PARTE: Automatic Program Partitioning for Efficient Computation over Encrypted Data

by

Meelap Shah

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2013

Author .... [illegible]
Department of Electrical Engineering and Computer Science
February 1, 2013

Certified by .... [illegible]
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by .... [illegible]
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

PARTE: Automatic Program Partitioning for Efficient Computation over Encrypted Data

by

Meelap Shah

Submitted to the Department of Electrical Engineering and Computer Science on February 1, 2013, in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science

Abstract

Many modern applications outsource their data storage and computation needs to third parties. Although this lifts many infrastructure burdens from the application developer, he must deal with an increased risk of data leakage (i.e. there are more distributed copies of the data, the third party may be insecure and/or untrustworthy). Oftentimes, the most practical option is to tolerate this risk. This is far from ideal and in case of highly sensitive data (e.g. medical records, location history) it is unacceptable. We present PARTE, a tool to aid application developers in lowering the risk of data leakage. PARTE statically analyzes a program's source, annotated to indicate types which will hold sensitive data (i.e. data that should not be leaked), and outputs a partitioned version of the source. One partition will operate only on encrypted copies of sensitive data to lower the risk of data leakage and can safely be run by a third party or otherwise untrusted environment. The second partition must have plaintext access to sensitive data and therefore should be run in a trusted environment. Program execution will flow between the partitions, levaraging third party resources when data leakage risk is low. Further, we identify operations which, if efficiently supported by some encryption scheme, would improve the performance of partitioned execution. To demonstrate the feasiblity of these ideas, we implement PARTE in Haskell and run it on a web application, hpaste, which allows users to upload and share text snippets. The partitioned hpaste services web request $1.2 - 2.5\times$ slower than the original hpaste. We find this overhead to be moderately high. Moreover, the partitioning does not allow much code to run on encrypted data. We discuss why we feel our techniques did not produce an attractive partitioning and offer insight on new research directions which could yield better results.

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

Many people have helped and motivated me through this project.

First my advisor, Professor Nickolai Zeldovich, whose patience and willingness to help seem inexhaustible. Ramesh has been a great friend and inspiration. Casual conversations with him often result in interesting ideas; this project is just one example. Austin is an endless source of ideas and information. He is very naturally able to explain arcane details of complex systems in fantastically simple ways. Raluca has been a solid source of support and snacks to encourage me to complete this project. Emily wrote the first lines of Haskell to concretize this project and get it rolling. Professor Adam Chlipala saved me countless hours of work by introducing me to some simple techniques in program analysis that heavily shaped this project. Elette was an amazing source of energy and drive. Thank you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many modern applications collect and use sensitive user data. For example, search engines might track browsing history to improve the quality of search results; smartphone applications might track location history to present timely traffic information; social network sites might collect personal information to help users stay up to date with their friends.

These applications need to maintain their own infrastructure or leverage third parties to meet their data storage and processing needs (e.g. Yelp [5] and foursquare [1] use Amazon's S3 and Elastic MapReduce services). Unfortunately, this means that there are more entrypoints through which an attacker can gain illicit access to user data. As a result, end users must trust their sensitive data with the application as well as third party servers.

Ideally, a user would be able to benefit from the services provided by an application while maintaining the confidentiality of his sensitive data. To achieve this, the user should need to trust only components that are under his control (such as his own computer). This requires eliminating the application and other third party servers from the trusted computing base (TCB).

One way in which the user can reduce the TCB is by storing all his sensitive data locally and never disclosing it to remote servers. In order to use an application, the user would have to receive the application code and run it locally over his data. This is very similar to an installed desktop application and is unsuitable in many scenarios: the user loses mobility since he can use the application only from a computer that has a copy of his data; existing applications would need to be rewritten to work in the user's environment; the user may have limited resources (CPU, storage, power). Alternatively, a trusted third party could receive data from the user and code from the application server, run the code over the data and return the result. However, this model offers little protection for users' data if the trusted third party is compromised by an attacker.

Another option is to use a fully homomorphic encryption (FHE) scheme [13]. An FHE scheme allows arbitrary computations over enrypted data. In this model, the user would encrypt all sensitive data under an FHE scheme locally before sending it to the application. The application code would need to be modified to work over encrypted data (for example, $+$ would be replaced with some other operation that can add two ciphertexts). The user would never reveal his data in the clear to the applcation or any third party in this model, protecting him from both malicious applications as well as compromises of remote servers. However, FHE schemes today incur too high of a space and time overhead to be usable for most applications [14].

The paper presents the design of PARTE, a tool to aid in modifying an existing application to work efficiently over encrypted data. PARTE builds on the idea proposed in CryptDB [16] of using various encryption schemes to enable executing code over encrypted data. CryptDB used various encryption schemes to build a SQL database in which queries could be executed over encrypted data–the DBMS never sees plaintext data; PARTE applies this idea to general applications. Rather than using FHE which allows any computation but is slow, PARTE will make use of a variety of encryption schemes that allow limited computation but are efficient so that the resulting application remains efficient enough to be practical.

A random (RND) encryption scheme (such as AES in CBC mode with a random initialization vector) does not allow any computation to be performed efficiently over ciphertexts. A deterministic (DET) encryption scheme (such as AES in a variant of CMC mode, details in [16]) allows two ciphertexts to be efficiently compared for equality. An order preserving (OPE) encryption scheme (such as [6]) allows comparisons ($<$, $>$) to be computed between two ciphertexts. A homomorphic (HOM) encryption scheme (such as Paillier [15]) allows addition of two ciphertexts or multiplication of a ciphertext with a plaintext. Various schemes exist to allow searching for a keyword in a body of text (such as [18]). Other specialized encryption schemes can potentially be developed to support other operations performed by an application.

These schemes are not all equally secure. For example, a RND ciphertext leaks no bits about its corresponding plaintext wheres an OPE ciphertext leaks half of the bits of the plaintext. It is important to choose the most secure encryption scheme that supports the needed operations to minimize data leakage.

With so many encryption schemes supporting different sets of operations, it can be tricky for the application developer to know how to apply them to his application. PARTE can aid the developer by analyzing the application and determining which scheme (if any) can be used and where. At a high level, this will work as follows. The developer first annotates data structures in the application code that will hold sensitive user data. PARTE will find all functions that use these data structures. For each such function, PARTE will determine what operations it applies to sensitive data and which encryption schemes (if any) support these operations. If at least one such encryption scheme exists, PARTE will output a modified version of the function that works on encrypted data structures as well as a wrapper function to encrypt data before invoking this modified function. If no encryption scheme exists, the function will need access to plaintext data. This partitions the application's functions into two groups: one group can operate on data encrypted under some scheme; the other must operate on plaintext. PARTE can partition the execution of the application (by inserting RPCs) so that functions that work on encrypted data run on the application/third party's servers and the remaining functions run in some trusted environment such as the user's local machine.

One point to note is that we do not want to depend on any changes to the runtime or the application's environment. We restrict PARTE to only change application source code. The reason for this is that since part of the application will run in an untrusted environment, we do not want to make any assumptions about that environment.

A second point to note is that PARTE must run statically. PARTE must know the code path particular inputs will follow ahead of time so it can encrypt those inputs with appropriate schemes.

For many applications, we believe that this partitioned execution is better than simply running the entire application on the user's machine. The application can still leverage the computing resources of the application/third party servers. In many cases, data can be stored encrypted on the application server and need not be transferred to the client in order to run the application. Allowing the application servers to store data grants the user mobility since he no longer needs to sync his data across all computers from which he might use the application. Moreover, different users can share data on the application server.

The high level challenges in making partitioned execution work are:

1. Identifying and distinguishing sensitive data from non-sensitive data in the application.

2. Identifying and separating application code that can operate on encrypted data from the code that must operate on plaintext.

3. Insert code to allow program execution to flow between the two partitions of the application code, encrypting data as necessary.

One challenge not addressed is how partitioned execution would work in multi-principal applications. If users encrypt their data with different keys, then it is not obvious how an application can compute on data from different users. One user may need to encrypt his data with another's key, but this might require that users trust each other. Or encryption schemes may need to support converting ciphertexts from one key to another, or operations on ciphertexts encrypted under different keys. It is unclear which approaches will work; multi-principal applications raises several important challenges which are left to future work.

We built a prototype of PARTE in Haskell, levaraging several features of the language to address the challenges listed above. We approximate identifying sensitive data by identifying sensitive data structures–structures that may hold sensitive data at runtime. We identify code that can operate on encrypted data by pairing encryption schemes with Haskell *type classes*. These techniques are described in detail in Chapter 3.

We run our prototype on several applications to demonstrate its feasiblity. We use a web appplication, hpaste [3], and a log analysis tool that we wrote. The results show that although the partitioned application works, the overhead is moderately high and the majority of the application needs to run on plaintext. We discuss why our results leave much to be desired in terms of performance, but also indicate where computations not supported efficiently by some encryption scheme occur in the application.

The rest of this thesis is organized as follows. Chapter 2 discusses related work addressing computation over encrypted data, program partitioning, and relevant program analysis techniques. Chapter 3 discusses the design of PARTE in detail and in particular how each of the challenges listed above is solved. Chapter 4 discusses a few implementation details of PARTE. Chapter 5 presents some results of running our prototype of PARTE on a few applications. Chapter 6 discusses why those results were observed and offers insight as to how they can be improved as well as directions for future work. Finally, Chapter 7 concludes.

Chapter 2

Related Work

Many systems have been built to address the issue of user data confidentiality. We break the related work down into three categories. Section 2.1 discusses work that encrypts sensitive data to ensure that it is not leaked. Section 2.2 discusses work that partitions applications so that senstive data and the code that operates on it remain in a trusted environment. Section 2.3 discusses work that uses techniques from program analysis to ensure that data is not revealed to unintended recipients.

2.1 Computation Over Encrypted Data

CryptDB [16] is a SQL database that only stores encrypted copies of data. It makes use of a variety of encryption schemes that each efficiently supports a small set of operations to execute many common types of SQL queries over encrypted data. Some queries need to perform operations that no crypto-scheme efficiently supports (e.g. matching text against a regular expression). In these cases, CryptDB transfers data to a trusted environment in which the data can be safely decrypted and computed upon.

A common pattern among applications that do use a SQL datastore is to perform several sequential, dependent database reads. In these cases, the application usually does some filtering (e.g. sort the results of one query and restrict the next query to just the top *n* items) or transformations (e.g. convert data to lower case) in between these database reads. While most logic of this type could be expressed in the SQL query, this is often not done in practice because complex SQL queries are difficult to understand and write. PARTE could enable application logic, such as code that runs between database queries, to run in the same, untrusted environment as the database. If the application and datastore share locality (which is often the case), this could significantly reduce the amount of data transferred between an application and its users.

Monomi [19] is a system that builds on CryptDB's approach by introducing split client/server query execution. For queries that involve computation not efficiently supported by some encryption scheme, CryptDB would incur a high overhead. Monomi alleviates this by splitting the execution of the query so that some of it executes on the database server over encrypted data and the rest of the query finishes executing on the client. However, Monomi assumes a certain type of workload and includes many optimizations to support this specific workload. One way in which our work is different is that we target general purpose programs so we cannot make any assumptions about the types of computation we will need to support.

Silverline [17] presents a system that aims to allow arbitrary programs to compute on sensitive data while maintaining confidentiality in untrusted environments. One limitation of Silverline is that it only considers programs that can work with data encrypted under a randomized scheme which fundamentally limits the types of programs Silverline can successfully work with. Moreover, Silverline's approach is to determine what data can be encrypted and encrypt just that; this may still expose sensitive data in clear while unnecessarily incurring overhead by encrypting non-sensitive data. PARTE's approach is to force sensitive data to remain in a trusted environment and allow it to leave only if it can first be encrypted; all non-sensitive data remains unencrypted and requires no extra processing.

Sporc [11] enables multiple users to collaboratively use an application hosted by an untrusted server. This system never exposes sensitive user data in plaintext to untrusted servers, thereby maintaining confidentiality. However, in Sporc the untrusted servers are only used to order, store, and broadcast client-generated operations. PARTE instead pushes as much of the application logic onto the untrusted server as possible to leverage its resources. Moreover, applications need to be specifically designed and written to take advantage of the Sporc framework whereas PARTE can work on a general class of applications.

Frientegrity [12] is a system similar to Sporc – it enables many users to form a social network in which data is encrypted and stored in an untrusted central server. This system is designed for a specific application which is reflected in the types of computation the untrusted server is allowed to do. In contrast, our work aims to support arbitrary programs and tries to make as much use of the untrusted server's resources as possible.

Some applications implement custom techniques to preserve data confidentiality. For example, `http://0bin.net` and `http://defuse.ca/pastebin.htm` are web applications that allow users to upload and share text snippets, but first encrypt these snippets on the client-side. Any client that wishes to view a stored snippet must have received the decryption key out of band. This approach does not generalize and requires much manual effort.

2.2 Program Partitioning

Jif/split [23] and Swift [10] are systems that partition application code to ensure data confidentiality. Sensitive data and application code that operates on it are pinned to one partition. Because these systems do not make use of encryption or other tools, sensitive data cannot leave the trusted environment without being leaked. These systems also require the developer to write the application in a Java-like language and specify information flow policies. They use both static and runtime techniques to ensure that these policies hold. PARTE also partitions applications, but differes in that it leverages cryptography to allow sensitive data to move between partitions while minimizing data leakage, and works with general purpose existing applications.

As mentioned in Section 2.1, Monomi also makes use of program partitioning. However, Monomi only partitions SQL queries whereas we support partitioning arbitrary applications.

Program partitioning has been used to solve a number of other problems. For example, Pyxis [7] improves the performance of database applications by partitioning to achieve better locality and reduce data transfer. Our goal of reducing the risk of data leakage is quite different and we take a different approach towards partitioning. However, it would be interesting to see if their static analysis techniques could be reworked to help realize our goal.

2.3 Program Analysis Techniques

Jeeves [21] is a programming language designed to give the developer control over how data is disclosed. The developer specifies a privacy policy that defines how data should be output in a given environment. This policy is enforced by the language runtime. Our work differs in that we maintain data confidentiality not just when data is being output but throughout the execution of the program. This allows us to run certain parts of the program in untrusted environments while preserving confidentiality.

Ur/Web [9] and UrFlow [8] provide a system that statically checks security policies for database backed web applications. The policies can control who is allowed to learn what information. This allows the system to prevent leaking data to untrusted users. Our work differs in that we trust the user but do not trust the application server.

Resin [22] is a language runtime that tracks data flow allows application developers to specify assertions that must check before the data is allowed to move across flow boundaries. This system could be used to track sensitive data and prevent it from flowing into untrusted environments. However, one way in which our work differs in that one of our goals is to enable safely transmitting sensitive data into untrusted environments to take advantage of that environments computational resources.

Chapter 3

System Design

In this chapter, we describe the design of PARTE.

3.1 Overview

The only input PARTE takes is the application's unmodified source code and annotations identifying data structures that will hold sensitive data. It outputs modified source code that enables parts of the application to execute over encrypted data. To achieve this, we must address the following problems.

1. Sensitive data must be differentiated from non-sensitive data throughout the application. Since our tool must work statically, we cannot rely on the value of data. Moreover, our tool has no sense as to what data should be considered sensitive. As a result, we approximate identifying sensitive data by having the application developer identify *sensitive data structures*. We assume that sensitive data is only ever stored in one of these data structures.

2. We must identify blocks of application code that operate on sensitive data structures. Functions serve as a natural way to break a well written application into logical chunks. We look at a function's *type signature* to determine if it operates on a sensitive data structure.

3. Each function in the application that operates on sensitive data structures must be analyzed to determine whether it can work on encrypted versions of that data. We use *type classes* to determine the operations the function performs.

   A type class defines a set of functions that a type can implement to support certain computation. For example, an `Int` type can implement an `Eq` type class to support $==$ and $\neq$. See [20] for more details about type classes.

   We match operations provided by type classes with operations efficiently supported by some encryption scheme. If there is a match, then we know that the function can be modified to work on encrypted data. If there is no match, we must run the function on plaintext.

4. The application must be partitioned into two pieces. The first piece should contain only function that can work over encrypted data and can therefore be safely run in an untrusted environment. The remaining functions need to run on plaintext in a trusted environment. We use a *data flow graph* in conjunction with the analyses from the previous challenges to determine just how to partition the functions.

We chose to implement a prototype of PARTE in Haskell because its static, implicit type system facilitates the analyses required to solve the challenges just described. Nevertheless, the ideas we describe here can be implemented in any other suitable language.

We use the code in Figure 3-1 as a running example throughout the following sections to illustrate how PARTE works. Suppose a company represents information about each employee in the `Employee` abstract data type (ADT). The application is run on a central server, but each employee's salary is considered sensitive information so we would like to avoid revealing it to the server in case it is compromised by an outside attacker. The following sections describe how PARTE can help us achieve this goal. Specifically, Section 3.2 describes how to differentiate sensitive from non-sensitive data; Section 3.3 describes how to find suitable encryption schemes for each function; Section 3.4 describes how to identify functions that operate on sensitive data; finally, Section 3.5 describes how to partition the application.

```haskell
data Employee = Employee {
    name :: String,
    salary :: Int,
    ...
}

highestPaid :: [Employee] -> (String,Int)
highestPaid employees = head . sort2 $
    [(name e, salary e) | e <- employees]

sort2 :: [(String,Int)] -> [(String,Int)]
sort2 s = case s of
            [] -> []
            (a:b) -> let p = snd a
                         l = filter ((<=p) . snd) b
                         h = filter ((> p) . snd) b
                     in (sort2 h)++[a]++(sort2 l)

averageSalary :: [Employee] -> Int
averageSalary e = average (map salary e)

average :: [Int] -> Int
average s = quot (sum s) (length s)
```

Figure 3-1: Haskell code representing an application to compute some simple statistics about employee salaries within a company.

3.2 Differentiating Sensitive From Non-Sensitive Data

The application developer needs to specify which data structures will hold sensitive data. In the example application, he would specify the tuple (“`Employee`”,“`salary`”). This tuple identifies the ADT and field which will represent a salary. It has type `Int`. To distinguish this from other data of type `Int` , we create a new type `SensitiveInt` that is simply a wrapper around an `Int` (Figure 3-2).

```
data SensitiveInt = MkSensitiveInt {
    unSensitiveInt :: Int
}

data Employee = Employee {
    name :: String,
    salary :: SensitiveInt,
    ...
}
```

Figure 3-2: PARTE defines `SensitiveInt` as a wrapper around an `Int`.

Certain data structures may at times hold sensitive data and at other times hold non-sensitive data. For example, an application may define a generic `Buffer` data type that is sometimes used to hold sensitive data but also used to hold non-sensitive data. We do not distinguish between these cases and treat all data of type `Buffer` as sensitive.

After wrapping the sensitive data in a unique type, the types of other functions in the application may need to change. For example, some functions that originally took an `Int` as input should now expect a `SensitiveInt`. Figure 3-3 shows what our example application looks like after making all such changes. We identify all functions that need to be changed as functions that operate on sensitive data.

More concretely, consider the call to `average` made from `averageSalary`. After changing the type of the salary field, `averageSalary` will now pass a list of `SensitiveInts` to `average`. This is a type mismatch since `average` expects a list of `Ints`. From this type mismatch we can conclude that `averageSalary` and `average` operate on sensitive data.

Polymorphic functions will not produce a type mismatch like this. For example, suppose `average` had type signature **`Fractional`** `a =>[a] -> a`. We need some other way to determine that this `average` operates on sensitive data. When we wrapped `Int` to create `SensitiveInt`, we did not make it an instance of any type class. As a result, the call to polymorphic average will not typecheck since there is no instance of `Fractional` for type `SensitiveInt`. Errors such as this allow us to determine whether polymorphic functions ever operate on sensitive data.

Sensitive data may sometimes be passed to a function through a higher order function. For example, suppose somewhere in our application we have the statement `(isWealthy . salary)(e :: Employee)` where `isWealthy` has type signature **`Int`** `->` **`Bool`**. In this case, the function composition `(.)` causes a type error. It has type `(.):: (b->c )->(a->b)->a->c`. When the compiler type checks this call, `salary` causes `b` to unify

```haskell
highestPaid :: [Employee] -> (String,SensitiveInt)
highestPaid employees = head . sort2 $
    [(name e, salary e) | e <- employees]

sort2 :: [(String,SensitiveInt)] -> [(String,SensitiveInt)]
sort2 s = case s of
            [] -> []
            (a:b) -> let p = snd a
                         l = filter ((<=p) . snd) b
                         h = filter ((> p) . snd) b
                     in (sort2 h)++[a]++(sort2 l)

averageSalary :: [Employee] -> Int
averageSalary e = average (map salary e)

average :: [SensitiveInt] -> Int
average s = quot (sum s) (length s)
```

Figure 3-3: Type signatures are modified to work with wrapped types.

with `SensitiveInt` but `isWealthy` causes it to unify with `Int`. This tells us that a function passed to `(.)` operates on sensitive data. To determine what this function is, we need to fix the type of `(.)` so that it typechecks and that its argument that operates on sensitive data does not typecheck. We take the error caused by `(.)` and use it to add an explicit type signature so that it checks. In this case, we get `(.):: (SensitiveInt ->c)-(a->SensitiveInt)->a->c`. Now `isWealthy` causes a type mismatch since it expected an `Int` but got a `SensitiveInt` and we can conclude that `isWealthy` operates on sensitive data. Adding explicit type signatures on higher order functions give us visibility into their functional arguments.

3.3 Identifying Suitable Encryption Schemes

Section 3.2 showed how to determine which functions operate on sensitive data. For each such function, we must determine if it can be modified to operate on encrypted data. To do this, we first determine the set of operations the function performs on the sensitive data and then match this set against operations supported by a variety of encryption schemes.

We determine the set of operations a function performs by representing classes of computation with Haskell type classes. For example, equality ($==$ and $/=$) operations are represented by the builtin `Eq` type class and comparison ($<$, $\leq$, $\geq$, $>$) operations are represented by the builtin `Ord` type class. We can represent addition by defining an `Addable` type class and keyword search over text by defining a `Searchable` type class.

Each of these type classes maps directly onto the operations efficiently supported by some encryption scheme. `Eq` maps to a deterministic scheme, `Ord` maps to an order preserving scheme [6], `Addable` maps to a homomorphic scheme [15], and `Searchable` maps to a text search scheme [18].

Application developers can easily take advantage of their own encryption schemes. They simply need to define and use a type class capturing the operations their schemes support, and provide PARTE with functions to encrypt and decrypt data using those schemes.

Haskell's type inference system allows us to automatically determine type class constraints on the arguments to a function. If a function has no explicit type signature, then Haskell will infer the most general signature that works. We can then unify type variables in the inferred signature with the concrete types in the explicit signature to determine which type class constraints apply to the sensitive inputs to the function.

Consider `sort2` from Figure 3-1. If we remove the explicit type declaration, its inferred signature will be `sort2 :: Ord b =>[(a,b)] -> [(a,b)]`. By matching the type variable `b` in the inferred signature with the `SensitiveInt` from the original, we can determine that `sort2` performs operations in `Ord` on `SensitiveInt`. `Ord` maps to an OPE scheme, so we can conclude that `sort2` can be modified to work on OPE encrypted sensitive data.

If there are multiple type class constraints on a sensitive data type, we intersect the set of schemes that suport each class. If there is such a scheme. we can modify the function to operate on sensitive data encrypted with that scheme. If there is no such scheme, the function must run on plaintext. We cannot simply send multiple copies of the sensitive data encrypted under different schemes that support each type class constraint because we do not know how the function uses these different operations. We would need to look inside the function to determine when to use each encryption, but our analysis does not see deeper than the function level. However, we can inform the application developer that if he can break the function into smaller functions each with fewer type class restrictions, they may be modifiable to work on encrypted data. Alternatively, if we used a different analysis that worked at a finer grain than the function level, then we might be able to automatically break a function into smaller pieces or use multiple schemes to encrypt sensitive data.

If there are no type class constraints on a sensitive data type, then we assume that the function does not poke at the data but treats it like a blackbox (i.e. performs no operations dependent on the value of the sensitive data). An example is taking the length of a list - this operation needs to know nothing about the type of elements in the list. As a result, we conclude that functions like this can be modified to work with data encrypted under any scheme.

Finally, there may be a type class constraint on a sensitive data type that does not map to any encryption scheme. In this case, we assume that the operations represented by the type class require access to plaintext data. For example, the developer may define a `RegexSearchable` type class to do regular expression matching against a type. We know of no efficient encryption scheme that would allow full regular expression search, so we conclude that the function cannot be modified to operate on encrypted data.

3.4 Determining Sensitive Dataflow

We saw how to determine suitable encryption schemes for individual functions in Section 3.3. Now we determine how sensitive data is passed between functions so that we know where to insert RPC and encryption functions. For example, if foo operates on plaintext but invokes bar which operates on OPE encrypted data, we must insert code to encrypt the data before bar is invoked.

To determine how sensitive data is passed between functions, we must first find all functions that operate on sensitive data. Section 3.2 showed how to find functions that directly take a sensitive data type as input. However, the application may define other ADTs which include sensitive data. In our running example, Employee includes a SensitiveInt. We can imagine defining another ADT Division containing a field of type [Employee] which contains SensitiveInts. We scan the entire program source to find all ADTs which contain a field of a sensitive type.

Now that we know all ADTs that contain sensitive data, we use phantom types to determine how they are passed between functions. First, a phantom parameter is added to the definition of every ADT containing sensitive information. In our example, we add the parameter p to ADT definitions. Next, a unique type is used to parametrize each occurrence of each ADT. The final result would look like Figure 3-4.

Now we look for type mismatch errors. For example, highestPaid passes a list of (String, SensitiveInt SensitiveInt1) to sort2 which expects a list of (String, SensitiveInt SensitiveInt3). The phantom parameters of the SensitiveInts do not match, resulting in an error which we can interpret as sensitive data flowing from highestPaid to sort2.

By interpreting all such type mismatches as data flow, we build a complete picture of a how a sensitive data type flows through an application. For our example application, we might get a picture as in Figure 3-5.

3.5 Partitioning the Application

Section 3.3 showed how to identify suitable encryption schemes for each function, and Section 3.4 showed how to determine the paths data takes through the application. We now combine this information to output a partitioned application.

We name the two partitions *Trusted* and *Untrusted.* Functions in the Trusted partition will have access to plaintext data while functions in the Untrusted partition will have access to only encrypted versions. This way, we can run functions in the Untrusted partition in an untrusted environment since they never operate on sensitive data in plaintext. The Trusted partition should be run in a trusted environment since sensitive data will be accessible in the clear.

Now we address the issue of assigning each function to a partition. We observe that the same sensitive data may be represented by different types throughout the execution of the program. For example, sensitive user input may initially be stored as a String until some function parses and constructs a developer-designated sensitive ADT with it. Similarly, sensitive data may be read out of one these ADTs into a different type. For example, showing an ADT might return a string containing the sensitive data.

Our analysis works at the level of types rather than data. Our dataflow graph tracks only functions dealing with sensitive types. We know nothing about the application's behavior on sensitive data in cases where its type is not one designated as sensitive by the developer.

As a result, we take a conservative approach when assigning functions that are not part of the dataflow graph and assign them all to the Trusted partition.

We can assign functions in the dataflow graph to a partition using our analysis from Section 3.3. Each function that we can modify to run on encrypted data goes in the Untrusted partition. The remaining functions go in the Trusted partition.

```
data SensitiveInt1
data SensitiveInt2
data SensitiveInt3
data SensitiveInt4
data SensitiveInt5
data Employee1
data Employee2
data Employee3

data SensitiveInt p = MkSensitiveInt {
    unSensitiveInt :: Int
}

data Employee p = Employee {
    name :: String,
    salary :: SensitiveInt SensitiveInt1,
    ...
}

data Division p = Division {
    employees :: [Employee Employee1],
    ...
}

highestPaid :: [Employee Employee2] ->
                (String, SensitiveInt SensitiveInt2)
highestPaid employees = head . sort2 $
    [(name e, salary e) | e <- employees]

sort2 :: [(String,SensitiveInt SensitiveInt3)] ->
            [(String,SensitiveInt SensitiveInt4)]
sort2 s = case s of
            [] -> []
            (a:b) -> let p = snd a
                         l = filter ((<=p) . snd) b
                         h = filter ((> p) . snd) b
                     in (sort2 h)++[a]++(sort2 l)

averageSalary :: [Employee Employee3] -> Int
averageSalary e = average (map salary e)

average :: [SensitiveInt SensitiveInt5] -> Int
average s = quot (sum s) (length s)
```

Figure 3-4: PARTE adds a phantom parameter to wrapped types and parametrizes it with unique types.

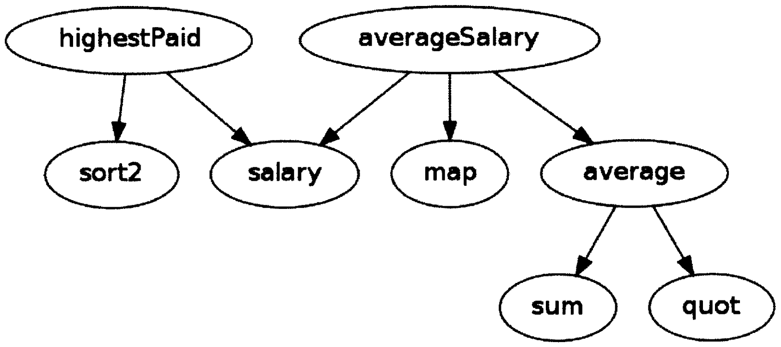

Figure 3-5: The data flow graph for our example application. Each oval is a function and the arrows indicate the flow of sensitive data.

Chapter 4

Implementation

In this chapter, we describe our prototype implementation of PARTE.

As described in Chapter 3, our design calls for an implementation that can define new types and modify existing types in a given codebase, type check a module and inspect any type errors. We chose to implement these capabilities in the simplest way possible that was still sufficient to evaluate PARTE as described in Chapter 5. Haskell has libraries available to parse and modify Haskell source code, and the Glasgow Haskell Compiler (GHC) exposes its internals, and the type checker in particular, through a nice API. We made use of these libraries and APIs to implement a prototype in approximately 3,500 lines of Haskell. We used version 7.2.1 of GHC. We used a third party implementation of AES, took the OPE implementation from CryptDB, and implemented the Paillier cryptosystem ourselves.

The GHC API provided convenient methods to type check our modified source code. However, it returned errors as strings, such as "Expected type: `SensitiveInt` Actual type: `Int` In the second argument of a call of foo". The location of the code that caused this error is also returned. Our implementation simply parses this error message to extract the type mismatch and determines the function that caused this error by searching for the function at the location of the error.

Our implementation is far from robust. Parsing error messages is clearly not a desirable way to extract information from the compiler. A more robust approach might hook into the compiler's type checking logic to extract precise error information. However, our implementation was sufficient to evaluate the ideas presented in this thesis.

In order to allow execution to flow between the two partitions, PARTE embeds a remote procedure call (RPC) server in the untrusted partition and an RPC client in the trusted partition. Program execution is expected to begin in the trusted partition. Functions that can run in the untrusted partition are invoked via RPC using the embedded client and server. PARTE inserts code to evaluate thunks and serialize data before being transmitted via RPC. One limitation of PARTE is that it requires all arguments and return values to be fully evaluated before being sent between partitions. Moreover, because Haskell has no notion of global variables PARTE does not have to deal with synchronizing global state between partitions. Mutating data (such as `MVars`) are not supported if they are used across partitions. These limitations could be overcome by using techiques such as those used by Pyxis [7] to synchronize global state across program partitions.

Chapter 5

Evaluation

In this chapter we evaluate our prototype of PARTE.

Our original goal was to reduce the risk of leaking sensitive data in a way that is efficient enough to be practical. Therefore, we evaluate our prototype along two dimensions:

1. Does PARTE actually reduce the risk of data leakage?

2. Is PARTE's output efficient enough to be practical?

To answer these questions, we run PARTE on a real application. Hpaste [3] is a web application that allows users to upload and share text snippets called pastes. Suppose we are using an untrusted third party to host the application. We would like to avoid leaking our pastes to the hosting provider and any attackers who might compromise them. We use PARTE to help us achieve this goal.

We describe the application and PARTE's partitioning of it in more detail in Section 5.1. Then we discuss the performance in Section 5.2 and the security in Section 5.3.

We notice that PARTE is particularly useful at identifying which functions in the application perform work that is not efficiently supported by any cryptosystems that we are aware of. This is useful to the application developer since it identifies complex parts of the application which, if simplified, could allow PARTE to work more effectively. It is also useful to cryptographers since it identifies operations that developers might like cryptosystems to support efficiently. We explore this idea further in Section 5.4 and demonstrate how it can be useful on several other toy applications that we wrote ourselves.

5.1 Hpaste

5.1.1 What is hpaste?

Hpaste is a web application that allows its users to upload snippets of text, called pastes. Each paste is accessible from the web application's home page or can be shared with a direct URL. Hpaste is implemented in approximately 3,500 lines of Haskell. It is built on top of the Snap [4] web framework and stores pastes in a Postgres database. A live version of the web application is available at `http://hpaste.org`.

5.1.2 Applying PARTE to Hpaste

PARTE requires a few pieces of input from the application developer, but otherwise partitions the application automatically. We need to identify the data structures that will hold sensitive data at runtime. In this application, we consider the contents of each paste to be sensitive. Looking through the source code, we see that there are two data structures that hold pastes. The first is the `pastePaste` field of an ADT named `Paste`. A paste that is retrieved from the database to be rendered into the web page returned to the user is stored in this structure. The second is the `pasteSubmitPaste` field of an ADT named `PasteSubmit`. A new paste submitted by the user is held in this structure while on its way to being wrtten to the database. We just need to specify the module, ADT name, and field name of each of these structures, so describing them to PARTE takes only two lines of annotations.

In order for PARTE to allow program execution to flow between partitions, the application's data structures need to be serializable. In Haskell, this can be done simply by deriving the `Read` and `Show` type classes. However, for other purposes the application had already defined instances of `Show` which could not be `read` back in unambiguously. Instead, we use the `Data.Binary` class to define our own serialization functions for the relevant data types in approximately 50 lines of code. Alternatively, packages like `cereal-derive` can be used in conjunction with GHC's `DeriveGeneric` extension to automatically derive serialization methods and avoid having to write any boilerplate code.

Running PARTE on hpaste to produce a partitioned application completes in 353 seconds. For comparision, simply building an unmodified copy of hpaste on the same machine takes 24 seconds. Because we iteratively compile the application, extract type error information, and then fix the error we expect to take several orders of magnitude longer than simply building the application. Table 5.1 breaks this time down according to each of the phases as described in Chapter 3.

Analysis Phase	Time (s)
Wrap sensitive data in unique type (Section 3.2)	62
Data flow with phantom types (Section 3.4)	158
Determine type class constraints (Section 3.3)	52
Assign functions to partitions (Section 3.5)	81
Total	353

Table 5.1: Time breakdown of running PARTE on hpaste.

PARTE partitioned the application into two pieces such that one piece sees only encrypted pastes. This piece primarily consists of the functions that read and write to the database and totalled approximately 800 lines of code. These functions do not transform the pastes at all, so a randomized encryption scheme (AES in CBC mode with a random initialization vector) is used to encrypt pastes before they are passed to these functions. We replace the server side code with this piece. We note that the application does not issue any SQL queries that depend on the content of pastes, and that the original pastes and encrypted pastes have the same type (strings) so we were able to use a standard Postgres backend without having to make any changes to the schema. If this were not the case, we would need to use a database that supports SQL queries over encrypted data such as CryptDB [16] or Monomi [19].

The other partition consists of the rest of the application which formats pastes and generates HTML to return to the client and consists of approximately 2,700 lines of code. It requires access to plaintext pastes because of the complex transformations it performs on pastes (e.g. syntax highlighting), so it must be run in a trusted environment. In this case, that environment is the client's machine. One way to preserve the user experience is to run the trusted partition in a browser extension on the client. The extension can transparently intercept all requests to the hpaste server and redirect them to the trusted partition. However, we instead run the trusted partition as an HTTP server on the client's machine and have the client's browser connect to this server. This is much simpler and sufficient to evaluate the performance and security of the partitioning.

5.2 Performance

In this section, we ask if PARTE's partitioning of hpaste is performant enough to still be practical. To answer this question, we compare the partitioned hpaste with the original hpaste. More specifically, we look at how partitioning has affected the application's response time (Section 5.2.1) and network usage (Section 5.2.2).

5.2.1 Processing Performance

To measure the time to service one request, we manually instrument both the unmodified and the partitioned version of the application to record the time when processing starts and ends. For the unmodified application, we measure the time elapsed between receiving a request and returning a response. The time to send the request for the application and the time for the end user to receive the response is covered in the next section on network performance. For the partitioned appliction, we measure the time elapsed in both the client and server side code. We do not include the time taken to perform RPCs since this is network performance which we cover in the next section.

We measure processing time for two types of requests: storing a paste and fetching a paste. We use the code in Figure 3-1 as the contents of the paste. We run each type of request 1,000 times and average the results in Table 5.2. We can see that partitioning causes a 1.2× slowdown for stores and a 2.5× slowdown for fetches.

There are two things of note here. The first is that the original unmodified application is quite slow to begin with. This is because of a bug in the Haskell library used by hpaste to communicate with Postgres that causes a delay of approximately 40 milliseconds for every database call. The second is that reading a past out of the database is slower than writing a paste. This is because the reading a paste performs more database calls and therefore induces more of the 40 millisecond delays. The rest of the overhead is attributable to the encryption and serialization overhead.

Request type	Unmodified hpaste (ms)	Partitioned hpaste (ms)
Store paste	168	214 (client 123, server 91)
Fetch paste	199	499 (client 162, server 337)

Table 5.2: Time taken for hpaste to serve one request.

5.2.2 Network Performance

To evaluate network performance, we measure the total number of bytes transferred over the network. In the unmodified application, the two components talking over a network are the web application's server and the end user's browser. In the partitioned application, the only network activity is between the two partitions. The trusted partition has locality with the end user's browser so we assume that incurs no additional network overhead. Moreover, we asusme the the backing database has locatity with the application server/untrusted partition so we assume communication with the database does not incur additional network overhead.

Table 5.3 shows the results. We can see that partitioning incurs no overhead in storing a paste but fetching a paste is almost twice as slow with partitioning. There are several things going on here. First, we note that in the unmodified version of hpaste, the client is a web browser making HTTP requests to the server. In the partitioned version of hpaste, the client is making RPCs to the server. In both cases, the content of the paste is transferred between client and server. Storing a paste takes roughly the same number of bytes in both cases because AES in CBC mode has minimal ciphertext expansion so the data is roughly the same number of bytes and the overhead of HTTP is roughly the same as the overhead for our RPCs. Fetching a paste shows markedly different numbers. In the unmodified version of hpaste, the client receives the paste in an HTTP response sent by the Snap server that hpaste is built on. The Snap server compresses its response which the client's browser decompresses which is why fetching a paste in the unmodified hpaste is smaller than both storing a paste in unmodified hpaste and fetching a paste in partitioned hpaste. Fetching a paste in partitioned hpaste is much larger because the RPC client receives a response from the RPC server which is not compressed. In fact, compression would not reduce the size anyways because the response largely consists of encrypted data.

Request type	Unmodified hpaste (bytes)	Partitioned hpaste (bytes)
Store paste	1861	1854
Fetch paste	1067	1758

Table 5.3: Bytes sent across the network to serve one request.

5.3 Security

In this section, we ask if PARTE's partitioning of hpaste has reduced the risk of data leakage. To answer this question, we manually audit and compare the partitioned hpaste with the original hpaste.

There are many ways in which sensitive user data can be leaked with the unmodified hpaste running on an untrusted server. Anyone who has compromised the server or otherwise has access to it can simply dump the database tables and read the pastes. They could also replace the application binary to forward all new pastes to interested third parties.

In the partitioned application, pastes are always encrypted under some encryption scheme before being sent to the untrusted server. In this case, anyone who has compromised the server or otherwise has access to it cannot read the pastes by dumping the tables because the pastes are encrypted. They also cannot learn the contents of the pastes by replacing the server binary because the server only sees encrypted pastes and never receives the decryption key. However, some information is leaked. For example, some encryption schemes leak some fraction of bits of the plaintext. The order preserving scheme we use leaks half of the bits of the original text. This may be enough for the attacker to learn useful information about the sensitive data or it may not. The amount of data leaked in this way is application dependent since it depends on the encryption schemes used. An application developer using PARTE on his application would have to manually audit the partitioning to determine the security benefit to his application.

Generally, we believe that the partitioned application greatly reduces the risk of leaking data since it only ever reveals encrypted data while the original application deals in plaintext.

5.4 Motivating New Cryptosystems

In this section, we discuss how PARTE can provide interesting insight as to what the complex functions in an application are and what types of encryption schemes, if developed, would be useful.

We notice that the functions that run in the trusted partition must do so because there is some operation that they perform which is not efficiently supported by any encryption scheme that we are aware of. It would be useful to know what these operations are because the application developer could possibly rewrite the logic to do something simpler that is efficiently supported by an existing encryption scheme. Even if this is not possible, it suggests operations that would be useful for encryption schemes to efficiently support so that applications that operate on sensitive data would no longer need to trust the environment in which they run.

It can be difficult to manually analyze an application to determine what these operations are. Without being intimately familiar with a codebase, searching for places where sensitive data is computed on and tracing how it is passed between functions poses quite a challenge. Without this knowledge, determining the precise set of operations which you would like an encryption scheme to efficiently support can be very difficult.

The data flow graph produced by PARTE can greatly simplify this task. We begin by looking at the line that PARTE draws through the data flow graph to partition the application's functions. We would like to move this line so that more functions fall in the partition to be executed by the untrusted party. In order to move this line, we look at the operations performed by functions that fall along the boundary but on the trusted side. These are the operations which we would like some encryption scheme to efficiently support.

In the following subsections, we identify desirable encryption schemes for hpaste and another simple application which we wrote but which performs computations that are common amongst real applications that leverage third party infrastructure [2].

5.4.1 hpaste

At a high level, the partitioning of hpaste described in Section 5.1 results in the untrusted server being able to do a minimal amount of work–reading pastes out of and writing pastes into the database–leaving the trusted client to do the rest. We would like for the server to do more of work to remove the burden from the client's resources.

From the dataflow graph, we can identify functions in the client's partition along the boundary separating the client from the server's partition. Quickly scanning these, two in particular stand out.

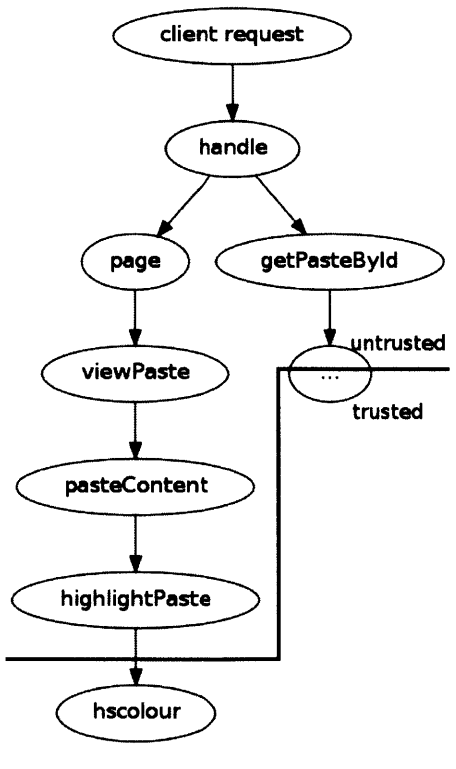

Figure 5-1: A part of hpaste's dataflow graph and PARTE's partitioning of it.

First is the function `Language.Haskell.HsColour.CSS.hscolour`. If a paste contains source code written in one of several programming languages, hpaste will colorize language keywords. Figure 5-1 shows a part of hpaste's dataflow graph and PARTE's partitioning of it. We see that just along the boundary on the trusted client's side is `hscolour`. To move the partitioning line so that the untrusted server does more work, we need an encryption scheme that supports the operations performed by `hscolour`. Looking at the body of this function, we see that it uses substring matching to identify language keywords and substring insertion to markup the paste so that those keywords are colorized. This suggests that an encryption scheme that supports text matching and insertion would be useful.

The second function that stands out is `Amelie.View.viewDiff`. As its name suggests, it takes two pastes as input and computes the diff between them. As a result, an encryption scheme that supports computing the diff between two strings would be useful.

5.4.2 Log Analysis

Third party resources are often used for log storage and analysis [2]. Stored logs are usually very large and need to be filtered to find relevant entries. A common way to filter is to match a pattern, such as "error", against each log entry. This is type of filtering is well suited to MapReduce style computation if the logs are distributed across a cluster of machines.

We wrote a simple Haskell application to do pattern based log filtering. We want to find out which operations limit what computation the untrusted party is able to do. As with hpaste, we use PARTE to generate a dataflow graph and we immediately see that the limiting function, `match`, compares a pattern against a log entry along the partition boundary. As a result, an encryption scheme that efficiently supports pattern matching could be useful to partition analytical workloads such as this.

5.4.3 Employee Application

We augmented our example application that dealt with employee salaries as shown in Figure 3-1 to take a set of employee records as input and output the name of the employee with the highest salary as well as the average salary. We used PARTE on this application which produced the partitioning shown in Figure 5-2. An order preserving scheme is used to encrypt the salary before sending it to the `highestPaid` function. `average` must remain in the trusted partition because it imposes the `Integral` type class restriction on its argument because of its use of `quot`, and this type class does not map to any encryption scheme that we know of. This suggests that the operations defined by the `Integral` type class which include integer division, quotient, and remainder functions might be useful for an encryption scheme to efficiently support.

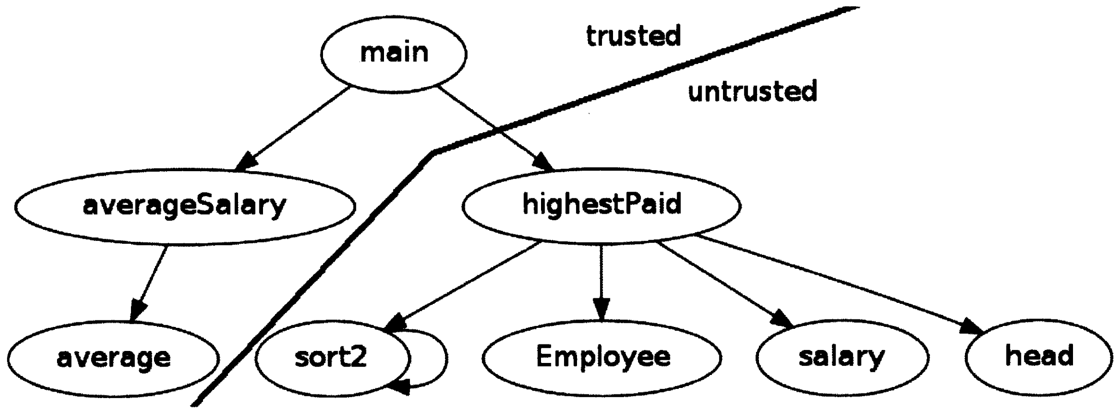

Figure 5-2: PARTE's partitioning of our example employee application.

Chapter 6

Discussion

In this chapter, we discuss why we observed the results that we did when evaluating PARTE, how we might do better, and compare and contrast PARTE's approach with those of other systems with similar goals.

The partitionings that PARTE produced on hpaste and several other toy applications we wrote included only minimal computation in the untrusted partition. This is far from optimal because untrusted parties (such as third party hosting services like Amazon EC2) are generally used for the resources that they provide. However, if PARTE only allows minimal computation to occur in the untrusted partition and forces the trusted partition to do the bulk of the work, then applications cannot take advantage of third party resources. In Section 6.1 we discuss why the untrusted partition was delegated a minimal amount of computation. In Section 6.2 we discuss approaches PARTE might use to produce more desirable partitionings. In Section 6.3 we discuss why CryptDB and Monomi, the most similar works in terms of goals and approaches, are able to perform much better. Finally, in Section 6.4, we discuss directions for future work.

6.1 Performance

In this section, we discus why we observed the partitionings that we did in Chapter 5. We believe that there are three main reasons why the untrusted partition was able to do so little computation.

1. As far as we know, only a very limited number of operations are efficiently supported by some encryption scheme. This fundamentally limits what PARTE can allow to run in the untrusted partition.

2. PARTE's function level analysis may be too coarse. Functions consist of many operations, some of which we can perform efficiently on encrypted data and some which we cannot. However, the presence of a single operation that we cannot perform efficiently on encrypted data causes PARTE to place the *entire* function in the trusted partition. This may unnecessarily place computation in the trusted partition when it could be safe to run in the untrusted partition.

3. Approximating sensitive data with sensitive data structures that may hold sensitive data at runtime severly limits PARTE's efficacy in placing functions in a partition. Sensitive data may change types at runtime (e.g., through a `toString` method). After such a type change, PARTE cannot trace the code path that the sensitive data follows. As a result, PARTE must be conservative and place all functions that do not explicitly deal with a sensitive data structure in the trusted partition. If such a function does not in fact interact with sensitive data, then this placement is unnecessary.

6.2 Improvements

In this section, we discuss how the three main problems identified with PARTE in Section 6.1 could be alleviated.

1. An efficient FHE scheme would certainly solve many problems since it would make any operation efficient to perform over encrypted data. In fact, such a development would likely give rise to more elegant solutions that solve the problem of data leakage. Next best would be more encryption schemes that efficiently support a limited set of operations. This would allow PARTE to place more application code in the untrusted partition. In fact, as discussed in Section 5.4, given an application PARTE can identify which operations an encryption scheme should be developed to efficiently supprt in order for the application to benefit more from partitioning.

2. Analyzing an application at a finer grain than the function level would allow PARTE to separate individual functions across the two partitions. Whereas before a function that performed lots of simple computations and one complex computation would have to run in the trusted partition, a finer grained analysis might allow the simple computations to be moved to the untrusted partition. This, however, may pose its own challenges if statements within a function depend on each other. However, these challenges have been addressed by previous works ([7, 23]).

3. Precisely tracking sensitive data instead of approximating this by tracking sensitive data structures would allow PARTE to be much less conservative when placing code in a partition. If PARTE could be sure that a piece of code did not operate on sensitive data, then it could place that code in the untrusted partition. However, the current approach has many false positives. A different approach might perform dynamic analysis offline in a trusted environment to taint sensitive data and observe the code paths it traverses. In this way, code that operate on sensitive data could be learned and the application could potentially be more effectively partitioned.

6.3 Similar Systems

In this section, we compare and contrast PARTE with two systems which we feel are most similar in approach and goal: CryptDB [16] and Monomi [19]. These two systems enable the execution of SQL queries over encrypted data. They pre-encrypt multiple copies of data using different encryptions schemes in what they refer to as onions. Monomi further precomputes certain values in anticipation of specific query types. Incoming queries are analyzed and based on the operations they perform, the data encrypted under the appropriate scheme is exposed. If a plaintext version of the data is required, CryptDB pulls all the data into a trusted environment, decrypts it and executes the query. Monomi partitions the query so that only the operations that need plaintext access to the data run in a trusted environment; the rest of the query runs in the untrusted database server.

These systems are similar in that both reduce the risk of data leakage in untrusted environments by computing over encrypted data. Monomi is further similar in that it partitions query execution for efficiency for specific workloads.

A primary difference between PARTE and these systems is that PARTE aims to be usable for arbitrary programs while these systems are only concerned with SQL queries (and in the case of Monomi, only analytical workloads). Although all of these systems have the same set of encryption schemes available for use, we believe that these SQL-oriented systems are much more successful in achieving their goals partly because SQL queries are much more amenable to the types of analysis needed to enable computation over encrypted data.

To begin, PARTE has to analyze arbitrary programs rather than SQL queries. A SQL query is easily parsed. The parse tree can be walked to determine the operations used and the order in which they are applied to data. In the database scenario, sensitive data is generally identified by column so there is little ambiguity as to whether a piece of data is sensitive or not. The problem that PARTE faces of distinguishing `Ints` that contain sensitive data from `Ints` that do not does not exist. Moreover, there are only a limited number of ways in which a query can refer to data (i.e., column name, an alias via the `AS` directive, and assignment). With little effort, a data flow graph can be extracted from a parse tree. Overall, the very structured and predictable format of SQL queries makes it easier to make them work over encrypted data than an arbitrary program.

6.4 Future Work

This section discuss some of the possible directions for future work. In Section 6.2 we discussed several ways in which PARTE might be improved. These form a good starting point for several avenues of future work.

One improvement not discussed in that section is to avoid having to fully evaluate thunks before sending them between partitions. One advantage of Haskell, and lazily evaluated languages in general, is that computations are stored as thunks and are only evaluated when needed. This allows the program to avoid performing any unnecessary computation. By forcing thunks to be fully evaluated, we lose this benefit. However, a drawback of thunks is that they consume a lot of memory. Therefore, while fully evaluating thunks before sending them across the network would likely significantly reduce the number of bytes sent across the network, it could also cause a lot of unnecessary computation to occur. Experimenting with this tradeoff to find an optimal balance is an interesting direction of future research.

Our analyses used specific features of programming languages such as phantom types and type classes. Not all programming languages support these constructs. Moreover, we observed that analysis at the level which we performed it at was too coarse grained. It would be interesting to further explore finer grained techniques that would work with more popular programming languages that may not necessarily support the language constructs that we used in this work.

Finally, we applied program partitioning to reduce the risk of data leakage. Our analysis could perhaps be used to optimize some other metric, such as network efficiency. Other works, such as [7], explore this. It would be interesting to see how our approach applied to different goals might compare with existing work.

Chapter 7

Conclusion

This thesis presented the design of PARTE, a tool to help lower the risk of data leakage when running applications in untrusted environments.

Our approach involves approximating the static tracking of sensitive data via sensitive data structures as indicated by the developer, using type classes to infer operations performed by functions, mapping type classes to operations efficiently supported by encryption schemes, and partitioning the application so that part of the application may safely execute over sensitive data in an untrusted environment.

We demonstrated our prototype implementation on hpaste, a web application, as well as two toy applications. Our evaluation showed that there is much room for improvement. We suggested several ways we might modify our approach to achieve more satisfactory results. We also showed how PARTE can offer insight as to what sorts of operations we would like encryption schemes to efficiently support.

Bibliography

[1] Aws case study: foursquare. URL `https://aws.amazon.com/solutions/case-studies/foursquare`.

[2] Powered by hadoop. URL `https://wiki.apache.org/hadoop/PoweredBy`. Accessed: January 16, 2013.

[3] hpaste. URL `http://www.hpaste.org`. Accessed: January 16, 2013.

[4] Snap framework. URL `http://snapframework.com`.

[5] Aws case study: Yelp. URL `https://aws.amazon.com/solutions/case-studies/yelp`.

[6] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, April 2009.

[7] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *Proc. VLDB Endow.*, 5(11):1471–1482, July 2012. ISSN 2150-8097. URL `http://dl.acm.org/citation.cfm?id=2350229.2350262`.

[8] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924943.1924951`.

[9] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 122–133, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806612. URL `http://doi.acm.org/10.1145/1806596.1806612`.

[10] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *SOSP*, October 2007.

[11] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proc. of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, October 2010.

[12] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with frientegrity: privacy and integrity with an untrusted provider. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 31–31, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362824`.

[13] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, May–June 2009.

[14] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. Cryptology ePrint Archive, Report 2012/099, 2012. `http://eprint.iacr.org/`.

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

[16] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, October 2011.

[17] K. P. N. Puttaswamy, C. Kruegel, and B. Y. Zhao. Silverline: toward data confidentiality in storage-intensive cloud applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 10:1–10:13, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038926. URL `http://doi.acm.org/10.1145/2038916.2038926`.

[18] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44 –55, 2000. doi: 10.1109/SECPRI.2000.848445.

[19] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, August 2013.

[20] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75283. URL `http://doi.acm.org/10.1145/75277.75283`.

[21] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 85–96, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103669. URL `http://doi.acm.org/10.1145/2103656.2103669`.

[22] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 291–304, New York, NY, USA, 2009.

ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629604. URL `http://doi.acm.org/10.1145/1629575.1629604`.

[23] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *SOSP*, October 2001.