A CONTEXT-SENSITIVE SECURITY TYPE SYSTEM FOR JAVA

By

Benjamin Kaiser

A Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved by the Examining Committee:

Ana Milanova, Thesis Adviser

Bülent Yener, Member

Carlos Varela, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2015
(For Graduation May 2015)

© Copyright 2015
by
Benjamin Kaiser
All Rights Reserved

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

Foremost, I would like to acknowledge my advisor Prof. Ana Milanova for her guidance and patience as I delved into a field I knew little about. I thank the other members of my thesis committee: Prof. Bülent Yener and Prof. Carlos Varela. Several of my fellow graduate students also deserve thanks for their roles as my sounding board: Jeremy Blackthorne, Alexei Bulazel, and Zach Jablons.

ABSTRACT

Existing cryptographic schemes can easily protect sensitive data in transit and while in storage. When it becomes necessary to compute over that data, there are a wide variety of cryptographic and language-based solutions that protect the data in different ways and from different adversaries. However, to date, there are few practical schemes that can fully guarantee the security of sensitive data when an untrusted machine performs operations over it.

This thesis presents the theoretical framework for a context-sensitive security type system for Java programs. The primary contribution is JSec, a two-stage protocol that prepares a program containing sensitive data to safely run on an untrusted machine. Given the program and a subset of its variables declared as sensitive, JSec first tracks information flow in order to infer what additional variables must be considered sensitive in order to ensure confidentiality of data. The use of a polymorphic type in this stage permits context-sensitivity, which allows us to type check a very broad class of Java programs. In the second stage, the sensitive variables determined in the first stage are encrypted using homomorphic encryption schemes that allow operations to be computed over ciphertexts. The final program can be safely executed by an untrusted host but must defer to a trusted host for key management, encryption, and decryption.

1. INTRODUCTION

Encryption protects sensitive data in transit and while it is stored on cloud servers. However, modern applications often not only store but operate on this data. To derive metrics from user records, for example, a company might run some analytic program on this sensitive data. That company might in turn outsource some of those computations to infrastructure services like Amazon EC2. Often these programs can only operate on unencrypted data, leaving potentially sensitive information vulnerable to leakage or theft. For users entrusting companies with their data and companies entrusting third-party services, the need to balance privacy and security with convenience and functionality is urgent.

The problem of executing programs on encrypted data has received substantial attention. Schemes based on cryptographic primitives such as oblivious RAM [2], fully homomorphic encryption [3], and multi-party computation [4] have achieved various degrees of security and efficiency (see §5.1 for discussion). However, the body of work addressing *language-based* techniques for encrypted computation is comparatively shallow. Such techniques build or augment type systems in order to impose information flow policies to govern the flow of sensitive data. A brief discussion of some of the immediate issues that any language-based encrypted computation scheme must address can be found in [5].

One particular language-based scheme is called *program partitioning* [6]. A program partitioning algorithm divides a program into partitions that are each run by different machines, called hosts. When run concurrently (and typically with some communication channel between them) these partitions compute the same functionality as the original program. When a program contains sensitive data that must not be revealed to one or more hosts, a partitioning can be generated that enforces that confidentiality requirement.

The company that wants to outsource expensive computations on sensitive user data requires a two-host partitioning. One partition will be thought of as "untrusted" by the company while the other is "trusted." The untrusted partition can be safely executed on a computer that the company does not trust without revealing the sensitive data in plaintext to the owners, administrators, or users of that machine. The trusted partition must be executed on a trusted machine.

Since the goal is to execute as much of the computation as possible on the untrusted computers of a third-party infrastructure service, a partitioning should attempt to maximize the untrusted portion of the program. Of course, it would be ideal to execute the entire program on the untrusted host, but due to the difficulty and scope of this problem, little research has been conducted in this area.

This paper proposes JSec, a context-sensitive type system for encrypted computation in Java. Given a program in which the author has denoted which variables are sensitive, JSec encrypts the program so that it can safely run in full on an untrusted host without leaking sensitive data. The trusted host is responsible for the storage of cryptographic keys and the execution of encryption and decryption routines as necessary. While this does not truly allow the program to execute independently of the trusted host, it is still a significant security guarantee.

In the first stage of JSec, a basic type inference algorithm determines what additional variables must be marked as sensitive to prevent data flow from a sensitive variable to a cleartext variable. In this stage, we also make use of a *polymorphic* type, which allows us to achieve context sensitivity. A method or variable designated as polymorphic can be instantiated in either encrypted form or cleartext form based on the context. This allows JSec to handle a broad class of programs that would otherwise be out of scope.

After sensitivity has been inferred, JSec assigns encrypted types to sensitive variables. These types each correspond to a cryptosystem, and every variable of

a given type will be encrypted with that type's corresponding encryption scheme. This is possible because some modern encryption schemes allow for operations on encrypted values. Therefore, those operations can be safely performed on sensitive data on the untrusted machine because that data can be in encrypted form.

It is worth noting that fully homomorphic encryption schemes [7] allow for arbitrary computation on encrypted data. However, existing implementations are too inefficient for practical use. JSec instead uses a set of encryption schemes, each of which supports a different operation.

These encrypted types are assigned to variables based on the operations those variables undergo. For example, if a sensitive variable is added to another value, it will be assigned a type that corresponds to an encryption algorithm that supports addition over encrypted values. Such cryptosystems are called *additively homomorphic*.

If that same variable later undergoes a comparison operation, it will be decrypted and re-encrypted with a new cryptosystem that supports that operation. This conversion process must be performed by the trusted host, who stores the encryption keys. However, the operations themselves can be executed by the untrusted host. At present, JSec supports the following operations over encrypted numeric values: addition ($+$), multiplication ($*$), equality ($==$), and order comparison ($\leq$).

This thesis's primary contribution is the theoretical exploration of the first context-sensitive encrypted type system for full Java programs. Prior works, of which there are few, explore other, more restricted languages like SQL [8], are platform-specific [9], or are restricted to a small functional subset of the language [10]. By handling a very broad set of programs as well as messy "side-effects" of Java such as context sensitivity and aliasing, JSec represents a significant step forward in type-based secure computation.

The organization of the rest of the thesis is as follows: §2 outlines the first stage of inference and the types, rules, and syntax thereof. Examples of basic type inference are demonstrated. §3 discusses the types and cryptosystems chosen for the second stage and the modifications to the rules and syntax necessary for encrypted type inference. §4 describes future direction for work on this subject, including discussions of implementation and optimization. §5 presents existing approaches to the problem of encrypted computation and other related works.

2. BASIC TYPE SYSTEM

Given a program and a subset of that program's variables denoted as sensitive, the first stage of JSec infers what additional variables must be marked as sensitive. This is done through the use of a *type system*, which consists of a series of rules that assign type qualifiers to variables. These rules track information flow between variables in order to ensure *type safety*, which means that no type errors will occur. These errors can represent anything; in common programming languages usage they primarily indicate when a value attempts to undergo an operation that it cannot support. Type checking can happen statically (at compile time) or dynamically (at run-time).

In JSec's type system, type errors represent flow from sources of sensitive data to cleartext variables. The rules in our type system enforce type safety against those errors. If a program type checks through JSec, it is guaranteed that no sensitive data flows to an unencrypted variable.

It is important to note that JSec currently only handles *direct information flow*: flow based on assignment statements. See the following code example:

```
1  int x = 10;
2  int y = x;
3  int z = x;
```

In this program, line 2 demonstrates direct information flow from x to y and line 3 demonstrates flow from y to z. Since information flow is transitive, there is also flow from x to z.

JSec ignores *indirect information flow*, which is the leakage of information through control flow. When a conditional is evaluated, it is sometimes possible to learn the value of one variable based on the value of another. For example:

```
1  int x = 10;
2  int y = 10;
3  if (x == 10) {
4     y = 5;
5  }
```

In this program, there is indirect information flow from x to y at line 4. If y's value is revealed to the user after that line has been executed, the user will be able to infer that $x = 10$ because the conditional must have been true.

2.1 Type Qualifiers

In JSec, every variable in a program is assigned a *basic type qualifier*. There are three:

- Sensitive (*sen*)

  A variable x is sensitive if it is marked as such by the program author or if there is flow to it from another sensitive variable. The initial set of sensitive variables as demarcated by the author can be thought of as *sources* of our type system. The value of a sensitive variable will always be in encrypted form.

- Cleartext (*clr*)

  Cleartext variables hold unencrypted values. Although a program author could explicitly declare an initial set of *clr* variables, it is more likely that we will have to infer what variables must have this qualifier. Our inference algorithm maximizes the number of *clr* variables. It considers `clr` to be the highest priority and, given a choice between assigning a variable *clr* or a different qualifier, it will always choose *clr*.

- Polymorphic (*poly*)

  Having a polymorphic type allows JSec to achieve *context sensitivity*. A variable x marked as *poly* will be interpreted as *clr* in some contexts and *sen* in others.

This is a powerful technique that offers significant efficiency gains. It is always faster to execute code over cleartext data than over encrypted data. A non-context sensitive scheme would frequently be forced to encrypt non-sensitive variables and evaluate methods over those encrypted variables. Context sensitivity allows JSec to avoid these extraneous operations by noting when a method or variable is instantiated as both *clr* and *poly* at different points in the program.

See §2.1.1 for additional discussion of context sensitivity and viewpoint adaptation.

Type systems typically include *subtyping relationships* between the qualifiers. A variable x can safely be assigned to a variable y that is its subtype. This means that any functions or operations that would have been performed on x (called the supertype) will be able to execute on y. More importantly for our purposes, information can flow from a subtype to a supertype but not from a supertype to a subtype.

Since our goal is to prevent flow from sensitive variables to cleartext variables, JSec enforces the subtyping hierarchy:

$$clr <: poly <: sen$$

That is, *clr* is a subtype of *poly* which is a subtype of *sen*. Subtyping is transitive, so *clr* is a subtype of *sen*. Each type is also considered to be a subtype of itself.

Because of this relationship, we can safely assign a cleartext variable to a sensitive variable:

```
clr int c = 10;
sen int s = c;
```

However, we cannot allow flow from a *sen* variable to a *clr* variable. The second statement in the following code snippet will **not** type check:

```
sen int s = 10;
clr int c = s;
```

2.1.1 Viewpoint Adaptation

The *poly* type allows JSec to achieve context sensitivity. *poly* is interpreted to either *clr* or *sen* through *viewpoint adaptation*. Viewpoint adaptation of a type q' from the viewpoint of a type q produces the adapted type q'' (written as $q \triangleright q' = q''$). The viewpoint adaptation operator $\triangleright$ is defined as follows:

$$\_ \triangleright sen = sen$$

$$\_ \triangleright clr = clr$$

$$q \triangleright poly = q$$

The underscore denotes an arbitrary type; therefore the first two statements indicate that the *sen* and *clr* types are context independent. The *poly* qualifier, however, will be interpreted to match the type of the viewpoint. This is primarily used to adapt parameters and return values from the viewpoint of the callsite.

The use of the polymorphic type is necessary in many cases. For example, the formal parameter of a function that is called once with a *sen* argument and once with a *clr* argument would need to be *poly* in order to assume both types. Consider the following sample code:

```
int id(int i) {
  return i;
}

sen int x = 10;
int y = 5;

x = id(x);
y = id(y);
```

When *id(x)* is called, *i* must be *sen*. But when *id(y)* is called on the following line, *i* must be *clr*. Thus it will be necessary to assign *i* the *poly* type. When this happens, we will create two copies of the *id()* function: one with the parameter *sen int i* and one with the parameter *clr int i*. Based on the type of the argument passed in the function call, we will choose which version of the function to evaluate. This is analogous to how templates are treated in C++. A parameterized type in that language is instantiated multiple times, each copy with a different type argument.

2.2 Program Syntax

Typing rules are defined over a *program syntax*. JSec's syntax is shown in Fig, 2.1. Terminal symbols c and d are class names, f is a field name, m is a method name, and x, y, and z are names of local variables or parameters.

C → class c extends d {F M} | int | float | double	class
F → T f	field
M → T m(T this, T x) {T y S; return y; }	method
S → S; S | x = y | x = y + z | x = y * z | x.f = y | x = y.f | x = y.m(z) | x = new B Q C() | while (B) { S } | if (B) { S } else { S }	statement
B → x ≤ y | x > y | x == y	boolean expression
Q → *sen* | *poly* | *clr*	qualifier
T → Q C	qualified type

Figure 2.1: The JSec program syntax.

The syntax gives us the syntactic structure of the program, and the next step is to construct typing rules for every syntactic construct. In this paper, trivial rules for methods, classes, and programs have been omitted as our contribution has to do with information flow through statements only. Syntactic constructs for loops and conditionals have been included to allow a broader class of programs to be

typed, although because JSec does not handle implicit flow, typing rules for these constructs have been omitted.

2.3 Typing Rules

JSec's typing rules enforce the restriction that information cannot flow from *sen* variables to *clr* variables. If a statement meets the premises below the line, the conditions above will be applied to that statement. A statement that meets the conditions of every typing rule that applies to it is considered type safe. A statement that fails to meet a condition of a typing rule that applies to it produces a type error.

$$\text{TNEW} \quad \frac{\Gamma(x) : \tau_x \qquad \tau_y <: \tau_x}{\Gamma \vdash x = \text{new } \tau_y \text{ C}}$$

$$\text{TASSIGN} \quad \frac{\Gamma(x) : \tau_x \qquad \Gamma(y) : \tau_y \qquad \tau_y <: \tau_x}{\Gamma \vdash x = y}$$

$$\text{TWRITE} \quad \frac{\Gamma(x) : \tau_x \qquad \Gamma(y) : \tau_y \qquad \Gamma(f) : \tau_f \qquad \tau_y <: \tau_x \rhd \tau_f}{\Gamma \vdash x.f = y}$$

$$\text{TREAD} \quad \frac{\Gamma(x) : \tau_x \qquad \Gamma(y) : \tau_y \qquad \Gamma(f) : \tau_f \qquad \tau_y \rhd \tau_f <: \tau_x}{\Gamma \vdash x = y.f}$$

$$\text{TCALL} \quad \frac{\begin{array}{c}\Gamma(m) : \tau_{this}, \tau_p \rightarrow \tau_{ret} \qquad \Gamma(x) : \tau_x \qquad \Gamma(y) : \tau_y \qquad \Gamma(z) : \tau_z \\ \tau^i \rhd \tau_{ret} <: \tau_x \qquad \tau_z <: \tau^i \rhd \tau_p \qquad \tau_y <: \tau^i \rhd \tau_{this}\end{array}}{\Gamma \vdash x = y.m^i(z)}$$

Figure 2.2: JSec's basic typing rules. Γ represents the typing environment, which is the mapping between variables and type qualifiers.

The TNEW rule applies to statements where new objects are instantiated. In practice, the *new* keyword and much of the class definition can sometimes be

abstracted away by a constructor function, so a statement like $int\ x = 10$; would be covered by this rule as well. It enforces the expected subtyping rule: that the object being instantiated must be a subtype of the left-hand side variable. The TASSIGN rule performs the same function for variable assignments: the RHS must be a subtype of the LHS.

TREAD and TWRITE utilize viewpoint adaptation as described in §2.1.1. Both rules adapt the field f from the viewpoint of the receiver y. From there, subtyping is enforced as in TNEW and TASSIGN.

For simplicity, we will also treat array elements as fields so that TWRITE will apply to the statement $x[z] = y$ and TREAD will apply to $x = y[z]$. When applying the rule in this way, it is important to remember that the array element $x[z]$ is the field, not the array index z.

TCALL is the most complicated rule, and it utilizes a variable i called the *callsite context*. This variable represents the context of the method call statement, and its viewpoint is used to adapt the method's parameters. TCALL's premise makes *this* explicit and assumes exactly one other formal parameter, but without loss of generality the rule can be applied in other cases.

Given a method call $x = y.m^i(z)$, TCALL adapts method m's parameters *this* and p and return value *ret* from the viewpoint of the callsite context. TCALL's first condition enforces the usual subtyping constraint: the (adapted) return value of the method call must be a subtype of the left-hand side of the statement. The next two conditions ensure that arguments are subtypes of formal parameters.

We must also account for known issues with subtyping when dealing with mutable references [11]. If the left-hand side of an assignment statement is mutable, we apply an equality constraint instead of a subtyping constraint. For example, if we apply TASSIGN to a statement $x = y$ where x is mutable, we enforce $\tau_y = \tau_x$

instead of $\tau_y <: \tau_x$.

2.4 Type Inference

Given a program, typing rules, and a set of sources, a *type inference* algorithm derives a valid typing for the program. This typing consists of a mapping from variables to type qualifiers that conforms to our typing rules. Note that JSec's type inference algorithm will never fail to produce such a typing. The algorithm's default case would be to simply assign the *sen* qualifier to every variable in the program, which constitutes a valid typing.

We compute a solution by initially assigning each variable a set of possible valid type qualifiers. For most variables, this set contains all three qualifiers: {*sen*, *poly*, *clr*}. For fields, however, the *sen* type is removed. This is because fields are always adapted from the viewpoint of their parent object, so for the field to be treated as *sen*, its actual type will be *poly* and its parent will be *sen*. Then, through viewpoint adaptation, $sen \triangleright poly = sen$.

After generating the initial qualifier sets, we apply typing rules statement-by-statement. Qualifiers are removed from sets if they don't satisfy the typing rule being applied. Note that for a qualifier τ to be removed from a variable x's set, there must be no valid assignment of types for which the statement type checks if the x is type τ.

A few circumstances exist in which a typing generated in this way will fail to remove all necessary qualifiers from a set. To resolve final conflicts of this kind, we impose additional constraints on the program called *method summary constraints*. These constraints link parameters to return values by searching for a transitive chain of subtyping from the parameter to the return value.

The constraint that is produced connects the argument (not the formal pa-

```
1  procedure SummarySolver
2    repeat
3      for each c in C do
4        SolveConstraint(c)
5        if c is τx <: τy ▷ τf and S(f) is {poly} then
6          Add τx <: τy into C
7        else if c is τx ▷ τf <: τy and S(f) is {poly} then
8          Add τx <: τy into C
9        else if c is τx <: τy then
10         for each τy <: τz in C do Add τx <: τz to C end for
11         for each τw <: τx in C do Add τw <: τy to C end for
12         for each τw <: τ^i ▷ τx in C and τi ▷ τy <: τz do
13           Add τw <: τz to C
14         end for
15       end if
16     end for
17   until S remains unchanged
18 end procedure
```

Figure 2.3: JSec's method summary constraint algorithm. It is adapted from [1].

rameter) to the LHS of the call assignment (not the return value). The algorithm that generates these constraints can be seen in Figure 2.3.

This algorithm handles four cases. Cases 1 and 2, on lines 5 and 7 respectively, add $\tau_x <: \tau_y$ to C because $\tau_y \triangleright poly$ always produces τ_y. Case 3 (on line 9) infers constraints based on transitivity; for example, if $\tau_x <: \tau_y$ and $\tau_y <: \tau_z$, we know that $\tau_x <: \tau_y$. Case 4 (line 12) finally connects method arguments with the left-hand-side of the method call assignment statement. If there is flow from the argument to a parameter, from the parameter to the return value, and from the return value to the LHS, this case enforces the constraint that the LHS is a subtype of the argument.

After every statement fully type checks and the algorithm runs, variables may still have more than one valid qualifier in their sets. In this case, the final qualifier is chosen according to the following ranking: $clr > poly > sen$.

```
1  class Identity {
2      int v, w, x, z;
3      sen int y;
4
5      public static void main(String []args) {
6          x = y;
7          z = id(x);
8          w = id(v);
9      }
10
11     int id(int i) {
12         return i;
13     }
14 }
```

Figure 2.4: Sample Java code for basic type inference.

For the example programs in this section, we have applied the inference algorithm by hand to demonstrate how it works. Although we process each statement only once, it should be noted that the actual inference algorithm will likely run through a program's statements multiple times before producing a final typing.

Each unqualified variable begins with a full qualifier set. Therefore, v, w, x, z, and i all map to the qualifier set $\{sen, poly, clr\}$ while y maps to the set $\{sen\}$. We will use the notation $S(a)$ to indicate a's qualifier set. On line 6, the TASSIGN rule states that τ_y must be a subtype of τ_x. Since the only qualifier in $S(x)$ that meets this criteria is *sen*, the *poly* and *clr* qualifiers are removed.

On line 7, we apply TCALL. We first consider the last two constraints: $\tau_x <: \tau^7 \rhd \tau_i$ and $\tau^7 \rhd \tau_{ret} <: \tau_z$. Since τ_x is *sen*, *clr* will be removed from $S(i)$ but *poly* will remain. This is because if τ_i is *poly* and τ^7 is *sen*, viewpoint adaptation will produce the type *sen* which meets our constraints. We must also consider the constraints $\tau_i <: \tau_{ret}$, which removes *clr* from $S(\tau_{ret})$, and $\tau_x <: \tau_z$, which assigns τ_z to *sen*.

TCALL is applied again on line 8. Consider the constraint $\tau_v <: \tau^8 \rhd \tau_i$. $S(v)$ contains all three qualifiers, meaning there is no restriction on τ_i. $S(w) =$

$\{sen, poly, clr\}$; this is unchanged by the constraint $\tau^8 \rhd \tau_{ret} <: \tau_w$ since $\tau^8 \rhd \tau_{ret}$ can be *sen*, *clr*, or *poly*.

Once this line-by-line inference is completed, any sets containing more than one qualifier will be reduced based on the preference hierarchy *clr* > *poly* > *sen*. Since $S(w) = S(v) = \{sen, poly, clr\}$, w and v will be *clr*. Since $S(i) = \{sen, poly\}$, i will be *poly*. Since $S(x) = S(y) = S(z) = \{sen\}$, all three variables are *sen*.

Let's look at another, more complex example (Figure 2.5). We will begin in the $main()$ function. On line 29, we have a call to $min\_list()$. We will apply TCALL: the constraint $\tau_{list1} <: \tau^{29} \rhd \tau_{list}$ removes *clr* from $S(list)$ (since τ_{list1} is *sen*). The function call on the following line is handled in the exact same way.

On line 11, $S(list) = \{sen, poly\}$. On line 12, we apply TREAD which states that $\tau_{list} \rhd \tau_{list[0]} <: \tau_{min}$. This removes *clr* from $S(min)$. Nothing is removed from $S(i)$ on line 13, and on line 15 the TREAD constraint $\tau_{list} \rhd \tau_{list[i]} <: \tau_{min}$ also checks out.

Now that we have typed the statements in the function, we can finish the TCALL applications. The constraints $\tau^{29} \rhd \tau_{min} <: \tau_{min1}$ and $\tau^{30} \rhd \tau_{min} <: \tau_{min2}$ are both met, meaning that $S(min1) = S(min2) = \{clr, poly, sen\}$. On line 31, we have three applications of TASSIGN from which we can see that τ_{min1}, τ_{min2}, and τ_x must all be subtypes of τ_{ans}. Since τ_x is *sen*, τ_{ans} must be *sen* as well.

On the final line of the program, we will apply TCALL to the $check\_ans()$ call. The constraint $\tau_{ans} <: \tau^{33} \rhd \tau_{val}$ removes *clr* from $S(val)$ because τ_{ans} is *sen*.

```
1  class Example{
2      sen static final int[] list1 = new int[2];
3      sen static final int[] list2 = new int[2];
4      sen static final int x = 3;
5
6      poly list1[0] = 9;
7      poly list1[1] = 3;
8      poly list2[0] = 1;
9      poly list2[1] = 7;
10
11     static int min_list(int[] list){
12         int min = list[0];
13         for(int i = 1; i<list.length; i++){
14             if (list[i] <= min){
15                 min = list[i];
16             }
17         }
18         return min;
19     }
20
21     static boolean check_ans(int val){
22         if (val == 20){
23             return true;
24         }
25         return false;
26     }
27
28     public static void main(String []args){
29         int min1 = min_list(list1);
30         int min2 = min_list(list2);
31         int ans = min1 + min2 + x;
32         ans = 2 * ans;
33         System.out.println(check_ans(ans));
34     }
35 }
```

Figure 2.5: Sample Java code for basic type inference.

Now we will compute the set-based solution to determine the final typing. $S(list) = S(min) = S(val) = \{poly, sen\}$, so $list$, min, and val are all *poly*. i is *clr*. Based on our preference hierarchy, $min1$ and $min2$ both become *clr*, but this assignment won't type check. The constraint $\tau_{29} \rhd min <: min1$ forces τ_{29} to be *clr*. But then the constraint $list1 <: \tau_{29} \rhd list$ is violated. To resolve this error, we must apply a method summary constraint.

Given the constraint $\tau_{list[0]} < \tau_{min}$, we can adapt both sides from the viewpoint of τ_{29} (because adaptation preserves subtyping). Now we have a transitive chain of subtypes: $\tau_{list1} <: \tau_{29} \rhd list[0] <: \tau_{29} \rhd \tau_{min} <: \tau_{min1}$. This implies $\tau_{list1} <: \tau_{min1}$, and this constraint sets $\tau_{min1} = sen$. This resolves our typing error and completes the type inference.

3. SECURE COMPUTATION

Once a program has completed basic typing, we must consider how to ensure the secrecy of data stored in sensitive variables. This section discusses the additions to the JSec type system that allow for confidentiality to be achieved. We will make use of a few different cryptographic primitives, but first we must define the strength and capabilities of our adversary. Our informal threat model is as follows:

We aim to hide sensitive data from an adversary who has access to the machine on which the program is running. This adversary can fully observe program execution and any stored data on disk. It can also observe all network traffic between the untrusted and trusted hosts. It does not have access to any of the information stored on the trusted host, which includes all encryption and decryption keys. We do not presently consider an active adversary who could modify code or data during execution.

To protect sensitive data from such an adversary, it must be encrypted. JSec makes use of several different encryption schemes in order to hide data through a variety of operations. However, some of these encryption schemes offer stronger security guarantees than others (see §3.2). JSec's encrypted type inference algorithm guarantees that the strongest available form of encryption will be used for a given operation.

3.1 Cryptography Background

In order to ensure that we choose the strongest available form of encryption, we must be able to evaluate and compare the strengths of our chosen cryptosystems. One common measure of the strength of a cryptosystem is *ciphertext indistinguishability*. We will briefly present the three formal definitions for indistinguishability security in order of increasing strength. Formal definitions for security are typically

represented as games where the cryptosystem is considered secure if no adversary can win with greater probability than guessing at random.

- **Indistinguishability under chosen-plaintext attack (IND-CPA)**
  In the IND-CPA security game, an adversary is allowed to encrypt a polynomially bounded number of messages. It must then submit two distinct plaintexts to a party known as the challenger. The challenger chooses one of the plaintexts at random, encrypts it, and sends it back to the adversary. The adversary may perform as many additional encryptions as desired, then must guess which of the two texts it has received the in encrypted form.

- **Indistinguishability under chosen ciphertext attack (IND-CCA1)**
  The IND-CCA1 security game is very similar to the previous game. The difference is that the adversary may not only encrypt arbitrary values but also decrypt as many ciphertexts as they like. They then submit two plaintexts to the challenger as in IND-CPA. However, after receiving the single challenge ciphertext in response, the adversary may not perform any further decryptions. It must then guess which of the two texts it has received in encrypted form.

- **Indistinguishability under adaptive chosen ciphertext attack (IND-CCA2)**
  The IND-CCA2 security definition includes the strongest adversary. The game is the same as IND-CCA1 up until the adversary receives the challenge ciphertext. This adversary is adaptive and at this point may decrypt any ciphertexts it chooses except for the challenge ciphertext. It must then guess which plaintext corresponds to the challenge ciphertext.

3.2 Type Qualifiers

Each type qualifier used in the second stage of JSec corresponds to a specific cryptosystem. Every variable of a given type will be encrypted with that type's corresponding encryption scheme. This section describes these types and encryption schemes.

- **Randomized encryption (*rnd*)**

  In JSec, randomized encryption offers the strongest security guarantee. In such an encryption scheme, the likelihood that two plaintexts will be encrypted to the same ciphertext is negligible. We could use a construction that attains the strongest form of indistinguishability security, IND-CCA2, but this is unnecessary. Our adversary does not have the ability to tamper with code or data, so CCA2's adaptive security is superfluous. An IND-CCA1 scheme such as AES-CBC will suffice. A random initialization vector and key will be generated and stored by the trusted host.

  No operations are supported by this type of encryption. Therefore, only sensitive variables that undergo no computation will be assigned to this type.

- **Additively homomorphic encryption (*ahe*)**

  The next strongest security guarantee is offered by additively homomorphic encryption. The Paillier cryptosystem is randomized, like the *rnd* type, but it achieves a slightly weaker indistinguishability definition. Specifically, it achieves IND-CPA security. An adversary who submits two messages for Paillier encryption and receives one ciphertext back cannot ascertain which message has been encrypted with greater probability than $\frac{1}{2}$.

  It is possible to multiply two Paillier ciphertexts together to generate a ciphertext that decrypts the sum of the original two plaintexts. That is, $enc(x) * enc(y) = enc(x + y)$. The private and public keys are used for encryption and decryption, respectively, and can be stored by the trusted host.

- **Deterministic (and multiplicatively homomorphic) encryption (*det*)**

  Deterministic encryption, by definition, cannot achieve any form of indistinguishability security. An adversary presented with the IND-CPA game could simply re-encrypt the original messages to determine which matches the challenge ciphertext. Because deterministic encryption leaks equality in this way, it can support equality comparisons over ciphertexts. If $enc(x) = enc(y)$, then

we know that $x = y$.

We will use the RSA cryptosystem, which is a deterministic public-key scheme. It has the added feature of being multiplicatively homomorphic. This means that it supports multiplication over encrypted values: $enc(x) * enc(y) = enc(x * y)$. The public key, which is used for encryption and the multiplication operation, must be known by the untrusted host. The private key used for decryption must be stored by the trusted host. Performing the equality check does not require the public key.

There are multiplicatively homomorphic encryption schemes that offer a stronger security guarantee than deterministic RSA. El Gamal, for example, achieves IND-CPA security. But because RSA supports both $*$ and $==$, we believe the gain in efficiency from having to change types less frequently will be worth the trade-off in security.

- **Order-preserving encryption (*ope*)**
  Order-preserving encryption provides the weakest form of security in JSec. Constructions for *ope* leak the ordering of plaintexts in addition to their equality. This allows us to compute inequality comparisons over ciphertexts.

  The OPE protocol proposed by Boldyreva et al. [12] is an order-preserving symmetric key scheme that suits our needs. A single key is used for both encryption and decryption, and it must be stored by the trusted host. The key is not required to perform order comparisons and therefore is not needed by the untrusted host.

  Although Popa et al.'s mutable OPE scheme [13] offers a stronger security guarantee than [12], its implementation is suitably complex as to be out of scope for this thesis. To encrypt new values, a time-consuming interactive protocol between the client and server is required, and occasional ciphertext

update operations must be run. Although the confidentiality guarantees of this type could be improved by implementing this scheme, this paper does not further address this.

To prepare a program to run on the untrusted host, we apply these encrypted types to variables. However, we must be able to handle the broad class of programs in which the same variable may successively undergo multiple distinct operations. When these operations require different encryption types, we have a problem.

To address this significant requirement, of note is the existence of fully homomorphic encryption [3], which supports arbitrary computation over encrypted data. One proposed solution to this problem makes use of FHE by applying it to all variables that undergo multiple operations [10]. However, there is currently no efficient scheme for fully homomorphic encryption. Therefore this solution is only viable on a small class of programs. The scheme mentioned uses Hadoop MapReduce programs as its benchmarks. Even these programs, which are broken down into a series of basic operations and thus are potentially well-suited for this solution, could only be successfully typed by such a scheme two-thirds of the time in experimental results.

Thanks to its subtyping relationship and type coercion capabilities, JSec is capable of typing the desired class of programs.

3.2.1 Subtyping and Type Coercion

The subtyping relationship between the encrypted types is as follows:

$$ope <: det <: ahe <: rnd$$

This relationship adheres to a weakest-to-strongest ordering of the types' corresponding cryptosystems. With the weakest security guarantees, *ope* is a subtype of all other types. The strongest type, *rnd*, is a supertype of all other types.

We must also define the sets of supported operations for the encrypted types. In conjunction with the subtyping hierarchy, these definitions allow JSec to choose the strongest type for a given operation.

Type	Operations
ope	$<$, $==$, $+$, $*$
det	$==$, $+$, $*$
ahe	$+$
rnd	

Figure 3.1: A table showing which operations are supported by each encrypted type.

The supported operation definitions serve an important purpose in JSec. They are not literal; for example, we know that addition cannot be performed on variables of the *det* type. However, we treat it as if it can because it is possible to change a *det* variable to a type that is capable of performing addition without compromising security.

In a broader sense, when a variable must undergo a series of operations that are supported by different types, we need some way to choose a type assignment that can perform all of the necessary operations. Consider the following code sample:

```
1  sen int x = 10;
2  if (x == 10) {
3     x = x + 1;
4  }
```

Figure 3.2: A brief Java program to demonstrate type coercion.

The variable x first undergoes an equality comparison. If the equality is true, the variable then undergoes an addition operation. We want to apply the most secure type that supports both $==$ and $+$; therefore x will be marked as *det*. However, we will need to change x's type from *det* to *ahe* in order to increment the variable on the following line. *Type coercion* allows for a subtype to be converted to a supertype for the purpose of executing a single statement. Thus, for this brief program to type check, we will assign x to *det* and insert coercion statements into

the program. These statements are discussed in §3.3.

This is why we define the types' supported operations the way we do. The *det* type must be thought of as supporting +, because it is possible to coerce it to a type that can perform this operation.

3.3 Program Syntax

In order to support encrypted types, some additions to the program syntax are necessary. Recall that the syntax defines the structure of the programs that JSec processes.

EQ $\rightarrow$ *rnd* |*mhe* | *rsa* | *ope*	encrypted qualifier
T $\rightarrow$ EQ C	qualified type
S $\rightarrow$ coerce(val,τ_{val},τ_t) | encrypt(c, τ_t)	statement

Figure 3.3: Additions to the JSec program syntax to allow for encrypted computation.

The first two rules simply add our new qualifiers to the syntax. This allows JSec to assign them to variables the same way it assigned basic type qualifiers in §2.

The third rule provides two new statements that allow for a few important functionalities. First is the type coercion statement *coerce(val,*τ_{val}*,*τ_t*)*, where *val* is a variable of type τ_{val} to be coerced to encrypted type τ_t. When this function is called, the untrusted host sends the arguments to the trusted host. The trusted host uses its locally stored keys to decrypt *val* from its existing type and re-encrypt it using type τ_t's corresponding encryption scheme. It sends the newly encrypted value back to the untrusted host, which coerces the right-hand side of the function call statement to τ_t.

Consider the sample program in Figure 3.2. As we showed, JSec infers that x must be marked as *det* and coerced to *ahe* on line 3. This will be done by inserting the statement *x = coerce(x,det,ahe);* immediately before the target line and the statement *x = coerce(x,det,ahe);* immediately after the line. The trusted host will

perform its computations, and the untrusted host will coerce x to *ahe* and store the newly encrypted value in it. It will execute the addition statement, then perform the second type coercion back to *det* in the same way.

The other statements in our third rule allows for the handling of constant values. Consider again the sample program in Figure 3.2, in which the variable x is incremented by 1. We can only perform an addition operation when both operands are encrypted by our additively homomorphic encryption scheme. We know that x will be encrypted properly because it will be coerced to *ahe*, but the constant value will need to be encrypted separately.

The new construct *encrypt(c, τ_t)* allows this to take place. We will insert this statement in place of the constant value; c is the constant to be encrypted and τ_t is the type that corresponds to the necessary encryption scheme.

3.4 Typing Rules

In this type system, a type safe statement is one whose left and right-hand sides both support the operation being performed. As with the basic types, typing rules are necessary to ensure type safety. However, these rules and their application are much simpler than the basic typing rules.

Recall that if a type supports an operation, we think of all of its subtypes as supporting that operation as well. When the time comes to actually perform the operation, we can coerce a subtype to the supertype that can actually compute the operation. This notion is what informs our typing rules. In general, they state that both sides of a statement must be subtypes of the type that is actually capable of performing the operation.

$$\text{TORDER:}\quad \frac{\Gamma(x) : ope \qquad \Gamma(y) : ope}{\Gamma \vdash x \leq y : boolean}$$

$$\text{TEQUAL:}\quad \frac{\Gamma(x) : \tau_x \qquad \Gamma(y) : \tau_y \qquad \tau_x <: det \qquad \tau_y <: det}{\Gamma \vdash x == y : boolean}$$

$$\text{TMULT:}\quad \frac{\Gamma(x) : \tau_x \qquad \Gamma(y) : \tau_y \qquad \tau_x <: det \qquad \tau_y <: det}{\Gamma \vdash x * y : det}$$

$$\text{TADD:}\quad \frac{\Gamma(x) : \tau_x \qquad \Gamma(y) : \tau_y \qquad \tau_x <: ahe \qquad \tau_y <: ahe}{\Gamma \vdash x + y : ahe}$$

$$\text{TCOERCE:}\quad \frac{\Gamma(x) : \tau_x \qquad \tau_x <: rnd \qquad \tau_t <: rnd}{\Gamma \vdash x = coerce(x, \tau_t) : \tau_t}$$

Figure 3.4: JSec's encrypted typing rules.

TORDER enforces type restrictions for order comparisons. Because *ope* has no subtypes, both sides of the comparison operation must be *ope* in order for the statement to type check. The output of the statement will be a boolean value in unencrypted form.

The next rule, TEQUAL, ensures that equality comparisons are type safe. In order for the statement to type check, both operands must be subtypes of *det*. Like TORDER, the output of the comparison is an unencrypted boolean value.

Because the same encrypted type handles equality comparisons and multiplication operations, TMULT is essentially the same as TEQUAL. Both sides of the operation must be subtypes of *det* for the statement to type check. Similarly, TADD requires both sides of an addition operation to be subtypes of *ahe*.

The final typing rule governs the type coercion statements defined in our program syntax. TCOERCE enforces the restriction that both the source and destination types of the coercion are subtypes of *rnd*. Since all encrypted types are subtypes of *rnd*, the rule essentially ensures that the statement is coercing from an encrypted type to an encrypted type.

We must also be able to infer encrypted flow through assignment and function call statements. For this purpose, we can reuse the typing rules for basic inference (Figure 2.2) with a few alterations. TNEW and TASSIGN can stay the same, but the other rules no longer need to be context sensitive as we do not have a polymorphic encrypted type. The modified inference rules are as follows:

$$\text{TWRITE\_ENC} \quad \frac{\Gamma(x) : \tau_x \qquad \Gamma(y) : \tau_y \qquad \Gamma(f) : \tau_f \qquad \tau_y <: \tau_f}{\Gamma \vdash x.f = y}$$

$$\text{TREAD\_ENC} \quad \frac{\Gamma(x) : \tau_x \qquad \Gamma(y) : \tau_y \qquad \Gamma(f) : \tau_f \qquad \tau_f <: \tau_x}{\Gamma \vdash x = y.f}$$

$$\text{TCALL\_ENC} \quad \frac{\begin{array}{cccc} \Gamma(m) : \tau_{this}, \tau_p \rightarrow \tau_{ret} & \Gamma(x) : \tau_x & \Gamma(y) : \tau_y & \Gamma(z) : \tau_z \\ \tau_{ret} <: \tau_x & \tau_z <: \tau_p & \tau_y <: \tau_{this} & \end{array}}{\Gamma \vdash x = y.m^i(z)}$$

Figure 3.5: Additional encrypted typing rules.

3.5 Type Inference

As with the basic types, a type inference algorithm applies the rules to a program one statement at a time. The algorithm we use is the same standard set-based algorithm demonstrated during basic typing. This section describes and demonstrates the application of this algorithm.

Each variable in the program maps to a set of qualifiers. At the beginning of type inference, every variable x is mapped to the full set $S(x) = \{rnd, ahe, det, ope\}$.

As typing rules are applied, subtyping constraints force qualifiers to be removed from the sets. When the algorithm terminates, each variable chooses its type from its corresponding set according to the following preference hierarchy: $rnd > ahe > det > ope$. This hierarchy enforces the intuition that we want to choose the strongest possible security type for each variable.

Once the algorithm chooses the type for each variable, we must iterate over the program one final time to insert coercion statements. As discussed in §3.2.1, these are wrapped around statements that require one or both operands to change type. After inserting these statements, the code sample shown in Figure 3.2 would look as follows:

```
1  det int x = 10;
2  if (x == 10) {
3      coerce(x, ahe);
4      x = x + encrypt(1, ahe);
5      coerce(x, det);
6  }
```

Figure 3.6: Sample Java code to demonstrate the placement of type coercion statements.

We will now demonstrate encrypted type inference on a sample program. Recall the program that we applied basic type inference to in Figure 2.4. Basic typing left us with the following type qualifier assignments:

sen:	x, y, z
poly:	i
clr:	v, w

Because we are only concerned with *sen* variables, this typing is very simple. All three variables x, y, and z will begin with the full set {*rnd*, *ahe*, *det*, *ope*}. Because no operations are performed on any of those variables, our typing rules will never be invoked. At the end of inference, all three variables will be marked as *rnd*

because it is the most preferential type.

Let's try a non-trivial example: the program in Figure 2.5. Our basic type assignment was as follows:

sen: $list1$, $list2$, x, ans
poly: $list$, min, $list1[0]$, $list1[1]$, $list2[0]$, $list2[1]$, $min1$, $min2$
clr: i

As discussed earlier, *poly* variables are duplicated, creating a *sen* copy and a *clr* copy. When applying rules to *poly* variables during this typing, we will be considering only the *sen* version of the variable.

We will begin by applying TORDER on line 14. By doing so, we see that min and $list[i]$ must both be subtypes of *ope*. This removes all other encrypted qualifiers from their sets. On line 15 we apply TASSIGN, and the constraint $\tau_{list[i]} <: \tau_{min}$ checks out.

On line 31, we apply TADD to see that τ_{min1}, τ_{min2}, and τ_x must be subtypes of *ahe*. This removes *rnd* from all three of the variables' sets. By TASSIGN, ans cannot be *ahe* either. On the following line, TMULT enforces $\tau_{ans} <:$ *det*, so $S(ans) = \{ope, det\}$.

On line 33, the TCALL_ENC constraint $\tau_{ans} <: \tau_{val}$ is met. Inside the function call, TEQUAL forces τ_{val} to be a subtype of *det*, making $S(val) = \{ope, det\}$.

We will now reduce qualifier sets based on the following preference hierarchy: $rnd > ahe > det > ope$. $S(min1) = S(min2) = S(x) = \{ope, det, ahe\}$, so all three variables will be set to *ahe*. $S(ans) = S(val) = \{ope, det\}$, so τ_{ans} and τ_{val} will be *det*.

The final step in the inference algorithm is coerce variables where needed. Recall that this process involves placing coercion statements before and after the statement in question. This example only requires coercion is two locations: lines 15 and 32. Because min and $list[i]$ are *ope*, on line 15 they must both be coerced

to *det* in order to perform the equality comparison. Likewise, on line 32, *ans* must be coerced from *det* to *ahe* to allow for the addition operation.

4. IMPLEMENTATION & FUTURE WORK

This thesis is primarily concerned with laying the theoretical groundwork for building an encrypted type system for general Java programs. While we have demonstrated the design of such a scheme, issues of implementation have been largely left to future work. This is for two reasons: first, because we felt it was important to maintain a reasonable scope for this thesis. With so little relevant work available, many minute details had to be addressed on the theoretical side before an implementation could even be discussed.

The second reason that development was not a priority is a lack of meaningful benchmarks on which to base an implementation. Good benchmark sets should contain a wide variety of programs with complicated flows and other edge cases. The only existing benchmark set that we could potentially adapt to our purposes is from MrCrypt [10], but the class of programs they are selected from is extremely narrow and would not produce particularly meaningful results for our scheme. A reasonable first step towards implementation in future works would be to adapt these benchmarks for JSec, but ideally a new set of more comprehensive benchmarks should be developed.

Although it was not a priority, steps have been made towards implementing our scheme. The first stage of JSec has been fully implemented using the type inference and checking framework established in [14]. This framework is built on top of the Checker Framework [15]. As we demonstrated by hand in §2.4, the framework takes Java source code as input, generates constraints by applying typing rules, then computes the set-based solution. It then outputs the final typing.

Development on the second stage is underway but will be a substantial undertaking. The first step would be to load the type inference engine with the encrypted types and associated typing rules. Although we did our best to cover edge cases

and side effects of Java operations, it is likely that significant debugging would be necessary in this stage. Once it is possible to produce a valid encrypted typing, the next step would be to implement the trusted host. This would require a network connection and protocols for the coercion, encryption, and decryption operations as described in §3.3. Finally, the trusted host would need to implement each of the cryptosystems necessary for encrypted computation. Open-source implementations of all four systems are available: AES-CBC and RSA are included in the Java Cryptography Architecture [16] while Paillier and OPE implementations can be found in CryptDB [8].

4.1 Optimization

Although we do not have efficiency estimates or benchmark results at this time, it is clear that the scheme we have described will impose significant overhead on the program's execution time. The largest source of overhead will likely be the trusted host, as encryption and decryption computations are quite expensive. Therefore, optimization strategies should seek to minimize the number of calls to the trusted host. We will briefly discuss a few ways to achieve this.

The first and most trivial optimization would be to combine duplicate coercion statements wherever possible. For example:

```
1 sen int x = 10;
2 if (x == 10){
3     x = x + 1;
4     x = x + 1;
5 }
```

Our inference algorithm would assign x as *det*. It would then coerce x to *ahe* for the first addition operation, then coerce it back to *det*. The same process would be performed for the second addition operation. Obviously the second coercion is

unnecessary, and a basic optimization algorithm would scan through the program after all coercion statement have been inserted and combine statements when possible.

Another optimization technique would be to cache encrypted values when they are being coerced to an immutable type. Since the value of the immutable variable will not change, the old value does not need to be refreshed when coercing back to the original type. We can simply restore the cached value, saving a trip to the trusted host.

At present, the only immutable type in JSec is *ope*, so the benefits of this optimization would be minimal. Equality comparisons do not require mutability, but our *det* type also handles multiplication, which does require mutability. The *det* type could be made immutable if we created a new type that handled multiplication and only used *det* for comparisons. Experimentation would be required to determine how much the efficiency gains from this optimization would be offset by the overhead of including a new type.

A final optimization would make use of a helpful feature of the Paillier cryptosystem to reduce interaction with the trusted host. Specifically, it is possible to transform a plaintext value and a Paillier ciphertext into a ciphertext that encrypts the sum of the two values. Given g as part of the public key, $enc(x) * g^y = enc(x+y)$. By applying this transformation, we could add unencrypted constant values or cleartext variables to *ahe* variables without involving the trusted host.

5. RELATED WORKS

A variety of solutions to the general problem of secure computation are presently being explored by researchers. This is not surprising given the assortment of cryptographic tools available and the vast landscape of architectures in which code needs to be securely executed. Some solutions, for example, are specific to web apps [9] or database languages like SQL [8]. Cryptographic techniques such as fully homomorphic encryption [7] and multi-party computation [4] have achieved strong theoretical results but little in the way of practical, implementable schemes.

This section will briefly survey fields related to JSec: encrypted computation, program partitioning, information flow security, and security typing systems.

5.1 Cryptographic Techniques

In cryptography literature, the problem of secure computation is defined a little differently than it is in programming languages literature. This is because cryptographic research generally deal with abstractions like mathematical functions, Turing machines, or circuits. The high degree of abstraction makes many cryptographic protocols too impractical for use in real software. However, there are some exceptions to this rule, and many of the impractical results are still significant and relevant.

The most general cryptographic secure computation problem is known as *secure multi-party computation* (MPC). In an MPC protocol, each party $P_1, \ldots, P_n$ has its own private input $x_1, \ldots, x_n$. The goal is to compute some function of these inputs $y = f(x_1, \ldots, x_n)$, but the parties are mutually distrustful and wish to conceal their inputs from each other. Therefore a party P_i should learn nothing from the protocol except y and whatever additional information can be "reasonably deduced" given y and x_i.

MPC protocols typically fall into one of two categories; they can either evaluate traditional Boolean circuits or arithmetic circuits (with + and × gates). Protocols for Boolean circuits often rely on *garbled circuits*, a secure two-party computation scheme created by Andrew Yao in 1982. This scheme is only secure against honest-but-curious adversaries, who follow the specifications of the protocol correctly but will attempt to learn any information available. An active or malicious adversary who tampers with the protocol could defeat the scheme. An efficient two-party computation protocol based on Yao's garbled circuits has been demonstrated [17].

Protocols for arithmetic circuits typically make use of additively homomorphic encryption schemes. Efficient protocols have been constructed this way, notably including the BeDOZa protocol [18]. This protocol involves some pre-processing computations that can be evaluated before the inputs or functions are even specified. This cleverly allows the evaluation of the actual function to be performed without requiring any cryptographic operations. It is therefore very efficient.

As discussed earlier in this paper, *fully homomorphic encryption* (FHE) allows for evaluation of arbitrary arithmetic circuits over encrypted data. The best known FHE system was developed by Craig Gentry in 2009 and made use of a mathematical construct called ideal lattices [3]. Many schemes following in Gentry's footsteps have been developed ([19], [20], and others), but they all share significant efficiency problems. The efficiency of an FHE scheme is primarily measured by three factors: ciphertext lengths, key lengths, and evaluation overhead, which is defined the difference between the time required to evaluate the encrypted circuit and the time required to evaluate the same circuit in plaintext form. For many schemes based on Gentry's work, this evaluation overhead is significant [21].

The study of the related field of *functional encryption* was initiated by Boneh et al. in 2010 [7]. Informally, a functional encryption scheme is capable of producing keys that enable the key holders to evaluate specific functions over encrypted data. The scheme should ensure that the key holders learn nothing about the data other

than the output of the function.

A groundbreaking construction for functional encryption was proposed in 2013 [22]. However, it makes use of computationally expensive tools like fully homomorphic encryption as building blocks, so it is extremely inefficient.

The field of *program obfuscation* seeks to transform a program's code in such a way that it becomes "unintelligible" or difficult to analyze. Notable theoretical results include *virtual black-box obfuscation* (VBB) [23]. Any information that can be determined given access to the source code of a VBB-obfuscated program could also be determined given input/output access to the original program.

Similarly to the functional encryption construction described above, a groundbreaking but inefficient candidate construction for VBB obfuscation has been proposed [24].

5.2 Language-based Techniques

Language-based solutions for secure computation typically utilize information flow control to quarantine sensitive data to trusted areas. This is most often done through a *security type system*, which assigns security types to program components and imposes rules to enforce some security guarantees over those types. JSec can be thought of as a security type system. Popa et al. [5] first proposed combining type-based tools like security type systems with encryption to achieve secure computation for general programs.

The static analysis tool MrCrypt [10] followed up on that proposal. MrCrypt secures Java programs by applying a set of encryption schemes that support different operations. This is similar to the second stage of JSec. However, without inferring sensitivity from a set of sources or including a polymorphic type, MrCrypt must encrypt the entire program. This means that the tool is unable to handle programs that include operations unsupported by its types. Over its benchmarks, which are

Hadoop MapReduce programs, MrCrypt achieves a successful typing two-thirds of the time.

Secure computation solutions that are more domain-specific have achieved more notable success. For example, CryptDB [8] allows for efficient computation of most SQL queries over encrypted databases. It does so by breaking queries up into a series of primitive operators such as comparisons, sums, and joins. Each of those simple operations can be performed over encrypted data, allowing the query to run seamlessly. The overhead is very low: only a 14% to 26% reduction in throughput.

The Jif [25] project is another successful secure computation scheme. Jif provides a security typing framework for Java that supports information flow control and access control. It allows for participants in a distributed computation scheme to express ownership of and restrictions on pieces of data. Static information flow policies enforce these conditions and ensure confidentiality and integrity of data.

Built on top of Jif, the Jif/split compiler [6] is a Java program partitioning framework that is similar to JSec in concept. It automatically partitions code into subprograms that can be run securely in a distributed setting. Each machine in the distributed network runs one of these subprograms, which are partitioned to enforce Jif's data restrictions. A partitioning is considered secure if the security of a participant's data can only be compromised by hosts the participants has declared as trusted.

The primary difference between JSec and Jif/split is that the latter only uses encryption in the communication channels between hosts. Confidentiality is ensured solely by the partitioning - data that a given participant is not authorized to view will never be computed over on their machine.

Swift [9] is another domain-specific secure computation scheme. Using Jif's security-typing framework, Swift secures web applications by partitioning them into

a Java program, which runs client-side, and a Javascript program, which runs server-side. By only running security-critical code on the server, Swift maximizes the size of the client partition.

LITERATURE CITED

[1] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," Tech. Rep., Rensselaer Polytechnic Institute, Troy, NY, USA, 2014.

[2] R. Ostrovsky, *Software Protection and Simulation on Oblivious RAMs*. PhD thesis, Dept. Comp. Sci., Mass. Inst. of Tech., Cambridge MA, 1992.

[3] C. Gentry, *A fully homomorphic encryption scheme*. PhD thesis, Dept. Comp. Sci., Stanford Univ., Stanford CA, 2009.

[4] A. C. Yao, "Protocols for secure computations," in *Proc. of SFCS '82*, 1982, pp. 160-164.

[5] M. Shah, E. Stark, R. A. Popa, and N. Zeldovich, "Language supp.ort for efficient computation over encrypted data," in *Proc. of Off the Beaten Track '12*, 2012.

[6] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Untrusted hosts and confidentiality: Secure program partitioning," in *Proc. of SOSP '01*, 2001, pp. 1 - 14.

[7] D. Boneh, A. Sahai, and B. Waters, "Functional encryption: Definitions and challenges," in *Proc. of TCC '11*, 2011, pp. 253-273.

[8] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *SOSP '11*, 2011, pp. 85-100.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web app.lications via automatic partitioning," in *Proc. of SOSP '07*, 2007, pp. 31-44.

[10] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein, "MrCrypt: Static analysis for secure cloud computations," in *Proc. of OOPSLA '13*, 2013, pp. 271-286.

[11] A. C. Myers, J. A. Bank, and B. Liskov, "Parameterized types for java," in *Proc. of POPL '97*, 1997, pp. 132-145.

[12] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *Proc. of Eurocrypt '09*, 2009, pp. 224 - 241.

[13] R. A. Popa, F. H. Li, and N. Zeldovich, "An ideal-security protocol for order-preserving encoding," in *Proc. of SOSP '13*, 2013, pp. 463-477.

[14] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst, "Inference and checking of object ownership," in *Proc. of ECOOP '12*, 2012, pp. 181-206.

[15] M. Papi, M. Ali, J. T. L. Correa, J. Perkins, and M. Ernst, "Practical pluggable types for java," in *Proc. of ISSTA '08*, 2008, pp. 201-212.

[16] Oracle, "*Java Cryptography Architecture (JCA) Reference Guide*." Oracle Java SE Documentation Manual, Available: http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.htm. Retrieved on: March 2, 2015.

[17] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," in *Proc. of ASIACRYPT '09*, 2009, pp. 250 - 267.

[18] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias, "Semi-homomorphic encryption and multiparty computation," in *Proc. of Eurocrypt '11*, 2011, pp. 169–188.

[19] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proc. of Eurocrypt '10*, 2010, pp. 24–43.

[20] N. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Proc. of PKC '10*, 2010, pp. 420-443.

[21] V. Vaikuntanathan, "Computing blindfolded: New developments in fully homomorphic encryption," in *Proc. of FOCS '11*, 2011, pp. 5-16.

[22] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *Proc. of FOCS '13*, 2013, pp. 40 - 49.

[23] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *Proc. of Crypto '01*, 2001, pp. 1-18.

[24] Z. Brakerski and G. N. Rothblum, "Virtual black-box obfuscation for all circuits via generic graded encoding," in *Proc. of TCC '14*, 2014, pp. 1-25.

[25] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic, "*Jif: Java information flow*." Software release. Available: http://www.cs.cornell.edu/jif, July 2001. Retrieved on: March 2, 2015.