Establishing Trust for Computation Offloading

Karthik Kumar, Yamini Nimmagadda, and Yung-Hsiang Lu
School of Electrical and Computer Engineering, Purdue University, USA

***Abstract*—Computation offloading can extend the battery lifetime of a portable device by migrating computation to grid-powered servers. The server may charge the portable device for the computation performed, thus providing offloading as a service. The portable device has to trust that the server has indeed performed the computation as claimed. We propose a protocol to establish trust for computation offloading. The protocol uses two keys to establish trust. The first key is part of the input for the computation; executing the program with the first key generates the second key. Without executing the program, it is difficult to generate the correct second key. We implement an algorithm that meets the requirements of the protocol on an HP iPAQ PDA and a server. We demonstrate through experiments how the algorithm establishes trust and quantify the overhead required to establish trust.**

I. Introduction

Battery-powered devices are used in various applications such as mobile phones. Since these systems are battery powered, energy conservation is important. One technique to conserve energy is to offload computation to a server. The server performs the computation and returns the result to the device. Since the server is usually much faster than the device, computation offloading may reduce the execution time of the program. Several research studies discuss the process of discovering servers in an unknown environment [1], [2], [3]. Many researchers [4], [5], [6], [7] develop infrastructures for these servers to support offloading as a service. We use the terms "device" and "client" interchangeably in this work.

We provide an example for offloading as a service for mobile devices. A tourist wishes to identify a monument encountered in a foreign country. The tourist captures an image of the monument on a PDA. In order to identify the monument, it is required to find the best match for the captured image from a collection of regional monuments available on the Internet. The computation is too intensive to be performed on the PDA. The tourist sends the captured image along with the desired image comparison program [8], [9] to a nearby server. The server performs the computation and returns the results to the PDA. The tourist is charged a certain cost based on the resources consumed on the server. Thus the server provides computation offloading as a service.

Offloading as a service opens several research questions. An important question is the possibility of cheating by the server. The user may be charged by the server; however it might return random images from the image collection and not execute the offloaded program. It could execute the program on just 100 images and charge the user for searching 1000 images. Since the user is being charged for offloading the computation, it is important to trust that the expected computation has been performed. This paper answers the question: *how can the device trust that the server actually performs the expected computation and does not cheat.* We believe this is the first work on establishing trust for computation offloading.

In this paper, we present a key-based protocol for establishing trust for computation offloading. The protocol specifies certain requirements for establishing trust. The device makes an estimate of the amount of computation to be performed on the server before offloading the program. The device then obtains information about the execution by using two keys to establish trust; executing the program with the first key gives the correct second key. The device compares its estimate with the information obtained from the second key returned by the server. Based on the comparison, the device decides whether to trust the output of the server.

We implement an algorithm that meets the requirements of the proposed protocol using C# on an HP iPAQ hw6945 PDA and a server. We simulate different types of servers which try to cheat by claiming more computation than what they actually perform. We show how our algorithm detects these different types of cheaters. We measure the execution time of the program on the iPAQ and on the servers, with and without our algorithm. Based on these measurements, we quantify the overhead required to establish trust.

II. Related Work

The related work can be classified into two categories (1) existing frameworks for computation offloading (2) mechanisms for secure execution. The former do not consider the problem of trusting that an offloaded program executed as expected. The latter are used to provide guarantees that untampered programs are executed. They are used to detect tampering programs. However, these are not specific to offloading and they cannot be applied directly.

A. Computation Offloading

Several works exist in the literature for offloading as a service. These can be classified into two categories (1) mechanisms to discover and select new servers for offloading (2) infrastructures for offloading as a service.

The mobility offered by portable devices has resulted in several studies on discovering servers for computation offloading in unknown environments. Anagnoustou et al. [1] present a context-aware service matching algorithm and give an architecture for service-discovery. Sivavakeesar et al. [2] propose a distributed middleware-based mechanism for discovering suitable service elements. Ou et al in [3] present a

978-1-4244-4581-3/09/$25.00 ©2009 IEEE

combined routing and surrogate selection algorithm with mobile handoffs. Guan et al. [4] present device-proxy-Grid system infrastructure for offloading as a service. They propose that pervasive devices be allowed to make use of Grid services to enable users to access computational resources automatically on demand. Goyal et al. [5] propose an offloading infrastructure with virtual machines that consider multiple clients and multiple servers. Wolski et al. [7] present a framework for offloading with Bayesian bandwidth prediction. Yang et al. [6] describe the requirements for using offloading as a service. Studies have been conducted for making offloading decisions [7], [10], [11]. This paper assumes that the client has decided to offload. Mechanisms exist for computation offloading, such as Java RMI (remote method invocation). However, none of the existing studies considers the problem of trusting the computation performed by a server.

B. Secure Execution

Secure execution frameworks propose mechanisms to ensure that untampered code is executed. Existing frameworks for secure execution can be classified into (1) hardware based (2) software based. The former such as TPM [12] requires additional hardware for security, and cannot be expected for a newly discovered server. Software-based mechanisms are proposed in [13] and [14]. The approach used by [13] is for microprocessors with no virtual memory. Seshadri et al. [14] propose Pioneer, a system for verifying code integrity and enforcing untampered code execution. However, their work is applicable only when the exact architectural details of the server are known. They ensure that the untampered executable is invoked by verifying the checksum over the executable. However, for applications like image, video or text retrieval [5], the search space is a variable as an input to the program. Hence the server could cheat by executing the unmodified program over a smaller search space and this would not be detected by their approach.

This paper has the following contributions (1) This is the first paper to study trust for computation offloading. (2) We propose a protocol for establishing trust and implement an algorithm that meet the requirements. We show how our method can detect cheating servers. (3) We analyze the performance of the offloaded program and quantify the overhead required to establish trust.

III. Establishing Trust

The first step towards establishing a trust model is examining how a server may cheat. In the following subsections, we build an attack model, a trust protocol, and describe our algorithm to establish trust.

A. Attack Model

In this paper, an "attack" means that a server charges a client for computation that is not performed. The possibilities include (a) The server may not execute the program and return random results for the offloaded computation. (b) The server may not execute the program and return the results of the previous execution. (c) The server may execute the program over a smaller subset of the input. (d) The server may execute only a part of the program. We make the following assumptions in our work: there is *insufficient time* for the server to analyze the program as this type of attack can be accomplished given enough time [15].

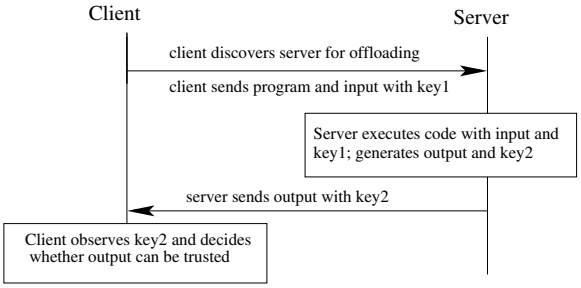

Fig. 1. Trust Protocol

B. Trust Protocol

We propose a trust protocol for computation offloading. The protocol uses two keys $key1$ and $key2$ to establish trust. The protocol has the following steps as given in Figure 1:

1) The client wants to offload a program and discovers a server that supports offloading.
2) The client inserts $key1$ as part of the input data; the client sends the program and the data to the server.
3) The server executes the program with the input, and generates the output and $key2$; $key2$ is considered as part of the output by the server.
4) The server returns the output including $key2$ to the client.
5) The client examines $key2$ and decides whether to trust the server.

A desired trust protocol needs to ensure that (1) the program is executed, (2) the execution is over the complete range of the input, and (3) the results returned for the program are for the current execution and not from the previous execution.

To provide these guarantees, $key2$ must have the following properties: (1) It is generated only if the program is executed. (2) It contains information about the extent of execution of the program. (3) A unique $key2$ is generated in response to a unique $key1$. This ensures that the results of the previous execution cannot generate an acceptable $key2$ if a different $key1$ is used.

In other words, if the program is executed, calculating $key2$ from $key1$ is easy. If the program is not executed, it is difficult to obtain $key2$. Since $key1$ is part of the input, the server may not be able to detect $key1$ easily. Several steganographic techniques for data [16] and images [17] have been proposed to hide $key1$ in the input.

Our trust protocol is similar to public key cryptography, where the user has two keys: a public key and a private key. A message encrypted with the public key can be decrypted with the corresponding private key. It is computationally difficult to deduce the private key from the public key. In our work,

"program execution" is analogous to the private key, and is required for "decryption" (generation of $key2$ from $key1$).

C. Trust Algorithm

1) Motivating Example: We show a simple example to motivate our algorithm. We consider a situation where a client offloads a program to add two arrays of numbers, using the following program segment in C#:

```
int[] add(int[] a, int[] b, int n)
{
  int[] c = new int[n+1];
  for(int i=1;i<n+1;i++)
    {
     c[i]=a[i]+b[i];
     c[0]+=a[0]+b[0];
    }
  return c;
}
```

In the above code segment, `a` and `b` are arrays passed to the server for addition; the results are returned in array `c`. However, the server may cheat by any of the mechanisms mentioned in the attack model; for example, it may add fewer than `n` elements. Our algorithm helps the client detect whether the offloaded program is executed. In this case, $key1$ is stored in `a[0]` and `b[0]` of the input arrays, and $key2$ is stored in `c[0]` of the output array. The client may observe $key2$ and decide whether to trust the output of the server. The value of $key1$, contained in `a[0]` and `b[0]`, is known to the client. Thus $key2$ should be `n` $\times$ (`a[0] + b[0]`), if the program is executed for `n` additions.

This is a simple example and the overhead imposed by the trust protocol is large. For most practical applications that require offloading, more complex calculation is performed and this makes the relative overhead acceptable.

2) Trust algorithm: Our trust algorithm dynamically generates $key2$ by executing the program using $key1$. We insert checkpoints at various locations in the program. The program uses $key1$ as part of its input. A unique $key1$ is generated by the client for each offloaded computation. The value of $key1$ is split into several subkeys, a set $K = \{k_i | 1 \leq i \leq m\}$, where m is the number of checkpoints. Another set $C = \{c_i | 1 \leq i \leq m\}$ stores $key2$ and all values are initialized to zero. When the program is executed at each checkpoint, unique computation is performed on the elements from the set K to generate elements of the set C. The set C is combined to $key2$ and sent to the client as part of the output.

We first consider a simple operation $c_i = k_i$ at each checkpoint. This gives information that the checkpoint is executed. However, it does not give information about how many times the checkpoint is executed. Thus we modify the operation to be $c_i += k_i$. This gives information about how many times each checkpoint is executed; however, it is a direct mapping from K to C. Each c_i is an integer multiple of k_i, and this may be easy to identify by a cheating server without analyzing the program. Hence, we make the operation a many-to-one mapping, such as $c_i += k_{i+1} + k_i$. In order to make the operation even harder to detect by the server, we multiply k_i by $(-1)^{k_i}$, thus making the operation depend on k_i and use the following operation to compute C.

$$c_i += k_{i+1} + (-1)^{k_i} \times k_i \tag{1}$$

It is possible to use even more complex operations. However, for the purpose of this paper we use equation (1) because of its low overhead. This operation meets the requirements of the protocol proposed in Section III-B. The protocol requires that $key2$ contain information about the execution of the program. In our algorithm, if a checkpoint is executed multiple times, k_i will either be added or subtracted from k_{i+1}. Thus by knowing k_i and k_{i+1}, and observing the value of c_i, it is possible to identify how many times the checkpoint is executed. Since k_i is determined by the client to be different for each execution, c_i is known to the client but unknown to the server without executing the program.

We choose addition as the repeated operation because the growth of its output value is linear; hence for a given number of digits for each c_i, we can detect more executions of the checkpoints. We want to limit the number of digits because $key2$ needs to be hidden in the output of the program.

We show how to establish trust using equation (1) for a program with two functions and two inputs: a variable a and a set of variables S. The program computes the size of S, given by the variable n. The program then executes a function, FUNCTION-A for n iterations, with each iteration performing some operations on a with an element of S. The output of FUNCTION-A, called $m1$, is passed FUNCTION-B. It performs some operations on $m1$ and returns r as the output.

The program is offloaded to the server, along with a and S. The result r is returned to the client. The client needs to detect whether the server cheats; for example, the server may remove some elements from S and then execute the program. In order to establish trust, a checkpoint is added inside each function. At each checkpoint, the operation described in equation (1) is performed; this records the information about the program's execution.

The client hides $key1$ with the original input a in a new variable in. When the program is executed with in, $key1$ is separated from a. The value of $key1$ gives three subkeys $k1$, $k2$, and $k3$; $k1$ and $k2$ are passed to FUNCTION-A, and store the information about its execution in $c1$. The subkeys $k2$ and $k3$ store the information about execution of FUNCTION-B in $c2$. The values of $c1$ and $c2$ give $key2$; $key2$ is hidden with the original output r in a new variable out. The value of out is returned to the client.

The client extracts r and $key2$ from out. The client expects r to be the output for the execution of the program for a and S of size s. The client can use $key2$ to decide whether to trust r. The client knows the values of $\{k1, k2, k3\}$; thus if the observed value of $key2$ is $\{s \times a \times (k2 + (-1)^{k1} \times k1),\ s/2 \times (k3 + (-1)^{k2} \times k2)\}$ the client can trust that the server has executed the program.

GENERIC-PROGRAM(in, S)

```
1  a, key1 ← in
   ▷ key1 is obtained from the input
2  k1,k2,k3 ← key1
3  ctr ← 0
4  c1, c2 ← 0
5  n ← |S|
6  if ctr < n
7      then ctr ← ctr + 1
8           [m1,c1] ← FUNCTION-A(a, S(ctr), k1, k2, c1)
            ▷ c1 represents execution of FUNCTION-A
9  [r,c2] ← FUNCTION-B(m1, n, k2, k3, c2)
   ▷ c2 represents execution of FUNCTION-B
10 key2 ← c1, c2
11 out ← r, key2
12 return out
```

FUNCTION-A($p, q, k1, k2, c1$)

```
1  for i ← 1 to p
2      do ▷ m1 ← (code of FUNCTION-A)(p,q)
3         c1 ← c1 + k2 +(−1)^k1 × k1 ▷ checkpoint
4  return [m1,c1]
```

FUNCTION-B($m1, n, k2, k3, c2$)

```
1  for i ← 1 to n/2
2      do ▷ r ← (code of FUNCTION-B)(m1)
3         c2 ← c1 + k3 +(−1)^k2 × k2 ▷ checkpoint
4  return [r,c2]
```

D. Programs with Unknown Number of Iterations

In the previous case, the client observes $key2$ and decides whether the server has executed the task. The number of iterations is known by the client in advance. However, sometimes the client has to estimate the amount of computation. An example is playing a chess game, where the program may search a number of solutions to decide the next move. The exact number of solutions searched may depend on the skill of the opponent. However, the client needs to know that at least n different moves have been considered, where n is the minimum number of searches required to trust that the program is genuinely playing the game and not returning random moves.

In such a scenario, the client estimates the number of iterations i_e required by the server, and compares the estimate with the observed number i_o returned by $key2$. The estimate may be obtained in several ways depending on the application. In some cases, the estimate may be given by some specific property of the application. Applications like chess may require a lower bound for trust. Sometimes, i_e may be a range of values; beyond this range, trust declines. This is true for applications where more iterations do not improve the result. When the number of iterations claimed by the server exceeds a certain value, trust declines. In other cases, the client may execute the program for some small values of inputs. The estimate i_e for larger inputs may be obtained by extrapolating the smaller observed values. The approximation error e_a is given as the average mean square distance of the data points from the approximating polynomial. The approximation with the least e_a may be selected. The expected value i_e for a given input is then obtained from the approximating function. We propose to calculate trust as a distance measure between the observed and the expected values of $key2$. The overall trust is given by considering the distance at all the checkpoints.

$$trust = \frac{1}{m} \times \sum_{i=0}^{m} D(i_e, i_o) \tag{2}$$

where m is the number of checkpoints, i_e is the expected number of executions and i_o is the observed number of executions. The function D returns a percentage for trust at each checkpoint. We allow a margin of error of e_a between the expected value and the actual value returned by the server. This is based on the error obtained during the approximation. Beyond the allowed margin, the distance is used to compute the percentage of trust. This may be formulated as

$$D(i_e, i_o) = \begin{cases} 1 & \text{if } i_e - e_a \le i_o \le i_e + e_a \\ 1 - (\frac{i_o - i_e - e_a}{i_e - e_a}) & \text{if } i_e + e_a < i_o \le 2i_e \\ \frac{i_o}{i_e - i_a} & \text{if } 0 \le i_o < i_e + e_a \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

In equation (3), we have three cases: First, when the distance of the observed value from the expected value is within the margin of error, the trust is 100%. Second, when the observed value is within a range from the expected value (on either side), the trust returned is a linear percentage based on the distance. Third, when the observed value is far from the expected value, the trust returned is 0%. If i_o is not an integer when computed from $key2$, the trust for that checkpoint is zero.

IV. EXPERIMENTS

A. Experimental Setup

We use an HP iPAQ hw6945 PDA as the portable device for our experiments. The PDA has a 480 MHz processor and 64 MB of RAM. We use a server with a 3 GHz processor and with 3 GB of RAM. The PDA communicates with the server using TCP sockets. We implement our trust algorithm in C# using the .NET framework, and we deploy the executable on the PDA, and the server.

B. Workload

The workload used to demonstrate our algorithm is the calculation of π for a precision of n digits after the decimal point. We choose π calculation because (i) The value π is a constant, and we know if the server returns the correct answer. (ii) Calculating π takes $O(n^2)$ time for n digits, making it a good candidate for computation offloading for large values of n. (iii) The value of π does not change; if the server returns the result of the previous execution the answer is correct. However, our method can detect whether the server performs the calculation to obtain the answer.

Several methods have been proposed for π calculation, we choose Machin's formula for the approximation. The formula is given below:

$$\pi = 16\arctan(1/5) - 4\arctan(1/239) \quad (4)$$

$$\arctan(x) = \sum_{i=0}^{\infty} \frac{(-1^i)x^i}{2i+1}. \quad (5)$$

Equation (4) shows π to be calculated as the difference of two terms: each is the sum of an infinite series. The stopping criterion for i to get n digits is when the variable computing the i^{th} term does not change the first n digits. This is a runtime stopping criterion; thus the relationship between n and i is not known beforehand. This makes π calculation fall under the category of applications described in Section III-D.

C. Evaluation Criterion

We calculate the trust for each case based on equations (2) and (3). We evaluate our algorithm against the attacks described in Section III-A. (a) The server does not execute the program and returns random numbers. (b) The client offloads the task for an input of size n, and obtains the result from the server. The next task is for an input of size $n+1$. The server does not execute this second task and returns the result of the first task (input n). In the reverse scenario, where the second task is for $n-1$ digits, the value of π for the previous execution is accurate. However, our method still detects whether the server performs the calculation. (c) The server is given a task of size n, and returns the results for m digits, $m < n$. (d) The server executes only a part of the program. This is accomplished by replacing a function with null operations.

We compute the time overhead for trust as the sum of four terms: (a) for running the program on the client over smaller values of input, (b) for calculating the linear fit for estimation, (c) for executing the program on the server with the checkpoints, and (d) for making a trust decision.

D. Implementation

We consider the situation when the client offloads the computation of π for a precision of n digits to a server. The server does not know the computation is to obtain the value of π. Nor does the server know that the client expects n digits of precision. We implement a π calculator based on equations (4) and (5). The execution times for the PDA and the server are shown in figure 2. Our program handles large numbers as arrays of unsigned integers. The calculation has 3 main steps (i) calculation of $16\arctan(1/5)$ (ii) calculation of $4\arctan(1/239)$ (iii) subtraction of (ii) from (i). Steps (i) and (ii) have an unknown number of iterations for calculating π upto n digits. Step (iii) is the subtraction of two arrays of n elements. The number of digits is appended with $key1$, and the two are indistinguishable to the server.

Each k_i in $key1$ takes a value from 0 to 9. The server executes the program and sends back the result to the PDA. The result has $key2 = C$ appended at a known location in the value of π. In our experiments, the maximum value of c_i can fit in 8 digits. We choose the 10^{th} to 33^{rd} digits to store $key2$, which is the set $C = \{c_i | i = 1, 2, 3\}$. (Alternately, the digit locations could also be given by $key1$). Since this arrangement is known to the client, $key2$ may be easily extracted.

E. Estimation of Key2

The PDA estimates the expected value of $key2$ by linear extrapolation of smaller values of the input and observing the number of times each checkpoint is accessed. For example, the number of iterations of i observed on the client in the calculation of $16\arctan(1/5)$ for 10 digits is 20, for 50 digits is 48 and for 500 digits is 364. By linearly extrapolating these values, we obtain the relationship $i = 0.71362 \times n + 10.85$. This gives the expected value of i for $n = 10,000$ to be 7146. The actual value is 7171. The estimates obtained by extrapolation to calculate π for n digits at the three checkpoints are given by $0.71362 \times n + 10.85$, $0.2098 \times n + 3.326$ and n. The margins of error we obtain based on extrapolation are 6%, 5.5%, and 0% respectively for the three checkpoints.

F. Detect Cheating Servers

The following are our observations for the evaluation:

(a) The server returns random results: We show an execution instance of this case from our experiments: input=100008527, corresponding to $n = 10000$, $k1 = 8$, $k2 = 5$, $k3 = 2$, and $k4 = 7$. The observed values: $c1 = +9672567$, $c2 = +7189036$, and $c3 = -3712389$ (random results from the server). The correct value (i_e) are 7146, 2101 and $10,000$. The observed $c1$ for the first checkpoint is 9672567, hence $i_o \times (8 + (-1)^5 \times 5) = 9672567$, which gives $i_o = 3224189$. The observed number of iterations for the other 2 checkpoints are 1027005.14 and 742477.8; since these two are fractions, and since for the first checkpoint $i_o \geq 2 \times i_e$, all the three return a trust value of 0. The client does not trust the server at all in this instance. We perform 50 such random attacks on various inputs n ranging from 5000 to $100,000$, and we observe that in 48 cases, the trust obtained is 0%. In the other two cases, the trust obtained are 0.66% and 3% respectively.

(b) The server returns the results of the previous execution: We offload two successive tasks for $n = 10,000$ and $n = 10,001$ digits respectively; the difference between the expected number of iterations for the two cases falls within the margin of error. We show an execution instance from our experiments. The expected number of iterations i_e for $10,000$ digits based on the model is 7146, 2101 and $10,000$. The first execution is performed by the server and has $k1 = 6$, $k2 = 7$, $k3 = 5$, $k4 = 1$. The observed values i_o at the checkpoints are 7171, 2107 and $10,000$ iterations. This corresponds to a trust of $(1/3) \times (1 + 1 + 1) = 100\%$, since the differences fall within the e_a calculated earlier. The server does not perform the second execution. For the second execution, the values of $k1, k2$, $k3$, and $k4$ are $9, 2, 8,$ and 7 respectively. Since this attack returns the previous observed values, i_o are recalculated to be $+651.9$, -421.4 and -40000. These result in a trust of 0%. We perform 50 such attacks in our experiments for $5000 \leq n \leq 100,000$ and we observe that in 45 cases, the trust obtained is 0% and the in the others, the observed trust $\leq 2.8\%$.

(c) The server executes the program over a smaller subset of input, for m digits, where m is a fraction of the received input. However, the received input contains $key1$ appended to n. Hence reducing the value of the input by a percentage will

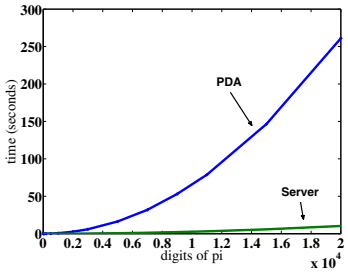

Fig. 2. Execution time for π calculation on the PDA and on the server

change the values of n and $key1$. We consider the case when the input is 100002368, and make the server randomly compute 50 different values from 0 to 100002368. We observe that in 42 cases, the trust is 0%, and in the others the observed trust $\leq 3\%$. The trust is low because the work done is not on the input given by the client because the server cannot distinguish $key1$ from the input.

(d) The server executes only a part of the program: The calculation of $4\arctan(1/239)$ is removed, and replaced by 0. In this case, we observe that the second checkpoint always returns a null value, thus giving the trust to be 66.67%, based on equation (2). In this case, the trust is high though the answer is wrong. This is because we do not compare the answer; we detect that the server has done two third of the work offloaded by the client. The client does not know the correct answer; otherwise, offloading would be unnecessary.

G. Overhead

To quantify the overhead, we measure the execution time for calculating π to $100,000$ digits. This computation takes more than 6 hours on the PDA; it takes 670 seconds on the server. We measure the overhead as described in Section IV-C: (1) For estimation, we use the values of π obtained for 10, 50, 100, 500 and 800 digits respectively. These are run on the PDA and take 0.88 seconds. (2) The estimation takes 2.02 seconds on the PDA. This is because we search a series of approximating polynomials for the estimation. We measure the approximation error for different fits such as linear, quadratic, etc upto the tenth degree and find linear estimation to have the least approximation error. (3) Calculating the correct $key2$ takes 0.8 seconds on the PDA. (4) The trust decision takes 0.2 seconds on the PDA. Thus the total overhead is 4 seconds. Offloading with trust has an overhead of $\frac{4}{670} = 0.6\%$ for $n = 100,000$.

H. Discussion

If checkpoints are placed inside conditional branches, conditional flow analysis may be required to determine the locations of the checkpoints. For example, a program can have a branch with two checkpoints. By checking the values of the subkeys at the checkpoints, we can know which branch is executed. For such applications, it may be required to formulate different expectations for the subkeys based on the branching conditions. This would be an extension of our work. If the stenographic technique is identified by the attacker, a plain text attack could attempt to identify the key computation function. We will analyze how more complex key computation functions perform in our future work. Our future work will also evaluate our algorithm using different mobile devices, with different memory and processing constraints.

V. Conclusion

We present a protocol to establish trust for computation offloading and implement an algorithm in C# to meet the requirements. We show that our algorithm detects cheating servers, and the overhead is negligible.

Acknowledgments

This work is supported in part by NSF CNS-0347466 and CCF-0541267. Any opinions, findings, and conclusions or recommendations in the project are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] ME Anagnostou, A. Juhola, and ED Sykas. Context Aware services as a step to pervasive computing. In *Lobster Workshop on Location based Services, Greece*, pages 4–5, 2002.

[2] S. Sivavakeesar, O.F. Gonzalez, and G. Pavlou. Service Discovery Strategies in Ubiquitous Communication Environments. *IEEE Communications Magazine*, 44(9):106–113, 2006.

[3] Shumao Ou, Kun Yang, and Liang Hu. Cross: A combined routing and surrogate selection algorithm for pervasive service offloading in mobile ad hoc environments. In *IEEE GLOBECOM*, pages 720–725, 2007.

[4] T. Guan, E. Zaluska, and DD Roure. Extending Pervasive Devices with the Semantic Grid: A Service Infrastructure Approach. In *Sixth IEEE Conference on Computer and Information Technology*, 2006.

[5] S. Goyal and J. Carter. A lightweight secure cyber foraging infrastructure for resource-constrained devices. In *Mobile Computing Systems and Applications*, pages 184–195, 2004.

[6] Kun Yang, Shumao Ou, and Hsiao-Hwa Chen. On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. In *Communications Magazine, IEEE*, pages 56–63, January 2008.

[7] R. Wolski, S. Gurun, C. Krintz, and D. Nurmi. Using bandwidth data to make computation offloading decisions. In *International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.

[8] H. Sonobe, S. Takagi, and F. Yoshimoto. Mobile computing system for fish image retrieval. In *International Workshop on Advanced Image Technology*, pages 33–37, 2004.

[9] M. Noda, H. Sonobe, S. Takagi, and F. Yoshimoto. Cosmos: convenient image retrieval system of flowers for mobile computing situations. In *IASTED*, pages 25–30, 2002.

[10] Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan Li. Adaptive computation offloading for energy conservation on battery-powered systems. In *ICPADS*, pages 1–8, December 2007.

[11] P. Rong and M. Pedram. Extending the lifetime of a network of battery-powered mobile devices by remote processing: a markovian decision-based approach. In *DAC*, pages 906–911, 2003.

[12] S.W. Smith. *Trusted Computing Platforms: Design and Applications*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2004.

[13] V. Gratzer and D. Naccache. Alien vs. Quine, the vanishing circuit and other tales from the industrys crypt. In *EUROCRYPT*, volume 4004, pages 48–58. Springer, 2006.

[14] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM symposium on Operating systems principles*, pages 1–16, 2005.

[15] Christian S. Collberg, Clark Thomborson, and Gregg M. Townsend. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, 29(6):35, 2007.

[16] FAP Petitcolas, RJ Anderson, and MG Kuhn. Information hiding-a survey. *Proceedings of the IEEE*, 87(7):1062–1078, 1999.

[17] R. Chandramouli and N. Memon. Analysis of LSB based image steganography techniques. In *ICIP*, volume 3, 2001.