Mobile Architecture for Distributed Brute-Force Attacks

Alexandru MARCULESCU
IT&C Security Master Program
Department of Economic Informatics and Cybernetics
The Bucharest University of Economic Studies
ROMANIA
alexandru.marculescu@gmail.com

Abstract: The focus of this paper centers on demonstrating that a distributed brute-force search of even a medium key-sized cryptosystem's keyspace is possible even using the increasingly present mobile devices, although not quite practical yet, except for extremely well-funded groups. It is also an operational model that also aims to be an explicit documentation source for building a distributed architecture involving a server and any number of mobile agents, as well as highlighting both the advantages and the drawbacks of executing such an algorithm on smartphones.

Key-Words: brute-force attack, distributed computing, smartphones

1. Introduction

The relatively new mobile computing paradigm that provides continuous network connectivity to devices regardless of their location, combined with the worldwide growth shown by smartphones over the past few years brings to light undeniable concerns and insecurities regarding the privacy and security of the information. With the increasing amount of mobile computing power available at lower and lower costs, today's cryptosystems must be able to withstand brute-force attacks that would have been unthinkable in the relatively recent past. To realize such a malicious aim, a distributed computing attempt would be implied. Regarding the feasibility of such a system, the main concern lies with the flexibility of the architecture, which should be able to support several different mobile platforms, as well as its scalability, while other objectives also arise, like reliability, security and, perhaps most important, overall speed and optimized performance. Encrypting private or sensitive data as it transfers over the public Internet is very important nowadays, whether we're talking about symmetric, asymmetric or hash algorithms. By publicly breaking them and raising awareness on the ones found out as vulnerable, people will know not to use them. Understanding more about this domain may help secure cryptography implementations in existing environments.

Besides the actual implementation, designing a clear web service and several smartphone clients (ideally an app for each of the most important platforms: Google's Android, Apple's iOS and Microsoft's Windows Phone) and assuring a communication between the components, the most challenging aspect of such a project is represented by optimizing client performance, given that mobile devices have tight constraints on power consumption and limited bandwidth.

2. Cryptographic aspects

Encrypting private or sensitive data as it transfers over the public Internet is very important nowadays, whether we're talking about symmetric, asymmetric or hash algorithms. By publicly breaking them and raising awareness on the ones found out as vulnerable, people will know not to use them. Understanding more about this domain may help secure cryptography implementations in existing environments.

The strength of an encryption algorithm is measured by two factors: the effective length of the key and its ability to withstand attack. It doesn't “cost” much more, in terms of computer cycles to encrypt something with 128 bits, instead
of 40 or 56. Yet, the level of security that is obtained as a result of that extra step is amazing, from being able to trivially decrypt a message in seconds to requiring more time than the age of the universe many times over.

To be “safe” means that the data that’s been encrypted should be worthless by the time someone is able to read it by using brute force. As an example, census data should be encrypted using a cryptosystem with a larger key than one required for a credit card that will expire in a year or two from now.

In computer science, brute-force (or exhaustive) search (or “generate and test”) is a trivial yet generic problem-solving technique that consists of systematically enumerating all possible candidates for the solution, until finding the first one that satisfies the problem’s statement. It’s simple to implement, suitable in most cases, will always find a solution if it exists, but its cost is proportional to the number of candidate solutions, so it’s typically used when the problem size is limited (or reduced through problem-specific heuristic methods), or when the simplicity of implementation is more important than speed.

Brute force is often used in known-plaintext or ciphertext-only attacks. It presumes checking all possible keys until the correct one is found is still an applicable method against any encrypted data (passwords and hashes are the most targeted), if the attacker is unable to target a weakness in the system that would make his task easier. In the worst case scenario, the algorithm has to check each available combination in order to find the correct one. Key length and data obfuscation can be used to render these attacks less effective (more time and/or resource consuming).

One challenge associated with a ciphertext-only brute force attack is determining when it is successful. If a 15-byte plaintext of “This is a secure message.” is encrypted with a one-time pad, a brute force attack reveals the plaintext, but it also unearths many additional possible plaintexts, such as “This is a yellow picture.”

3. Distributed computing

Speeding up a computation can be achieved by "parallelizing" it - dividing it into pieces that can be worked on by separate processors at the same time. Most modern supercomputers work this way, using many processors in one box. Another way of reaching the same result is through distributed computing, a method of computer processing in which different parts of a program, or different portions of data, are processing simultaneously on two or more computers.

The majority of the world’s computing power is no longer in institutional machine rooms and supercomputer centers. Instead, it is now distributed in the hundreds of millions of personal computers and mobile devices all over the world. In a few more years, other consumer devices like game consoles and television set-top boxes may comprise a large fraction of total computing power, so distributed computing can definitely be considered the most powerful emerging information technology. The sheer power, agility to avoid denial-of-service attacks, and overall ability to automate across heterogenous and asymmetrical information systems cannot be ignored.

At the same time the cost of computing is going down, the availability of massive computational power is increasing, eroding the cryptographic algorithms’ effectiveness. Given the ubiquity of both smartphones and the Internet, and the fact that key search is easily parallelizable, it’s relatively easy to harness the power of many thousands of computers and/or mobile devices of all types. Also, this easier access to large numbers of machines tends to blur the distinctions among the classifications. Now, individuals and small groups can muster the resources equivalent to several large organizations. Data that was once considered to be vulnerable only to an attack by a “large, well-equipped organization” may now be vulnerable to a just few people with friends on the Internet and no budget. This ability may especially affect policies that have assumed a feasible attack on AES would
require an investment in specialized hardware.

Since the 1990s, both the PCs (see Moore’s Law) and the Internet expanded to the consumer market. Public computing can provide more computing power than any supercomputer, cluster, or grid, and the disparity will only grow over time, benefiting certain projects (for example, science research can be significantly accelerated through this approach – see SETI@home) that intensively require this virtual supercomputer’s aggregated processing power. By distributing the grid (or neural) network, it gains new capabilities, greater fault tolerance, and an easier implementation, while also eliminating many of the costs associated with a local supercomputer, like hardware (even upgrading), electricity, cooling, floor space and system administrators.

Conducting a public computing project requires adapting an application program to various platforms, implementing server systems and databases, keeping track of user accounts and dealing with redundancy and error conditions.

To be amenable to public computing, a task must be divisible into independent pieces whose ratio of computation to data is high (otherwise the cost of Internet data transfer may exceed the cost of doing the computation centrally). For example, medical and genetics projects that involve searching a set of millions or billions of molecules, or matching a set of proteins with a DNA sequence, which represent easily parallelizable tasks [3]. Distributed.net is a worldwide distributed computing effort attempting to solve large scale problems using idle CPU or GPU time. It’s in fact the second project of this kind, launched in early 1997 (a year after the Great Internet Mersenne Prime Search). It has famously solved DES Challenge II-1 in 41 days in early 1998, while also brute forcing RC5 messages encrypted with 56-bit and 64-bit keys, and is currently working on cracking a 72-bit key; as of March 2013, 2.77% of the keyspace has been searched, meaning it would take approximately 120 years to test every possible remaining key, at the current rate [5]. Recent distributed.net statistics emphasize the vastly superior capabilities of GPUs in this domain, currently completing almost 87% of all work units each day [6]. On high-end NVIDIA video cards, upwards of 600 million keys/second has been reported [7]. Considering a very high end single CPU working on RCS-72 may achieve 50 million keys/second, the CUDA advancement represents a performance increase of roughly 1000%. AMD’s FireStream for ATI provides key rates in excess of 1.8 billion keys/second on some of the products in the Radeon HD 5000 and 6000 series [2].

The DESCHALL effort endeavored to crack RSA’s DES Challenge by means of a large-scale DESCHALL’s approach centered on a single “key server” which kept track of which blocks of keys had been tested [4]. Clients would then contact the server, via the Internet, to request work and report the results.

The Berkeley Open Infrastructure for Network Computing (BOINC) is an open source middleware system for volunteer and grid computing. Now a platform for distributed applications in diverse science areas (ranging from mathematics and medicine to molecular biology, climatology and astrophysics), it was originally developed to support the SETI@home project. Another established distributed computing project is Folding@home [8], whose research is focused on bio molecular phenomena (primarily protein folding, hence the name). Although the name would suggest that it also belongs to the BOINC platform, it’s a separate project, developed and operated at a Stanford University laboratory, known for pioneering the use of PlayStations and Message Passing Interface (used in parallel programming on multi-core processors), alongside GPUs, for distributed computing [1].

4. Problem solution

The solution described below represents an option for a brute-force attack distributed between any numbers of mobile devices. The reasoning is that such a threat is very real and feasible nowadays.
Figure 1. Proposed architecture

The application package consists of a web service that can be accessed through the SOAP protocol (“create once, consume everywhere”) by two mobile clients (namely Android 2.3 and Windows Phone 7.5), and not only. Clients can connect to the web service using the designated installed application and start generating keys accordingly in order to test them against the provided ciphertext, before uploading the results. The architecture is somewhat similar to the other client – server platforms used by the distributed projects presented earlier.

Although usually the web services’ objective is to offer clients the ability to remotely execute certain methods, thus eliminating the need for downloading separate API, the present case represents quite the opposite: resources are being distributed by the server in exchange for additional computational power, provided by the clients’ parallelized processing of the same shared data.

Although usually the web services’ objective is to offer clients the ability to remotely execute certain methods, thus eliminating the need for downloading separate API, the present case represents quite the opposite: resources are being distributed by the server in exchange for additional computational power, provided by the clients’ parallelized processing of the same shared data.

Since sending large amounts of data over the Internet is not advisable, both for reasons involving security and, certainly more important in the present case, performance (because the mobile clients communication with the server isn’t designed to work just when being connected to a Wi-Fi hotspot, but also when using cellular data, and the signal is not great – Edge, for example). This potential bottleneck has been dispatched by using only a block’s number on the server side, with the client being able to generate the corresponding range of keys, for it knows the size of both the block and the key.

In the current implementation, they keys are handled using the BigInteger data format, for ease of use. While the long data type would have sufficed in the case of a DES implementation, it’s also arguable that incrementing a byte array through shifting might reap greater benefits in terms of performance.

In the presented Electronic Code Book scenario, which is simpler (by discarding the initialization vector) and faster than CBC (but also less secure), matching with just the first decrypted block is considered sufficient, thus saving processing time.

Endianness is a key aspect that needed addressing, specifically on the client side, where Java’s use of big endian would conflict with Microsoft’s C# little-endian (although the former arguably represents the more natural ordering).

Such an architecture presents many advantages, from which the following stand up:

- **Scalability** – new devices can be added incrementally in order to gain processing strength, through growth;
- **Reliability** – even if one device crashes, the rest of them remain unaffected and the system can survive as a whole;
- **Flexibility** – both mobile clients are fairly easy to install and run (as apps), while new features added on the server backend side will be immediately available and easily accessible;
- **Resources sharing** – the server only generates block numbers, which are swiftly distributed to the clients, where the actual key generation takes place;
Overall speed and performance – as mentioned above, actual computing is done on the clients, and a collection of processors can potentially provide more power than even a so called “supercomputer”;

However, it might also present several disadvantages, such as software support (more components often result in a higher change of errors occurring, while troubleshooting and diagnosing problems are a bit more cumbersome, sometimes requiring remote access) and network load (lots of communication messages).

![Figure 2. Algorithm flowchart](image)

The system’s data flow is best described from the web service’s perspective, through its public methods:

The third method represents the central part of the system, with the client proceeding to request blocks with the designated, fixed number of keys, once all prerequisites are satisfied. As the algorithm iterates through the collection described above, it tries every key against the encrypted text. If the result is positive, the server will be swiftly called, but in the other (overwhelming) number of cases, the results will be uploaded (basically by sending a negative message), before requesting another block number (the proper manner in which is done is explained later on).

There are two primary vulnerabilities associated with this kind of architecture:

- The presence of a malicious client infiltrated in the structure, that would deny finding a match, and upload fake results (a “no match found”, in our case);
- The off chance of a client blocking a key, by not sending back the result (this could be intentional or not, by losing network access or battery power);

While the former can be resolved partially through authentication, it might be cumbersome to manage, and therefore insufficient. DESCHALL used to prevent sabotage by using a different approach: the client’s “Not found” message contained additional data, calculated during its search, to allow the server to verify that the client had actually searched the assigned key space. BOINC also does some further calculations on the server side to verify the client’s work, but this would be really time consuming, and also not suitable for the present project’s area of interest.

Implementing a duplex service would be a method of settling the latter problem, as the server would periodically check if a client is still “alive”, and retrieve its assigned keys in case no response is received. However this type of approach is described in technology as “overkill”, because it would also be time consuming, and the lightweight architecture concept is all about speed and performance.

Instead, these reliability issues have been addressed through a simple, yet efficient approach, which could be considered the core of the architecture, its most important part. The answer lies with the “Attempts” field in the data table managing the key blocks, which helps in keeping the status of each and every one of them. It’s designed as an integer, because a Boolean type would have rendered the whole system vulnerable to the upload of false results, by lacking a method to identify then.

Upon service start, as clients begin to request key blocks, they are also inserted in the database. During this stage, the selected key is always the one just after the last inserted one, even if not all the already distributed ones are definitely ruled out.
Once the keyspace is covered by inserting all the keys in the database, key blocks will be distributed in a heuristic manner ("greedy algorithm"). This means that if a one key block remains lagging behind in terms of the number of confirmed attempts, it is automatically considered the best candidate for the solution, and therefore be assigned to all clients, until at least one of them actually tests it.

For example, if even after dispatching the keys that remained unattended after the first iteration through the keyspace the message is not decrypted, the algorithm will proceed, making it virtually impossible for saboteurs to discard the same matching key over and over again.

The solution might be improved in the future, through features like load balancing, where clients would automatically increase the size of the key blocks they requested so that they would gradually reach a compromise between the number of tested keys, and the time required for browsing through one block (which will always be $2^N$ keys in size, where $N$ is generally between 16 and 25). However, one must also take into consideration a scenario in which an unusually powerful client would get to request a proportionally large key block, which could then potentially block the whole system in case of failing the completion of both the processing and the upload.

One way around this would be to upgrade the architecture to allow clients to request more than one key block at a time, storing the results until capable of uploading the results.

5. Conclusions
Although encryption algorithms with fixed-length key sizes will always be subject to brute-force key search attacks, up to this moment, no 128-bit key for AES (or any cryptosystem) has been successfully found in a brute force search (although neither did any scientific community project attempt it, to our knowledge). The following table has been compiled after a series of tests run on different devices:

<table>
<thead>
<tr>
<th>Block size</th>
<th>Sony Ericsson Xperia Arc S (1.4 Ghz, single core)</th>
<th>Samsung Galaxy S II (1.2 Ghz Dual-core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{9} - 2^{10}$</td>
<td>512</td>
<td>776</td>
</tr>
<tr>
<td>$2^{11} - 2^{14}$</td>
<td>682</td>
<td>1034</td>
</tr>
<tr>
<td>$2^{16} - 2^{22}$</td>
<td>697</td>
<td>1057</td>
</tr>
</tbody>
</table>

It’s easy to observe that the overall performance tends to increase with the block size, but only up to a certain point. Although we would be tempted to start with a very large block size, a cautious approach needs to be taken, because in case of failures (Internet access loss, for example) it would of course be much worse losing a $2^{20}$ key block’s work. Also, an eye must be kept on the power consumption, in respect to battery performance. Extrapolating, if such results could be obtain on a decent device, imagine what could be achieved with a client dedicated (by adding multithreading capabilities and other such performance features) to suit the cutting edge technology available in today’s state of the art smartphones. For example, Samsung recently launched a ~1.5GHz octo-core CPU [9]. Also, given the fact that there are now more than 1 billion smartphones in the world, outnumbering the more basic feature phones in worldwide shipments for the first time this last quarter, the potential is there for all to see. On average, the right key will be found after searching about half way through all the key space (meaning $2^{127}$ keys have to be tried).

One might imagine that, to be extra secure, the key length might be doubled as to use a 256-bit one instead. $2^{256}$ is enormously bigger than $2^{128}$ (in fact, precisely $2^{128}$ times larger). Although the number of keys needed to be tried would get mind-bogglingly bigger, it would be wrong to think that this made it more secure, because searching through the whole keyspace was already unimaginably hard, and there is no gain in making it harder. Presuming we had a trillion of such slightly-better-than-average mobile devices under a unified direction, that might be able to search $2^{50}$ (about one quadrillion) keys per second. There are about 31557600 seconds per year, so working together like this, about $2^{75}$ (or 10 septillion) keys could be checked per year.

At that rate it would take $2^{53}$ (10 quadrillion) years to work through half of the $2^{128}$ key space. If we take the universe to be about 15 billion years old, then the amount of time it would take this distributed network, working faster than the combined power of a trillion super computers, would be more than 600,000 times the age of the universe. In case this analogy has gone too far astray, quick (and really high) estimations have suggested that all of the computing power on Earth turned to trying AES keys couldn’t check more than $2^{75}$ keys per year. At that rate, it would take more than half a million times the age of the universe to go through half of the $2^{128}$ possible AES keys. There are other kinds of attacks AES needs to defend against, but nothing could be gained by increasing the key size, because even with all the resources on Earth contributing, not even a dent could be made before the universe comes to an end.

The chances of finding the key is as close to zero as we could possibly want. Let’s name them $\epsilon_1$ (epsilon 1), a really small number. In case of a 256-bit key, the chances are represented by another number as close to zero as we could possibly want, called $\epsilon_2$. Sure, $\epsilon_2$ is many times smaller than $\epsilon_1$, but both $\epsilon_1$ and $\epsilon_2$ are already as close to zero as we
could possibly want. The practical security gain in using the larger key size is pretty much the difference between $\epsilon_1$ and $\epsilon_2$. That difference, for all meaningful purposes, is zero.

We all know that computers consume electricity. As it happens, computation (and inspecting keys) has to consume energy (Landauer’s principle [10]). It is actually the destruction (or overwriting) of information that necessarily consumes energy, but that happens when the previously searched range of keys is disposed, and a new one is generated. If this could be done using the absolute theoretical minimum energy for a single computation, $2.85 \times 10^{-21}$ J, distributed network of billion super-fast (and now unfathomably efficient) devices would require about $1/100^{th}$ of the total amount of energy humanity uses in a year [11] to work through half of the 2128 key space. While using this type of effort to do a brute force attack on a 128-bit key might seem difficult today, the potential for performing very large computations besides using expensive, dedicated hardware, or supercomputers can be seen already, so it will certainly be possible in the not so distant future, as more people become involved in these efforts, and as processor speeds increase according to Moore’s Law. At the same time, attacks on smaller key lengths will become easier and easier. Although current distributed projects, like the present one, have not been a direct threat to computer security (i.e., we didn’t decipher actual, sensitive data), there is no reason that a similar project type could not be assembled by the “underworld” of computer crackers for their own use. Such an attempt would require the attacker to not only be relatively well-funded, but also dedicated enough to make a real investment of their time.

Understanding cryptographic attacks is not only important to the science community, but it also serves to improve algorithms: IBM was aware of the technique of differential cryptanalysis before first publishing its DES algorithm in 1977, and used this knowledge to improve it, by adding substitution boxes to thwart these types of attacks.

**Acknowledgement**

Parts of this research have been published in the Proceedings of the 7th International Conference on Security for Information Technology and Communications, SECITC 2013.

**References**


