Privacy-Preserving Genomic Computation Through Program Specialization

Rui Wang[1], XiaoFeng Wang[1], Zhou Li[1], Haixu Tang[1], Michael K. Reiter[2], Zheng Dong[1]
[1]Indiana University at Bloomington.
[2]University of North Carolina at Chapel Hill.

ABSTRACT

In this paper, we present a new approach to performing important classes of genomic computations (e.g., search for homologous genes) that makes a significant step towards privacy protection in this domain. Our approach leverages a key property of the human genome, namely that the vast majority of it is shared across humans (and hence public), and consequently relatively little of it is sensitive. Based on this observation, we propose a privacy-protection framework that partitions a genomic computation, distributing the part on sensitive data to the data provider and the part on the pubic data to the user of the data. Such a partition is achieved through program specialization that enables a biocomputing program to perform a concrete execution on public data and a symbolic execution on sensitive data. As a result, the program is simplified into an efficient query program that takes only sensitive genetic data as inputs. We prove the effectiveness of our techniques on a set of dynamic programming algorithms common in genomic computing. We develop a program transformation tool that automatically instruments a legacy program for specialization operations. We also demonstrate that our techniques can greatly facilitate secure multi-party computations on large biocomputing problems.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Security

Keywords

Privacy-Preserving Computation, Program Specialization, Human Genome, Symbolic Execution, Dynamic Programming, Secure Multi-Party Computation

1. INTRODUCTION

Recent progress in the study of the human genome has led to a revolution in biomedical science, which promises a profound impact on many aspects in people's lives. These advances, however, do not come without introducing new concerns: genomic data carry sensitive personal information such as genetic markers for diseases, whose confidentiality is threatened by the increasing collection and distribution of those data for medical research. To protect genome privacy, prior research suggests anonymizing genome data before releasing them, through techniques such as DNA lattice anonymization [47]. Such an approach, however, reduces the information in the original data and as a result, undermines their utility for genome research.

A straightforward approach that avoid disclosing sensitive data involved in a computation to the party using the data (called *data consumer* or DC) is to simply delegate all the computation tasks to the party providing the data (called *data provider* or DP). This centralized treatment, however, is unviable because the DP can easily become a performance bottleneck. In the case that the DC also holds sensitive inputs to its computation, a secure multi-party computation (SMC) [69, 29] needs to be performed between these two parties. Unfortunately, recent research shows that even optimized SMC cannot handle genome computations of a realistic scale [34], which often involve millions of nucleotides.

Many important genome studies, such as search of homologous genes [6, 7, 38, 58, 9], comparison of syntenic regions across multiple genomes [63, 45], and protein identification in proteomics [61, 67, 32, 17], utilize dynamic programming [64] and other algorithms to compare a query DNA or protein sequence with genomic sequences in a genome database. For example, in order to determine the level of variation of a specific gene in the population, a DC may request to compare a query gene sequence from a reference genome with its homologous gene sequences from all individual genomes in a personal genome database. The privacy problem here is that these genomes contain some sensitive genetic variations, which are mostly related to single nucleotide polymorphism (SNP) [43], a DNA variation that differs between members of a species. These variations can be used to identify individuals and their personal health information such as genetic diseases, and therefore should not be exposed to the DC. On the other hand, it is well known that genetic variations represent only a small fraction of the entire human genome, as indicated by prior research [4] (0.5 percent between two unrelated persons), and our analysis in the Appendix (0.01 percent among a population of 1 million). Though the sensitivity of individual SNPs [43] are yet to be determined, we can adopt a conservative approach that treats all SNPs as sensitive data. Even in this case, nearly all human genes consist of a vast majority of common (and definitely non-sensitive) nucleotides that serve as part of the inputs to aforementioned research.

The above observation can be leveraged to protect sensitive genetic information involved in a genomic computation, through distributing the computation between the DP and the DC: the DP un-

CCS'09, November 9–13, 2009, Chicago, Illinois, USA.

dertakes a small portion of the computation related to sensitive data while the DC works on the rest of a genome sequence involving only nonsensitive nucleotides. As a result, the computation can be accomplished without revealing sensitive nucleotides to the DC. Partitioning a computation for privacy protection has been studied in prior research [18, 16]. For example, Swift [18] uses information-flow analysis to separate an application into the parts that work on the data with different security levels. However, such an approach can be less applicable to genome algorithms that intertwine the operations on both public and sensitive data. A prominent example is a category of dynamic programming algorithms (DPA) common in the aforementioned research: once a DPA encounters a sensitive nucleotide, all the follow-up computation will all be related to it. As a result, an information-flow analysis will tell us to put the whole computation on the DP side.

In this paper, we propose a new technique that applies *program specialization* [35] to partition a genomic computation according to the sensitivity levels of the genome data it works on. Our approach allows the DC to compute over the genome sequences sanitized by the DP, on which sensitive nucleotides are replaced with symbols. This is achieved through a *mixed execution*: a concrete execution on public data and a symbolic execution [42] on those symbols. As a result, a biocomputing program can be specialized into a "query" program for the DP, which takes nothing but sensitive nucleotides as its inputs. Given that sensitive nucleotides only take a very small portion of the data a program processes, its specialized query program is typically much more efficient, and can be easily computed by the DP. An efficient query not only saves the DP's resources but also significantly reduces the cost for performing an SMC protocol, should the DC also have sensitive inputs involved in the computation. To control information leaks from the outcome of a computation, we treat the program as a database query, and use a query auditor (Section 2.4) to mediate its answers. To efficiently retrofit legacy biocomputing code with the capability to perform such distributed computations, we also designed a source-to-source transformation tool that automatically analyzes a legacy program and instruments it for mixed executions.

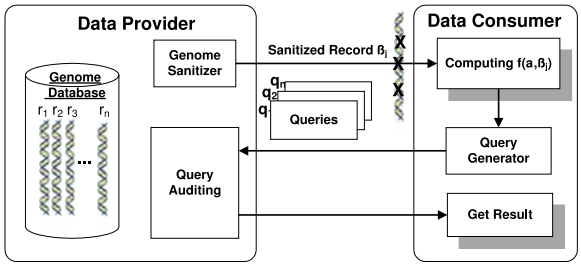

Figure 1: Framework.

We outline the contributions of this paper as follows:

•*A privacy protection framework.* We propose a distributed framework for privacy-preserving genomic computing, as illustrated in Figure 1. Our framework distributes a computation task between the DP and the DC, and lets the DP handle a small portion of the task related to its sensitive data. This avoids expensive SMC when the DC does not have sensitive inputs, and significantly reduces the overheads for running such a protocol when it does.

•*Computation partitioning for privacy protection.* We propose novel techniques that use public data to specialize a genomic computing program into a much more efficient query program for processing sensitive data on the DP. We theoretically analyze the effectiveness of these techniques over a category of DPAs that are extensively used in bioinformatics.

•*Source-to-source transformation tool.* We design a new tool that automatically analyzes a legacy genomic program and instrument it with the code to perform specialization.

•*Implementation and evaluations.* Our evaluations show that our techniques enable many important genome algorithms [64, 56, 54, 6] to work on a large amount of data at small overheads. We implemented a prototype of our tool and successfully applied it to transform three biocomputing libraries. We also studied use of SMC protocols over query programs, and observed a significant improvement in performance compared with a direct application of them to unspecialized algorithms.

Our technique is based upon partition of a computation task according to the sensitivity of its nucleotide inputs. We are fully aware that identifying sensitive SNPs is still an ongoing research [33]. However, prior research does indicate that such SNPs take only a very small portion of human genome [27], and many important genome studies [6, 7, 38, 58, 9, 63, 45, 61, 67, 32, 17] work on contiguous genome sequences that involve only small amount of SNPs. The classification of sensitive/nonsensitive nucleotides only serves as an input to our approach.

The rest of the paper is organized as follows. Section 2 and 3 describes our query generation techniques and transformation tool. Section 4 reports on an evaluation of our approach. Section 5 discusses the limitations of our current design. Section 6 presents the related prior research, and Section 7 concludes the paper.

2. COMPUTATION PARTITIONING

In this section, we present the techniques that partition a genomic computation task according to the sensitivity levels of genome data. Our approach is based upon program specialization (a.k.a., partial evaluation), a technique that uses partial inputs of a program to produce a new program that only accepts the rest of the inputs [35]. In our research, we developed the specialization techniques for genome computing, which reduces an algorithm to a query program using a sanitized DNA record. As a first step, our current focus is on a set of dynamic programming algorithms [64, 56, 54, 6] that are common in genome computing.

2.1 Overview

The general idea of our techniques can be illustrated through a simple example in Figure 2. The example computes the *edit distance* between genome sequences α and β, i.e., the minimal number of edit operations, including delete, insert and replace, to convert one sequence to the other. This is done through dynamic programming over a two-dimension matrix $D(0 \cdots n, 0 \cdots m)$, where n and m are the lengths of α and β respectively. Specifically, the algorithm first initializes the matrix by setting $D(i, 0)$ to i for $0 \leq i \leq n$, and $D(0, j)$ to j for $0 \leq j \leq m$. Then, it recursively fills the matrix as follows:

$$D(i,j) = \min(D(i-1,j)+1, D(i,j-1)+1, \\ D(i-1,j-1)+s(i,j)) \quad (1)$$

where $s(i, j)$ is a score function that has a value 1 if $\alpha[i]$, the i nucleotide on α, is different from $\beta[j]$, the jth nucleotide on β, and a value 0 otherwise. The minimal edit cost between these sequences is recorded in $D(n, m)$ and the edit process that incurs that cost is described by a path from the entry $(0, 0)$ to (n, m).

Figure 2 presents an example with α=`ATC` and β=`ACC`. The edit distance here is $D(3, 3) = 1$, and the optimal edit path is $(0, 0) \rightarrow (1, 1) \rightarrow (2, 2) \rightarrow (3, 3)$, as each cell on the chain provides the smallest edit cost to the next one according to Equation 1. This algorithm is implemented by a program $P1$ that iteratively computes the values of the cells in the matrix.

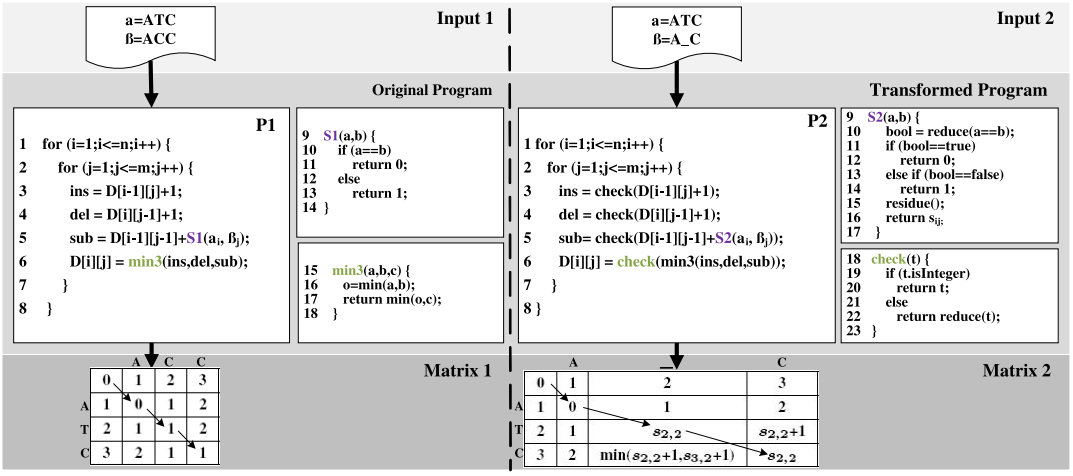

Figure 2: A simple example.

Suppose that $\beta[2]$ is a sensitive nucleotide that is replaced by a symbol. This prevents $P1$ from accomplishing the computation, because it cannot get the values for the third and fourth columns in the matrix. To solve this problem, we transfer the program to another program, $P2$, to perform a mixed execution. Specifically, the statements at Line 3, 4, 5 and 6 of $P1$ are all modified to work on both concrete and symbol inputs: all the operations go as normal if the input to a statement contains only concrete values; otherwise, symbolic execution [42] is performed to generate an expression as its output. Such an expression is further *reduced* through, for example, combining all the constants. The score function $S1$ of $P1$ is also converted into $S2$: if a branch condition contains symbols (Line 15 in $S2$), $S2$ exports the branch condition in $S1$ and both of its branches to a *residual* program, and returns a symbol $s_{i,j}$, where i and j are the indices of nucleotide inputs. The same transformation happens to the $\min$ operation at Line 6 of $P1$. Its counterpart statement simplifies the expressions the operation involves through unfolding symbols into expressions, combining constants and comparing two expressions using common symbols and value ranges. In the end, the reduced expressions in $D(3,3)$ is exported to the residue program, which serves as the query for the DP.

Matrix 2 shows the process of computing over β in the presence of an unknown nucleotide. Consider $D(3,3)$ as an example. Computing its value using Equation 1 results in an expression that seeks the minimal one among four expressions: $e_1 = s_{2,2}$, $e_2 = s_{2,2}+2$, $e_3 = s_{3,2}+2$ and $e_4 = s_{2,2}+2$. This expression is further reduced as follows. We first find that e_1 is smaller than e_2 and e_4, as all of them describe a sum between $s_{2,2}$ and a constant, and e_1 has the smallest constant. Then, e_1 is compared with e_3 using the value range of symbols $s_{2,2}$ and $s_{3,2}$, which is either 0 or 1, though their exact values are unknown. As a result, the query program we generate only contains a very simple expression, e_1, along with the part of the score function for computing $s_{2,2}$. It is evident that the cost for answering such a query on the DP side is far lower than running $P1$. More interestingly, the DC can even figure out the optimal edit path without consulting the DP: as we can observe from Matrix 2, the value of $D(3,3)$ can be traced back to $D(2,2)$, and again to $D(1,1)$ and $D(0,0)$ according to Equation 1; this can be done without knowing the content of $\beta[2]$.

The above specialization techniques are elaborated in Section 2.2. Their effectiveness is theoretically analyzed in Section 2.3. The transformation from $P1$ to $P2$ can be achieved automatically using program analysis techniques, which is discussed in Section 3.

2.2 Specialization Techniques

To specialize a program, we need to locate its statements that work on sensitive nucleotides and transform them into the form that specialization operations can be performed. Those "tainted" statements are identified by a taint analysis, which we describe in Section 3. Here we first present our specialization techniques.

Specialization operations. Our approach converts every tainted statement into a program snippet that checks the input it receives: if the input does not contain symbols, the original statement is executed; otherwise, a symbolic expression is built through symbolic execution [42] and further simplified by a reduction function before being exported as an output. Denote the specialization operations on a program P by $specialize(P)$. Such operations happen to following program elements:

• *Assignment.* An assignment $a = exp$ is changed to $a = reduce(exp)$ if the expression exp involves symbols, where $reduce()$ is a reduction function.

• *Branching.* A branching statement is in the form "`if` exp `then` P, `else` P'", where exp is the branch condition, and P and P' are the statements to be executed on the two branches. Such a statement is transformed to a set of statements that first checks $reduce(exp)$: if the outcome is either true or false, the program proceeds as normal; otherwise, the following statement is exported to a residue program: "`if` $reduce(exp)$ `then` $specialize(P)$, `else` $specialize(P')$". Also exported are the state of the program prior to the branching, including the values of the variables to be used in P and P'.

To evaluate $specialize()$ on both P and P' online, we need to set a checkpoint prior to the branching statement and roll back after exploring one branch. This can incur significant performance overhead. An alternative is to symbolically execute both branches offline to acquire their symbolic expressions, and replace the symbols in the expressions with concrete values online. Further complicating the specialization efforts is the fact that a branch can include other tainted branching statements, which makes the cost of evaluation high. A simple solution can be exporting all statements of a branch if it contains other tainted branches.

• *Loop.* A loop is residualized if its exit condition is symbolic and cannot be evaluated after proper reduction. When this happens, we can choose to specialize the body of the loop if it does not involve tainted branches.

• *Function.* When part of input parameters to a function's are symbols, the function needs to be specialized using the techniques described above. When this happens, a symbolic expression can be returned. If a function is repeatedly called with different parameters, we can choose to residualize it without specialization.

• *Tainted address.* Programs may read or write a memory location whose address depends on the values of sensitive nucleotides. For example, the index of an array can be determined by unknown symbols, and a pointer in a C program can be tainted by sensitive inputs. When a tainted address is encountered, we can simply export all the statements that directly or transitively rely on the address to the residue program.

Another treatment of a tainted address is to explore all possible values it can take. A nucleotide can only assume four values: A, T, C and G. Therefore, a read from the address involving one symbol gives four possible outcomes at most, which can be represented by a new symbol. Writing to the address is more complicated, as we need to create four threads, each handling one possible version of data. This can be problematic when multiple symbols are present, which causes the number of the threads to increase exponentially. Further study of this problem is left as our future research.

Reduction. Key to specialization is reduction [35] that serves to simplify symbolic expressions. A typical reduction technique is constant folding that combines all the constants in an expression. This is achieved by taking advantage of the properties of a computation, such as commutativity, associativity and distributivity. For example, $10 + a + 6$ can be reduced to $a + 16$. In some cases, an expression can be simplified by unfolding a symbol into the expression it represents. As an example, consider an expression $a+b+10$ with $b = a + 6$. Unfolding b reduces it to $2a + 16$.

A Boolean expression can be evaluated even when it contains symbols. For example, we know that a branch condition $a + 10 \geq a+6$ is true even when the value of a is unknown, as the symbols on both sides of the inequality cancel each other and only the concrete value $4 \geq 0$ is left. This approach can be applied to the comparison between two linear expressions that contain the same set of symbols and each of them has the same coefficient. More generally, combining multiple occurrences of the same symbols when possible can help simplify an expression.

In our research, we design another reduction technique that evaluates a Boolean expression using the value ranges of the symbols it contains. Specifically, our approach identifies the maximal and minimal values a symbol can take and then propagate this range to a symbolic expression. Whenever a comparison between two expressions happens and the ranges of these expressions do not overlap, its Boolean outcome can be determined. For example, consider expressions exp= $a + 9$ and exp′= $b + 6$. Given the ranges of a and b are from 0 to 1, we know that exp is between 9 and 10, while exp′ falls in the range from 6 to 7. As a result, the Boolean expression exp≥exp′ is true. This technique is particularly effective on dynamic programming based genome computing, which we discuss in Section 2.3.

Symbol unfolding. As described above, unfolding a symbol can help simplify that expression. This, however, does not work always. Consider the following example: $d = \min(b + c_1, b + c_2, b + c_3)$ with $b = \min(a_1, a_2, a_3)$. If b is unfolded in the expression of d, we need to compare 9 values to get d. In contrast, if we first get b and then compute d, only 6 comparisons are needed. In our research, we propose a new reduction rule that unfolds a symbol only when an expression does not contain new symbols. In the above example, we can unfold b and c_i if $c_{i=1,2,3}$ only contains a_1, a_2, a_3 or constants: suppose $c_1 = a_1 + 5$, $c_2 = a_1 + 6$ and $c_3 = a_1 + 8$, such an unfolding gives us $d = \min(2a_1+5, a_1+a_2+5, a_1+a_3+5)$, which needs only 3 comparisons to compute. Application of this rule to a dynamic programming algorithm can reduce it to a much simpler residue program that is also dynamic programming, as elaborated in Section 2.3.

2.3 Analysis

Dynamic programming [14] is an optimization technique widely used in bioinformatics, particularly for solving fundamental genome computing problems such as sequence alignment, structural alignment and RNA secondary structure prediction. These problems typically involve two genome sequences, $\alpha[1 \cdots n]$ and $\beta[1 \cdots m]$, and are modeled over an $n + 1$ by $m + 1$ matrix D. The objective is to find an optimal path from $(0, 0)$ to (n, m) that maximizes or minimizes the scores accumulated from those incurred by individual moves from (i, j) to $(i+1, j)$ or $(i, j+1)$ or $(i+1, j+1)$. Such a modeling can also be generalized to a multidimensional graph for the problem such as *multiple sequence alignment* [26], where the goal is to find an optimal path in the graph. The DPAs for solving these problems are usually in the following form:

$$\begin{aligned} D(i,j) = \min(&D(i-1,j) + s_1(i,j), \\ &D(i,j-1) + s_2(i,j), \\ &D(i-1,j-1) + s_3(i,j), C) \end{aligned} \quad (2)$$

where $D(i, j)$ is the score for the optimal path from $(0, 0)$ to (i, j), $s_1(i, j)$, $s_2(i, j)$ and $s_3(i, j)$ are the functions that compute a score given $\alpha[i]$ and $\beta[j]$, and C is a constant. This form of optimization describes many important bioinformatics algorithms, including the famous Needleman-Wunsch [56] and the most widely-used BLAST 2 [66]. Note that throughout this paper we focus on an improved version of the DPA first introduced by Gotoh [30], which reduces the complexity of the DPAs like Needleman-Wunsch and Smith-Waterman algorithms from $O(mn^2)$ to $O(mn)$, and thus are commonly used in today's genome research.

Let ρ be the *ratio* of sensitive nucleotides on β, and $\beta[x_{t=1\cdots\rho m}]$ be these nucleotides. The effectiveness of our specialization techniques on a DPA is described by Theorem 1.

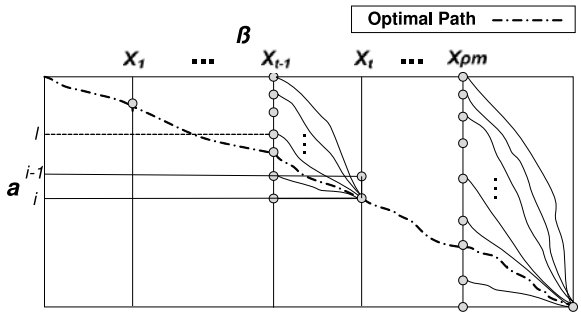

Figure 3: Proof illustration.

THEOREM 1. *The query $q(\beta[x_1], \cdots, \beta[x_{\rho m}])$ generated by specializing a DPA described in Equation 2 is still a DPA. The computational, spatial and communication complexities for answering the query are at most $O(\rho mn^2)$.*

Figure 3 illustrates the general idea of the proof, whose full content is presented in a longer version of the paper [68], due to space limit. Informally, every unknown nucleotide $\beta[x_t]$ corresponds to one column x_t in the $(n + 1) \times (m + 1)$ matrix D. Consider two neighboring columns x_{t-1} and x_t. A path from $(0, 0)$ to (i, x_t), a cell in x_t, must go through one of the cells $(0, x_{t-1}), \cdots, (i, x_{t-1})$

in x_{t-1}. We call a path from $(0,0)$ a *connection path* for (l, x_{t-1}) $(0 \leq l \leq i)$ and (i, x_t) if the path passes both cells and does not pass any other cells in column x_{t-1} or x_t between these two cells. The optimal connection path (the one with the minimal score) is composed of the optimal path from $(0,0)$ to (l, x_{t-1}), and the path segment between (l, x_{t-1}) and (i, x_t) with the lowest score. Its score can be represented as a linear expression with the symbol $D(l, x_{t-1})$ and the symbol related to x_t, and simplified using the fact that all nucleotides between the two columns are known. Particularly, an expression that compares the scores of two different connection paths can often be reduced: for example, we know that a path with a score $D(l, x_{t-1}) + C_1 + s_1(i, x_t)$ is better than the one with $D(l, x_{t-1}) + C_2 + s_1(i, x_t)$ if the constant C_1 is smaller than C_2. The optimal path to (i, x_t) is either one of the $i+1$ optimal connection paths from $(0, x_{t-1}), \cdots, (i, x_{t-1})$ or the path passing $(i-1, x_t)$. Seeking the optimal path from $(0,0)$ to (n, m), we need to first find values for column $x_{\rho m}$, which depends on column $x_{\rho m-1}$ and so on. This forms a DPA (See Equation 3 in [68]). Computing $D(i, x_t)$ requires comparing the scores of $i+2$ paths ($i+1$ optimal connection paths and an additional path from $(i-1, x_t)$). Therefore, the complexity for computing unknown column x_t is $O(n^2)$. Since there are totally ρm unknown columns, the complexity for answering the query becomes $O(\rho m n^2)$.

Discussion. The complexities of an unspecialized DPA is $O(mn)$ for both computation and space. More often than not, the optimal path with at least m elements needs to be delivered from the DP to the DC if the whole computation task is delegated to the DP. On the other hand, most genome computing tasks involve a short α, on the order of 10^2, and a long β, from 10^6 (a chromosome) to 10^9 (the whole genome sequence of a human). Therefore, given $\rho < 10^{-4}$, the query program generated by our approach can be hundreds of times more efficient than the original program in terms of computation and space. Our approach incurs extra communication overheads: the complexity of the communication from the DC to the DP can be $O(\rho m n^2)$. This weakness, however, is compensated by the efficiency of the communication from the DP to the DC, which is only $O(\rho m)$. This is because to empower the DC to figure out the whole optimal path, the DP only needs to disclose the intersections between the optimal path and unknown columns $x_1, \cdots, x_{\rho m}$, and for every intersection (i, x_t), the one of the $i+2$ paths that contributes to the value of the cell.

Actually, the theoretic result turns out to be too pessimistic, because our analysis does not consider the reduction achievable using the value ranges of expressions: due to the scarcity of unknown nucleotides, the differences between the constants in the expressions for two optimal connection paths can easily overwhelm the deviations caused by an unknown symbol; as a result, optimal connection paths from different cells in x_{t-1} can often be compared and many of them can be removed from the reduced expression of $D(i, x_t)$. In our experiment, we observed that a query was at least thousand times more efficient than the original program, in terms of computation, space and communication (Section 4).

DPA extensions. DPAs used in genome computing can be extended to improve their performance. Two prominent examples are Divide-and-Conquer, which is optimized for space efficiency, and BLAST, which is designed for high performance. The Divide-and-Conquer algorithm (DCA) [54] first runs a DPA to compute the first half of matrix D column by column until j, the column in the middle of the matrix, and then compute the second half backwards from column n to j. As a result, the intersection between the optimal path and column j is identified. Denote the intersection by (i, j). The same process happens to the matrix between $(0,0)$ and (i, j) and the one between (i, j) and (n, m) to find other members on the optimal path, which further divides these matrices into smaller ones. As such, the algorithm can determine every member on the path. Since computing a column only needs the information in the prior column, DCA reduces the spatial complexity of a DPA to $O(m+n)$, at the cost of doubled computation overheads.

The DCA needs to run a DPA over the whole matrix once, which makes the complexities of the query generated from specialization stay at $O(\rho m n^2)$. Apparently, this suggests that the query program loses the edge in space efficiency. Again, such a theoretic result is deceiving: the query built upon real data is actually much more efficient, as observed in our research.

BLAST is a widely-used algorithm for fast searching. It first searches for high scoring subsequence matchings between the sequences α and β by seeking *words*, a subsequence typically containing 11 nucleotides, with scores above a threshold. Then, the algorithm extends these words using a DPA to find a locally optimal alignment. Our specialization techniques generate queries for extending words, which is much more efficient than running the whole algorithm on the DP. A problem is that the score of a word is usually calculated using exact match. When a word matches a sequence involving sensitive nucleotides, these nucleotides will be exposed, which could cause a computation to fail. Fortunately, the number of sensitive nucleotides in a given β is usually very small, and as a result, the chance that a word in a short α matches a sequence involving such nucleotides is very low.

2.4 Query Auditing

Our framework adopted a simple security policy to control information leaks from the outcomes of a computation. The policy specifies a *threshold* for a query, the maximal number of SNPs whose values can be revealed. For each query, the DP first runs a query auditor to evaluate the amount of information that could be leaked out by the answer: if it goes above the threshold, the DP refuses to respond; otherwise, the query is allowed to be answered. The query auditor can be as simple as a constraint solver: given a query and its answer as a constraint, it attempts to determine whether the constraint can only be satisfied when some SNPs take unique values; when this happens, these SNPs are deemed disclosed if the answer is given to the DC. For example, consider a query q for an edit distance, whose answer is 5; if the auditor finds that to satisfy the constraint "$q = 5$", a SNP must be '`A`', it concludes that the SNP will be disclosed by the answer. In Section 4, we demonstrate that this simple technique actually worked on realistic computations.

The action of denying a query itself can leak information: at the very least, an attacker knows that the answer to her query can be used to determine at least t SNPs, where t is the threshold. However, by setting the threshold well below the number of SNPs involved, we can make it difficult for the attacker to find out exactly which t SNPs can be determined. In general, however, we do not want to claim that this approach is a perfect solution. Instead, it is just a component of our framework and can be replaced with other existing technologies for query auditing [55, 40] and inference control [40, 55, 50, 51, 31, 24]. Study of these technologies' efficacy under our framework is left as future research.

2.5 Secure Multi-party Computation

The DC's sequence α may contain sensitive nucleotides that cannot be revealed to the DP. When this happens, a query needs to be answered without leaking out sensitive inputs from both α and β, which can be achieved using secure multi-party computation [69, 29]. Direct application of SMC on α and β, however, can introduce huge performance overheads, making the approach hard to scale [34]. Our solution is to use nonsensitive data on both α and β

to specialize a computation, reducing its complexity. Specifically, let the set of sensitive nucleotides on α be $\{\alpha[y_\tau]\}$, and the set for β be $\{\beta[x_t]\}$. These nucleotides are all marked as symbols on the sequences. Performing a mixed execution on them, the DC can acquire a query q with $\{\alpha[y_\tau]\}$ and $\{\beta[x_t]\}$ as inputs. Such a query is typically much more efficient than the original program, as demonstrated in our experimental studies (Section 4). To seek the answer for q, the DC converts it into a circuit Q and further encrypts it to create a "garbled circuit" Q'. Over Q', the DC and the DP can run an SMC to compute the answer to the query. Compared with the prior work [34], our approach is much more efficient, as Q' can be very small, and therefore can handle a computation task with a much larger scale (on the order of tens of thousands of nucleotides).

A problem here is that SMC does not offer protection to the information revealed by the outcome of a computation. A solution can be to let the DP evaluate the unencrypted circuit Q without access to $\{\alpha[y_\tau]\}$ before SMC happens. This is feasible because α is usually very short, involving only a few hundreds of nucleotides, and as a result, typically no more than 5 of them are SNP [43]. Therefore, the DP can check all 4^5 possible combinations of these nucleotides to ensure that none of them will cause the answer for q to violate privacy policies, for example, exposing more nucleotides than permitted by a threshold. In the case that the size of $\{\alpha[y_\tau]\}$ is large, a solution could be randomly sampling some of combinations for policy verification. Note that we can hide the outcome of such a computation from the DP, which eliminates the concern of leaking the DC's data to the DP through the outcomes. The effectiveness of such an approach, however, needs to be further studied in the future research.

3. PROGRAM TRANSFORMATION

This section describes a tool for transforming legacy biocomputing code into a new program to perform mixed executions on sanitized genome sequences. Our current design is for converting Java programs, but the idea behind it can work on the programs in other languages. We also implemented a prototype using Java.

To transform a Java program, our tool takes the following steps. It first runs a transformation tool such as Java2XML [3] to convert the source code into an *abstract syntax tree* (AST) that describes the structure of the program [12]. The AST representation clearly indicates different elements of the program, including variables and statements, and their relations, in particular execution flows, over which a *taint analysis* is performed to find out all the elements tainted by sensitive nucleotides. These elements are further instrumented with specialization code to support mixed executions. Finally, the transformed AST is converted into a new Java program through XSLT stylesheet [12].

3.1 Taint Analysis

The objective of taint analysis is to identify all statements and variables affected by sensitive nucleotides. The statements in a program that import these data are manually annotated as taint sources. Starting from them, our approach statically analyzes the propagation of tainted data on the AST in accordance with a set of propagation rules. Such a rule is in the form of `(s,i,o,e)`, in which `s` is a statement, `i` and `o` represent the input and the output of the statement respectively, and `e` is a Boolean value that indicates whether execution of the statement will cause taint to be propagated from `i` to `o`. For instance, the rule `(=, value, variable, true)` specifies that an assignment statement ("=") will propagate taint from its input (`value`) to its output (`variable`).

Let V be the set of tainted variables and S be the set of tainted statements. These sets include only the taint sources at the beginning of an analysis. During the analysis, our analyzer checks every element on the AST according to the execution flow of the program, identify tainted variables and the statements that operate on these variables using propagation rules, and put them to V and S respectively. Some statements need special treatment. Specifically, our analyzer forks threads to explore different branches of a branching statement to the point where they converge. For a loop statement, we need to consider the propagation of taint across different iterations. Consider the example in Figure 2 from Line 15 to 18 of $P1$, in which $\min(a, b, c)$ is computed by first comparing a and b to find the smaller one and then comparing it with c. These operations are embedded in the loop from Line 2 and 7. An interesting observation is that if c is tainted, the first iteration of the loop only taints the statement at Line 17 and array D. However, the next iteration sees the statement at Line 16 also become tainted because this time, D is tainted. Our solution to the problem is to statically analyze the loop iteration by iteration, until no new tainted variables or statements are discovered.

Another important issue we had to deal with is propagation of taint through *control flow*. This happens when a branch condition becomes tainted. As a result, sensitive inputs could affect the use of the statements and variables within the *scope* [5] of the branching, that is, part of the program between the condition and the program location where all branches converge. For example, the score function $S1$ in Figure 2 contains a branch that a comparison between two nucleotides, one of which can be sensitive, determines the score it returns. In this case, we taint all the variables within the scope of the branching to be used posterior to the statement. For the example in Figure 2, the output of $S1$ is tainted.

3.2 Code Instrumentation

```
Original code
1   int I;
2   I = 5;

Transformed code snippet
1  IntSymbol I=new IntSymbol();
2  I.assign(5);
3  class IntSymbol {
4    Object value;
5    void assign(int i) { value = i;  }
6    void assign(exp e) {  value = reduce(e); }
7  }
```

Figure 4: Integer variable transformation.

Tainted program elements need to be transformed to enable a mixed execution. This was achieved in our research through replacing a tainted variable with a class that accepts both concrete values and a symbolic expression, and transforming tainted statements into the forms that can work on these variables. Figure 4 presents an example, in which an integer variable I is converted into a new type `IntSymbol`, a class accepting both concrete and symbolic values. To perform an operation on such a variable, proper instrumentation needs to be done to operators, such as assignment and addition. In Figure 4, an assignment of a value to I is modified to be performed by `assign()`,the method of `IntSymbol`: the method does the normal assignment when its input is concrete, and maintains and reduces an expression when the input is symbolic.

A tainted statement is replaced with a code snippet according to its type, as described in Section 2.2. A problem is that a program could call a function from other libraries whose source code may not be available. This is tackled by our instrumentation tool through redirecting such a call to a wrapper of the function being called. The wrapper checks the parameters of the call: if any of them is symbolic, it returns a new symbol to enable the follow-up

operations and residualizes the call; otherwise, it passes the parameters to the callee.

4. EVALUATION

This section reports an empirical study of the techniques we propose. The genome sequences used in our study came from the human genome dataset in UCSC Genome Browser [39], the latest Build 36.1 assembled on March 2006. We extracted segments from the dataset and truncated them into sequences of different sizes for our experiments. These sequences were sanitized by replacing their SNP nucleotides, as indicated by the International HapMap Project [27], with symbols.

4.1 Program Transformation

We ran our program-transformation prototype on 7 Java-based DPA implementations, including 3 bioinformatics libraries and 4 synthesized programs, as illustrated in Table 1. Our prototype transformed all synthesized programs and most part of the libraries. The new programs and the queries they generated were evaluated using genome data, and their outcomes were found to be identical with those produced by running the original programs on unprotected sequences. This indicates that the transformation was sound. Following we describe our experiences with the Java libraries.

NeoBio [22] is a Java library including three pair-wise alignment algorithms, Needleman-Wunsch [56], Smith-Waterman [64], and Crochemore-Landau-Ziv-Ukelson [21]. Our tool failed to transform the last one because it intensively uses tainted addresses: it performs computation upon a double-linked list constructed based on the values of individual nucleotides. As a result, our analyzer found that nearly all the statements of the algorithm had to be residualized. This problem comes from the limit support our current design offers for symbolic addresses, which will be addressed in our follow-up research.

Argo genome browser [1] includes 48 class files to support both global and local alignment algorithms. Most of the classes, however, are different designs of score functions, which can be residualized without incurring noticeable performance overheads to a query program. The library was successfully converted by our prototype and evaluated in our experiments. The same success also happened to JAligner [2], a Java implementation of the Smith-Waterman algorithm with Gotoh's improvement. An interesting property of this algorithm is that it maintains a $(n+1)\times(m+1)$ matrix to record the neighbor of each cell that contributes to its value. This simplifies the "backtracking" process for identifying the optimal path. During the program's runtime, our specialization code assigned symbols to cells after unknown nucleotides were encountered. The concrete values of these cells were calculated from the DP's answer to the query exported by the program, which included the intersections between the optimal path and unknown columns, and the symbolic expressions contributing to the values of these intersections.

4.2 Performance

We ran the transformed programs on real genome sequences to study their performance. Our experiments were conducted on two laptops, each with a 1.8G Intel Core 2 Duo CPU and 2 GB memory. One of these laptops was used as the DP, and the other as DC. They communicated with each other through a local network. In the experiments, we measured the computation time and memory use for both mixed executions on sanitized data that happened on the DC side, and executions of the queries generated thereby on the DP side. Such information was compared with the computational and spatial overheads for directly running the original programs on unprotected data, which served as baselines. We also recorded the communicational overheads incurred by the interactions between the DP and the DC.

Table 2 illustrates the experimental results, in which the problem sizes are described as (n, m), where n and m represent the sizes of α (the DC's sequence) and β (the DP's sequence) respectively. Our experiments include an edit distance (row 1), 2 global alignments (row 2 and 3), 4 local alignments (row 4 to 7), longest common sequence (LCS) identification (row 8) and 1 multiple alignment. The multiple alignment algorithm computes over three sequences. The last one belongs to the DP and contains one SNP. The problem sizes we chose ranged from hundreds of nucleotides to a million of nucleotides. The number of sensitive nucleotides on β varied according to the problem sizes, from a single one to 1056. In the table, the baseline results are labeled as "Native".

The table shows that the mixed execution did take a noticeable toll on the DC's performance. Compared with the baseline, transformed programs were typically one order of magnitude slower and consumed more memory. Such a raise of overheads culminated in the experiment involving the Needleman-Wunsch algorithm from the NeoBio library, which brought in a slow down factor of 64 and used 28 times more memory. However, the DC's cost seems to be compensated by the huge performance gain on the DP side: the query programs generated by the DC were so efficient that they were at least 10000 times faster than the baseline and typically consumed much less memory. Actually, computing the answer for a query never took more than 100 microseconds. Particularly, the transformed Divide-and-Conquer algorithm (row 8) even enabled the DC to accomplish the computation without querying the DP at all. This is because in that experiment, the constant in Equation 2 was found to be below the value ranges of all symbolic expressions, and as a result, a concrete outcome ensued. Note this would not be possible without specialization. Moreover, the communication overheads were also found to be very low. This is in a stark contrast with the conservative estimate made in our theoretic analysis (Section 2.3), which predicts much higher overheads.

4.3 Information Leaks

We also evaluated the information leaks that can be caused by releasing the outcomes of query programs, using a query auditor built upon a constraint solver [25]. The outcomes are shown in Table 3. This study was conducted under three scenarios: an answer includes only a value (e.g., an edit distance), a path for optimal alignment or both. From the table, we can see that the amount of information disclosed by answers is pretty low: ranging from 0% to 1.8%. The performance of constraint solving was also reasonable: from 0.001 to 0.3 seconds.

4.4 Secure Multi-party Computation

We studied how our specialization techniques could facilitate secure multi-party computation when the DC's sequence α also contains sensitive nucleotides. In our experiment, we ran the transformed edit-distance program on the sanitized sequences α and β, whose SNP nucleotides were replaced by symbols. This produced a query program, which was converted into a "garbled circuit" using a tool we developed. After that, the DP and the DC ran an SMC protocol [34] to evaluate the circuit. In the experiment, we measured the accumulated computation time and memory use on the DC side, including those for program specialization and running the SMC protocol, as well as the overheads on the DP side for performing its part of the protocol. These results were compared with the overheads of running an optimized SMC protocol [34] directly on α and β. The optimized SMC protocol we used is an implementation of Protocol 3 proposed in the prior work [34]. The protocol is recommended for computing large-size problems, as it strikes a

Table 1: Transformed programs.

Program Name	Source	# of Class Files	Included Algorithms
NeoBio	library	22	Needleman-Wunsch,Smith-Waterman,Crochemore-Landau-Ziv-Ukelson
Argo genome browser	library	48	Global Alignment, Local Alignment
JAligner	library	16	Smith-Waterman algorithm with Gotoh's improvement
Edit Distance	synthesized	1	Edit Distance
Blast	synthesized	2	Blast
Divide-and-Conquer	synthesized	2	Divide-and-Conquer
Multiple Alignment	synthesized	1	Multiple Alignment

Table 2: Performance.

Algorithm	Problem Size	SNP	Native		DC		DP		Bandwidth (KB)
			Time(s)	Mem(MB)	Time	Mem	Time	Mem	
Edit Distance	400×400	2	0.523	3.665	28.526	46.915	0.000033	1.536	1.841
NeoBio Needleman	400×400	2	0.665	2.052	42.465	56.897	0.000078	2.740	2.760
Argo Global Alignment	400×400	2	0.801	3.432	46.151	44.736	0.000054	2.740	2.535
NeoBio Waterman	200×100000	55	95.664	87.830	1009.029	626.167	0.000014	2.740	1.968
Blast	200×100000	55	2.416	18.624	64.286	49.893	0.000019	2.740	2.017
Argo Local Alignment	200×100000	55	109.132	133.521	1512.368	661.880	0.000017	2.740	1.996
JAligner Waterman	200×100000	55	27.056	124.215	1637.066	604.712	0.000016	2.740	1.968
Divide-and-Conquer	200×1000000	1056	646.909	34.738	6857.100	168.808	0	0	0
Multiple Alignment	100×100×100	1	6.545	6.606	394.865	113.188	0.000021	2.052	2.038

balance between computation time and memory use [34]. We also recorded the bandwidth consumptions for both our approach and the prior approach. The results are presented in Table 4.

As illustrated by Table 4, the optimized SMC protocol took more than an hour and 2.56GB bandwidth to deal with a 200×1000 problem. This is actually not necessary, as real genome sequences of such sizes typically contain very few sensitive nucleotides. In contrast, our approach first specialized the computation to a much smaller problem and then performed SMC on it. Though the SMC protocol we used was not optimal, we achieved a significantly better performance: a little more than 106 seconds in computation time and merely 5.2 KB in bandwidth usage. When the problem size grew to 300×10000, direct application of SMC to the whole sequences could not finish the computation in 3 hours, while our approach accomplished the task within 411 seconds with 13.9KB bandwidth consumption. This result demonstrates that our approach can offer more practical privacy protection for genome computing.

5. DISCUSSION

Beside the threshold policy, our framework is open to other privacy policies [46, 44, 33] for regulating information leaks from outcomes of a computation. We feel that there is reason to expect these policies and their enforcement, which have been extensively studied in database security, to be successfully combined into our approach, as the interactions between the DP and the DC does not have any substantial difference from those between clients and a database. However, it remains to be seen how effectively and efficiently these techniques work under our framework, which is left as future research.

Though DPAs are among the most important building blocks for computational genomics, there are many other algorithms that need to be further studied. For decades, program specialization has been proven to be valuable to the research on a wide variety of areas, including compiler generation [36], computer graphics [49] and others. We expect the same success in applying the techniques to genomic computations. On the other hand, it is also important to understand the limitations of specialization techniques. For example, some data structures such as suffix trees [48] can be hard to specialize because to correctly build these structures, the DC may have to know the exact values of every genome strings, including those carrying sensitive information.

Our program-transformation tool is designed for Java. Actually, `C` is more pervasive in genomic computation, due to its high performance. Given its complicated structure, in particular, extensive use of pointers, `C` programs can be more difficult to transform. Prior research [8] studies specialization of part of `C`, which we can take advantage of. Such work, however, is more about partial evaluation at compiling time than offline transformation for run-time specialization of existing code, which is our focus.

6. RELATED WORK

Privacy preserving computations over genome data have been studied recently. Most prior approaches are based on cryptographic protocols [10, 34, 15]. A prominent example is the recent work that optimizes SMC techniques for computing DPA-based bioinformatics algorithms [34]. This approach significantly improves the efficiency of SMC and is demonstrated to be very effective on small-scale computing tasks, such as global alignment involving hundreds of nucleotides. However, it is unable to deal with a large-scale computation, as demonstrated in Section 4. Another approach is distributed Smith-Waterman algorithm [65] that decomposes a computation problem into small sub-problems and allocates them to multiple problem solvers. This technique, however, leaks more information than what is revealed by the outcome of a computation, and offers little privacy guarantee. In contrast, our approach takes advantage of the fact that a genome sequence is actually a mixture of public and sensitive data, and only a very small portion of it needs protection. As a result, we can simplify a computation to the extent that millions of nucleotides can be easily handled and information leaks can be effectively assessed.

Information flow security was proposed decades ago [23, 13] and its application to programming languages like Java has also been studied for many years. A prominent example is Jif [52, 53], a security-typed programming language that supports information-

Table 3: Information Leakage.

Algorithm	Problem Size	SNP	Value		Path		Value and Path	
			Leakage(%)	Time(s)	Leakage(%)	Time(s)	Leakage(%)	Time(s)
Edit Distance	400×400	2	0	0.053	0	0.063	0	0.070
NeoBio Needleman	400×400	2	0	0.274	0	0.209	0	0.304
Argo Global Alignment	400×400	2	0	0.249	0	0.196	0	0.291
NeoBio Waterman	200×100000	55	1.8	0.010	1.8	0.010	1.8	0.005
Blast	200×100000	55	1.8	0.007	1.8	0.003	1.8	0.011
Argo Local Alignment	200×100000	55	1.8	0.003	1.8	0.005	1.8	0.001
JAligner Waterman	200×100000	55	1.8	0.008	1.8	0.008	1.8	0.004
Divide-and-Conquer	200×1000000	1056	0	0	0	0	0	0

Table 4: Facilitation of secure multi-party computation.

Problem Size	SNP		Optimized SMC [34]			Our Approach				
	DP	DC	Time	Mem	Bandwidth	DP		DC		Bandwidth
						Time	Mem	Time	Mem	
200 × 1000	3	1	1h5min	5.95MB	2.56GB	392.5ms	5.12MB	106.2s	62.3MB	5.2KB
300 × 10000	4	2	>3h	n/a	n/a	537ms	5.12MB	410.5s	223.2MB	13.9KB

flow (IF) control within Java. Jif is designed for enforcing different IF policies in a program, which is more than we need. Therefore, we did not build our prototype over it, and instead, implemented our own lightweight tool for taint analysis. Based on Jif, Swift [18] uses IF analysis to separate an application according to security policies. This is insufficient for our purpose, because for the bio-computing algorithms like DPA, the part of the computation on the sensitive data can be intertwined with that on the public data. For example, a static IF analysis on a DPA could taint all the statements once the program receives a single sensitive nucleotide. As a result, the whole program has to be placed on the DP side, which is exactly what we intend to avoid.

Program specialization and partial evaluation have been studied for decades [35, 19, 20, 62, 60, 28], and are extensively used in compiler generation, real-time systems and many other areas [36, 57, 19, 20, 37, 41, 49, 11]. To our knowledge, our approach is the first attempt to apply program specialization to privacy-preserving genomic computations. For this purpose, we propose new specialization techniques tuned to the properties of genome data, including the rules for reducing Boolean expressions with value ranges and determining when to unfold a symbol. We prove the effectiveness of these techniques on a category of DPAs that are common in genome computing. Moreover, different from existing partial evaluators [35] that work at the compiling stage, our approach specializes an algorithm at runtime. This aligns our approach with the techniques for dynamic code generation [59]. However, unlike these techniques, we do not rely on a language to define the way to generate new programs, and instead, use program analysis to retrofit legacy programs for specialization operations.

7. CONCLUSION

This paper presents an innovation that makes an important step toward practical privacy-preserving genomic computations. Our approach is based upon the fact that only a very small portion of human genome contains sensitive information. Therefore, a data provider can announce sanitized genome sequences to enable computations on pubic data, and answers the queries about sensitive data when its privacy policy permits. These queries are generated in our research through program specialization. We theoretically analyzed the effectiveness of our approach on a set of DPAs common in computational genomics, and experimentally demonstrated its capability to handle the computation tasks with practical scales. We developed a program transformation tool to automatically convert existing bioinformatics programs to the forms capable of performing privacy-preserving operations. We also studied how our techniques can facilitate SMC on genome computing problems.

8. ACKNOWLEDGEMENTS

We thank Louis Kruger, Somesh Jha, and Vitaly Shmatikov for sharing with us the code for optimized SMC [34]. This work was supported in part by the National Science Foundation the Cyber Trust program under Grant No. CNS-0716292.

9. REFERENCES

[1] Argo genome browser. http://www.genome.wi.mit.edu/annotation/argo/.

[2] Jaligner: java implementation of the smith-waterman algorithm for biological sequence alignement. http://jaligner.sourceforge.net/.

[3] Java2xml : A java to xml converter. https://java2xml.dev.java.net/.

[4] Genetic variation program. http://www.genome.gov/10001551, 2008.

[5] F. E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, 1990.

[7] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res*, 25(17):3389–3402, Sep 1997.

[8] L. O. Andersen. Program analysis and specialization for the c programming language. Phd thesis, Department of Computer Science, University of Copenhagen, May 1994.

[9] S. Artzi, A. Kiezun, and N. Shomron. miRNAminer: a tool for homologous microRNA gene search. *BMC Bioinformatics*, 9:39, 2008.

[10] M. J. Atallah, F. Kerschbaum, and W. Du. Secure and private sequence comparisons. In *WPES '03: Proceedings of the 2003 ACM workshop on Privacy in the electronic society*, pages 39–44, New York, NY, USA, 2003. ACM.

[11] W.-Y. Au, D. Weise, and S. Seligman. Generating compiled simulations using partial evaluation. In *DAC '93: Proceedings of the 28th Design Automation Conference*, pages 205–210, New York, NY, USA, 1991. IEEE.

[12] G. J. Badros. Javaml: a markup language for java source code. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 159–177, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[13] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, Hanscom AFB, Bed-ford, Mass., November 1973.

[14] R. Bellman. Dynamic programming. *Science*, 153(3731):34 – 37, 1966.

[15] F. Bruekers, S. Katzenbeisser, K. Kursawe, and P. Tuyls. Privacy-preserving matching of dna profiles. Technical Report Report 2008/203, ACR Cryptology ePrint Archive, 2008.

[16] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[17] N. E. Castellana, S. H. Payne, Z. Shen, M. Stanke, V. Bafna, and S. P. Briggs. Discovery and revision of Arabidopsis genes by proteogenomics. *Proc. Natl. Acad. Sci. U.S.A.*, 105:21034–21038, Dec 2008.

[18] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.

[19] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 493–501, New York, NY, USA, 1993. ACM.

[20] C. Consel and S. C. Khoo. Semantics-directed generation of a prolog compiler. *Sci. Comput. Program.*, 21(3):263–291, 1993.

[21] M. Crochemore, G. M. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 02)*, 2002.

[22] S. A. de Carvalho Junior. Neobio - bioinformatics algorithms in java. `http://neobio.sourceforge.net/`.

[23] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[24] J. Domingo-Ferrer, editor. *Inference control in statistical databases: From theory to practice*. Springer, 2002.

[25] B. Dutertre and L. Moura. The YICES SMT Solver. `http://yices.csl.sri.com/`, as of 2008.

[26] R. C. Edgar and S. Batzoglou. Multiple sequence alignment. *Current Opinion in Structural Biology*, 16(3):368–373, 2006.

[27] R. Gibbs. The international hapmap project. *Nature (London)*, 426:789, 2003.

[28] R. Glück and J. Jorgensen. Efficient multi-level generating extensions for program specialization. In *PLILPS '95: Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 259–278, London, UK, 1995. Springer-Verlag.

[29] O. Goldreich, S.Micali, and A.Wigderson. How to play any mental game. In *STOC*, 1987.

[30] O. Gotoh. An improved algorithm for matching biological sequences. *J Mol Biol*, 162(3):705–708, December 1982.

[31] V. Goyal, S. K. Gupta, and A. Gupta. A unified audit expression model for auditing sql queries. In *Proceeedings of the 22nd annual IFIP WG 11.3 working conference on Data and Applications Security*, pages 33–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] N. Gupta, S. Tanner, N. Jaitly, J. N. Adkins, M. Lipton, R. Edwards, M. Romine, A. Osterman, V. Bafna, R. D. Smith, and P. A. Pevzner. Whole proteome analysis of post-translational modifications: applications of mass-spectrometry for proteogenomic annotation. *Genome Res.*, 17:1362–1377, Sep 2007.

[33] J. N. Hirschhorn and M. J. Daly. Genome-wide association studies for common diseases and complex traits. *Nature Reviews Genetics*, 6(2):95–108, February 2005.

[34] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *2008 IEEE Symposium on Security and Privacy*, 2008.

[35] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation, C.A.R. Hoare Series*. Prentice-Hall, 1993.

[36] N. D. Jones, P. Sestoft, and H. Sondergaard. An experiment in partial evaluation: the generation of a compiler generator. In *Proc. of the first international conference on Rewriting techniques and applications*, pages 124–140, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[37] J. Jorgensen. Generating a compiler for a lazy language by partial evaluation. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 258–268, New York, NY, USA, 1992. ACM.

[38] O. Keller, F. Odronitz, M. Stanke, M. Kollmar, and S. Waack. Scipio: using protein sequences to determine the precise exon/intron structures of genes and their orthologs in closely related species. *BMC Bioinformatics*, 9:278, 2008.

[39] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at ucsc. *GENOME RESEARCH*, 25(6):996–1006, 2002.

[40] K. Kenthapadi, N. Mishra, and K. Nissim. Simulatable auditing. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 118–127, New York, NY, USA, 2005. ACM.

[41] S. C. Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In *PEPM '91: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 211–222, New York, NY, USA, 1991. ACM.

[42] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[43] L. Kruglyak and D. Nickerson. Variation is the spice of life. *Nat. Genet.*, 27:234–236, Mar 2001.

[44] N. Li and T. Li. t-closeness: Privacy beyond k-anonymity and âĎŞ-diversity. In *In Proceedings of IEEE International Conference on Data Engineering*, 2007.

[45] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, Mar 2002.

[46] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1):3, 2007.

[47] B. Malin. Protecting dna sequence anonymity with generalization lattices. Technical Report CMU-ISRI-04-134, Carnegie Mellon University, As of October 2007.

[48] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[49] T. Mogensen. The appliation of partial evaluation to ray-tracing. Master thesis, DIKU, University of Copenhagen, 1986.

[50] R. Motwani, S. Nabar, and D. Thomas. Auditing a batch of sql queries. *Data Engineering Workshop, 2007. IEEE 23th International Conference on*, pages 186–191, April 2007.

[51] R. Motwani, S. Nabar, and D. Thomas. Auditing sql queries. *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 287–296, April 2008.

[52] A. C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, pages 228–241, 1999.

[53] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[54] E. W. Myers and W. Miller. Optimal alignments in linear space. *CABIOS*, 4:11–17, 1988.

[55] S. U. Nabar, B. Marthi, K. Kenthapadi, N. Mishra, and R. Motwani. Towards robustness in query auditing. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 151–162. VLDB Endowment, 2006.

[56] W. C. Needleman SB. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3):443–453, 1970.

[57] V. Nirkhe and W. Pugh. Partial evaluation of high-level imperative programming languages with applications in hard real-time systems. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 269–280, New York, NY, USA, 1992. ACM.

[58] G. Pavesi, F. Zambelli, C. Caggese, and G. Pesole. Exalign: a new method for comparative analysis of exon-intron gene structures. *Nucleic Acids Res.*, 36:e47, May 2008.

[59] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. C and tcc: a language and compiler for dynamic code generation. *ACM Trans. Program. Lang. Syst.*, 21(2):324–369, 1999.

[60] T. W. Reps and T. Turnidge. Program specialization via program slicing. In *Selected Papers from the Internaltional Seminar on Partial Evaluation*, pages 409–429, London, UK, 1996. Springer-Verlag.

[61] R. G. Sadygov, D. Cociorva, and J. R. Yates. Large-scale database searching using tandem mass spectra: looking up the answer in the back of the book. *Nat. Methods*, 1:195–202, Dec 2004.

[62] U. P. Schultz, J. L. Lawall, C. Consel, and G. Muller. Towards automatic specialization of java programs. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 367–390, London, UK, 1999. Springer-Verlag.

[63] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller. Human-mouse alignments with blastz. *Genome Res*, 13(1):103–107, Jan 2003.

[64] W. M. Smith TF. Identification of common molecular subsequences. *J Mol Biol*, 147:195, 1981.

[65] E. Szajda, M. Pohl, J. Owen, and B. Lawson. Toward a practical data privacy scheme for a distributed implementation of the smith-waterman genome sequence comparison algrotihm. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 06)*, 2006.

[66] T. A. Tatusova and T. L. Madden. Blast 2 sequences - a new tool for comparing protein and nucleotide sequences. *FEMS Microbiology Letters*, 174:247–250, 1999.

[67] D. Tsur, S. Tanner, E. Zandi, V. Bafna, and P. A. Pevzner. Identification of post-translational modifications by blind search of mass spectra. *Nat. Biotechnol.*, 23:1562–1567, Dec 2005.

[68] R. Wang, X. Wang, Z. Li, H. Tang, M. K. Reiter, and Z. Dong. Privacy-preserving genomic computation through program specialization. Technical Report IUCS-TR679, Indiana University, 2009.

[69] A. Yao. How to generate and exchange secrets. In *FOCS*, 1986.