JCrypt: Towards Computation over Encrypted Data

Yao Dong
Rensselaer Polytechnic Institute
Troy, New York
dongy6@rpi.edu

Ana Milanova
Rensselaer Polytechnic Institute
Troy, New York
milanova@cs.rpi.edu

Julian Dolby
IBM Thomas J. Watson Research Center
Yorktown Heights, New York
dolby@us.ibm.com

ABSTRACT

Cloud computing allows clients to upload data and computation to untrusted servers, which leads to potential violations to the confidentiality of client data. We propose JCrypt, a static program analysis which transforms a Java program into an equivalent one, so that it performs computation over encrypted data and preserves data confidentiality. JCrypt minimizes computation over encrypted data. It consists of two stages. The first stage is a type-based information flow analysis which partitions the program so that only sensitive parts need to be encrypted. The second stage is an inter-procedural data-flow analysis, similar to the classical Available Expressions. It deduces the appropriate encryption scheme for sensitive variables. We implemented JCrypt for Java and showed that our analysis is effective and practical using five benchmark suites. JCrypt encrypts a significantly larger percentage of benchmarks compared to MrCrypt, the closest related work.

CCS Concepts

•Security and privacy → Information flow control; **•Theory of computation → Program analysis;**

Keywords

Information Flow, Encryption Scheme Inference, Polymorphism, Data Confidentiality, Security

1. INTRODUCTION

With the booming of internet-based business, the total number of applications developed over the Cloud has increased dramatically over the past few years. These applications have created a new challenge. Cloud-based applications usually outsource the computation and data storage to third parties such as Amazon EC2 and S3 which can lead to violations of the confidentiality and integrity of sensitive data.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPPJ '16, August 29-September 02, 2016, Lugano, Switzerland

© 2016 ACM. ISBN 978-1-4503-4135-6/16/08...$15.00

DOI: http://dx.doi.org/10.1145/2972206.2972209

One approach to address this problem is to encrypt sensitive data before sending to the cloud server and perform all computation (application functionality) back on the trusted client side [5, 16]. However, this approach is not suitable for all kinds of applications. Furthermore, it is difficult to transform an existing server-side application into this form. In addition, this approach gives away the advantage of cloud computing. A theoretical solution is fully homomorphic encryption (FHE) [6]. This cryptographic scheme makes it possible to compute arbitrary functions over encrypted data on the server. Unfortunately, current implementations of fully homomorphic encryption schemes are prohibitively expensive by orders of magnitude [3, 7, 8]. Furthermore, FHE "is a theoretical achievement" but putting it into practice requires design of circuits specific to the algorithm [23]. Therefore, execution of arbitrary programs over encrypted data remains an open problem.

Another approach is to encrypt data using specialized encryption schemes. Some specialized encryption schemes are more efficient for specific computations over encrypted data. Therefore, if some data is only involved in certain operations (e.g., addition), it can be encrypted using a specialized (and efficient) encryption scheme as in CryptDB [22] and MrCrypt [26]. Shah et al. [25] conjectured that program partitioning [2, 27] and other program analysis techniques can help (1) minimize computation on the untrusted server, (2) deduce efficient encryption schemes (for data that is involved only in operations supported by a given scheme) and (3) deduce re-encryption points (for data that is involved in multiple operations, not all supported by any given scheme).

We propose novel program analysis techniques to address this problem. Specifically, we present JCrypt, a type-based approach that *automatically* analyzes and transforms a Java program into an equivalent one that performs secure computations over encrypted data on an untrusted server. Given a Java program in which the user has marked certain variables as *sensitive*, JCrypt deduces the appropriate encryption schemes for the sensitive variables based on the operations performed on those variables. Like CryptDB [22] and MrCrypt [26], JCrypt chooses more efficient schemes instead of fully homomorphic encryption scheme to encrypt sensitive data.

JCrypt uses explicit information flow analysis and parametric polymorphism to *minimize* encryption. The information flow analysis guarantees that the sensitive source propagates to as few variables as possible. Parametric polymorphism enables *context-sensitive* methods which is instrumental for precision. In addition, JCrypt uses an Avail-

able Expressions-like analysis to infer what kind of encryption scheme (perhaps more than one) is needed of sensitive sources; this helps minimize costly conversions between encryption schemes.

There are two stages of analysis in JCrypt. The first stage is information flow analysis. It partitions the variables in the program into two parts: sensitive variables, which must be encrypted, and cleartext ones which may remain in cleartext. The key insight is that if there is information flow from the user-provided sensitive data (i.e., the *sources*) to a variable, then this variable must be encrypted. Conversely, if there is no flow from sensitive data to a variable, then this variable and its operations may remain in cleartext form. JCrypt maximizes the part that remains in cleartext.

The second stage is data-flow analysis where JCrypt determines which encryption schemes should be used to encrypt sensitive data according to the operations performed on the variables. One efficient encryption scheme only supports one specific operation, such as addition, equality check or order comparison. If the same data is involved in multiple operations, a decryption and re-encryption process, which we call *conversion*, will be needed. JCrypt can reduce and even avoid such conversions through inter-procedural data-flow analysis. The key idea is that if there is no definition of a variable between two different operations, then there is no need of conversion at the second operation. The approach is that we can provide the variable with two versions of ciphertext which are encrypted by the two corresponding encryption schemes.

JCrypt requires annotations only on *sources*. Once the sources are marked in a program, JCrypt performs completely automatic analysis without any input from the user. We have implemented JCrypt for Java programs and evaluated it on five sets of benchmarks, which include 18 general Java programs and 35 MapReduce programs. The MapReduce benchmarks are highly relevant for cloud computing. The experimental results show that our approach is effective: JCrypt can identify encryption schemes for a large number of programs (29 out of 35 MapReduce benchmarks) to perform operations on encrypted data without requiring conversions.

In summary, we make the following contributions:

- JCrypt, a novel type-based analysis. It combines information flow analysis with cryptography towards the problem of running computations over encrypted data. The advantage of information flow analysis is that it is not necessary to encrypt all data in a program. This can save the cost of encrypted computation without hurting security.
- JCrypt is context-sensitive, which allows for the same method to be applied on encrypted values and on cleartext values.
- JCrypt leverages data-flow analysis to minimize conversions between different encryption schemes.
- JCrypt handles full Java, including standard object-oriented features unlike prior work.
- Publicly available implementation and evaluation of JCrypt.

The remainder of the paper is structured as follows. Sect. 2 gives an overview of JCrypt. Sect. 3 describes the information flow analysis, and Sect. 4 describes the data-flow analysis. Sect. 5 presents the implementation and experimental results. Sect. 6 discusses related work, and Sect. 7 concludes.

```
1   public class Data {
2     poly int d;
3     poly int get(poly Data this) {
4       if (this.d < 0)
5         this.d = this.d + 1;
6       return this.d;
7     }
8     void set(poly Data this, poly int p) {
9       this.d = p;
10    }
11  }
12  public class Example {
13    public void main() {
14      sensitive Data ds = new Data();
15      sensitive int s = ...; // sensitive source
16      ds.set(s);
17      sensitive int ss = ds.get();
18      clear Data dc = new Data();
19      clear int c = ...;
20      dc.set(c);
21      clear int cc = dc.get();
22      ...
23    }
24  }
```

Figure 1: A sample Java program.

2. OVERVIEW

This section provides an illustrative example and gives an overview of JCrypt. There are two stages of analysis. The first stage is information flow analysis where JCrypt determines which variables should be marked as sensitive. Essentially, this stage partitions the variables in the program into two parts: sensitive variables, which must be encrypted, and cleartext ones which may remain in cleartext. Consider the program in Fig. 1. Only variable s of line 15 in main is marked sensitive by the user, meaning that it contains sensitive information and must be manipulated in encrypted form. Since there is *information flow* from s to ss, the variable ss becomes sensitive. Variable c is clear and cc is clear as well. Note that the method set and get are polymorphic, as they operate over a sensitive argument at line 16, and over a clear argument at line 20. In order to preserve confidentiality, s and ss must be encrypted while c and cc may remain in cleartext. More details about our information flow analysis appear in Sect. 3.

The second stage is encryption analysis where JCrypt determines which encryption schemes should be used to encrypt sensitive variables. For example, variable s flows to field d which undergoes a comparison operation in line 4. Therefore s should be encrypted by an order-preserving encryption scheme (OPE) which supports comparisons over encrypted data (but does not support addition or multiplication). In line 5, d is incremented by 1 so we should use an additively homomorphic encryption scheme (AH) which allows for the addition operation (but does not allow for

```
1  public class Data {
2    int d;
3    int d_OPE, d_AH;
4    int get_Sen(Data this) {
5      if (this.d_OPE < 0)
6        this.d_AH = this.d_AH + 1;
7      return this.d_AH;
8    }
9    void set_Sen(int[] p) {
10     this.d_OPE = p[0];
11     this.d_AH = p[1];
12   }
13   int get(Data this) {
14     if (this.d < 0) this.d = this.d + 1;
15     return this.d;
16   }
17   void set(int p) { this.d = p; }
18 }
19 public class Example {
20   public void main() {
21     Data ds = new Data();
22     int[] s = ...; // s[0] is OPE and s[1] is AH
23     ds.set_Sen(s);
24     int ss = ds.get_Sen();
25     Data dc = new Data();
26     int c = ...;
27     dc.set(c);
28     int cc = dc.get();
29     ...
30   }
31 }
```

Figure 2: The transformed program. Methods get_Sen and set_Sen are the sensitive versions of get and set. The array s has two elements which are the ciphertext encrypted by the two encryption schemes (OPE and AH). Fields d_OPE and d_AH store the two versions of the encrypted data.[2]

comparison or multiplication). It is worth noting that the same data in d is involved in two different operations. A straight-forward approach would encrypt s and d using a single encryption scheme. However, no matter which encryption scheme we use to encrypt the data, we must decrypt the ciphertext and re-encrypt it using the other scheme. We call this a *conversion* or *re-encryption*. Our goal is to *minimize and even completely eliminate conversions* because the re-encryption process is expensive. The key observation is that in this case, we can avoid conversion if we encrypted the value of sensitive s by both OPE and AH on the secure client end and then send the two versions of ciphertext to the untrusted server. This is possible because the new value of d obtained after the addition in line 5 is not needed for the comparison. Or in other words, the original OPE-encrypted value of sensitive input s is *available* at line 4. The program will choose the OPE version at line 4 and the AH version at line 5 *without any conversion*. We have developed a data-flow analysis, inspired by classical Available expressions analysis, to perform this task. The details appear in Sect. 4.

After the two stages of analysis, JCrypt transforms the original program in Fig. 1 into a functionally equivalent one which performs computation entirely over encrypted data using only efficient encryption schemes. The transformed program is shown in Fig. 2.

There are two notable parts in the transformed program of Fig. 2. The first one is that we have two versions of method set and get. The encrypted versions in line 4 and 9 are necessary to compute over the encrypted data of sensitive source s. The cleartext versions in line 13 and 17 remain the same as in Fig. 1 which are used to compute over the cleartext data in variable c. The merit of the cleartext version is that it reduces the computation overhead: first, it avoids unnecessary encryption (e.g., if set had only the encrypted version, encryption of the cleartext c becomes necessary), and second, computation over cleartext is more efficient than computation over encrypted values (c does not require encryption and can be operated in cleartext). Correspondingly, we have two versions of field d. The encrypted version in line 3 stores the ciphertext data, which is used in encrypted versions of methods get_Sen and set_Sen. The clear version in line 2 stores the cleartext data which is used in the clear versions of those methods. The second notable part is that we have two versions of encrypted field d. The OPE version d_OPE is used for comparison in line 5 while the AH version d_AH is used for addition in line 6. Therefore, no conversion is required for the encrypted data at the two different operations.

As in CryptDB [22], we use 4 efficient encryption schemes: (1) RND: a randomized encryption scheme for variables not used in any operations; (2) AH: an additively homomorphic encryption scheme for variables only used in addition operations; (3) DET: a deterministic encryption scheme for variables involved in equality checking; and (4) OPE: an order-preserving encryption scheme for variables used in comparison operations.

3. INFORMATION FLOW TYPE SYSTEM

This section describes the information flow type system and type inference of JCrypt. Given a program where a set of variables are marked as sensitive by the user (i.e, a set of *sources*), the system automatically infers types for the rest of the variables aiming to *maximize* the number of cleartext variables.

This type system is built upon our previous work on taint analysis for Android and the DFlow/DroidInfer system [12]. Both DFlow/DroidInfer and JCrypt are instances of our framework for inference and checking of pluggable types [9, 18], which we have used to define and implement many practical type-based analyses [9, 13, 10, 12].

There are important differences. DFlow/DroidInfer takes a set of *sources* and *sinks* and either types the program, which guarantees the absence of *explicit flow* from sources to sinks, or conversely, issues type errors that signal potential flow from sources to sinks. In contrast, in JCrypt there are no sinks. The goal of the information flow is to propagate the sensitive sources minimally, affecting as few variables as possible, thus maximizing the cleartext portion of the program. Sect. 3.3 states the delta to DroidInfer that is necessary to make this work.

[2]The types of all sensitive variables are not int in the real program since variables holding ciphertext could be any type depending on specific Java implementation of encryption schemes. The common cases are byte[] and BigInteger. We use int here for brevity.

$$
\begin{array}{lll}
cd & ::= \text{class C extends D } \{\overline{fd}\ \overline{md}\} & \textit{class} \\
fd & ::= t\ \mathsf{f} & \textit{field} \\
md & ::= t\ \mathsf{m}(t\ \mathsf{this},\ t\ \mathsf{x})\ \{\ \overline{t\ \mathsf{y}}\ \mathsf{s};\ \mathsf{return\ y}\ \} & \textit{method} \\
\mathsf{s} & ::= \mathsf{s};\mathsf{s} \mid \mathsf{x} = \mathsf{new}\ t() \mid \mathsf{x} = \mathsf{y} \mid \mathsf{x} = \mathsf{y.f} & \textit{statement} \\
 & \quad \mid \mathsf{y.f} = \mathsf{x} \mid \mathsf{x} = \mathsf{y.m(z)} & \\
t & ::= q\ \mathsf{C} & \textit{qualified type} \\
q & ::= \mathsf{sensitive} \mid \mathsf{poly} \mid \mathsf{clear} & \textit{qualifier}
\end{array}
$$

Figure 3: Syntax. 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, formal parameters, or parameter this. For simplicity, we assume all names are unique.

The section begins with the type qualifiers (Sect. 3.1) and proceeds to define the typing rules (Sect. 3.2) and the inference analysis (Sect. 3.3). To keep the paper self-contained, we have included some description similar to [12].

3.1 Type Qualifiers

As it is typical for type-based approaches, each variable is typed by a type qualifier. There are three qualifiers: sensitive, clear, and poly.

- sensitive: A variable x is sensitive, if there is flow from a sensitive source to x. The sensitive sources are a set of initial variables that the client marked as sensitive and these values should be in encrypted form.
- clear: A variable x is clear, if there is no flow from any sensitive sources. A clear variable holds a cleartext value.
- poly: The polymorphic qualifier is used to achieve context sensitivity. It is interpreted as sensitive in some contexts and as clear in other contexts.

The subtyping relation between the qualifiers is

$$\mathsf{clear} <: \mathsf{poly} <: \mathsf{sensitive}$$

where $q_1 <: q_2$ denotes q_1 is a subtype of q_2. For example, it is allowed to assign a clear variable to a poly or sensitive one, but it is not allowed to assign a sensitive variable to a poly or clear one.

3.1.1 Context Sensitivity

JCrypt expresses context sensitivity using the polymorphic type qualifier, poly, and *viewpoint adaptation* [4]. The concrete value of poly is interpreted by the viewpoint adaptation operation. Viewpoint adaptation of a type q' from the viewpoint of another type q, results in the adapted type q''. This is written as $q \rhd q' = q''$. Viewpoint adaptation adapts fields, formal parameters, and method return values from the viewpoint of the *context* at the field access or method call. JCrypt defines viewpoint adaptation below:

$$
\begin{array}{cccc}
_ & \rhd & \mathsf{sensitive} & = \mathsf{sensitive} \\
_ & \rhd & \mathsf{clear} & = \mathsf{clear} \\
q & \rhd & \mathsf{poly} & = q
\end{array}
$$

The underscore denotes a "don't care" value. Qualifiers sensitive and clear do not depend on the viewpoint (context).

$$
\text{(TNEW)}\ \frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}
\qquad
\text{(TASSIGN)}\ \frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}
$$

$$
\text{(TWRITE)}\ \frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}
$$

$$
\text{(TREAD)}\ \frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}
$$

$$
\text{(TCALL)}\ \frac{\begin{array}{c}\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_{\mathsf{p}} \rightarrow q_{\mathsf{ret}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{y}} <: q^i \rhd q_{\mathsf{this}} \quad q_{\mathsf{z}} <: q^i \rhd q_{\mathsf{p}} \quad q^i \rhd q_{\mathsf{ret}} <: q_{\mathsf{x}}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m}^i(\mathsf{z})}
$$

Figure 4: Typing rules for information the flow type system. Function *typeof* retrieves the JCrypt types of fields and methods, Γ is a type environment that maps variables to one of clear, poly or sensitive. Callsite qualifier q^i is the context of adaptation at call site i.

Qualifier poly depends on the viewpoint: e.g., if the viewpoint (context) is sensitive, then poly is interpreted as sensitive.

The type of a poly field f is interpreted from the viewpoint of the *receiver* at the field access. If the receiver x is sensitive, then x.f is sensitive. If the receiver x is clear, then x.f is clear.

The type of a poly parameter or return value is interpreted from the viewpoint of q^i, the context at the method call. Consider the example in Fig. 1, where method set is typed as follows:

```
void set(poly int p)
```

This enables context sensitivity because set can take as input a sensitive int as well as a clear one. poly is interpreted as sensitive at callsite 16, and as clear at callsite 20.

3.2 Typing Rules

We define our typing rules over a syntax in "named form" where the results of field accesses, method calls, and instantiations are immediately stored in a variable. The syntax is shown in Fig. 3. Without loss of generality, we assume that methods have parameter this, and exactly one other formal parameter. The JCrypt type system is *orthogonal* to (i.e. independent of) the Java type system, which allows us to specify typing rules over type qualifiers q alone.

The typing rules are defined in Fig. 4. Rules (TNEW) and (TASSIGN) enforce the expected subtyping constraints. The rules for field access, (TREAD) and (TWRITE), adapt field f from the viewpoint of *receiver* y and then enforce the subtyping constraints. Recall that the type of a poly field f is interpreted in the context of the receiver y. If the receiver y is sensitive, then y.f is sensitive. If the receiver y is clear, then y.f is clear.

The rule for method call, (TCALL), adapts formal parameters this and p and return value ret from the viewpoint of *callsite context* q^i, and enforces the subtyping constraints that capture flows from actual arguments to formal parameters, and from return value to the left-hand-side of the call

assignment.

The callsite context q^i is any of {sensitive, poly, clear}. Consider the example in Fig. 1. At callsite 17, q^{17} is sensitive and $q^{17} \triangleright$ poly is interpreted as sensitive. The following constraints generated at callsite 17 are satisfied[3]:

$$\mathsf{ds} <: q^{17} \triangleright \mathsf{poly} \quad q^{17} \triangleright \mathsf{poly} <: \mathsf{sensitive}$$

At callsite 21, q^{21} is clear and $q^{21} \triangleright$ poly is interpreted as clear. Therefore, the constraints at callsite 21 are satisfied:

$$\mathsf{dc} <: q^{21} \triangleright \mathsf{poly} \quad q^{21} \triangleright \mathsf{poly} <: \mathsf{clear}$$

We compose JCrypt with ReIm, a reference immutability type system [13]. This is necessary to overcome known issues with subtyping in the presence of mutable references [1, 24]. If the left-hand-side of an assignment (explicit or implicit) is *immutable* according to ReIm, we enforce a subtyping constraint; otherwise, we enforce an equality constraint. For example, at (TASSIGN) x = y, if x is immutable, i.e. x is not used to modify the referenced object, we enforce $q_\mathsf{y} <: q_\mathsf{x}$; otherwise, we enforce $q_\mathsf{y} = q_\mathsf{x}$. The more variables are proven immutable, the more subtyping constraints there are, and hence, the more precise JCrypt is [19].

Method overriding is handled by the standard constraints for function subtyping. If n overrides m, we require *typeof*(n) <: *typeof*(m) and thus

$$(q_{\mathsf{this}_\mathsf{n}}, q_{\mathsf{p}_\mathsf{n}} \rightarrow q_{\mathsf{ret}_\mathsf{n}}) \;<:\; (q_{\mathsf{this}_\mathsf{m}}, q_{\mathsf{p}_\mathsf{m}} \rightarrow q_{\mathsf{ret}_\mathsf{m}})$$

This entails $q_{\mathsf{this}_\mathsf{m}} <: q_{\mathsf{this}_\mathsf{n}}$, $q_{\mathsf{p}_\mathsf{m}} <: q_{\mathsf{p}_\mathsf{n}}$, and $q_{\mathsf{ret}_\mathsf{n}} <: q_{\mathsf{ret}_\mathsf{m}}$.

The type system guarantees that there is no *explicit* flow from a sensitive variable to a cleartext one. Soundness is argued exactly as in [11]. We focus on explicit flows because it enables lightweight and efficient analysis. Furthermore, implicit flows are rarely seen in our benchmarks. The only related work in this space that we are aware of, MrCrypt [26], considers explicit flows as well.

3.3 Type Inference

Given a set of sources, that is, a set of sensitive variables, type inference derives a *valid typing*, i.e. an assignment from program variables to type qualifiers that type checks with the typing rules in Fig. 4. Unfortunately, there are many valid typings. For example, one trivial (and completely useless) typing assigns sensitive to every variable in the program. We derive a typing that *maximizes* the number of clear variables.

The inference first computes a *set-based solution* S, which maps variables to sets of potential type qualifiers. Then it uses *method summary constraints*, a technique that refines the set-based solution to derive a valid typing.

3.3.1 Set-Based Solution

The set-based solution is a mapping S from variables to sets of qualifiers. For instance, if $S(\mathsf{x}) = \{\mathsf{sensitive}, \mathsf{poly}\}$, that means variable x can be sensitive or poly, but not clear. Programmer-annotated sensitive variables (i.e., the sources) are initialized to the singleton set {sensitive}. Fields f are initialized to $S(\mathsf{f}) = \{\mathsf{clear}, \mathsf{poly}\}$. All other variables and callsite contexts q^i are initialized to the maximal set of qualifiers, i.e. $S(\mathsf{x}) = \{\mathsf{sensitive}, \mathsf{poly}, \mathsf{clear}\}$.

The inference then creates constraints for all program statements according to the typing rules in Fig. 4. It takes into account the mutability of the left-hand-side of assignments as discussed in the end of Sect. 3.2. The *set-based solver* iterates over constraints c and removes *infeasible* qualifiers from the set of variables in c [13]. Consider constraint c: $q_\mathsf{y} <: q_\mathsf{x}$ where $S(\mathsf{y}) = \{\mathsf{sensitive}\}$ and $S(\mathsf{x}) = \{\mathsf{sensitive}, \mathsf{poly}, \mathsf{clear}\}$ before solving the constraint. The solver removes poly and clear from $S(\mathsf{x})$, because there does not exist a $q_\mathsf{y} \in S(\mathsf{y})$ that satisfies $q_\mathsf{y} <: \mathsf{poly}$ and $q_\mathsf{y} <: \mathsf{clear}$. The solver keeps removing infeasible qualifiers for each constraint until it reaches a fixpoint.

[3]For brevity and clarity, we omit q when dealing with variables from code examples, i.e., we write y instead of q_y.

```
1    class Example {
2      static poly int min(poly int[] list) {
3        poly int m = list[0];
4        for (clear int i = 1; i < list.length; i++) {
5          if (list[i] < m) {
6            m = list[i];
7          }
8        }
9        return m;
10     }
11     public static void main(String[] args) {
12       sensitive int[] list1 = ...; // marked by user
13       clear int[] list2 = ...;
14       sensitive int min1 = min(list1);
15       clear int min2 = min(list2);
16       sensitive int ans = min1 + min2;
17       ...
18     }
19   }
```

Figure 5: Method summary constrains example.

3.3.2 Method Summary Constraints

Unfortunately, the set-based solver reaches fixpoint before it has removed all infeasible qualifiers, and in general we cannot derive a typing from that fixpoint solution. Therefore, we derive additional constraints called *method summary constraints* to remove additional infeasible qualifiers. The algorithm for generating and solving method summary constraints is described in detail in [11]. Below, we give the intuition.

The method summary constraints "connect" the formal parameter to the return value of a method. These summary constraints are then instantiated at call sites to connect the actual argument to the left-hand-side of the call assignment. Consider the min method in Fig. 5. Based on the typing rules in Fig. 4, we have constraints list <: m and m <: ret for statements at line 3 and 6, and line 9 respectively. (In the following paragraph, we explain what happens with the subscript operator list[] at these statements.) Due do transitivity, we have list <: ret, which implies $q^{14} \triangleright \mathsf{list} <: q^{14} \triangleright \mathsf{ret}$. Therefore

$$\mathsf{list1} <: q^{14} \triangleright \mathsf{list} \quad <: \quad q^{14} \triangleright \mathsf{ret} <: \mathsf{min1}$$

which gives us list1 <: min1. The inference adds list1 <: min1, connecting the actual argument list1 and the left-hand side min1 at callsite 14 and since list1 is sensitive, min1 becomes sensitive. Similarly, the inference adds list2 <: min2 at callsite 21. Such new constraints remove additional infeasible qualifiers: in the example, this constraint removes

	Java program	Constraints	Set-based solution
1	`public class Data {`		
2	`  int d;`		$S(\mathsf{d}) = \{\mathsf{poly}\}$
3	`  int get(Data this) {`		$S(\mathsf{ret_{get}}) = \{\mathsf{clear}, \mathsf{poly}, \mathsf{sensitive}\}$
4	`    if (this.d < 0) {`		$*S(\mathsf{ret_{get}}) = \{\mathsf{poly}, \mathsf{sensitive}\}$
5	`      this.d = this.d + 1;`		$S(\mathsf{this_{get}}) = \{\mathsf{clear}, \mathsf{poly}, \mathsf{sensitive}\}$
6	`    }`		$*S(\mathsf{this_{get}}) = \{\mathsf{poly}, \mathsf{sensitive}\}$
7	`    return this.d;`	$\mathsf{this_{get}} \rhd \mathsf{d} <: \mathsf{ret_{get}}$	
8	`  }`	$*\mathsf{this_{get}} <: \mathsf{ret_{get}}$	
9	`  void set(Data this, int p) {`		$S(\mathsf{p}) = \{\mathsf{poly}, \mathsf{sensitive}\}$
10	`    this.d = p;`	$\mathsf{p} <: \mathsf{this_{set}} \rhd \mathsf{d}$	
11	`  }`	$*\mathsf{p} <: \mathsf{this_{set}}$	$S(\mathsf{this_{set}}) = \{\mathsf{poly}, \mathsf{sensitive}\}$
12	`}`		
13	`public class Example {`		
14	`  public void main() {`		
15	`    Data ds = new Data();`		$S(\mathsf{ds}) = \{\mathsf{clear}, \mathsf{poly}, \mathsf{sensitive}\}$
16	`    `**sensitive**` int s = ...;`		$S(\mathsf{s}) = \{\mathsf{sensitive}\}$
17	`    ds.set(s);`	$\mathsf{s} <: q^{17} \rhd \mathsf{p}$	$*S(\mathsf{ds}) = \{\mathsf{sensitive}\}$
18		$\mathsf{ds} = q^{17} \rhd \mathsf{this_{set}}$	
19	`    int ss = ds.get();`	$\mathsf{ds} <: q^{19} \rhd \mathsf{this_{get}}$	$S(\mathsf{ss}) = \{\mathsf{clear}, \mathsf{poly}, \mathsf{sensitive}\}$
20		$q^{19} \rhd \mathsf{ret_{get}} <: \mathsf{ss}$	$*S(\mathsf{ss}) = \{\mathsf{sensitive}\}$
21	`    Data dc = new Data();`	$*\mathsf{s} <: \mathsf{ds}$	$S(\mathsf{dc}) = \{\mathsf{clear}, \mathsf{poly}, \mathsf{sensitive}\}$
22	`    int c = ...;`	$*\mathsf{ds} <: \mathsf{ss}$	$S(\mathsf{c}) = \{\mathsf{clear}, \mathsf{poly}, \mathsf{sensitive}\}$
23	`    dc.set(c);`		
24		$\mathsf{c} <: q^{23} \rhd \mathsf{p}$	
25	`    int cc = dc.get();`	$\mathsf{dc} = q^{23} \rhd \mathsf{this_{set}}$	$S(\mathsf{cc}) = \{\mathsf{clear}, \mathsf{poly}, \mathsf{sensitive}\}$
26	`    ...`	$\mathsf{dc} <: q^{25} \rhd \mathsf{this_{get}}$	
27	`  }`	$q^{25} \rhd \mathsf{ret_{get}} <: \mathsf{cc}$	
28	`}`	$*\mathsf{c} <: \mathsf{dc}$ $*\mathsf{dc} <: \mathsf{cc}$	

Figure 6: The example to explain the type inference for information flow analysis. The first column is the Java program. The second column shows the corresponding constraints the statement generates. Those starting with * are the method summary constraints. The third column is the set-based solution. Those starting with * are the solutions updated based on method summary constraints.

qualifiers clear and poly from $S(\mathsf{list1})$.

Array access is handled as field access with the array component treated as a special field, which is standard. The "field" of array list1 is poly because sensitive values are stored into the array. (This part is elided from the code in Fig. 5.) The array reads at line 3 and 6 are handled by rule (TREAD) resulting in constraint $\mathsf{m} <: \mathsf{list} \rhd \mathsf{poly}$ which is $\mathsf{m} <: \mathsf{list}$.

When the algorithm terminates, the inference derives a concrete typing by picking up the *maximal* element of $S(\mathsf{x})$ for each variable x according to the ranking clear > poly > sensitive. This results in a typing, called a *maximal typing* [9], with a maximal number of clear variables [9]. This works towards the goal of performing as much computation on cleartext values as possible. We note that the ranking in the opposite direction of subtyping and the argument that this maximal typing type checks, are major differences with DFlow/DroidInfer.

The value of a callsite qualifier q^i is not selected according to the above ranking because this could cause the program not type checked. For example, for the method call $\mathsf{x} = \mathsf{y.m(z)}$, if $S(\mathsf{z}) = \{\mathsf{sensitive}\}$, then based on the constraint $\mathsf{z} <: q^i \rhd \mathsf{p}$, only clear can be removed from $S(\mathsf{p})$. The inference result is that $S(q^i) = \{\mathsf{sensitive}, \mathsf{poly}, \mathsf{clear}\}$ and $S(\mathsf{p}) = \{\mathsf{sensitive}, \mathsf{poly}\}$. We pick up the maximal element poly as the type of p. But if we select clear for q^i in the same way, the constraint will become $\mathsf{sensitive} <: \mathsf{clear} \rhd \mathsf{poly}$ which fails to satisfy the subtyping relation.

For each callsite qualifier q^i, we select the maximal value from $S(q^i)$ which is able to satisfy all related constrains. This is done *after* we have selected the types for all variables as described above. Thus, in the above example, this maximal value is sensitive. In theory, it is possible that no value exists for q^i to satisfy all related constraints. For example, suppose ret is poly and x is clear in constraint $q^i \rhd \mathsf{ret} <: \mathsf{x}$ for the same method call, then sensitive q^i cannot satisfy this constraint. One can handle this problem by relaxing method call constraints to $\mathsf{z} <: q^i_1 \rhd \mathsf{p}$ and $q^i_2 \rhd \mathsf{ret} <: \mathsf{x}$, i.e., q^i_1 and q^i_2 need not be the same [20]. Fortunately, in the real world programs rarely require this relaxation. In our experiments q^i always exists.

Our type inference can be useful in other type systems. For example, EnerJ [24] has a similar goal to ours. EnerJ gives each variable an approximate or precise type (an approximate variable indicates that computation over that variable can be done approximately, thus achieving energy savings). It enforces $\mathsf{precise} <: \mathsf{approximate}$ meaning that a precise variable can flow to an approximate one, but an approximate variable cannot flow to a precise one.

The programmer annotates certain variables as approximate; these are the sources. The goal is to minimize the impact of approximate variables on the precise partition of the program. Currently, EnerJ does not perform type inference; therefore it burdens the programmer to manually propagate the initial approximate annotations until the pro-

gram type checks. Our inference system can be applied directly to EnerJ, which would ease the annotation burden and make EnerJ more practical.

3.3.3 Inference Example

Consider the example in Fig. 6 which is copied from Fig. 1. The Data object at line 15 stores sensitive values, while the one at line 21 stores cleartext values. Therefore, class Data is polymorphic; it is interpreted as sensitive in the context of the first Data object, and as clear in the context of the second one.

At line 16, s is annotated as sensitive by the programmer. Therefore, line 17 forces p to be {poly, sensitive}, then 10 forces $\mathsf{this}_{\mathsf{set}}$ to be {poly, sensitive} and d to be {poly}. (Recall that field d is initialized to {poly, clear }; 10 removes clear.) Now the set-based solver reaches a fixpoint and no more infeasible qualifiers can be removed. If we picked the maximal type from each set, the program will not type-check.

Therefore, we use method summary constraints to remove additional infeasible qualifiers. Since field d is poly, constraint $\mathsf{this}_{\mathsf{get}} \rhd \mathsf{d} <: \mathsf{ret}_{\mathsf{get}}$ leads to method summary constraint $\mathsf{this}_{\mathsf{get}} <: \mathsf{ret}_{\mathsf{get}}$, which in turn leads to $\mathsf{ds} <: \mathsf{ss}$ due to the call at 19. Similarly, $\mathsf{p} <: \mathsf{this}_{\mathsf{set}} \rhd \mathsf{d}$ leads to $\mathsf{p} <: \mathsf{this}_{\mathsf{set}}$, which in turn leads to $\mathsf{s} <: \mathsf{ds}$ due to the call at 17. Since s is {sensitive}, ds is updated to {sensitive} and then ss is updated to {sensitive} as well. Similarly, we can derive the method summary constraints $\mathsf{c} <: \mathsf{dc}$ and $\mathsf{dc} <: \mathsf{cc}$ in line 27 and 28 from the constraints of line 23-26. We update all solutions based on the new constraints. In the end we choose the maximal typing from each set as the concrete type for each variable. We can see that variable ss has a type of sensitive which is expected since sensitive s propagates to it through methods set and get in class Data. On the other hand, variable cc remains clear since c is clear even though it flows through the same methods. Callsite qualifiers q^{17} and q^{19} are inferred as sensitive, and qualifiers q^{23} and q^{25} are inferred as clear (for clarity these are not shown in Fig. 6). The example demonstrates the importance of polymorphism in our analysis.

After the information flow analysis of the first stage, every variable has a type sensitive, poly or clear. All methods that have poly parameters (include implicit parameter this) will have two versions, one for computation over encrypted data and the other for computation over cleartext data. The transformed program is shown in Fig. 7.

```
1   public class Data {
2     int d;
3     int d_Sen;
4     int get_Sen(Data this) {
5       if (d_Sen < 0)
6         this.d_Sen = this.d_Sen + 1;
7       return this.d_Sen;
8     }
9     void set_Sen(int p) { this.d_Sen = p; }
10    int get(Data this) {
11      if (this.d < 0) this.d = this.d + 1;
12      return this.d;
13    }
14    void set(int p) { this.d = p; }
15  }
16  public class Example {
17    public void main() {
18      Data ds = new Data();
19      int s = ...;
20      ds.set_Sen(s);
21      int ss = ds.get_Sen();
22      Data dc = new Data();
23      int c = ...;
24      dc.set(c);
25      int cc = dc.get();
26      ...
27    }
28  }
```

Figure 7: The partitioned program. The methods get_Sen and set_Sen are the sensitive versions of get and set. The field d_Sen stores the encrypted data which is used in sensitive versions of methods.

4. DATA-FLOW ANALYSIS

This section describes the data-flow analysis for the second stage of JCrypt. In this stage, JCrypt determines the encryption scheme for each sensitive variable and minimize conversions using an inter-procedural data-flow analysis.

As mentioned in the overview section, we can eliminate conversions by encrypting the sensitive input data with several efficient encryption schemes. The program would choose the appropriate *available* version of ciphertext, as in Fig. 2. In this example, the OPE version of the data is available at line 5, that is, there are no operation expressions to change d before the point. Similarly, the AH version is also available at line 6. However, the addition expression at line 6 redefines d which "kills" all other encryption schemes except AH. Therefore, if there is any operations on d other than addition afterwards, then conversions there will be unavoidable. Our analysis is similar to classical Available Expressions data-flow analysis. The difference is that our lattice elements are sets of available encrypted data, rather than sets of available expressions.

While the analysis in the first stage Sect. 3 is *context-sensitive* and flow-insensitive, the analysis in the second stage is context-insensitive and *flow-sensitive*. One way to think of the analysis is as a classical intra-procedural Available Expression where the sensitive partition of the program constitutes the one "procedure". Sect. 4.1 defines the standard components of the data-flow analysis. Sect. 4.2 illustrates the analysis with a detailed example.

4.1 Definitions

We say that an encrypted version (RND, AH, DET or OPE) is available for variable x at program point p if there are no operations to recompute the value of x in every path from the entry node of the program to p. An assignment statement $\mathsf{x} = \mathsf{y} + \mathsf{z}$ *kills* RND, DET and OPE for x and *generates* AH for x. The expressions for equality checking and comparison do not kill or generate any encryption version since they do not change any involved variable. Initially, each variable x is initialized to all four versions $S(\mathsf{x}) = \{\text{RND, AH, DET, OPE}\}$. The transfer functions, defined over the named form syntax from Sect. 3, are as follows:

- $\mathsf{x} = \mathsf{y} \quad \Rightarrow \quad S(\mathsf{x}) = S(\mathsf{y})$
- $\mathsf{x} = \mathsf{y.f} \quad \Rightarrow \quad S(\mathsf{x}) = S(\mathsf{f})$

- $\mathsf{x.f = y} \quad \Rightarrow \quad S(\mathsf{f}) = S(\mathsf{f}) \cap S(\mathsf{y})$
- $\mathsf{x = y + z} \quad \Rightarrow \quad S(\mathsf{x}) = \{\text{AH}\}$
- $\mathsf{x = y.m(z)} \quad \Rightarrow \quad S(\mathsf{p}) = S(\mathsf{p}) \cap S(\mathsf{z}),\ S(\mathsf{x}) = S(\mathsf{ret_m})$

$\mathsf{ret_m}$ and p are the return value and parameter of m. The analysis is field-based, that is, field f is considered as a global variable.

For local variables, the set of available encryption versions is equal to the right-hand-side of the assignment. However, for fields and method parameters, we use intersection to update their sets because the analysis is context-insensitive. Therefore, field writes and method calls are treated as standard merge points. There is no rule for this in these transfer function since this is of reference type while our available encryption analysis only apply to the data stored in a variable.

All other components of the data-flow analysis are as in standard Available Expressions. That is, the direction of the data-flow is Forward, and the meet operator is the intersection, by virtue of Available Expressions being a Must data-flow analysis.

JCrypt examines the operands of arithmetic operations. If the corresponding encryption version is available then there is no need for conversion. If it is not available, then we cannot avoid the type conversion.

As we mentioned, the first stage of the analysis, i.e., the information flow analysis in Sect. 3, partitions a program into two parts, as in Fig. 7. The data-flow analysis is only applied to the sensitive partition, that is, methods set_Sen and get_Sen. The clear versions of methods set and get are not analyzed. Therefore, partitioning a program not only reduces the need for encryption (by marking a large number of variables clear), but it also aids the data-flow analysis by minimizing the part of the program that needs to be processed.

4.2 Example

Consider the example of Fig. 7 which is partitioned based on the result of the first stage analysis. The control flow graphs are shown in Fig. 8. The figure does no include the clear part, that is, variable dc, c and cc and method set and get. The initial value for the entry point of main is a map of all local variables and fields to a set of all encryption schemes, that is $S(\mathsf{ds}) = S(\mathsf{s}) = S(\mathsf{ss}) = \{\text{RND, AH, DET, OPE}\}$. There is no fields in class Example. Block n_1 and n_2 do not modify the data-flow value. The call-entry of set_Sen has an initial value for its local variable p and field d_Sen, and $S(\mathsf{p})$ is updated by the intersection of its initial set and the set of argument s according to the transfer function. Here the set keeps unchanged. Similarly, the call-entry flow value of get_Sen is $S(\mathsf{d_Sen}) = \{\text{RND, AH, DET, OPE}\}$. Note here, if the previous method set_Sen modifies the set of field d_Sen, then the value here should have the same change since fields are considered global for the whole program. Block n_5 changes the set of d_Sen to $\{\text{AH}\}$ based on the transfer function for addition operation. At block n_6 we intersect the two sets obtained from the two paths and get $S(\mathsf{d_Sen}) = \{\text{AH}\}$. The return value propagates to ss at call site c_2 which forces $S(\mathsf{ss}) = \{\text{AH}\}$. At block n_4 we check if OPE is available for the comparison. Similarly, we check the availability of AH at block n_5. Both of them are available, hence the program does not require any conversion to execute over encrypted data.

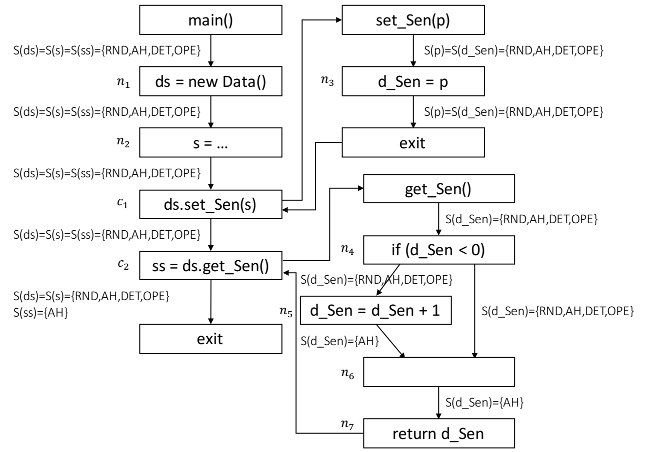

Figure 8: The control flow graphs annotated with data-flow values.

Suppose the piece of code of line 5 and 6 in Fig. 7 becomes a loop instead of a condition as follows:

```
while (this.d < 0) {
  this.d = this.d + 1;
}
```

Then a conversion is necessary since there will be a control flow from the addition statement back to the condition so that OPE is no longer available at the comparison.

5. EXPERIMENTS

We built the first stage analysis of JCrypt in our type inference and checking framework [18]. Using the framework, programmers provide parameters to instantiate their own type system. Then the framework infers the "best" typing. We have instantiated several type systems and their corresponding inferences [13, 9, 17, 10, 12]. Our framework includes two front-ends. One takes as input the Java source code while the other takes the Jimple code transformed by Soot. JCrypt uses the Soot-based front-end. We built the second stage analysis of JCrypt on top of a general-purpose inter-procedural analysis framework for Soot [21]. The call graph is built by the SPARK engine [14] in Soot with a context-insensitive pointer analysis. The framework provides program representations for Soot's Jimple IR which is compatible with our checker framework used in the first stage analysis.

The work flow of JCrypt is shown in Fig. 9. First, users mark certain variables as *sensitive* sources in the Java program to be analyzed. Next, the JCrypt information flow analysis module takes as input the compiled Java class files and outputs the intermediate result showing the inferred type (sensitive, poly or clear) for each variable. This module consists of two components: (1) the JCrypt front-end takes as input the Jimple files transformed by Soot and outputs the constraints generated according to the typing rules in Fig. 4, and (2) the JCrypt type inference engine takes the constraints and infers a valid typing for the analyzed program. Then the transformer module utilizes the intermediate result to transform the original program into a partitioned one as shown in Fig. 7. In the end, the data-flow analysis engine analyzes the transformed program to determine

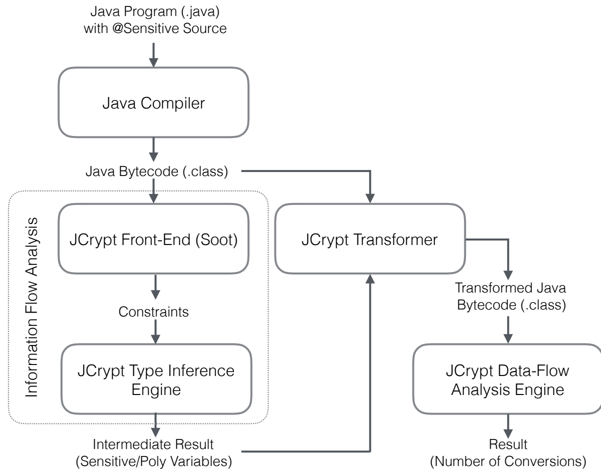

Figure 9: The work flow of JCrypt.

Benchmark	#Line	#Conversion	#Conversion/100-line
bh	1136	8	0.7
bisort	355	0	0.0
em3d	467	0	0.0
health	569	0	0.0
mst	471	0	0.0
power	773	126	16.3
treeadd	200	0	0.0
tsp	563	12	2.1
voronoi	1003	4	0.4

Figure 10: Results on the JOlden benchmarks. #Line shows the number of lines of the benchmarks, including blank lines and comments. #Conversion gives the number of conversions required by the benchmarks. #Conversion/100-line shows the average number of conversions per 100 line of code.

encryption schemes for each sensitive variable and identify conversions, if any. The source code of JCrypt is publicly available at https://github.com/proganalysis/type-inference.

All experiments run on a MacBook Pro with Intel® Core i5 CPU @2.6GHz and 8 GB RAM. The software environment consists of Oracle JDK 1.7 and the Soot 2.5.0 nightly build.

5.1 Benchmarks

We evaluate JCrypt on five benchmark suites which are respectively listed in Fig. 10, 11, 12, 13 and 14. For each benchmark, we list the number of source lines of code.

Java Olden (JOlden)[4] is a benchmark suite of 10 programs which implement different algorithms using the tree data structure. If these programs are used on a server to provide clients with computations, we can consider the values in the tree nodes as sensitive. Therefore, we added sensitive annotations on the data fields of the tree node classes. We analyzed 9 of the 10 programs because we could not identify meaningful sensitive data in the one remaining benchmark, perimeter. This benchmark computes the total perimeter of a region in a binary image by counting the number of tree nodes which represent blocks of the image and there is no data stored in tree nodes.

[4]https://github.com/farseerfc/purano/tree/master/src/jolden

Benchmark	#Line	#Conversion	#Conversion/100-line
FFT	168	8	4.8
SOR	36	5	13.9
MonteCarlo	59	3	5.1
SparseMatMult	38	1	2.6
LU	283	9	3.2
ZXing	26171	297	1.1
jMonkeyEngine	5962	159	2.7
ImageJ	156	16	10.3
Raytracer	174	29	16.7

Figure 11: Results on the EnerJ benchmarks.

Benchmark	#Line	#Conversion	#Conversion/100-line
L1	137	0	0.0
L2	148	0	0.0
L3	185	0	0.0
L4	141	0	0.0
L5	169	0	0.0
L6	139	0	0.0
L7	158	0	0.0
L8	170	1	0.6
L9	196	0	0.0
L10	245	0	0.0
L11	184	0	0.0
L12	218	0	0.0
L13	182	0	0.0
L14	183	0	0.0
L15	188	0	0.0
L16	134	0	0.0
L17	259	3	1.2

Figure 12: Results on the PIGMIX2 benchmarks.

Sampson et al. [24] propose EnerJ, a type system for energy efficient computation. EnerJ partitions the program into a precise and an approximate components where the approximate component can be processed more cheaply realizing energy savings. They evaluate EnerJ by annotating 9 Java programs, which we refer to as "the EnerJ suite." The authors of EnerJ explicitly annotated certain variables with the maximal @Approx qualifier. We reuse these annotations by replacing @Approx with sensitive.

The remaining three benchmark suites come from MrCrypt [26]. They include the PIGMIX2 suite (17 programs), the Brown suite (5 programs) and the PUMA suite (13 programs). These benchmarks are Hadoop MapReduce programs widely used in cloud computing [15]. Since clients upload MapReduce programs and input data together to the server, we consider all input data as sensitive. Each MapReduce program has a map method that passes the input file through a parameter of type org.apache.hadoop.io.Text. Hence, we mark this parameter as the sensitive source.

5.2 Experimental Result

In our experiments, we aim to evaluate the benefits of JCrypt. Specifically, we would like to answer two questions: (1) How often can programs be executed over efficiently encrypted data without any conversions? (2) How many conversions does a program require when conversion cannot be avoided?

We consider a decryption and re-encryption process as one conversion. For example, the expression $\mathsf{x} < \mathsf{y}$ needs two conversions if $S(\mathsf{x}) = \{\text{AH}\}$ and $S(\mathsf{y}) = \{\text{AH}\}$. Assuming that only AH is available before the expression, x and y each need one conversion from AH to OPE. In addition, we count the number of conversions statically. For example, if the above expression is inside a loop, we still count it as

Benchmark	#Line	#Conversion	#Conversion/100-line
Grep	214	0	0.0
Benchmark1	113	0	0.0
Benchmark2	162	0	0.0
Benchmark3	855	4	0.5
Benchmark4	95	0	0.0

Figure 13: Results on the Brown benchmarks.

Benchmark	#Line	#Conversion	#Conversion/100-line
Word Count	88	0	0.0
Inverted Index	126	0	0.0
Term Vector	187	0	0.0
Self Join	136	0	0.0
Adjacency List	157	0	0.0
K-Means	428	7	1.6
Classification	228	5	2.2
Histogram Movies	132	3	2.3
Histogram Ratings	115	0	0.0
Sequence Count	124	0	0.0
Ranked Inverted Index	127	0	0.0
Tera Sort	192	0	0.0
Grep	55	0	0.0

Figure 14: Results on the PUMA benchmarks.

two conversions no matter how many times the expression is executed at run time.

JCrypt takes 109 seconds per benchmark on average. It takes less than 2 minutes on 36 of the 53 benchmarks, and between 2 and 3 minutes on 15 benchmarks. The 2 outliers run in 5 and 9 minutes.

5.2.1 Results

The result on JOlden (Fig. 10) shows that 56% of the programs (5 out of 9) do not require conversions. This shows that JCrypt is applicable achieves very good result on general-purpose Java programs. Except for one benchmark, power, which requires 126 conversions, the remaining 3 programs require only a few conversions (less than 3 per 100 lines).

All EnerJ benchmarks require conversion (Fig. 11). This is because these programs are computationally intensive required to evaluate EnerJ from the point of view of power saving. However, if we consider the size of each benchmark, the number of conversions is relatively small (less than 17 per 100 lines).

The result on MapReduce is shown in Fig. 12, 13 and 14. 29 out of 35 benchmarks (83%) require no conversions. A typical example of conversion is shown in Fig. 15 which is from the benchmark Histogram Movies from the PUMA suite. Here rating comes from an array of sensitive data.

```
1   ...
2   while ( ... ) {
3     sumRatings += rating; // rating is sensitive
4     totalReviews++;
5   }
6   avgReview = sumRatings / totalReviews;
7   absReview = Math.floor(avgReview);
8   fraction = avgReview − absReview;
9   if (fraction < (division ∗ i)) {
10    outValue = absReview + division ∗ i;
11  }
```

Figure 15: A slice of program requiring conversions.

The addition operation makes $S(\text{sumRatings}) = \{\text{AH}\}$. The type for the division [5] is not available at sumRatings / totalReviews; hence, a conversion from AH to the division encryption version is necessary at line 6. Similarly, another conversion from AH to OPE is also required at line 9 for variable fraction.

We manually examined all conversions reported by JCrypt. Our analysis result is *optimal* in the sense that the conversions are unavoidable even if we had the most precise static analysis. E.g., consider the conversion in Fig. 15. Our results show that for a significant percentage of programs data is separable; therefore, these programs can benefit from efficient encryption schemes. Unfortunately, for an equally significant percentage of programs, data is not separable. In our future work, we will investigate program analysis and partitioning techniques for efficient conversion.

5.2.2 Comparison with MrCrypt

To illustrate the benefit of our data-flow analysis, we compare out result with MrCrypt on the three MapReduce benchmark suites, PIGMIX2, Brown and PUMA. MrCrypt infers encryption schemes for input data in the program. If the program performs multiple operations on the same data, MrCrypt considers that the data requires fully homomorphic encryption (FH). This has the same meaning as conversion in JCrypt.

In MrCrypt, 66.7% of the programs (24 of 36) can be executed without requiring FH. In JCrypt, 83% of the programs (29 of 35) can be executed without conversion. The benchmarks used in JCrypt are the same as in MrCrypt, except for the Brown suite where 3 benchmarks may be different. Among the common benchmarks, 5 need FH in MrCrypt but no conversion in JCrypt.

For example, the two benchmarks L15 and L16 require FH in MrCrypt because the sensitive data involves two operations: equality checking and addition. The relevant piece of code is as follows:

```
HashSet<Text> hash = new HashSet<Text>();
while (iter.hasNext()) {
  List<Text> vals = Library.splitLine(iter.next(), ' ');
  hash.add(vals.get(0));
}
for (Text t : hash) rev += Double.valueOf(t.toString());
```

The equality check is implicit from the hashset. JCrypt results show that these two benchmarks do not need any conversion because DET is available at hash.add and AH is also available at the addition.

MrCrypt is not publicly available and we cannot compare directly on the JOlden and EnerJ benchmarks. Based on the paper [26], we can extrapolate that MrCrypt only works on a functional subsets of Java programs, such as MapReduce. In contrast, JCrypt can handle any Java programs.

In addition, MrCrypt is context-insensitive which could lead to imprecise inference result. Consider the example in Fig. 1 again. Just like the earlier discussion, MrCrypt concludes that the program requires FH since the field involves two operations (comparison and addition) in method get.

[5] We do not specify an encryption type for the division operation in order to simplify our description. We can extend our system with any other operation. For example, we can simply add a DIV type for the division operation to the initial set and a transfer function $S(\text{x}) = \{\text{DIV}\}$ for statement x = y / z.

However, even if we remove the operations for the sensitive data, MrCrypt would still get the same conclusion. Suppose we modify the example of Fig. 1 as follows:

```
1   public class Data {
2     int d;
3     int get(Data this) {return this.d;}
4     void set(Data this, int p) {this.d = p;}
5   }
6   public class Example {
7     public void main() {
8       Data ds = new Data();
9       sensitive int s = ...; // sensitive source
10      ds.set(s);
11      sensitive int ss = ds.get();
12      Data dc = new Data();
13      clear int c = ...;
14      dc.set(c);
15      clear int cc = dc.get();
16      if (cc < 0) cc++;
17    }
18  }
```

We remove the if block from method get and add a similar block to the end of method main. Now the program does not require FH or conversion because it performs operations only over clear data. However, due to MrCrypt is context-insensitive, it detects flow from sensitive source to field d at line 10, which flows to cc at line 15 by calling method get. In the end, MrCrypt finds the two operations on cc and comes to the conclusion that it needs FH to encrypt sensitive data. In contrast, JCrypt is able to identify that cc is clear through the first stage of information flow analysis and directly concludes that the program does not require conversions, even without the second stage of data-flow analysis.

6. RELATED WORK

JCrypt is related to MrCrypt [26], a system that provides secure computations over encrypted data on the cloud server. MrCrypt statically analyzes Java programs and infers the encryption type for each variable which is similar to JCrypt. The key difference between these two systems is that JCrypt performs analysis on arbitrary Java program while MrCrypt handles only a small functional subset of Java, such as MapReduce programs which require only a small set of operations. Furthermore, JCrypt uses information flow analysis and parametric polymorphism to minimize the computation of encryption, while MrCrypt has to encrypt the entire program which results in a relatively larger encryption overhead. Finally, JCrypt uses Available Expressions analysis to avoid conversions so that it can handle programs where more than one operations apply on the same variable, while MrCrypt fails on such programs.

CryptDB [22] is another system that can peform computation over encrypted data. It analyzes SQL queries and rewrites them to execute on SQL databases. CryptDB encrypts data using all necessary encryption schemes and stores them in the database so that it can choose different encypted data according to different operations. This is similar to JCrypt for the cases where a program performs multiple operations on the same sensitive data. However, like MrCrypt, CryptDB is also limited to a specific application — database queries, while JCrypt is applicable to any Java program.

Program partitioning techniques have been used to provide efficient computation in other domains. EnerJ [24] is a type-based system that partitions a program into a precise component and an approximate component. Computing on approximate data is less expensive than computing on precise data. Therefore running the program can save energy at some accuracy cost. Similarly to EnerJ, JCrypt partitions the program based on information flow. One key difference is that EnerJ requires significant amount of programmer-provided annotations, while JCrypt automatically infers a partition with a maximal number of cleartext variables.

Another work related to program partitioning is Swift [2], an approach to partition a program into two parts, one running on the client and the other running on the server. The difference between JCrypt and Swift is that Swift physically isolates security-critical code from the untrusted environment to provide security, while JCrypt marks sensitive code and encrypts the data so that the whole program can run on the untrusted server. Further, Swift requires programmers to use a security-typed programming language, Jif/split [27] to write programs. This increases the burden on programmers and it may not be suitable for existing programs. In contrast, JCrypt only requires a few annotations by the programmer to indicate sensitive sources.

7. CONCLUSIONS

We presented JCrypt, a system to provide data confidentiality for programs running in an untrusted environment. JCrypt performs two stages of analysis on Java programs to minimize the encrypted computing and identify encryption schemes for sensitive data. Our experiments show that the approach is effective.

One limitation of our work is that JCrypt assumes that library calls do not perform operations on encrypted data. While this is largely the case for our benchmarks, which operate on integer data, in general it may be unsafe. Another limitation is that currently we do not measure overhead of the encrypted version over the unencrypted one. At this point our goal was to establish that program analysis enables encryption for a large number of programs. We plan to address these limitations in future work.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback on our work. This work was supported by NSF Award CCF-1319384.

9. REFERENCES

[1] J. A. Bank, A. C. Myers, and B. Liskov. Parameterized types for Java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 132–145, New York, NY, USA, 1997. ACM.

[2] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 31–44, New York, NY, USA, 2007. ACM.

[3] M. Cooney. IBM touts encryption innovation: New technology performs calculations on encrypted data without decrypting it. *Network World*, June 2009.

[4] W. Dietl and P. MÃijller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[5] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[6] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[7] C. Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, Mar. 2010.

[8] C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, EUROCRYPT'11, pages 129–148, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and checking of object ownership. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 181–206, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] W. Huang, Y. Dong, and A. Milanova. Type-based taint analysis for Java web applications. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*, pages 140–154, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[11] W. Huang, Y. Dong, and A. Milanova. Type-based taint analysis for Java web applications. Technical report, Rensselaer Polytechnic Institute, Department of Computer Science, 2014.

[12] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 106–117, New York, NY, USA, 2015. ACM.

[13] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. Reim & ReImInfer: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 879–896, New York, NY, USA, 2012. ACM.

[14] O. Lhoták and L. Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[15] N. Liu, X. Yang, X. H. Sun, J. Jenkins, and R. Ross. YARNsim: Simulating Hadoop YARN. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 637–646, May 2015.

[16] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4):12:1–12:38, Dec. 2011.

[17] A. Milanova and W. Huang. Dataflow and type-based formulations for reference immutability. In *19th International Workshop on Foundations of Object-Oriented Languages*, FOOL'12, 2012.

[18] A. Milanova and W. Huang. Inference and checking of context-sensitive pluggable types. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 26:1–26:4, New York, NY, USA, 2012. ACM.

[19] A. Milanova and W. Huang. Composing polymorphic information flow systems with reference immutability. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, FTfJP '13, pages 5:1–5:7, New York, NY, USA, 2013. ACM.

[20] A. Milanova, W. Huang, and Y. Dong. CFL-reachability and context-sensitive integrity types. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 99–109, New York, NY, USA, 2014. ACM.

[21] R. Padhye and U. P. Khedker. Interprocedural data flow analysis in Soot using value contexts. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13, pages 31–36, New York, NY, USA, 2013. ACM.

[22] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[23] S. Rass and D. Slamanig. *Cryptography for Security and Privacy in Cloud Computing*. Artech House, Inc., Norwood, MA, USA, 2013.

[24] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[25] M. Shah, E. Stark, R. A. Popa, and N. Zeldovich. Language support for efficient computation over encrypted data. In *Off the Beaten Track Workshop: Underrepresented Problems for Programming Language Researchers*, Philadelphia, PA, January 2012.

[26] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein. MrCrypt: Static analysis for secure cloud computations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 271–286, New York, NY, USA, 2013. ACM.

[27] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 1–14, New York, NY, USA, 2001. ACM.