsf{ABB}}, \mathcal{Z}, \mathcal{A}](P)$ are indistinguishable;*

Security *the views of $\mathcal{Z}$ in systems $\mathcal{Z}\|\mathcal{P}_1\|\cdots\|\mathcal{P}_n\|\mathcal{F}_{\mathsf{ABB}}\|\mathcal{A}_\mathcal{S}$ and $\mathcal{Z}\|\mathcal{P}_1\|\cdots\|\mathcal{P}_n\|\pi_{\mathsf{ABB}}\|\mathcal{A}$ are indistinguishable.*

Proof Follows directly from π_{ABB} being at least as secure as $\mathcal{F}_{\mathsf{ABB}}$. Consider the composition of $\mathcal{Z}, \mathcal{P}_1, \ldots, \mathcal{P}_n$ as the environment $\mathcal{Z}'$ for π_{ABB}. For this environment and the adversary $\mathcal{A}$, there exists $\mathcal{A}_\mathcal{S}$, such that the views of $\mathcal{Z}'$ are indistinguishable in systems $\mathcal{Z}'\|\pi_{\mathsf{ABB}}\|\mathcal{A}$ and $\mathcal{Z}'\|\mathcal{F}_{\mathsf{ABB}}\|\mathcal{A}_\mathcal{S}$. Both claims of the theorem state the indistinguishablity of certain parts of these views.

3.7 Security of Information Flow

The protection domains give us a simple discipline for information flow control. An observer able to access data only in certain protection domains will learn nothing about data in other domains, as long as no operation explicitly transfers data between these protection domains.

The semantics (3) specifies the order in which system calls are made. The actual execution of the program depends on the implementations of these system calls. The used ABB $\mathcal{F}_{\mathsf{ABB}}$, the environment $\mathcal{Z}$ and the adversary $\mathcal{A}_\mathcal{S}$ define the semantics $[\![f]\!]$ of an arity-k system call f as a

function with the type $\mathbf{PD} \times (\mathbf{PD} \times \mathbf{Val})^k \times \mathcal{W} \to \mathbf{Val} \times \mathcal{W}$, where **PD** is the set of all protection domains and **Val** is the set of possible values the program operates on. For simplicity of exposition, we assume that we have a single type (with the set of values **Val**) and all system calls are defined for all protection domains. The result of the system call depends not only on its arguments and their (and the result's) protection domains, but also (e.g. when inputting values) on the outside environment $\mathcal{Z}\|\mathcal{A}_\mathcal{S}$. We let $\mathcal{W}$ denote the set of possible states of the environment. This state can change due to the system call, thus codomain of $[\![f]\!]$ also contains $\mathcal{W}$.

The initial state of the environment is distributed according to $\boldsymbol{\mathcal{W}} \in \mathcal{D}(\mathcal{W})$, this distribution is known to everyone ($\mathcal{D}(X)$ denotes the set of probability distributions over the set X). The distribution $\boldsymbol{\mathcal{W}}$ and the semantics of system calls define a probability distribution over the set of traces $[\![P]\!]$. In this setting, the semantics of all system calls can be assumed to be deterministic, as the random coins they might use may be considered to be a part of $\mathcal{W}$.

To speak about information flow security, we partition the set of protection domains into "low" and "high" domains: $\mathbf{PD} = \mathbf{PD}_L \dot\cup \mathbf{PD}_H$, with $\texttt{public} \in \mathbf{PD}_L$. Similarly, the state of the world has "low" and "high" part: $\mathcal{W} = \mathcal{W}_L \times \mathcal{W}_H$; these parts must be independent in the initial distribution $\boldsymbol{\mathcal{W}}$. For each label $\kappa \equiv v = f(d, d_1, v_1, \ldots, d_n, v_n)$ we define its *low-slice* $\overline{\kappa}$ by replacing each v_i by a placeholder $*$ iff $d_i \in \mathbf{PD}_H$; this definition straightforwardly extends to traces. A label κ is a *declassification label* if $d \in \mathbf{PD}_L$, but $d_i \in \mathbf{PD}_H$ for some i. We require that the semantics of system calls $[\![f]\!]$ only has the information flows that we expect it to have: the low-part of the state of the environment output by $[\![f]\!]$ may only depend on the low inputs to $[\![f]\!]$, and the value v output by $[\![f]\!]$ may depend on the high-part of the state of the environment only if $d \in \mathbf{PD}_H$. It is natural to require that the environment $\mathcal{Z}$ ensures the absence of other information flows even without the cooperation of the adversary $\mathcal{A}_\mathcal{S}$.

We can show that unless κ is a declassification label, the execution of a system call labeled κ does not increase the low-adversary's (that can observe the low-slices of traces) knowledge about the high-part of the initial state of the environment. We define a probabilistic notion of the knowledge —similar to [1]— of the adversary after observing the low-slice of the trace so far. The adversary's knowledge $\mathcal{KN}_P^{\overline{T},w_L}$ is a probability distribution that assigns to each pair (w', T'), where $w' \in \mathcal{W}$ and T' is a trace, the probability that the execution of the program P started in state w' and proceeded along the trace T', given that the low-slice of the trace so far has been $\overline{T}$ and the initial low-part of the state of the environment was w_L. We can show (see Appendix D) that if κ is not a declassification label, then $\mathcal{KN}_P^{\overline{T},w_L} = \mathcal{KN}_P^{\overline{T;\kappa},w_L}$.

4 Monomorphic Language

The monomorphic language is a WHILE-language with functions and PDs. The notion of protection domains remains in the language but the run-time behavior no longer depends on them. All dynamic dispatches have been resolved; the function calls in the monomorphic language are statically bound. The majority of program optimizations and static analysis should be done at the level of the monomorphic language.

Syntactically the target language has only few differences from the high level language (1) and (2). The first is that functions can no longer be polymorphically quantified over protection domains, and every function definition is uniquely indexed to distinguish between the instances of the same polymorphic language function. We also assume that every function call refers to the unique index ι.

4.1 Dynamic Semantics

Operational semantics of the monomorphic language (Appendix B) is defined in very similar style to the polymorphic language semantics. A major difference is that environment no longer stores domain types of values, and due to the lack of dynamic dispatch, the semantic rules do not depend on the instance environment Δ. We shall not define all of the language concepts because they are almost identical to those presented for the polymorphic language semantics.

Trace semantics of the monomorphic language program is defined using the same notion of traces as defined for the polymorphic language. It's possible because the traces hide the individual configurations between the transitions, but the labels are same between the language semantics. Trace semantics of monomorphic language program P' is defined by a set of sequences of labels $[\![P']\!] \subseteq \mathcal{T}^\infty$ analogously to the polymorphic language trace semantics. The resulting semantics is executable in the same way as described in Sec. 3.6.

5 Translation

This section covers a type-directed method for translating programs of the polymorphic language to the programs of the monomorphic language. The type-directed translation methods model compilation to intermediate representation.

5.1 Translation Methods

Type-directed methods for translating statements are in the form $\Delta; P; d\,t; \Gamma \vdash s \rightsquigarrow s'$ where s is a statement of the polymorphic language and s' is a statement of the monomorphic language. The context Δ contains instantiations of functions to some particular protection domains, and Γ is the type environment for variables, d and t denote the return type of the current function. Translation rules of expressions are in the form $\Delta; P; \Gamma \vdash e \rightsquigarrow e' : d\,t$, where e is expression of the polymorphic language, e' is expression of the monomorphic language, d is the domain type, and t is the data type of the expression.

The translation methods follow the type checking rules directly. There are, however, few notable points. The first difference is that in the monomorphic language we refer to the concrete instances of polymorphic functions using the index $\iota = (\ell, \theta)$ where ℓ is the label of function in the polymorphic language and θ is the substitution to concrete protection domains. Due to unambiguity this index is unique for every function call. Second notable point is that the translation does not always produce a well-typed monomorphic language program — the instantiation context might contain instantiations that do not type. If given smallest Δ such that $\Delta \vdash P$ and $\Delta \vdash P \rightsquigarrow P'$ then P' type checks.

5.2 Functional Correctness

Informally we wish to show that the original program and the translated program give rise to the same set of sequences of system calls. First we will define equivalence relation between the polymorphic and monomorphic language program configurations. Next we will assert a lemma that intuitively states that equivalent program configurations will eventually transition into equivalent configurations. Finally, we will state a theorem to establish that the translation to monomorphic form preserves semantics.

To establish an equivalence relation between polymorphic and monomorphic language configurations we define equivalence relation between: environments, singleton configurations, and program configurations. Two environments $\Gamma \vdash \gamma \equiv \gamma'$ are equivalent under the given type environment Γ if, for every $(x_i : d_i\, t_i) \in \Gamma$, there exists $v_i \in \mathbf{Val}$, such that $\gamma(x_i) = d_i\, v_i$ and $\gamma'(x_i) = v_i$. Additionally, if a variable does not occur in the type environment Γ it must not occur in either of the environments. Given equivalent environments the equivalence of singleton configurations rises naturally, and equivalence of configurations and program configurations follows from statement translation rules.

$$\frac{\Delta; P; d_0'\, t_0'; \Gamma' \vdash C \equiv C' \qquad \Gamma \vdash \gamma \equiv \gamma' \qquad \Delta; P; d_0\, t_0; \Gamma \vdash \mathcal{S}[d_0\, \bot] \rightsquigarrow \mathcal{S}'[\bot]}{\Delta; P; d_0\, t_0; \Gamma \vdash (\mathcal{S}, d_0, \gamma) : C \equiv (\mathcal{S}', \gamma') : C'}$$

$$\frac{\mathcal{G} \vdash C \equiv C' \qquad \mathcal{G} \vdash s \rightsquigarrow s'}{\mathcal{G} \vdash \langle C, s\rangle \equiv \langle C', s'\rangle}$$

We write $C^\circ \xrightarrow{\kappa^*} C^\bullet$, for both polymorphic and monomorphic program configurations, if it's possible to transition from the source configuration to the destination configuration by performing the action κ followed and preceded by some finite number of empty transitions. For empty transitions we allow that $C^\circ \xrightarrow{\tau^*} C^\circ$.

Lemma 5.1 (Weak bisimulation) *Let $\mathcal{G}$ be statement type checking context. For every two equivalent program configurations $\mathcal{G} \vdash C^\circ \equiv C'^\circ$ the following conditions hold:*

1. *For every label κ and configuration $C^\bullet$ if $\Delta \vdash C^\circ \xrightarrow{\kappa} C^\bullet$ there exists $C'^\bullet$ such that $\vdash C'^\circ \xrightarrow{\kappa^*} C'^\bullet$ and $\mathcal{G} \vdash C^\bullet \equiv C'^\bullet$.*
2. *For every label κ and configuration $C'^\bullet$ if $\vdash C'^\circ \xrightarrow{\kappa} C'^\bullet$ there exists $C^\bullet$ such that $\Delta \vdash C^\circ \xrightarrow{\kappa^*} C^\bullet$ and $\mathcal{G} \vdash C^\bullet \equiv C'^\bullet$.*

Proof By structural recursion over the translation methods.

Finite traces are said to be equivalent if they are equal without their empty transitions. We say that two infinite traces $T, T' \subseteq \mathcal{T}^\omega$ are equivalent $T \cong T'$ if for every prefix of T there exists equivalent prefix of T'. Notice that, according to our definition, a finite trace can never be equivalent to an infinite trace. This matches with intuition that, given enough time, a non-terminating program can always be distinguished from a terminating program no matter the system calls they perform. Sets of traces are equivalent if for every trace from one set there exists equivalent trace in another. More formally $\mathcal{T} \cong \mathcal{T}'$ if for every $T \in \mathcal{T}$ there exists $T' \in \mathcal{T}'$ such that $T \cong T'$, and for every $T' \in \mathcal{T}'$ there exists $T \in \mathcal{T}$ such that $T \cong T'$.

Theorem 5.2 *For every unambiguously well-typed polymorphic language program $\Delta \vdash P$ and a monomorphic language program P', the following holds. If $\Delta \vdash P \rightsquigarrow P'$ then the trace semantics of the two programs are equivalent: $[\![P]\!] \cong [\![P']\!]$.*

The equivalence of trace sets of the polymorphic language program P and its compilation to monomorphic language P' implies the indistinguishability of their executions, using either the arithmetic black box $\mathcal{F}_{\mathsf{ABB}}$ or its concrete implementation π_{ABB}. Indeed, the behaviour of the machines $\mathcal{P}_1, \ldots, \mathcal{P}_n$ described in Sec. 3.6 and referred to in Thm. 3.1 depends only on the symbolic

trace of the program P, which is determined by the trace semantics $[\![P]\!]$. As $[\![P]\!] \cong [\![P']\!]$, the instructions given to $\mathcal{F}_{\mathsf{ABB}}$ (or π_{ABB}) by the execution of P or P' are the same and the executions proceed in lock-step.

5.3 Security

The secure information flow property we stated in Sect. 3.7 was specified in terms of the set of program traces making up the semantics of the program. In particular, the property did not directly refer to the program that generated those traces. One could then also ask for a particular program in the intermediate language whether it satisfies that information flow property; some of them would satisfy it for a particular choice of $\mathbf{PD}_L$, while the others wouldn't. But as the translation from the polymorphic language to the monomorphic language preserves the semantics of the program, we immediately obtain that the translation of any polymorphic language program satisfies the secure information flow property stated in Sect. 3.7.

6 Implementing Protection Domains

6.1 Using Secure Multiparty Computation

Programs that use protection domains require a runtime that enforces the data separation protection domain separation. Techniques for implementing such a runtime include secure multiparty computation, trusted hardware and virtualization. In our work, we focus on the first of the three. Secure multiparty computation is a cryptographic method for n different parties evaluate a function $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ so that each party $\mathcal{P}_i$ provides the input x_i and learns the output y_i so that no party can learn the inputs or outputs of another party. The first protocols for secure multiparty computation were proposed by Yao [34]. The techniques have been developed since then and several practical implementations of programmable secure computation have been created [21, 5, 15, 11].

Protection domains can be implemented with secure multiparty computation techniques. A suitable technique must provide the following elements:

1. A list of data types used in the protection domain.
2. For each data type in a protection domain we need:
   (a) a *classification* function that converts a public value into a protected value of the same type,
   (b) a *declassification* function that converts a protected value into a public value of the same type,
   (c) functions that perform meaningful operations on protected values that output protected values and

The functions performing secure operations should be universally composable so they could be combined into programs [7].

An example of a suitable technique is *homomorphic encryption*. A homomorphic encryption scheme allows encrypted values to be transformed into new encrypted values that can represent sums or products of the original value. Data values would be classified by encrypting them and

stored as ciphertexts. Declassification would mean the decryption of the values. Operations on protected data would use the transformation functions defined for the homomorphic encryption scheme.

Constructing a useful protection domain from secure multiparty computation is non-trivial, as many such schemes only support a few secure operations. It may be possible to construct more complex secure computation operations by composing simpler ones, but dedicated protocols have been shown to be more efficient in practice. We have identified several protocol suites that are suitable for implementation as protection domain runtimes with some work, including, but not limited to [2, 5, 10].

6.2 Deploying Protection Domains

A secure multiparty computation technique may require several computing parties to be able to provide its security guarantees. This means that a runtime using protection domains may need to be a distributed system. Furthermore, individual protection domains may require a different number of servers, so their concurrent use in a single program requires that the runtime is aware of all protection domains and can coordinate their execution. We will now describe one option for deploying multi-node protection domains.

The first option lets each computing node in the runtime make use of a protection domain. Let m be the maximum number of nodes required by a deployable protection domain. We choose $n \geq m$ as the number of computing nodes that we need to use (and have available). We call these nodes $\mathcal{N}_1, \ldots, \mathcal{N}_n$.

Assume that a protection domain $\mathcal{PD}_k$ requires m_k nodes named $\mathcal{P}_1, \ldots, \mathcal{P}_{m_k}$. We now assign each protection domain node $\mathcal{P}_i$ to a physical node $\mathcal{N}_j$ so that no physical node hosts more than one protection domain node. There will remain physical nodes, where a protection domain node is not assigned. These physical nodes will be "placeholder nodes" that know the locations of the other nodes that are actually involved in the computation. The benefit of this approach is that each physical node is aware of each protection domain so a program containing references to protection domains can be loaded on each physical node unchanged.

Alternatively, if we prefer not to use placeholder nodes, the program may need to be recompiled for each physical node so that it will have to run code with protection domains that are available at that node.

6.3 Implementation

Our compiler translates from the described polymorphic language to the described monomorphic language, performs certain program optimizations on the translated code, and then translates it further to a bytecode that is handled by our distributed SMC runtime. The runtime interfaces with PDK implementations (using system calls) that are also distributed among the nodes executing the runtime. Each PD requires a number of nodes (that we call "roles" by an analogue with cryptographic protocols) to run securely. To deploy an application, we select for each role of each PD in that application a physical node that executes the code for that role in this PD. Currently, we have implementations of two PDKs, based on [6, 29]. The PDs in these two PDKs use two or three parties, respectively.

Our compiler can successfully translate polymorphic code to bytecode that is targeted to both protection domains. This code can successfully be executed on our runtime and tests show that the

execution and results are correct for both domains. The compiler and the runtime are not publicly available at this time, as the implementation is still on a research prototype level.

7 Applications

SecreC has been used in the following applications.

1. In [18], Kamm and Willemson used Sharemind and SecreC to build a satellite collision prediction tool that keeps the trajectories of satellites confidential.
2. In [4], Bogdanov, Laud and Talviste used Sharemind and SecreC to implement oblivious sorting algorithms using secret sharing. The study evaluates the theoretical performance and discusses the practical implications of the different approaches.
3. In [3], the authors present a privacy-preserving statistical analysis toolkit built using SecreC.

8 Conclusion

Efficient combining of different SMC techniques is crucial for obtaining acceptable performance for privacy-preserving outsourced computations. Equipping the programmer with easy-to-use tools to guide the combination is a big step towards this. In this paper, we have presented a language that allows fine-grained orchestration of different techniques with minimum effort by the programmer.

References

[1] Askarov, A., Sabelfeld, A.: Gradual release: Unifying declassification, encryption and key release policies. In: IEEE Symposium on Security and Privacy. pp. 207–221. IEEE Computer Society (2007)

[2] Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In: Simon, J. (ed.) Proceedings of the 20th Annual ACM Symposium on Theory of Computing. STOC'88. pp. 1–10 (1988)

[3] Bogdanov, D., Kamm, L., Laur, S., Pruulmann-Vengerfeldt, P.: Secure multi-party data analysis: end user validation and practical experiments. Cryptology ePrint Archive, Report 2013/826 (2013), `http://eprint.iacr.org/`

[4] Bogdanov, D., Laur, S., Talviste, R.: Oblivious Sorting of Secret-Shared Data. Tech. Rep. T-4-19, Cybernetica, `http://research.cyber.ee/`. (2013)

[5] Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A Framework for Fast Privacy-Preserving Computations. In: Jajodia, S., Lopez, J. (eds.) Proceedings of the 13th European Symposium on Research in Computer Security, ESORICS '08. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer (2008)

[6] Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. International Journal of Information Security 11(6), 403–418 (2012)

[7] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings of the 42nd Annual Symposium on Foundations of Computer Science. FOCS'01. pp. 136–145. IEEE Computer Society (2001)

[8] Damgård, I., Jurik, M.: A Length-Flexible Threshold Cryptosystem with Applications. In: Safavi-Naini, R., Seberry, J. (eds.) Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia, July 9-11, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2727, pp. 350–364. Springer (2003)

[9] Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 2729, pp. 247–264. Springer (2003)

[10] Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) Proceedings of the 32nd Annual Cryptology Conference. CRYPTO'12. Lecture Notes in Computer Science, vol. 7417, pp. 643–662. Springer (2012)

[11] Geisler, M.: Cryptographic Protocols: Theory and Implementation. Ph.D. thesis, Aarhus University (February 2010)

[12] Gennaro, R., Rabin, M.O., Rabin, T.: Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In: PODC. pp. 101–111 (1998)

[13] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) STOC. pp. 169–178. ACM (2009)

[14] Goldreich, O., Micali, S., Wigderson, A.: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In: STOC. pp. 218–229. ACM (1987)

[15] Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS'10. pp. 451–462. ACM (2010)

[16] Ishai, Y., Paskin, A.: Evaluating Branching Programs on Encrypted Data. In: Vadhan, S.P. (ed.) TCC. Lecture Notes in Computer Science, vol. 4392, pp. 575–594. Springer (2007)

[17] Jagomägis, R.: SecreC: a Privacy-Aware Programming Language with Applications in Data Mining. Master's thesis, Institute of Computer Science, University of Tartu (2010)

[18] Kamm, L., Willemson, J.: Secure Floating-Point Arithmetic and Private Satellite Collision Analysis. Cryptology ePrint Archive, Report 2013/850 (2013), `http://eprint.iacr.org/`

[19] Kerschbaum, F., Schneider, T., Schröpfer, A.: Automatic Protocol Selection in Secure Two-Party Computations. In: 20th Network and Distributed System Security Symposium (NDSS) (2013)

[20] Li, P., Zdancewic, S.: Encoding information flow in haskell. In: CSFW. p. 16. IEEE Computer Society (2006)

[21] Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - Secure Two-Party Computation System. In: Proceedings of the 13th USENIX Security Symposium. USENIX'04. pp. 287–302. USENIX (2004)

[22] Mitchell, J.C., Sharma, R., Stefan, D., Zimmerman, J.: Information-flow control for programming on encrypted data. In: Chong, S. (ed.) CSF. pp. 45–60. IEEE (2012)

[23] Myers, A.C.: JFlow: Practical Mostly-Static Information Flow Control. In: Appel, A.W., Aiken, A. (eds.) POPL. pp. 228–241. ACM (1999)

[24] Myers, A.C., Liskov, B.: Complete, safe information flow with decentralized labels. In: IEEE Symposium on Security and Privacy. pp. 186–197. IEEE Computer Society (1998)

[25] Nielsen, J.D., Schwartzbach, M.I.: A domain-specific programming language for secure multiparty computation. In: Hicks, M.W. (ed.) Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security. PLAS'07. pp. 21–30. ACM (2007)

[26] Paillier, P.: Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In: Stern, J. (ed.) Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding. Lecture Notes in Computer Science, vol. 1592, pp. 223–238. Springer (1999)

[27] Pottier, F., Simonet, V.: Information flow inference for ml. In: Launchbury, J., Mitchell, J.C. (eds.) POPL. pp. 319–330. ACM (2002)

[28] Pullonen, P.: Data representations in secure multi-party computations (2012), seminar in cryptography, University of Tartu

[29] Pullonen, P., Bogdanov, D., Schneider, T.: The design and implementation of a two-party protocol suite for sharemind 3. Tech. Rep. T-4-17, Cybernetica AS, Tartu, `http://research.cyber.ee/`. (2012)

[30] Rabin, M.: How to exchange secrets by oblivious transfer. Tech. Rep. TR-81, Aiken Computation Laboratory, Harvard University (1981)

[31] Ristioja, J.: An analysis framework for an imperative privacy-preserving programming language. Master's thesis, Institute of Computer Science, University of Tartu (2010)

[32] Schröpfer, A., Kerschbaum, F., Mueller, G.: L1 - An Intermediate Language for Mixed-Protocol Secure Computation. In: Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference. COMPSAC'11. pp. 298–307. IEEE Computer Society (2011)

[33] Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (1979)

[34] Yao, A.C.C.: Protocols for Secure Computations (Extended Abstract). In: 23rd Annual Symposium on Foundations of Computer Science. FOCS'82. pp. 160–164. IEEE (1982)

$$\frac{}{\gamma \vdash \langle c_t \rangle \Rightarrow \langle c^+ \rangle} \qquad \frac{}{\gamma \vdash \langle x \rangle \Rightarrow \langle \gamma(x) \rangle}$$

Figure 3: Polymorphic language expression evaluation rules

$$\frac{}{\Delta \vdash \langle P \rangle \to \langle \bot, \varepsilon, \mathit{body}(P) \rangle} \qquad \frac{\gamma' = \gamma[x \mapsto d\,\bot]}{\Delta \vdash \langle \gamma, \{x : d\,t\,;\ s\} \rangle \to \langle \gamma', s \rangle}$$

$$\frac{\Delta \vdash \langle C, s \rangle \xrightarrow{\kappa} \langle C', s' \rangle}{\Delta \vdash \langle (\mathcal{S}, \gamma) : C, s \rangle \xrightarrow{\kappa} \langle (\mathcal{S}, \gamma) : C', s' \rangle} \qquad \frac{\Delta \vdash \langle C, s \rangle \xrightarrow{\kappa} C'}{\Delta \vdash \langle (\mathcal{S}, \gamma) : C, s \rangle \xrightarrow{\kappa} (\mathcal{S}, \gamma) : C'}$$

$$\frac{\Delta \vdash \langle \gamma, s_1 \rangle \xrightarrow{\kappa} \gamma'}{\Delta \vdash \langle \gamma, s_1\,; s_2 \rangle \xrightarrow{\kappa} \langle \gamma', s_2 \rangle} \qquad \frac{\Delta \vdash \langle \gamma, s_1 \rangle \xrightarrow{\kappa} \langle \gamma', s' \rangle}{\Delta \vdash \langle \gamma, s_1\,; s_2 \rangle \xrightarrow{\kappa} \langle \gamma', s'\,; s_2 \rangle} \qquad \frac{}{\Delta \vdash \langle \gamma, x \texttt{ = } d\,v \rangle \to \gamma[x \mapsto d\,v]}$$

$$\frac{}{\Delta \vdash \langle \gamma, \texttt{if true}^+ \texttt{ then } s_1 \texttt{ else } s_2 \rangle \to \langle \gamma, s_1 \rangle} \qquad \frac{}{\Delta \vdash \langle \gamma, \texttt{if false}^+ \texttt{ then } s_1 \texttt{ else } s_2 \rangle \to \langle \gamma, s_2 \rangle}$$

$$\frac{}{\Delta \vdash \langle \gamma, \texttt{while } e \texttt{ do } s \rangle \to \langle \gamma, \texttt{if } e \texttt{ then } (s \texttt{ ; while } e \texttt{ do } s) \texttt{ else skip} \rangle}$$

$$\frac{}{\Delta \vdash \langle \gamma, \texttt{skip} \rangle \to \gamma} \qquad \frac{}{\Delta \vdash \langle (\mathcal{S}', \gamma') : \gamma, \mathcal{S}[\texttt{return } d\,v] \rangle \to \langle \gamma', \mathcal{S}'[d\,v] \rangle} \qquad \frac{\gamma \vdash \langle e \rangle \Rightarrow \langle e' \rangle}{\Delta \vdash \langle \gamma, \mathcal{S}[e] \rangle \to \langle \gamma, \mathcal{S}[e'] \rangle}$$

Figure 4: Polymorphic language statement evaluation rules

A Dynamic Semantics of the Polymorphic Language

A.1 Expressions

The transition rules for expressions (Fig. 3) are in form $\gamma \vdash \langle e \rangle \Rightarrow \langle d\,v \rangle$. The expression evaluation rules can only be applied to expressions with fully evaluated subexpressions. The rules are completely standard apart from function calls which are handled by statement evaluation rules.

A.2 Statements

Transition rules for statements (Fig. 4) are either in the form $\Delta \vdash \langle C, s \rangle \xrightarrow{\kappa} \langle C', s' \rangle$ or, if evaluation of the statement halts, in the form $\Delta \vdash \langle C, s \rangle \xrightarrow{\kappa} C'$. Most rules are fairly standard and we only highlight some of them. Throughout this section we denote pairs that consist of protection domain d and environment γ with just γ. Every rule apart from the one for system calls and the one for function calls passes the protection domain along with the environment. We have also omitted the program P from the context of every rule.

If a return statement, with evaluated sub-expression, is reached within arbitrary statement evaluation context. The top-most stack frame is removed, and the value is inserted into the evaluation context of the caller. The current statement evaluation context is discarded entirely.

System calls are somewhat similar to return statement, but have the effect of calling the system function with currently active stack frame. The system call gets entire stack frame as argument which is unpacked to fit the form defined previously. For environment $\gamma = \varepsilon[x_1 \mapsto d_1\,v_1, \ldots, x_n \mapsto d_n\,v_n]$ the call $f(d_0, \gamma)$ is equal to $f(d_0, d_1\,v_1, \ldots, d_n\,v_n)$.

$$\frac{d_0\, t_0 = \mathsf{ret}_P^\theta(\ell) \qquad \gamma \vdash_d \mathcal{S} : d_0 \qquad (\ell, \theta) = \Delta(f, d_0, d_1, \ldots, d_n) \qquad \gamma' = \varepsilon[x_i \mapsto d_i\, v_i \mid x_i : d_i\, t_i \in \mathsf{arg}^\theta(\ell)] \qquad s = \mathsf{body}_P^\theta(\ell)}{\Delta \vdash \langle \gamma, d, \mathcal{S}[f\,\texttt{(}d_1\, v_1\,\texttt{,}\ \ldots\ \texttt{,}d_n\, v_n\texttt{)}]\rangle \to \langle (\mathcal{S}, d, \gamma) : (d_0, \gamma'), s\rangle}$$

$$\frac{d_0\, t_0 = \mathsf{ret}_P^\theta(\ell) \qquad \gamma \vdash_d \mathcal{S} : d_0 \qquad (\ell, \theta) = \Delta(f, d_0, d_1, \ldots, d_n) \qquad \kappa \;\equiv\; v_0 = f(d_0, d_1, v_1, \ldots, d_n, v_n) \qquad \bot = \mathsf{body}_P^\theta(\ell)}{\Delta \vdash \langle \gamma, d, \mathcal{S}[f\,\texttt{(}d_1\, v_1\,\texttt{,}\ \ldots\ \texttt{,}d_n\, v_n\texttt{)}]\rangle \xrightarrow{\kappa} \langle \gamma, d, \mathcal{S}[d_0\, v_0]\rangle}$$

Figure 5: Polymorphic language function and syscall evaluation

$$\frac{\gamma \vdash_d \mathcal{S} : d'}{\gamma \vdash_d \mathcal{S}\ \texttt{;}\ s_2 : d'} \qquad \frac{\gamma(x) = d\, v \qquad \gamma \vdash_d \mathcal{E} : d'}{\gamma \vdash x\ \texttt{=}\ \mathcal{E} : d'} \qquad \frac{\gamma \vdash_d \mathcal{E} : d'}{\gamma \vdash_d \texttt{return}\ \mathcal{E} : d'} \qquad \frac{}{\gamma \vdash_d [] : d}$$

$$\frac{\gamma \vdash_{\texttt{public}} \mathcal{E} : d'}{\gamma \vdash \texttt{if}\ \mathcal{E}\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : d'} \qquad \frac{\gamma \vdash \mathcal{E} : d'}{\gamma \vdash f\texttt{(}d_1\, v_1\,\texttt{,}\ \ldots\ \texttt{,}\mathcal{E}\texttt{,}e_{i+1}, \ldots\ \texttt{,}e_n\texttt{)} : d'}$$

Figure 6: Domain of the evaluation context

A.3 Function Calls

Function calls – with fully evaluated parameters – are handled by statement evaluation rules. If such function call is reached within some statement evaluation context $\mathcal{S}$ the statement evaluation context will be pushed onto the return stack, the rule transitions into evaluating the function body, and new stack frame is constructed. The new environment within the stack frame maps the formal parameters to the values of the actual parameters.

The appropriate protection domain of the return type is selected dynamically using judgments in the form $\gamma \vdash_d \mathcal{S} : d'$ (see Fig. 6), where d is the current domain context. The judgment will match with every possible security domain d' that the evaluation context can reside in. The rule for function call could lead to ambiguous behaviour but all function calls of an unambiguously-typed program can be evaluated unambiguously.

B Monomorphic Language Semantics

See Fig. 8 for the transitions of the statements of the monomorphic language. The evaluation of expressions is defined in Fig. 7.

$$\frac{}{\gamma \vdash \langle x\rangle \Rightarrow \langle \gamma(x)\rangle} \qquad \frac{}{\gamma \vdash \langle c_t\rangle \Rightarrow \langle c\rangle}$$

Figure 7: Monomorphic language expression evaluation rules

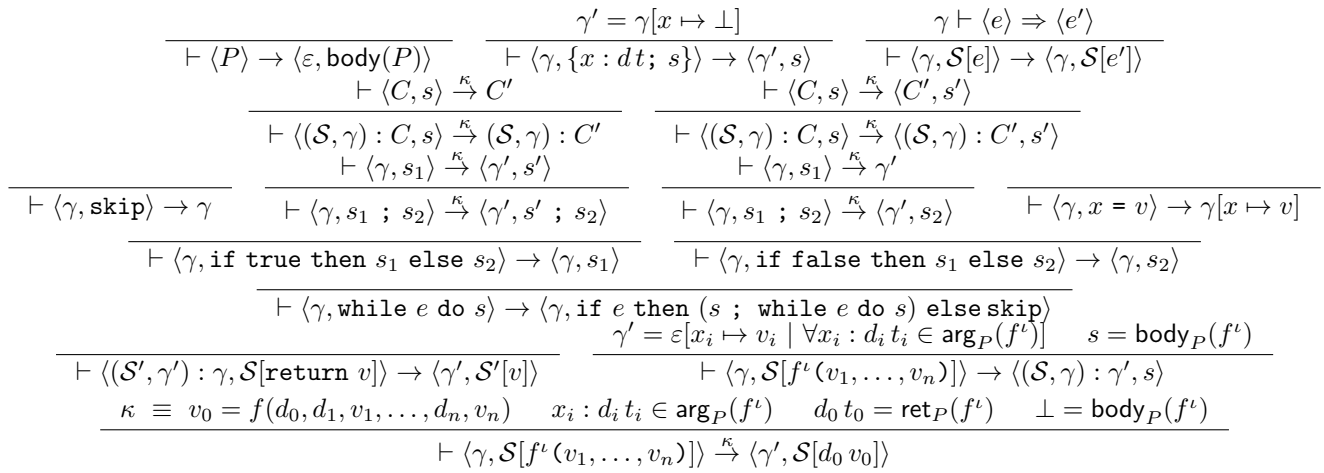

Figure 8: Monomorphic language statement evaluation rules

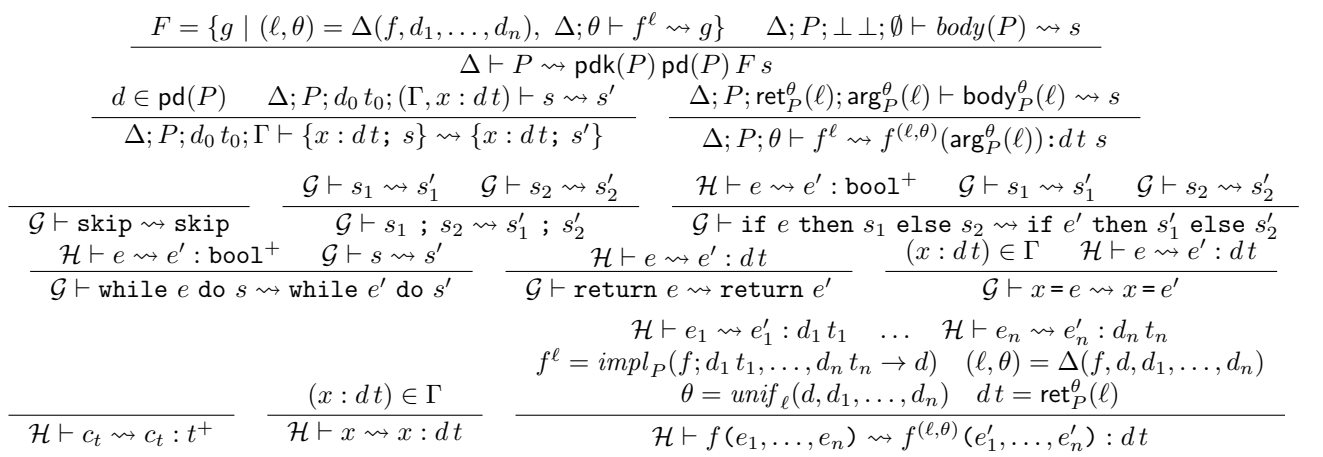

Figure 9: Translation methods

C Translation Rules

See Fig. 9 for translation rules from polymorphic language to monomorphic language.

D Proof of Information Flow Security

The initial state $w \in \mathcal{W}$ of the environment of the program P is distributed according to $\boldsymbol{\mathcal{W}}$. The adversary is assumed to know the low-part w_L of the initial state. During the execution of P, the adversary is also assumed to see the low-slice of the label of each occurring transition.

Besides the low-slices of transition labels and traces (according to the partitioning $\mathbf{PD}_L \dot{\cup} \mathbf{PD}_H$ of the set of protection domains $\mathbf{PD}$), we can also speak about low-slices of program configurations. Recall that a program configuration is a pair $\langle C, s\rangle$ or an item C, where s is a statement (Sect. 3.1) and C is a sequence $(\mathcal{S}_1, d_1, \gamma_1) : \cdots : (\mathcal{S}_n, d_n, \gamma_n) : (d_{n+1}, \gamma_{n+1})$, where $n \geq 0$, $\mathcal{S}_i$ is a statement

evaluation context (Fig. 2), d_i is the protection domain of the value to be returned from this frame and γ_i is a mapping from variables to values annotated with protection domains. All values appearing in s and $\mathcal{S}_i$ are annotated with their protection domains as well. To form the low-slice $\overline{\langle C, s\rangle}$ or $\overline{C}$, we replace all annotated values $d\,v$ in s and $\mathcal{S}_i$ with the placeholders $d\,*$, if $d \in \mathbf{PD}_H$. We have, that the low-slice of the execution of the program P can be recovered from the low-slices of the labels of the transitions taken during the execution.

Lemma D.1 *Let P be a well-typed program and $C_1^\circ \xrightarrow{\kappa_1} C_1^\bullet$ and $C_2^\circ \xrightarrow{\kappa_2} C_2^\bullet$ two of possible steps that it may make. If $\overline{C_1^\circ} = \overline{C_2^\circ}$ and $\overline{\kappa_1} = \overline{\kappa_2}$, then $\overline{C_1^\bullet} = \overline{C_2^\bullet}$.*

Proof The low-slice of a context contains full information about the control flow of the program, hence C_1° and C_2° both make the same step. If it is an assignment of a value to a variable, and the assigned value has a low protection domain, then the value must be the same in C_1° and C_2°, hence the variable will have the same value in both $C_1^\bullet$ and $C_2^\bullet$. If the assigned value has a high protection domain, then it is replaced with the placeholder $*$ in both $\overline{C_1^\circ}$ and $\overline{C_2^\circ}$, as well as in $\overline{C_1^\bullet}$ and $\overline{C_2^\bullet}$. If the step is corresponds to a choice in some `if-` or `while`-statement then the value branched on must be public, it will not be replaced with $*$ in $\overline{C_1^\circ}$ and $\overline{C_2^\circ}$, and hence both C_1° and C_2° branch in the same direction. If C_1° and C_2° make a function call or return a value, then the passed values are either equal (if they have low protection domains) or replaced with the placeholder in the low-slice (if they have high protection domains). A low-security value returned from a system call can be found from the low-slice of the labels $\overline{\kappa_1}$ and $\overline{\kappa_2}$. ∎

For each possible initial environment $w \in \mathcal{W}$, the semantics $[\![f]\!]$ of the system calls f and the program P uniquely determine the trace (an element of $[\![P]\!]$) of the execution. Denote this trace by $\mathrm{Tr}^{[\![\cdot]\!]}(w, P)$. Let $\overline{\mathcal{T}}$ be the set of all low-slices of transition labels. For each $\overline{T} \in \overline{\mathcal{T}}^*$ and each $w_L \in \mathcal{W}_L$, let $\mathcal{I}_P^{\overline{T},w_L}$ be the set of possible initial environments, the low-part of which is w_L and the execution from which starts with transitions, the low-slices of which make up $\overline{T}$. Formally,

$$\mathcal{I}_P^{\overline{T},w_L} = \{w' \mid w'_L = w_L \wedge \overline{T} \sqsubseteq \overline{\mathrm{Tr}^{[\![\cdot]\!]}(w', P)}\},$$

where $\sqsubseteq$ denotes that a sequence is a prefix of the other one. The knowledge of an adversary observing the execution of P is characterized by a probability distribution $\mathcal{KN}_P^{\overline{T},w_L}$ obtained by projecting $\boldsymbol{\mathcal{W}}$ to $\mathcal{I}_P^{\overline{T},w_L}$. To project a probability distribution $D \in \mathcal{D}(X)$ to a set $Y \subseteq X$ means setting the probabilities of all $x \in X \backslash Y$ to 0 and rescaling the probabilities of all $x \in X$, such that their sum is still equal to 1. After observing the prefix of the trace $\overline{T}$ and knowing that the low-part of the initial environment was equal to w_L, the adversary knows that the initial environment had to belong to the set $\mathcal{I}_P^{\overline{T},w_L}$. The relative probabilities of the elements in this set are the same as their relative probabilities in $\boldsymbol{\mathcal{W}}$.

The set of possible initial states is not changed by observing the low-slice of one more transition, unless it is a declassification transition. Namely,

Lemma D.2 *Let $w_L \in \mathcal{W}_l$, $\overline{T} \in \overline{\mathcal{T}}^*$ and $\kappa \in \mathcal{T}$, such that κ is not a declassification label. Then either $\mathcal{I}_P^{\overline{T};\overline{\kappa},w_L} = \mathcal{I}_P^{\overline{T},w_L}$ or $\mathcal{I}_P^{\overline{T};\overline{\kappa},w_L} = \emptyset$.*

Proof We first note that the low-slice of the program context and the low-part of the environment are the same for all $w' \in \mathcal{I}_P^{\overline{T},w_L}$ after the execution that produces the trace with the low-slice

$\overline{\kappa}$. Indeed, they do not depend on w' at the beginning of the execution (the low-part of the environment is w_L and the low-slice of the context is empty) and, according to Lemma D.1, the contexts stay equal after each step of the program. Similarly, the low-part of the environment in different executions evolves in the same way (due to the constraints placed on the semantics of the system calls, its changes can be deduced from the low-slice of the execution trace).

We now note that the low-slice of the current program context, as well as the low-part of the current environment determine the low-slice of the label of the next transition, unless this label is a declassification label. Indeed, the current program point is uniquely determined, hence it is determined whether the next transition is silent or a system call $f(d_0, d_1\, v_1, \ldots, d_n\, v_n)$. If it is a system call then the domains $d_0, \ldots, d_n$ are determined by the low-slice of the program context. In the low-slice of the label κ' of the next transition, the values $v_1, \ldots, v_n$ are included if the corresponding domains $d_1, \ldots, d_n$ are in $\mathbf{PD}_L$. In this case, these values can be found from the low-slice of the program context. If some d_i is in $\mathbf{PD}_H$, then only a placeholder is included in $\overline{\kappa'}$ instead of the value v_i. The value v_0 returned by the system call is included in $\overline{\kappa'}$ only if $d_0 \in \mathbf{PD}_L$. In this case, v_0 can be computed from the values v_i ($i \in \{1, \ldots, n\}$) included in $\overline{\kappa'}$ and the low-part of the current environment. The low-part of the environment after this step can also be computed from the same values and the low-part of the current environment.

As $\overline{\kappa'}$ is uniquely determined, we have $\mathcal{I}_P^{\overline{T};\overline{\kappa},w_L} = \mathcal{I}_P^{\overline{T},w_L}$ if $\overline{\kappa} = \overline{\kappa'}$, and $\mathcal{I}_P^{\overline{T};\overline{\kappa},w_L} = \emptyset$ otherwise. ∎

Considering the definition of $\mathcal{KN}_P^{\overline{T},w_L}$, the previous lemma immediately implies

Proposition D.3 *Let $w_L \in \mathcal{W}_l$, $\overline{T} \in \overline{\mathcal{T}}^*$ and $\kappa \in \mathcal{T}$, such that κ is not a declassification label. Then $\mathcal{KN}_P^{\overline{T};\overline{\kappa},w_L} = \mathcal{KN}_P^{\overline{T},w_L}$.*