Privacy-Preserving Remote Diagnostics

Justin Brickell Donald E. Porter Vitaly Shmatikov Emmett Witchel
The University of Texas at Austin
{jlbrick,porterde,shmat,witchel}@cs.utexas.edu

ABSTRACT

We present an efficient protocol for privacy-preserving evaluation of diagnostic programs, represented as binary decision trees or branching programs. The protocol applies a branching diagnostic program with classification labels in the leaves to the user's attribute vector. The user learns only the label assigned by the program to his vector; the diagnostic program itself remains secret. The program's owner does not learn anything. Our construction is significantly more efficient than those obtained by direct application of generic secure multi-party computation techniques.

We use our protocol to implement a privacy-preserving version of the Clarify system for software fault diagnosis, and demonstrate that its performance is acceptable for many practical scenarios.

Categories and Subject Descriptors

E.3 [**Data**]: Data Encryption; I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems

General Terms

Algorithms, Security, Performance

Keywords

Privacy, Data Mining, Diagnostics, Branching Programs

1. INTRODUCTION

Diagnostic programs, typically represented as decision trees or binary branching programs, are the cornerstone of expert systems and data analysis tools. Learning and evaluating diagnostic programs which classify data on the basis of certain features are among the most fundamental data mining tasks.

Evaluation of a diagnostic program on a remote user's data often presents privacy risks to both the user and the program's owner. The program's owner may not want the user to learn the entire contents of the diagnostic program, while the user may not want to reveal his local data to the program's owner. For example, consider a medical expert system, where the diagnostic program is the realization of a substantial investment, and the data on which the program is evaluated contain information about the user's health.

Another example is remote software fault diagnosis, which is an increasingly popular support method for complex applications. The details of remote software diagnostic systems differ (see Section 5), but there are many commonalities. An application does something undesirable (crashes, becomes slow or unresponsive, quits with an obscure error message), and the runtime system gathers some data about the problem. The software manufacturer uses this information to diagnose the problem, usually by reading a small subset of the data. Users are typically required to ship all fault-related data to the manufacturer. For example, most users of Microsoft Windows have encountered the (in)famous "send error report" button.

The data gathered by the runtime system may contain sensitive information, such as passwords and snippets of the user's documents. Many users are not willing to reveal this information to the software manufacturer. On the other hand, software manufacturers often view their proprietary diagnostic programs as valuable intellectual property. Diagnostic programs may reveal the application's support history, unpatched security vulnerabilities, and other information about the implementation and internal structure of the application that the manufacturer may prefer to keep secret.

This paper describes a method for *privacy-preserving evaluation of diagnostic branching programs*. This problem is different from privacy-preserving *learning* of decision trees, which has been the subject of much research [1, 4, 22]. We assume that the diagnostic program already exists, in the form of a binary decision tree or branching program, and investigate how to *apply* it to the user's data in such a way that the program is not revealed to the user, and the user's data are not revealed to the program's owner.

Our contributions. We present a practical, provably secure interactive protocol for privacy-preserving evaluation of branching programs. The protocol takes place between a Server, in possession of a binary branching program T, and a User, in possession of an attribute vector v. The User learns $c = T(v)$, the diagnostic label that T assigns to v. The Server may or may not learn the label—we consider both variants.

Our protocol does not reveal any useful information except the outcome of the computation, which is the diagnostic label in this case. In particular, the User does not learn *how* the branching program arrived at the diagnosis, nor which of the User's attributes it considered, nor the topology of the branching program, nor any other diagnostic labels that it may contain. The Server, on the other hand, learns nothing whatsoever about the User's local data.

We emphasize the strong privacy properties achieved by our protocol. For example, secrecy of the program being evaluated is *not* the standard requirement of secure multi-party computation, which usually assumes that the program is public, and only the par-

CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.

ties' respective inputs are secret. In many of our applications, the user should not learn *which* of his attributes are considered by the branching program. If the attribute vector is very large (as is the case, for example, in software fault diagnostics, where the attribute vector is a record of the user's runtime environment), achieving these security properties efficiently is a difficult challenge.

Our branching program evaluation protocol combines in a novel way several cryptographic techniques such as homomorphic encryption, blinding, and Yao's "garbled circuits" method. Yao's method is used in a somewhat unusual way, not simply as a black-box realization of secure circuit evaluation. We exploit the details of circuit representation in Yao's protocol to implement a *conditional oblivious transfer* primitive needed by our protocol.

We present a substantial case study, in which we use our method to implement a privacy-preserving version of Clarify [17], a system for remote diagnosis of software faults. We apply our protocol to the decision trees generated by Clarify for several large, real-world applications such as `gcc` and `latex`, and demonstrate that its performance is efficient for many practical scenarios.

While there have been many theoretical results in the field of secure multi-party computation, actual implementations and working systems are extremely rare. Experimental evaluation of our prototype implementation demonstrates that our protocol performs significantly better than the generic methods.

The paper is organized as follows. We describe related work in Section 2, and our cryptographic toolkit in Section 3. In Section 4, we present our protocol. In Section 5, we apply it to privacy-preserving software fault diagnosis, and analyze its performance in Section 6. Conclusions are in Section 7.

2. RELATED WORK

This paper follows a long tradition of research in the *secure multi-party computation* (SMC) paradigm. Informally, security of a protocol in the SMC paradigm is defined as computational indistinguishability from a simulation of some *ideal functionality*, in which the trusted third party accepts the parties' inputs and carries out the computation. Formal definitions for various settings can be found, for example, in [2, 7, 15].

Any probabilistic polynomial-time multi-party computation, represented as a circuit or a binary decision diagram, can be converted into a "privacy-preserving" one using generic techniques of Yao [40] and Goldreich *et al*. [16]. Generic constructions, however, tend to be impractical due to their complexity (*e.g.*, see the comparison of our techniques with the generic approach in Section 4.5). Recent research has focused on finding more efficient privacy-preserving algorithms for problems such as computation of approximations [10], auctions [31], set matching and intersection [12], surveys [11], and various data mining problems.

Some SMC research has used branching programs instead of circuits as function representation [14,29]. It is still the case, however, that when computing $f(x, y)$ securely, f is assumed to be known to both parties, while x and y are their private inputs. In our scenario, we are computing $g(y)$, where g is the private input of the first party and y is the private input of the second party. This can be implemented by making f a generic function evaluator and x a description of the particular function g. As we show in Section 4.5, this approach does not scale to the size of branching programs that arise in real-world applications. Selective private function evaluation [8] considers evaluation of functions on large datasets, but the functions are much simpler than the branching programs considered in this paper. To achieve practical efficiency, our protocol fundamentally relies on the structure of branching programs.

Many papers investigated the problem of privacy-preserving decision tree *learning*, both in cryptographic [22] and statistical [1,4] settings. Decision tree learning is a machine learning technique for building a compact classifier that best represents a set of labeled examples. The problem considered in this paper is privacy-preserving *evaluation* of decision trees and branching programs, which is fundamentally different (and complementary) to the problem of tree learning. For example, in our protocol the tree is an input, whereas in the privacy-preserving learning protocols the tree is the output.

Concurrently and independently of this work, Ishai and Paskin presented a protocol for evaluating a branching program P on an encrypted input x in such a way that only $P(x)$ is revealed to the evaluator [19]. The representation of P must contain only single-bit decision nodes and output a single bit. This protocol appears impractical for scenarios such as remote software diagnostics where the user's input contains thousands of 32-bit values.

Crypto-computing [36] considers the problem of a circuit owner obliviously evaluating the circuit on encrypted inputs. While in our construction the User evaluates an encrypted branching program, in the crypto-computing paradigm the Server would perform the computation on the encrypted attribute vector. It is not clear whether the theoretical techniques of [36] lend themselves to a practical implementation, and shifting the burden of evaluation to the Server is not desirable in practical applications.

Several papers considered problems which are superficially similar to our remote software diagnostics scenario. The Scrash system [5] removes sensitive information from the crash data, thus enabling users who are concerned about privacy to assist in *building* a software crash diagnostic. Scrash requires re-compilation, and assumes that users have access to the program's source code. By contrast, we focus on privacy-preserving *evaluation* of "black-box" fault diagnosis programs. Unlike Scrash, our system can be applied to commercial software applications, which are compiled without symbols and distributed without source code.

The Friends Troubleshooting Network system [18, 38] allows participants in a trust network to collaborate in order to diagnose software errors. By contrast, we assume that the diagnostic tool is controlled by a single party (*e.g.*, the software manufacturer). Furthermore, our protocol is cryptographically secure.

3. CRYPTOGRAPHIC TOOLS

3.1 Oblivious transfer

Oblivious transfer (OT) is a fundamental cryptographic primitive [21, 35]. A 1-out-of-2 oblivious transfer, denoted as OT_2^1, is a protocol between two parties, the Chooser and the Sender. The Chooser's input is the index $i \in \{0, 1\}$. The Sender's inputs are the values x_0, x_1. As a result of the protocol, the Chooser learns x_i (and *only* x_i), while the Sender learns nothing.

In our constructions, we use oblivious transfer as a "black-box" primitive, *i.e.*, our constructions do not depend on a particular OT implementation. In our implementations, we employ the Naor-Pinkas constructions for OT_2^1 [30]. Each instance of OT_2^1 requires one online and one offline modular exponentiation for the sender, and one online and one offline modular exponentiation for the chooser. Amortization techniques of [30] can achieve fewer than one exponentiation per oblivious transfer, but reducing the number of exponentiations by more than a constant factor requires an impractical increase in the communication complexity.

3.2 Homomorphic encryption

A *homomorphic encryption scheme* is a semantically secure cryptosystem that permits algebraic manipulations on plaintexts given their respective ciphertexts. In this paper, we require an en-

cryption scheme with an *additively homomorphic* property, which allows $E[x_1 + x_2]$ to be computed from $E[x_1]$ and $E[x_2]$.

In our prototype implementation, we use the Paillier cryptosystem [34]. This is sufficient when the participants are semi-honest. If security against malicious participants is required, the homomorphic encryption scheme needs the additional property of *verifiability*: there should exist efficient zero-knowledge proof systems which enable a participant to prove certain relationships between the encrypted plaintexts and previously committed values (see Section 4.4). Such efficient proof systems are not known for the Paillier cryptosystem. In the malicious case, the protocol should be implemented with a homomorphic, verifiable cryptosystem, *e.g.*, the homomorphic version of the Camenisch-Shoup cryptosystem [6, 20].

3.3 Garbled circuits

Garbled circuits are a fundamental technique in secure multiparty computation. Originally proposed by Yao [40], the garbled circuits method enables secure constant-round computation of any two-party functionality. We only give a brief overview here; a detailed explanation can be found in [23].

Let C be a boolean circuit which receives two n-bit inputs $x = x_1 \dots x_n$ and $y = y_1 \dots y_n$, and outputs bit $C(x, y) \in \{0, 1\}$. (If the circuit takes only one input, we denote the other input as $\perp$.) Consider Alice and Bob who wish to securely compute $C(x, y)$, where x is Alice's input, y is Bob's input. Yao's method transforms any C into a secure *garbled* circuit C', which enables computation of $C(x, y)$ without revealing x to Bob or y to Alice.

For each wire i of the circuit, Alice generates two random *wire keys* w_i^0 and w_i^1. These wire keys are used as labels encoding, respectively, 0 and 1 on that wire. Now consider a single gate g in C, described by some boolean function $g : \{0, 1\} \times \{0, 1\} \to \{0, 1\}$. Let the two input wires to g be labeled A and B, and let the output wire be labeled C. The corresponding wire keys are $w_A^0, w_A^1, w_B^0, w_B^1, w_C^0, w_C^1$.

The garbled gate g' of circuit C' is defined by a random permutation of the following four ciphertexts, where $\{x\}_\kappa$ is a symmetric-key encryption of plaintext x under key κ (see [23] for the properties that the encryption scheme must satisfy).

$$c_{00} = \{\{w_C^{g(0,0)}\}_{w_B^0}\}_{w_A^0} \qquad c_{01} = \{\{w_C^{g(0,1)}\}_{w_B^0}\}_{w_A^1}$$
$$c_{10} = \{\{w_C^{g(1,0)}\}_{w_B^1}\}_{w_A^0} \qquad c_{11} = \{\{w_C^{g(1,1)}\}_{w_B^1}\}_{w_A^1}$$

Alice garbles all gates of the circuit in this manner, and sends the entire garbled circuit to Bob.

Garbled circuit evaluation proceeds as follows. For each input wire i associated with Alice, Alice simply sends to Bob the wire key $w_i^{b_A}$ encoding Alice's input bit b_A on that wire. This leaks no information about the value of b_A because the wire keys are random. For each input wire j associated with Bob, Alice and Bob engage in OT_2^1 protocol. Alice's inputs as the sender are the two wire keys w_j^0 and w_j^1, and Bob's input as the chooser is his bit b_B on that wire. As a result of the OT protocol, Bob learns the wire key $w_j^{b_B}$ encoding his input without revealing b_B to Alice.

Bob evaluates the circuit starting from the gates where he has a wire key for each input wire. For each such gate, Bob can decrypt exactly one of the four ciphertexts, and learn the key w_C encoding the value of the gate's output wire. If the output wire is used as an input into another gate, Bob continues the process. This evaluation procedure maintains the invariant that, for each circuit wire i, Bob learns exactly one wire key w_i^b. This wire key is random and thus leaks no information about the bit b it "represents."

In the standard Yao's method, Alice provides a mapping for the wire keys w_{out}^0 and w_{out}^1 of each circuit output wire *out* to 0 and 1, respectively. This allows circuits transformed by Yao's method to be used as "black boxes" which have the same functionality as normal circuits, but hide the parties' respective inputs.

In our constructions, we use Yao's garbled circuits to implement secure integer comparison (see Section 3.4). In contrast to the standard black-box functionality, Alice does not provide a mapping from w_{out}^0 and w_{out}^1 to 0 and 1; instead, we consider w_{out}^0 or w_{out}^1 to be Bob's final output from evaluating the circuit. Furthermore, w_{out}^0 and w_{out}^1 can be arbitrary strings of our choosing rather than random strings. Using Yao's method in this non-standard way allows us to implement a *conditional oblivious transfer*, in which Bob learns exactly one of two values depending on the output of the function encoded by the circuit.

Our prototype implementation uses the Fairplay implementation of Yao's method [24] to construct the integer comparison circuits, which are described in detail in Section 3.4. We refer to it as the YAO subroutine, which, on input of a two-party functionality, produces its garbled-circuit implementation.

To achieve security against malicious (as opposed to semi-honest) participants, it is necessary to use an implementation of Yao's method which is secure in the malicious model and operates on committed inputs [20].

3.4 Privacy-Preserving Offset Integer Comparison

Our protocol for the evaluation of branching programs requires a secure sub-protocol for the comparison of integer values. The Privacy-Preserving Offset Integer Comparison protocol takes place between two parties, Alice and Bob. Bob has an ℓ-bit integer x, while Alice has ℓ-bit integers b and t, and output keys h^0 and h^1. At the end of the protocol execution, Bob learns h^0 if $x - b \mod 2^\ell < t$ and h^1 otherwise, while Alice learns nothing. We denote this functionality as $\text{COMPARE}(x, b, t, h^0, h^1)$. Note that in the special case where $b = 0$, and h^0 and h^1 are the single bits 0 and 1, this is the same as Yao's millionaires' problem.

This problem is also known as *conditional oblivious transfer* with a "greater than" predicate [3]. We assume that participants are computationally bounded, since the encryption scheme used in our protocol is only computationally secure. In this case, Yao's method is the most efficient currently known approach [3].

Yao's garbled circuits provide a relatively efficient protocol for integer comparison because the circuit needed to compare two ℓ-bit integers is relatively small. As described in Section 3.3, in our protocol the circuit creator does not provide the mappings from output-wire keys to actual output-wire bits, so that Bob learns one of the two keys, but not the actual result of the comparison.

4. SECURE EVALUATION OF BRANCHING PROGRAMS

We now describe our protocol for the secure evaluation of binary branching programs. The protocol is executed between a Server, in possession of a branching program (formally defined in Section 4.1), and a User, in possession of an attribute vector. Let k be the number of nodes in the branching program, and n be the number of attributes.

In most practical scenarios, n is significantly larger than k; our protocol is optimized for this case. In particular, the size of the securely transformed branching program is independent of n.

4.1 Branching programs

In this section, we formally define branching programs, which include binary classification or decision trees as a special case. Let

$\mathcal{V} = v_1, \ldots, v_n$ be the vector of User's attributes. Each attribute value is an ℓ-bit integer. (In our experiments, $\ell = 32$, which appears to be sufficient for most practical scenarios.)

A binary branching program T is a triple $\langle \{P_1, \ldots, P_k\}, L, R \rangle$. The first element is a set of nodes. For $i \leq l$, P_i are *decision nodes*. For $i > l$, P_i are *classification nodes*.

Decision nodes are the internal nodes of the program. Each decision node is a pair $\langle t_i, \alpha_i \rangle$, where α_i is the index of an attribute, and t_i is the threshold value with which v_{α_i} is compared in this node. The same value of α may occur in many nodes, *i.e.*, the same attribute may be evaluated more than once. For each decision node i, $L(i)$ is the index of the next node if $v_{\alpha_i} \leq t_i$; $R(i)$ is the index of the next node if $v_{\alpha_i} > t_i$. Functions L and R are such that the resulting directed graph is acyclic.

Classification or diagnosis nodes are the leaf nodes of the program. Each leaf node consists of a single classification label $\langle d_i \rangle$.

To evaluate the branching program on some attribute vector $\mathcal{V}$, start at P_1. If $v_{\alpha_1} \leq t_1$, set $h = L(1)$, else $h = R(1)$. Repeat the process recursively for P_h, and so on, until reaching one of the leaf nodes and obtaining the classification.

4.2 Security requirements

The objective of our protocol is to *securely* evaluate T on $\mathcal{V}$. The protocol should reveal nothing to the Server. The User should learn $T(\mathcal{V})$, which is a classification label contained in one of the leaves of the branching program T. The User is also permitted to learn the total number of nodes of T (see the discussion in Section 4.6) and the length of the path from the root node of T to the leaf containing the result of evaluation, *i.e.*, the label assigned by T to $\mathcal{V}$.

The User should not learn anything else about T. In particular, the User should not learn *which* attributes from $\mathcal{V}$ have been considered by T, with what threshold values they have been compared, the outcome of any comparison, and so on.

The requirement that attribute selection be oblivious precludes a naïve application of secure multi-party computation (SMC) techniques. In standard SMC, each participant knows which of his inputs have been used in the computation. While it is possible to create a circuit that takes all of the User's attributes as inputs and ignores those not used by T, this circuit would be impractically large ($\mathcal{V}$ may contains tens of thousands of attributes). A detailed discussion can be found in Section 4.5.

4.3 Secure branching program protocol

The protocol runs in three phases.

Phase I (offline): Creation of the secure branching program. This is an offline pre-computation executed by the Server. Using Algorithm 1, the Server converts the original branching program T into its secure equivalent T'. Algorithm 1 does not require any interaction with the User or knowledge of the User's identity. For example, the Server may maintain a large store of secure branching programs (all representing differently randomized transformations of the same T), which is replenished during idle periods when the Server's machines have many spare cycles.

Algorithm 1 converts the nodes in the branching program T into secure nodes in the branching program T'. Each classification node is replaced by an encryption of its classification label so that its contents will remain unknown to the User unless the appropriate decryption key is obtained. Each decision node is replaced by a small garbled circuit implementing offset integer comparison (see Section 3.4). This circuit enables the User to learn one of two keys, depending on the comparison between the User's attribute value (offset by a blinding value) and the decision node's threshold value. The revealed key decrypts the next node on the evaluation path.

Input: Branching program $T = \langle \{P_1, \ldots, P_k\}, L, R \rangle$ (see Section 4.1). For $i \leq l$, P_i is a decision node $\langle t_i, \alpha_i \rangle$. For $i > l$, P_i is a classification node containing label $\langle d_i \rangle$.
Outputs:
(i) Secure branching program T'
(ii) k random $\ell + \ell'$-bit blinding values $b_1, \ldots, b_k$
(iii) $2 \cdot k \cdot \ell$ random wire keys w_{ij}^0, w_{ij}^1 for $1 \leq i \leq k, 1 \leq j \leq \ell$

CREATESECUREPROGRAM

1: **let** Q be a random permutation of the set 1, ..., k with $Q[1] = 1$
2: Generate random keys $\kappa_1, ..., \kappa_k$ to be used for encrypting the decision nodes.
3: **for** $i = 1$ to k **do**
4: Generate $2 \cdot \ell$ random wire keys w_{ij}^0, w_{ij}^1 for $1 \leq j \leq \ell$ (to be used for encoding the User's input into the garbled threshold comparison circuit).
5: Generate a random $\ell + \ell'$-bit blinding value b_i; store b_i and $b'_i = b_i \mod 2^\ell$.
6: **let** $\tilde{i} = Q[i]$
7: **if** P_i is a classification node $\langle d_i \rangle$ **then**
8: **let** $S_{\tilde{i}} = \{\text{"label"}, d_i\}_{\kappa_{\tilde{i}}}$, where $\{y\}_\kappa$ is the encryption of y under key κ using a semantically secure symmetric-key encryption scheme. (We assume that all plaintexts are padded so that the ciphertexts of decision nodes and classification nodes have the same size.)
9: **else if** P_i is a decision node $\langle t_i, \alpha_i \rangle$ **then**
10: Use the subroutine YAO for generating garbled circuits (see Section 3.3) to generate a secure circuit C_i for the offset integer comparison functionality (see Section 3.4) COMPARE$(x, b'_{\tilde{i}}, t_i, \mathbf{L}, \mathbf{R}) =$ if $x - b'_{\tilde{i}} \mod 2^\ell < t_i$ then return $\mathbf{L}$ else return $\mathbf{R}$
where $\mathbf{L} = (Q[L(i)], \kappa_{Q[L(i)]})$,
$\mathbf{R} = (Q[R(i)], \kappa_{Q[R(i)]})$
Use $w_{\tilde{i}j}^0, w_{\tilde{i}j}^1$ $(1 \leq j \leq \ell)$ to encode, respectively, 0 and 1 on the ℓ wires corresponding to input x.
11: **let** $S_{\tilde{i}} = \{C_i\}_{\kappa_{\tilde{i}}}$
12: **end if**
13: **end for**
14: **return** $T' = \langle \{S_1, ..., S_k\}, \kappa_1 \rangle$

Algorithm 1: Convert a branching program into a secure branching program

Because the User should not know *which* attribute is being compared to a threshold, the User's input to the garbled circuit is *blinded* by the Server (in phase II, described below) by adding a random $(\ell + \ell')$-bit value that the User does not know. Here ℓ' is the statistical security parameter, set to 80 bits in our implementation. The blinding values $b_1, \ldots, b_k$ are generated randomly by the Server in Phase I. They will be subtracted from the User's input to the circuit before it is compared to the threshold.

Phase II: Oblivious attribute selection. In this phase, the User obtains the blinded attribute values which will be used as inputs to the COMPARE circuits in the secure decision nodes created in Phase I. First, the User creates an instance of the additively homomorphic public-key encryption scheme, and encrypts each attribute in his attribute vector with the public key (this can take place offline). The User sends the entire encrypted attribute vector to the Server along with the public key.

For node i, the blinding value chosen in Phase I is b_i, and the attribute to be compared is α_i. Thus, the User needs to learn $v_{\alpha_i} + b_i$. The Server cannot compute this value directly without learning v_{α_i}

User's input: Attribute vector $v_1, \ldots, v_n$ with ℓ-bit attribute values
Server's input: For each node t_i of T', α_i is the index of the User's attribute which is being compared in this node (if t_i is not a decision node, α_i is chosen randomly); b_i is the random $(\ell + \ell')$-bit value generated as part of CREATESECUREPROGRAM.
Outputs for the User:
(i) $s_1, \ldots, s_k$ where $\forall i\ s_i = v_{\alpha_i} + b_i \mod 2^\ell$
(ii) For each i, wire keys $w_{i1}, \ldots, w_{i\ell}$ encoding $s_i = v_{\alpha_i} + b'_i \mod 2^\ell$ on the input wires of circuit C_i (see Algorithm 1).
Output for the Server: $\perp$

OBLIVIOUSATTRIBUTESELECTION

1: The User generates a public/private key pair of a homomorphic encryption scheme, and sends the public key to the Server.
2: **for** $i = 1$ to n **do**
3: The User sends $E[v_i]$ to the Server.
4: **end for**
5: **for** $i = 1$ to k **do**
6: Server computes $E[v_{\alpha_i} + b_i]$ from $E[v_{\alpha_i}]$ and b_i using the homomorphic property of the encryption scheme, and sends this value to the User.
7: The User decrypts to find $v_{\alpha_i} + b_i$ and then computes $s_i = v_{\alpha_i} + b_i \mod 2^\ell = v_{\alpha_i} + b'_i \mod 2^\ell$
8: **for** $j = 1$ to ℓ **do**
9: The Server and the User execute OT_2^1 oblivious transfer protocol.
The User acts as the chooser; his input is $s_i[j]$, *i.e.*, the jth bit of s_i.
The Server acts as the sender; his inputs are wire keys w_{ij}^0 and w_{ij}^1, encoding, respectively, 0 and 1 on the jth input wire of threshold comparison circuit C_i (see Algorithm 1).
10: **end for**
11: As the result of ℓ oblivious transfers, the User learns wire keys $w_{i1}, \ldots, w_{i\ell}$ encoding his input s_i into the circuit C_i. Note that the User cannot yet evaluate C_i because he does not know the key κ_i under which C_i is encrypted.
12: **end for**

Algorithm 2: Oblivious attribute selection

(which violates the User's privacy), but he can compute $E[v_{\alpha_i} + b_i]$ since he knows $E[v_{\alpha_i}]$ and the encryption is homomorphic. He computes this encrypted value and sends it to the User.

The random blinding value b_i added by the Server to the encrypted ℓ-bit attribute v_{α_i} is ℓ' bits longer than v_{α_i}. Therefore, it statistically hides v_{α_i} (and thus does not reveal which attribute the Server chose) when ℓ' is sufficiently large (80 bits in our implementation). Note that $2^{\ell+\ell'}$ is much smaller than the order of the group in which plaintext addition is done under encryption.

The User uses his private key to decrypt $v_{\alpha_i} + b_i$. By taking $v_{\alpha_i} + b_i \mod 2^\ell$, the User obtains s_i, his ℓ-bit input into the garbled offset integer comparison circuit.

Next, the User acts as the chooser in ℓ instances of 1-out-of-2 oblivious transfer with the Server to learn the garbled wire keys corresponding to his input value s_i. Note that this does not reveal s_i to the Server. Now the User has all the wire key values he needs to evaluate T' in phase III.

Phase III: Evaluation of the secure branching program. In the last phase, the User receives the secure branching program T' from the Server along with κ_1, and evaluates it locally by applying Algorithm 3 on inputs $(T', 1, \kappa_1)$.

Inputs: Secure program T', node index h with corresponding node encryption key κ_h, and, for each i such that $1 \le i \le k$, wire keys $w_{i1}, \ldots, w_{i\ell}$.
Output: Classification label c such that $c = T(\mathcal{V})$

EVALUATESECUREPROGRAM(T', h, κ_h)

1: Use key κ_h to decrypt node S_h of T' and obtain C_h.
2: **if** $C_h = \langle$"label", $d\rangle$ **then**
3: C_h is a classification node.
return label d.
4: **else if** C_h is a garbled circuit **then**
5: Evaluate C_h on inputs $w_{h1}, \ldots, w_{h\ell}$.
6: As the result of evaluation, obtain the pair $(h', \kappa_{h'})$ encoding the output wire value.
7: **return** EVALUATESECUREPROGRAM$(h', \kappa_{h'})$.
8: **else**
9: **error** "Secure program is not properly formed!"
10: **end if**

Algorithm 3: Evaluation of secure branching program

Evaluation does not reveal anything to the User except the label at the end of the evaluation path. At each step, the User applies one of the comparison circuits C_h to the value s_h (encoded as a set of wire keys—see Section 3.3), but he does not know which of his attributes is hidden in s_h. The User thus learns the index of the next node and the decryption key, but not the result of the comparison.

The only information leaked by the evaluation procedure is (i) the total number of nodes in the program T', (ii) the number of nodes that have been evaluated before reaching a classification node (note that in a full decision tree this number does not depend on the path taken), and (iii) the classification label d.

If the usage scenario requires the Server to learn the classification label, too, the User simply sends d to the Server. If the Server should learn the classification label and the User should learn nothing, then the Server can replace the labels with ciphertexts encrypting the labels under the Server's public key; when the User obtains a ciphertext at the end of evaluation, he sends it to the Server.

We emphasize that the User cannot simply re-run the program evaluation algorithm of Phase III on the same secure program T' and a *different* attribute vector, thus learning more about the original branching program. After learning the wire keys corresponding to his (blinded) attributes during Phase II, the User can evaluate only a *single* path in the branching program—that corresponding to the attribute vector he used as his input into the protocol. There is no way for the User to learn the random wire keys encoding other possible inputs to the program.

In order to evaluate T on a different attribute vector, the User must re-run the entire protocol starting from Phase I. He will then obtain a different secure program T'' and a different set of wire keys. Our protocol maintains the invariant that, for every secure branching program, there is only one path that can be evaluated by the User, and this path appears random to the User.

4.4 Security properties

The protocol presented in Section 4.3 is secure in the *semi-honest* model, *i.e.*, under the assumption that participants faithfully follow the protocol, but may attempt to learn extra information from the protocol transcript. (The proof is standard, and omitted to save space.) Even if the User is malicious rather than semi-honest, he cannot learn anything about the diagnostic program except the final diagnostic label, since the underlying oblivious transfer protocol is secure against malicious choosers.

The protocol of Section 4.3 can be transformed, at a constant cost, to achieve security in the malicious model. We only sketch the transformation here due to lack of space. Both parties must commit to their respective protocol inputs, including the Server's branching program and blinding values, and the User's attribute values. Each instance of oblivious transfer (OT) must be replaced with an instance of *committed oblivious transfer*, during which the parties prove in zero-knowledge that their inputs into OT are consistent with their previous commitments. Similarly, each instance of Yao's protocol must be replaced with an instance of *secure two-party computation on committed inputs*, during which the Server proves in zero-knowledge that the offset integer comparison circuits have been formed correctly, and both parties prove that their inputs are consistent with their commitments. The homomorphic encryption scheme must be *verifiable*, *i.e.*, it must enable the encryptor to prove that the plaintext is consistent with a previous commitment. Finally, the commitment scheme must enable efficient proofs of certain relationships between committed values.

The cryptographic tools which satisfy the above requirements, *i.e.*, (1) homomorphic, verifiable encryption scheme, (2) commitment scheme, (3) committed oblivious transfer protocol, (4) secure two-party computation protocol on committed inputs, and (5) efficient zero-knowledge proof systems for the required relationships between protocol components can be found in [20].

Even security in the malicious model does not prevent a malicious User from claiming that his attribute vector has changed and repeatedly re-running the protocol on different committed inputs in an attempt to learn the entire diagnostic program. To prevent this, the Server can rate-limit the number of protocol invocations with each User. This is easy to enforce by refusing to accept new commitments from the User until a specified period expires.

4.5 Efficiency and comparison with generic techniques

In the secure branching program created by Algorithm 1, each of the decision nodes of the original program is replaced by a garbled Yao circuit for comparing two (offset) ℓ-bit integers. Each such circuit requires $\log \ell$ gates, for a total of $k \cdot \log \ell$ gates (this is a conservative estimate, since some of the nodes are classification nodes). Note that the size of the circuit is independent of the number of the User's attributes n. Algorithm 2 requires $k \cdot \ell$ OT_2^1 oblivious transfers to transfer the wire keys corresponding to the User's (blinded) inputs into each of the k nodes.

An alternative to using our protocol from Section 4.3 is to use generic techniques that enable secure computation of any two-party functionality, represented either as a boolean circuit [23, 40] or a binary decision diagram [14] (the latter may be a better choice for branching diagnostic programs).

A naïve way to implement the secure evaluation of binary branching programs using generic techniques would be to have the Server take his specific branching program, transform it into an equivalent secure program using, say, the standard garbled circuit techniques (see Section 3.3), and have the User evaluate the garbled circuit on his attribute vector.

This does *not* satisfy our security requirements. First of all, the topology of the program is revealed to the User. In generic secure multi-party computation (SMC), it is usually assumed that the function to be computed is known to both parties. Yao's garbled circuit technique works even if the circuit evaluator does not know the truth tables associated with the individual gates, but it reveals the topology of the circuit being evaluated. By contrast, our protocol only reveals the length of the evaluation path and the total number of nodes; it leaks no other information about the rest of the branching program. Even worse, with the naïve approach the User learns on *which* of his attributes the program was evaluated, thus violating one of our core security requirements (see Section 4.2).

To ensure obliviousness of the User's input selection, the SMC functionality must be defined so that it takes *any* branching program of a given size (as opposed to the specific Server's program) and securely applies it to *any* attribute vector of a given length.

We have attempted to implement such a functionality using the Fairplay compiler [24], which converts any two-party functionality into an equivalent garbled circuit. Unfortunately, the Fairplay compiler is memory-bound, and in our experiments it was unable to compile functionalities that would allow us to apply branching programs of realistic size to realistic attribute vectors. On a machine with 4 Gigabytes of RAM, the compiler runs out of memory when attempting to compile the functionality that applies a 63-node branching program to a 400-attribute vector.

Table 1 gives comparative measurements of online computation and communication for a few sample configurations. Our experimental setup, along with the detailed performance analysis of our protocol, can be found in Section 6.

Some of the negative aspects of Fairplay, such as running out of memory even on relatively small configurations, may be due to the particular compiler implementation rather than the inherent flaws of the generic approach. Nevertheless, our protocol described in Section 4.3 provides a superior solution for the specific task of secure branching program evaluation.

4.6 Achieving complete privacy

The protocol of Section 4.3 reveals the total number of nodes in the branching program and the length of the evaluation path corresponding to the User's attribute vector. This information appears harmless in practice, but, if necessary, it can be hidden, provided that there exist upper bounds B on the number of nodes and P on the length of the longest evaluation path.

To hide the size of the branching program, the Server can create $B - k$ random ciphertexts (which will never be decrypted by the User), and mix them randomly with the real encrypted nodes of the secure program T'. Semantic security of the encryption scheme used to encrypt individual nodes guarantees that the User cannot tell the difference between an encryption of a real node that he did not reach in his evaluation, and a random ciphertext of the same size. When padded in this way, secure versions of *all* branching programs will contain exactly B ciphertexts.

To hide the length of evaluation paths, first transform the branching program into a decision tree, so that each node has a fixed depth. Then transform it into a full tree of depth P by replacing classification nodes at depth $p < P$ with full trees of depth $(P - p + 1)$, in which every leaf contains the original classification. In the resulting tree, every evaluation path has length P.

5. REMOTE SOFTWARE DIAGNOSTICS

In this section, we give a brief introduction to the problem of remote software fault diagnostics, and then use our protocol of Section 4 to implement a privacy-preserving version of Clarify, a practical system for software fault diagnosis [17].

Microsoft error reporting is an example of a remote software diagnosis tool [28]. A Microsoft error report has two purposes. The first purpose is to gather extensive information about a software failure to enable Microsoft engineers to fix the software problem. We emphasize that we do *not* focus on this problem, since there exist many standard techniques, both privacy-preserving and not, for creating decision trees (*e.g.*, see [17] and Section 2).

The second purpose is to improve the user's experience by pro-

Nodes	Attrib.	Server		User	
		Computation	Communication	Computation	Communication
15	100	*67* vs. **2** sec	*1,292* vs. **263** KB	*76* vs. **3** sec	*528* vs. **98** KB
63	100	*72* vs. **7** sec	*1,799* vs. **1121** KB	*79* vs. **14** sec	*528* vs. **351** KB
15	1000	*605* vs. **2** sec	*11,388* vs. **263** KB	*706* vs. **4** sec	*5,277* vs. **255** KB
63	1000	*X* vs. **7** sec	*X* vs. **1121** KB	*X* vs. **12** sec	*X* vs. **508** KB
Cursive - Fairplay, **Bold** - our protocol, *X* - failed to compile					

Table 1: Comparison of protocols for the evaluation of branching programs

viding a message describing the user's problem and how the user can avoid the problem in the future. Our case study addresses the second purpose, where a server provides feedback to the user about the user's problem. Windows Vista includes a prominent item on the control panel called "Problem reports and solutions" [27] that allows users to get the latest information about a particular software problem from Microsoft's web site. The Ubuntu Linux distribution also contains new features to generate more information about software failures to help users [39].

Microsoft's privacy statement about the information it collects for software problem diagnosis [26] acknowledges that problem reports can compromise users' privacy. Problem reports contain the contents of memory for the program that failed, and this memory "might include your name, part of a document you were working on or data that you recently submitted to a website." The policy says that users concerned about the release of personal or confidential information should *not* send problem reports. Of course, users who do not send problem reports cannot benefit from remote fault diagnostics. Corporate users in particular have expressed concern that remote diagnostics could reveal their intellectual property [32]. Ubuntu's documentation also acknowledges security and privacy risks associated with data-rich fault reports.

The protocol presented in this paper enables the user to obtain a support message in a privacy-preserving fashion. The user does not reveal anything to the software manufacturer about his or her local data, and the software manufacturer does not reveal to the user how the user's local data was mapped to a diagnostic message.

Privacy of diagnostic programs. It may appear that the software manufacturer should simply send the diagnostic program to the user or, better yet, integrate it directly into the supported application. Many software manufacturers, however, view their diagnostic programs as valuable intellectual property. They state this explicitly in the legal documents that accompany the diagnostic software [13] and sue competitors who obtain access to their diagnostic programs [33]. Updating widely deployed software with new support messages and diagnostic tools is not always feasible, either, since many users simply don't install patches.

Moreover, diagnostic programs can reveal vulnerabilities in deployed software. For example, a single message from Microsoft's Dr. Watson diagnostic tool was sufficient to reveal to any user who experienced a particular fault an exploitable buffer overflow [25] (had the entire Dr. Watson diagnostic tree been shipped to every Windows user instead of being evaluated on Microsoft's servers, even users who did not experience the fault could discover the vulnerability by analyzing the diagnostic tree). In the diagnostic trees produced by the Clarify toolkit for `gzprintf`, which is one of our benchmarks, the inner nodes of the diagnostic tree directly point to the function that contains a security vulnerability. With a lively, semi-legal market in information about software vulnerabilities [37], software manufacturers have a strong disincentive to completely reveal all known faults and bugs in their applications.

We emphasize that we do not promote "security by obscurity." Software manufacturers should patch the bugs and vulnerabilities in their programs as soon as practicable. From a purely pragmatic perspective, however, they should not be forced to choose between not providing diagnostic support, or else revealing every internal detail of their applications and diagnostic tools. In reality, when faced with such a stark choice, many will decide to not provide support at all, resulting in a poorer experience for the users who are not willing to disclose their own local data.

Runtime data collection and fault diagnosis. Typically, the runtime environment records some abstraction of the program's behavior. Different abstractions have different cost and accuracy tradeoffs (*e.g.*, see [9, 17]). For instance, one abstraction counts how many times each function in the program was called, another counts function call sites that satisfy a certain predicate (*e.g.*, equal to zero) on the function's return value, and so on.

For the purposes of this paper, we abstract from the details of the program behavior "dumps" generated by the runtime environment, and refer to individual data items simply as *attributes*. We assume that the vector of attributes has a fixed maximum size, which can be quite large—for example, a vector of function callsite counters may include dozens of thousands of attributes. Note that the online computational complexity of the User's algorithm in the protocol of Section 4 does not depend on the number of attributes.

Diagnostic programs evaluate the data dump produced by the runtime environment and diagnose the problem. In this paper, we use diagnostic programs generated by the Clarify system [17]. We emphasize that Clarify is a "black-box" diagnostic system, and thus not simply an alternative to debugging. Commercial applications are often distributed as packed binaries, compiled without symbols and not accompanied by source code. Investigation of a fault in such a binary by manual debugging is a laborious process, whereas even an unsophisticated user can benefit from fast diagnostics provided by systems like Clarify.

In general, the diagnostic program can be manually created by human experts, or constructed automatically by training a machine learning classifier on previously labeled program behaviors (supplied, for example, by beta testers who are not concerned about privacy of their data). Clarify takes the latter approach. If necessary, standard methods for privacy-preserving decision tree construction can be used to protect data suppliers' privacy [1, 22]. The number of users whose data are used in constructing the diagnostic program is typically orders of magnitude lower than the number of users who apply the resulting program to their data. Therefore, we focus on privacy-preserving *evaluation* of diagnostic programs.

The diagnostic program usually has the form of a classification tree or a branching program. In each internal node, one of the attributes is compared with some threshold value. Leaves contain diagnostic labels. For example, Figure 1 shows a diagnostic branching program created by Clarify using function counting for the mp3 player `mpg321`. The application itself does not give any consistent error messages for any of these error cases. The model has four diagnostic labels, including normal execution (no error), file format

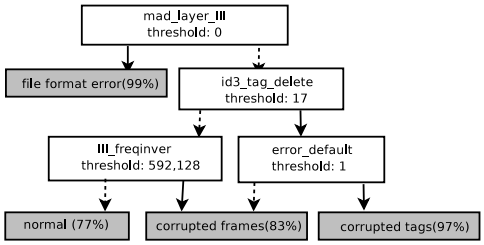

Figure 1: Diagnostic branching program for the `mpg321` benchmark. Dotted lines are taken when the attribute (normalized count of the feature value) is less than or equal to the threshold listed in the box, while the solid line is taken when it is greater than the threshold. The threshold is determined automatically for each benchmark by the decision tree algorithm, and can be different for each node in the tree. Clear boxes are decision nodes. Shaded boxes are classification nodes.

error (trying to play a wav file as if it were an mp3), corrupted tag (mp3 metadata, *e.g.*, artist name is stored in ID3 format tags), and corrupted mp3 frame data. The diagnostic branching program distinguishes between these three failure modes and normal execution.

At the root, the function `mad_layer_III` provides almost perfect discriminative information for the wav error class (trying to play a wav as if it were an mp3): the `mad_layer_III` routine is part of the `libmad` library and is called when the audio frame decoder runs. Since the *wav* format is among the formats not supported by `mpg321`, it will not successfully decode any audio frames, and the `libmad` library will never call `mad_layer_III`.

The `id3_tag_delete` routine differentiates between the corrupted tag and and other classes. The ID3 tag parser in the `libid3tag` library dynamically allocates memory to represent tags and frees them with `id3_tag_delete`. If tag parsing fails, the memory for a tag is not allocated. Since no tag parsing succeeds in the corrupted frames case, `id3_tag_delete` is never called to free the tag memory, making its absence discriminative for that class. The `libmad` audio library's default error handler `error_default` is used if the application does not specify one. `mpg321` does not specify its own error handler, so the presence of the function indicates corrupted audio frames, and its absence indicates corrupted id3 tags. Finally, `III_freqinver`, which performs subband frequency inversion for odd sample lines, is called very frequently as part of normal decoding of audio frame data. When there are corrupted frames, this function is called less frequently, and the decision tree algorithm finds an appropriate threshold value to separate the normal from the corrupted case.

Table 2 shows the parameters of diagnostic programs for several benchmark applications.

6. PERFORMANCE

We evaluated our prototype implementation using PCs with an Intel Pentium D 3 GHz processor, 2 GB of RAM, and 2MB cache. This is a realistic approximation of what the User might use, but we expect that the Server would maintain a more powerful dedicated server to process remote diagnostics requests.

In our analysis of scaling behavior, we created artificial data sets with varying numbers of nodes and attributes, and measured the offline and online time separately. Offline time includes all calculations that can be performed independently of the other party,

App.	Attributes	Nodes	# Errs	Accuracy
gcc	2,920	37	5	89.2%
latex	395	1,107	81	85.1%
mpg321	128	9	4	87.5%
nfs	292	17	5	93.3%
iptables	70	9	5	98.5%

Table 2: For each benchmark, we give the length of the attribute vector, number of nodes in the diagnostic tree, number of errors distinguished by the decision tree, and the tree's classification accuracy.

	Server		User	
Application	Time	Bytes	Time	Bytes
`foxpro` (7 nodes, 224 attrs)	1s	119 KB	2s	78 KB
`iptables` (9 nodes, 70 attrs)	2s	155 KB	2s	61 KB
`mpg321` (9 nodes, 128 attrs)	2s	155 KB	2s	71 KB
`nfs` (17 nodes, 292 attrs)	2s	298 KB	4s	142 KB
`gzprintf` (23 nodes, 60 attrs)	3s	506 KB	5s	133 KB
`gcc` (37 nodes, 2920 attrs)	5s	656 KB	7s	707 KB
`latex` (1107 nodes, 395 attrs)	113s	19,793 KB	189s	5,908 KB

Table 3: Privacy-preserving evaluation of Clarify diagnostic programs: computation and communication costs.

while online time includes the calculations that depend on the information sent by the other party earlier in the protocol. Online time is the more important metric, since it dictates how long the two parties must maintain a connection. Offline calculations can be performed during idle times when the CPU is in low demand.

We first analyze the scaling behavior of the Server algorithm, as presented in Figures 2 and 3. Here we see that the Server's computation and bandwidth requirements are independent of the number of attributes, which is an attractive property in the software diagnostic scenario where the attribute vector can be quite large and contain a lot of information which is not relevant for all diagnostic programs. Furthermore, the Server's algorithm scales linearly with the number of nodes in the branching program, which is as good as one can realistically hope to achieve.

Scaling behavior of the User's algorithm is shown in Figures 4 and 5. As is the case with the Server, the User's online computation time depends linearly on the size of the branching program, but is independent of the number of attributes in the attribute vector. Unlike the Server, the User's offline computation time and bandwidth requirements do depend on the number of attributes. This is because the User must encrypt the entire attribute vector offline, and then transmit it as part of the protocol.

We also evaluate our prototype implementation on several real applications, as shown in Table 3. These benchmarks have been chosen because they are common, heavily-used programs that either contain security vulnerabilities which would be revealed by the diagnostic program (*e.g.*, `gzprintf`), or report misleading error messages (or none at all) for non-exotic error conditions, and therefore would benefit the most from remote diagnosis. As our diagnostic programs, we used classification trees generated by Clar-

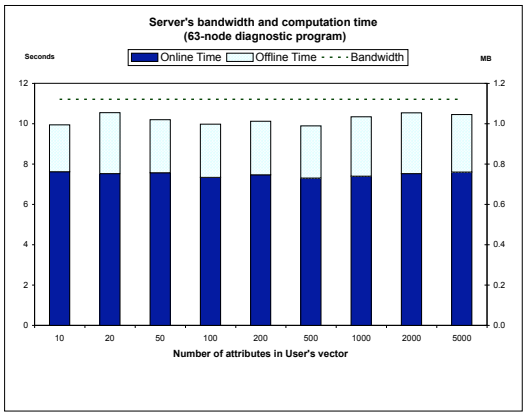

Figure 2: Server algorithm: scaling with the number of attributes.

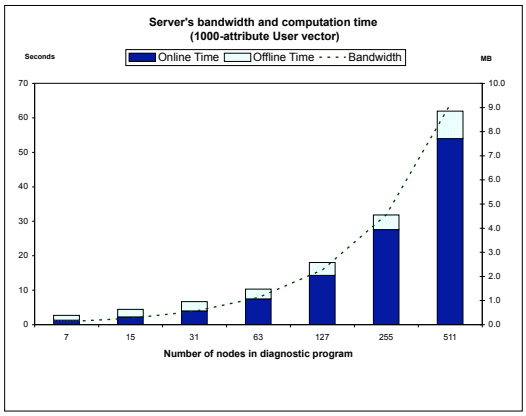

Figure 3: Server algorithm: scaling with the size of the diagnostic program.

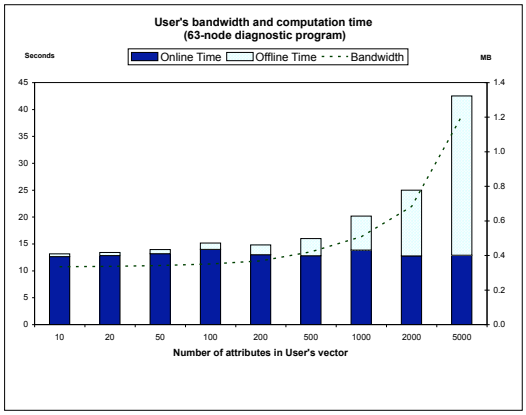

Figure 4: User algorithm: scaling with the number of attributes.

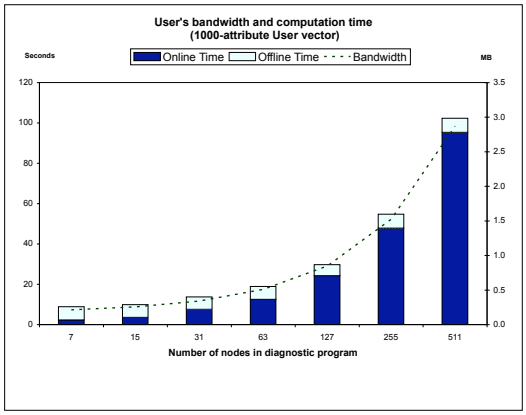

Figure 5: User algorithm: scaling with the size of the diagnostic program.

ify [9, 17], and as attribute vectors 32-bit invocation counters for each function of the application.

For all applications, the computation and communication cost of executing our privacy-preserving protocol is acceptable in many practical scenarios.

7. CONCLUSIONS

We presented a practical, provably secure protocol which enables a User to evaluate the Server's branching program on the User's local data without revealing any information except the diagnostic label. We applied our prototype implementation to several realistic benchmarks, using diagnostic decision trees produced by the Clarify system as our branching programs, and demonstrated that it performs well in many practical scenarios.

An interesting topic of future research is applying the techniques for oblivious evaluation of branching programs developed in this paper to other problems in privacy-preserving computation.

Acknowledgements. This material is based upon work supported by the National Science Foundation under grants CNS-0509033, IIS-0534198 and CNS-0615104, and the ARO grant W911NF-06-1-0316.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 439–450. ACM, 2000.

[2] D. Beaver. Foundations of secure interactive computing. In *Proc. Advances in Cryptology - CRYPTO 1991*, volume 576 of *LNCS*, pages 377–391. Springer, 1992.

[3] I. Blake and V. Kolesnikov. Strong conditional oblivious transfer and computing on intervals. In *Proc. Advances in Cryptology - ASIACRYPT 2004*, volume 3329 of *LNCS*, pages 515–529. Springer, 2004.

[4] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *Proc. 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 128–138. ACM, 2005.

[5] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Proc. 12th USENIX Security Symposium*, pages 273–284. USENIX, 2003.

[6] J. Camenisch and V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *Proc. Advances in Cryptology - CRYPTO 2003*, volume 2729 of *LNCS*, pages 126–144. Springer, 2003.

[7] R. Canetti. Security and composition of multiparty cryptograpic protocols. *J. Cryptology*, 13(1):143–202, 2000.

[8] R. Canetti, Y. Ishai, R. Kumar, M. Reiter, R. Rubinfeld, and R. Wright. Selective private function evaluation with applications to private statistics. In *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 293–304. ACM, 2001.

[9] J. Davis, J. Ha, C. Rossbach, H. Ramadan, and E. Witchel. Cost-sensitive decision tree learning for forensic classification. In *Proc. 17th European Conference on Machine Learning (ECML)*, volume 4212 of *LNCS*, pages 622–629. Springer, 2006.

[10] J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. Strauss, and R. Wright. Secure multiparty computation of approximations. In *Proc. 28th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of *LNCS*, pages 927–938. Springer, 2001.

[11] J. Feigenbaum, B. Pinkas, R. Ryger, and F. Saint-Jean. Secure computation of surveys. In *Proc. EU Workshop on Secure Multiparty Protocols*, 2004.

[12] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Proc. Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 1–19. Springer, 2004.

[13] Gateway. System management services agreement. `http://www.gateway.com/about/legal/warranties/20461r10.pdf`, 1999.

[14] E. Goh, L. Kruger, D. Boneh, and S. Jha. Secure function evaluation with ordered binary decision diagrams. In *Proc. 13th ACM Conference on Computer and Communications Security (CCS)*, pages 410–420. ACM, 2006.

[15] O. Goldreich. *Foundations of Cryptography: Volume II (Basic Applications)*. Cambridge University Press, 2004.

[16] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229. ACM, 1987.

[17] J. Ha, C. Rossbach, J. Davis, I. Roy, H. Ramadan, D. Porter, D. Chen, and E. Witchel. Improved error reporting for software that uses black box components. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 101–111. ACM, 2007.

[18] Q. Huang, D. Jao, and H. Wang. Applications of secure electronic voting to automated privacy-preserving troubleshooting. In *Proc. 12th ACM Conference on Computer and Communications Security (CCS)*, pages 68–80. ACM, 2005.

[19] Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In *Proc. 4th Theory of Cryptography Conference (TCC)*, volume 4392 of *LNCS*, pages 575–594. Springer, 2007.

[20] S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *Proc. Advances in Cryptology - EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 97–114. Springer, 2007.

[21] J. Kilian. Founding cryptography on oblivious transfer. In *Proc. 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 20–31. ACM, 1988.

[22] Y. Lindell and B. Pinkas. Privacy preserving data mining. *J. Cryptology*, 15(3):177–206, 2002.

[23] Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. `http://eprint.iacr.org/2004/175`, 2004.

[24] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – a secure two-party computation system. In *Proc. 13th USENIX Security Symposium*, pages 287–302. USENIX, 2004.

[25] G. McGraw and J. Viega. Making your software behave: Security by obscurity. `http://www.ibm.com/developerworks/java/library/s-obs.html`, 2000.

[26] Microsoft. Privacy statement for the Microsoft error reporting service. `http://oca.microsoft.com/en/dcp20.asp`, 2006.

[27] Microsoft. Reporting and solving computer problems. `http://windowshelp.microsoft.com/Windows/en-US/Help/d97ba15e-9806-4ff3-8ead-71b8d62123fe1033.mspx`, 2006.

[28] Microsoft. How to: Configure microsoft error reporting. `http://msdn2.microsoft.com/en-us/library/bb219076.aspx`, 2007.

[29] M. Naor and K. Nissim. Communication preserving protocols for secure function evaluation. In *Proc. 33rd ACM Symposium on Theory of Computing (STOC)*, pages 590–599. ACM, 2001.

[30] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 448–457. SIAM, 2001.

[31] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proc. 1st ACM Conference on Electronic Commerce*, pages 129–139. ACM, 1999.

[32] R. Naraine. Dr. Watson's Longhorn makeover raises eyebrows. `http://www.eweek.com/article2/0,1759,1822142,00.asp`, 2005.

[33] Oracle. Oracle sues SAP. `http://www.oracle.com/sapsuit/index.html`, 2007.

[34] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. Advances in Cryptology - EUROCRYPT 1999*, volume 1592 of *LNCS*, pages 223–238. Springer, 1999.

[35] M. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Aiken Computation Laboratory, Harvard University, 1981.

[36] T. Sander, A. Young, and M. Yung. Non-interactive CryptoComputing for NC1. In *Proc. 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 554–566. IEEE, 1999.

[37] B. Stone. A lively market, legal and not, for software bugs. New York Times, Jan 30 2007.

[38] H. Wang, Y.-C. Hu, C. Yuan, Z. Zhang, and Y.-M. Wang. Friends troubleshooting network: Towards privacy-preserving, automatic troubleshooting. In *3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 3279 of *LNCS*, pages 184–194. Springer, 2004.

[39] J. Weideman. Automated problem reports. `https://wiki.ubuntu.com/AutomatedProblemReports`, 2006.

[40] A. Yao. How to generate and exchange secrets. In *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.