International Journal of Research
Available at
https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 05 Issue 16
June 2018

Dropping Denial-Of-Service Attacks Using Software Puzzle

J. Tulasi Rajesh[1], Dr. G. P Saradhi Varma[2]
Assistant Professor[1], Professor[2],
Department Information Technology[1, 3.]
S.R.K.R Engineering College, Bhimavaram, Andhra Pradesh, India[1, 2.]

ABSTRACT

Denial-of-service (DoS) and distributed DoS (DDoS) are the major threats in cyber-security. As a countermeasure to such threats client puzzle scheme is implemented. The client puzzle demands a client to perform computationally expensive operations before being granted services to the client from a server. However, an attacker can inflate the capability of DoS/DDoS attacks with fast puzzle solving software and/or built-in graphics processing unit (GPU) hardware to significantly weaken the effectiveness of client puzzles. In order to prevent DoS/DDoS attackers from inflating the puzzle-solving capabilities, a new client puzzle referred to as software puzzle is implemented. Unlike the existing client puzzle schemes, which publish their puzzle algorithms in advance, a puzzle algorithm in the implemented software puzzle scheme is randomly generated only after a client request is received at the server side and the algorithm is generated such that: a) an attacker is unable to prepare an implementation to solve the puzzle in advance and b) the attacker needs considerable effort in translating a central processing unit puzzle software to its functionally equivalent GPU version such that the translation cannot be done in real time.

INTRODUCTION

DENIAL of Service (DoS) attacks and Distributed DoS (DDoS) attacks attempt to deplete an online service resources such as memory, network bandwidth, and computation power by devastating the service with bogus requests. [1] For example, a malicious client sends a large number of garbage requests to an HTTPS bank server. As the server has to spend a lot of CPU time in completing SSL handshakes, it may not have sufficient resources left to handle service requests from its customers, resulting in lost businesses and reputation. DoS and DDoS attacks are not only theoretical, but also realistic. DoS and DDoS are effective if attackers spend much less resources than the victim server or are much more powerful than normal users [1]. In the example stated above, the attacker spends negligible effort in producing a request, but the server has to spend much more computational effort in HTTPS handshake (*e.g.*, for RSA decryption). In this case, conventional cryptographic tools do not enhance the availability of the services; in fact, they may degrade service quality due to expensive cryptographic operations. The seriousness of the DoS/DDoS problem and their increased frequency has led to the advent of numerous defense mechanisms[6]. Let γ denote the ratio of resource consumption by a client and a server. Obviously, a countermeasure to DoS and DDoS is to increase the ratio γ , *i.e.*, increase the computational cost of the client or decrease that of the server. Client puzzle[7] is a well-known approach to increase the cost of clients as it forces the clients to carry out heavy operations before being granted services.

Generally a client puzzle scheme consists of three steps:

i) puzzle generation

ii) Puzzle solving by the client

iii) Puzzle verification by the server.

Hash-reversal is an important client puzzle scheme which increases a client cost by forcing the client to crack a one-way hash instance. Technically, in the puzzle generation step, given a public puzzle function *P* derived from one-way functions such as SHA-1 or block cipher AES, a server randomly chooses a puzzle challenge *x*, and sends *x* to the client. In the puzzle-solving and verification steps, the client returns a puzzle response (x, y), and if the server confirms $x = P(y)$, the client is able to obtain the service from the server. In this hash-reversal puzzle scheme, a client has to spend a certain amount of time t_c in solving the puzzle (*i.e.*, finding the puzzle solution *y*), and the server has to spend time t_s in generating the puzzle challenge *x* and verifying the puzzle solution *y*. Since the server is able to choose the challenge such that $t_c >> t_s$ for normal users, *i.e.*, $\gamma >> 1$, an attacker cannot start DoS attack efficiently by solving many puzzles. Alternatively, an attacker can merely reply to the server with an arbitrary number y' so as to exhaust the server's time for verification. In this case, although $\gamma < 1$ such that defense effect of client puzzle is weakened, the server time t_s is still much smaller than the service preparation time (*e.g.*, RSA decryption) or service time (*e.g.*, database process) as the returned answer will be rejected at a high probability. Therefore, in either case, a client puzzle can significantly reduce the impact of DoS attack because it enables a server to spend much less time in handling the bulk of malicious requests. Of course, optimizing the puzzle verification mechanism is very important and doing so will undoubtedly improve the server's performance[8]. The existing client puzzle schemes assume that the malicious client solves the puzzle using legacy CPU resource only. However, this assumption is not always true. Presently, the many-core GPU (Graphic Processing Unit) component is almost a standard configuration in modern desktop computers (*e.g.*, ATI FirePro V3750 in Dell T3500), laptop computers (*e.g.*, nVidia Quadro FX [2] 880M in Lenovo Thinkpad W510), and even smart phones (*e.g.*, PowerVR SGX540 in Samsung I9008 Galaxy*TM* S). Therefore, an attacker can easily utilize the "free" GPUs or integrated CPU-GPU to inflate his computational capacity[5]. This renders the existing client puzzle schemes ineffective due to the significantly decreased computational cost ratio γ . For example, an attacker may amortize one puzzle-solving task to hundreds of GPU cores if the client puzzle function is parallelizable (*e.g.*, the hash reversal puzzle), or the attacker may simultaneously send to the server many requests and ask every GPU core to solve one received puzzle challenge independently if the puzzle function is non-parallelizable (*e.g.* modular square root puzzle and Time-lock puzzle[10]). This parallelism strategy can dramatically reduce the total puzzle-solving time, and hence increase the attack efficiency. Green et al. examined various GPU-inflated DoS attacks, and showed that attackers can use GPUs to inflate their ability to solve typical reversal based puzzles by a factor of more than 600. Moreover, in order to defeat GPU-inflated DoS attack to client puzzles, the scheme was introduced to track the individual client behaviour through client's IP address Nonetheless, if IP tracking is effective to thwart the GPU inflation, IP filtering can be used to defense against DoS attacks directly without utilizing client puzzles. In other words, their defense against GPU-inflated DoS attacks may not be attractive in practice. The scheme dynamically embeds client-specific challenges in web pages, transparently delivers server challenges and client responses. However, this scheme is vulnerable to DoS attackers who can implement the puzzle function in real-time. Technically, an attacker can rewrite the puzzle function $P(\cdot)$ with a native language such as C/C++ such that the cost of an attacker is much smaller than that the server expects. Even worse, a GPU-inflated DoS attacker can realize the fast software implementation on the many-core GPU hardware and run the software in all the GPU cores simultaneously such that it is easy to defeat the web-based client puzzle scheme. Obviously, if a puzzle is designed

International Journal of Research
Available at
https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 05 Issue 16
June 2018

based on client's GPU capability, the GPU-inflation DoS does not work at all. However, it is not recommend to do so because it is troublesome for massive deployment due to (1) not all the clients have GPU-enabled devices; and (2) an extra real-time environment shall be installed in order to run GPU kernel. By exploiting the architectural difference between CPU and GPU, presents a new type of client puzzle, called software puzzle, to defend against GPU-inflated DoS and DDoS attacks. Unlike the existing client puzzle schemes which publish a puzzle function in advance, the software puzzle scheme dynamically generates the puzzle function *P(.)* in the form of a software core *C* upon receiving a client's request. Specifically, by extending DCG technology which produces machine instructions at runtime[4] the present software puzzle scheme randomly chooses a set of basic functions, assembles them together into the puzzle core *C*, constructs a software puzzle C_{0x} with the puzzle core *C* and a random challenge *x*. If the server aims to defeat high-level attackers who are able to reverse-engineer software, it will obfuscate C_{0x} into an enhanced software puzzle. After receiving the software puzzle sent from the server, a client tries to solve the software puzzle on the host CPU, and replies to the server, as the conventional client puzzle scheme does. However, a malicious client may attempt to offload the puzzle-solving task into its GPU. In this case, the malicious client has to translate the CPU software puzzle into its functionally equivalent GPU version because GPU and CPU have totally different instruction sets designed for different applications. Note that this translation cannot be done in advance since the software puzzle is formed dynamically and randomly. As rewriting/translating a software puzzle is time-consuming, which may take even more time than solving the puzzle on the host CPU directly, software puzzle thwarts the GPU-inflated DoS attacks.

A. NOTATIONS USED

x: challenge chosen by server.

m: message collected by server from outside environment.

y: solution to the puzzle challenge x.

(x', y'): puzzle response returned from client.

P(.): puzzle algorithm such that x=P(y , m).

C: software implementation of P(.) which is considered as puzzle core.

C_{0x} : puzzle which embeds the information of x into C.

C_{1x} : Obfuscated C_{0x}.

γ: ratio of resource consumption by a client and a server.

t_s: time taken by server to generate the puzzle x.

t_c: time taken by client to solve the puzzle x.

>>: much greater than.

B. DIFFERENCE BETWEEN CPU AND GPU

Unlike modern CPUs, modern GPU executes massively data parallel programs in quite predictable way. Although both CPU and GPU software can be implemented using the same higher language such as C, their low level instruction sets are totally different. Particularly some instruction operations are not supported in GPU software. For example, self-modifying code, widely used in software protection, modifies the software itself on the fly so as to raise the bar of hacking. As all the GPU cores share the same kernel, if one thread modifies the kernel, the final software output is hard to predict due to independent threads.

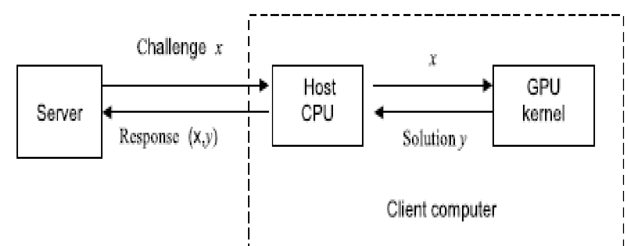


Fig 4.1 GPU – Inflated DoS attack against data puzzle

A CPU processor is usually much slower than a GPU processor, but one CPU core is much faster than one GPU core. In addition, one CPU dominates its resources such as memory and cache, but all GPU cores share resources including memory and cache. If a GPU kernel were to ask many shared resource, the number of cores used in the application would be much smaller than the available cores such that the potential of GPU would not be fully utilized. In this case, GPU may be slower than CPU.

III. SOFTWARE PUZZLE

A client puzzle is classified into two types: data puzzle and a software puzzle.

DATA PUZZLE: If a puzzle function P, is fixed and disclosed in advance, which is similar to all existing client puzzle schemes[8], then it is said to be data puzzle.

SOFTWARE PUZZLE: If a puzzle function P, is arbitrary and not known in advance, then it is said to be Software puzzle.

Data puzzle aims to enforce the client's computation delay of the inverse function $P^{-1}(x)$ for a random input x; while a software puzzle aims to deter an adversary from understanding / translating the implementation of a random puzzle function P(.). Unlike a data puzzle challenge, which includes a challenge data only, a software puzzle challenge includes a dynamically generated software C(.) including a data puzzle function as a component. Although a software puzzle scheme does not publish the puzzle function in advance, it follows the Kerchoff's Principle because an adversary knows the algorithm for constructing software puzzles, and is able to reverse-engineer the software puzzle $C1_x$ to know puzzle function P(.) several minutes later after receiving the software puzzle.

A. GPU-INFLATED DOS ATTACK

When a client wants to obtain a service, she/he sends a request to the server. After receiving the client request, the server responds with puzzle challenge x. If the client is genuine, she/he will find the puzzle solution y directly on the host CPU and sends the response (x,y) to the server. However , there may be a chance for the malicious user who contrls the host may send the challenge x to GPU and exploit the GPU resource to accelerate the puzzle-solving process.

B.SOFTWARE PUZZLE FRAMEWORK

In order to defeat the GPU-inflated DoS attack, data puzzle is extended to software puzzle. At the server, the software puzzle scheme has a code block warehouse W storing various software instruction blocks. Additionally, it includes two modules: generating the puzzle C_{Ox} by randomly assembling code blocks extracted from the warehouse, and obfuscating the puzzle C_{Ox} for high security puzzle C_{1x}.

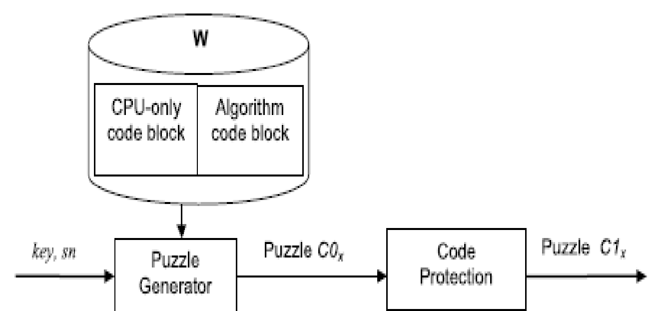


Fig 4.2 Software Puzzle generated with secret key and nonce

C. CONSTRUCTION OF CODE BLOCK WAREHOUSE

The code block warehouse W stores compiled instruction blocks in Java bytecode or C binary code. The purpose to store compiled codes rather than source code is to save server's time as the server has to take extra time to compile source codes into compile codes in the process of puzzle generation. The requirements for each block are:

International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 05 Issue 16
June 2018

In order to assemble the code blocks together, each block has well-defined input and output parameters such that the output from one block can be used as the input for the following blocks.

The size of each code block is decided by the security parameter k. Given that the size of software puzzle is constant, if the block size is smaller, there are more blocks on average such that more puzzles can be generated. Thus smaller block size implies higher security level because an attacker has to spend more effort to figure out a puzzle. The disadvantage of small block size is that the server has to spend more time in extracting the basic blocks and assembling the extracted blocks into software puzzle.

Preferably, the warehouse stores both Java bytecode and the C binary code. Because the former is applicable to different OS platforms but slow, it is suitable to deliver the software puzzle to the client in the format of Java bytecode. In contrast, the later is fast and is used by the server for generating the stored pair (x,y). As a result, this Java C hybrid scheme ensures that the server has advantage over the client in terms of resource consumption, as well as the support of cross platform deployment. In general, code blocks can be classified into two categories namely CPU-only instruction block and data puzzle algorithm block.

CPU-ONLY INSTRUCTION BLOCK: Unlike CPU, GPU is designed for the predictable graphic processing such as matrix operations, not generic logic processing. As branching operations such goto, try-catch-finally are inherently non-predictable and are non-paralleable, executing them in GPU is slow such that the major merit of GPU cannot be exploited by the attacker. Moreover, some hardware related operations such as reading hardware input and surfing network cannot be performed on GPU. Further, the state of the art GPUs do not support dynamic thread generation. Finally, the high speed shared memory is shared by all the GPU thread blocks together such that the size of fast accessible memory available to each thread is small. Therefore, if the puzzle kernel demands large shared memory, the GPU parallelism potential will be restricted seriously, or the threads have to access the global memory at a much slower speed.

Instruction	Difference Exploited
Create new thread	GPU does not support child thread
Create new class	GPU does not support dynamic code
Human machine interface	GPU cannot support human machine
Network interface	GPU does not support networking function
Goto (address)	GPU cannot support branch
Try-catch	GPU does not support exception handling
Allocate large memory	GPU has much smaller memory than CPU
Read local cookie	GPU cannot directly read CPU storage

Table I Examples of CPU only instructions

II. DATA PUZZLE ALGORITHM BLOCK: Similar to the blocks in data puzzle, algorithm blocks perform the mathematical operations only. For ex. In an AES round, ShiftRows code block outputs a transformed message matrix which can be used as input of any other operation such as MixColumn code block without incurring parameter mismatch errors.

D.SOFTWARE PUZZLE GENERATION

In order to construct a software puzzle, the server has to execute three modules namely: puzzle core generation, puzzle challenge generation, software puzzle encrypting.

PUZZLE CORE GENERATION: From the code block warehouse, the server first choses n code blocks based on hash functions and a secret, eg., the jth instruction block b_{i_j} , where $i_j = H_1(y , j)$, and $y = H_2(key , s_n)$, with one-way functions $H_1(.)$ and $H_2(.)$, key is the server's secret and s_n is nonce or timestamp.

All the chosen blocks are assembled into a puzzle core, denoted as $C(.) = (b_{i1}, b_{i2}, \ldots b_{in})$.

PUZZLE CHALLENGE GENERATION:

Given some auxiliary input messages such as IP addresses and in-line constants, the server calculates a message m from public data such as their IP addresses, port numbers and cookies, and produces a challenge $x = C(y,m)$, which is similar to encrypting plaintext m with key y to produce cipher text x.

As the attacker does not know the puzzle core C(.) in advance, it cannot exploit GPU to solve the puzzle C_{Ox} in real time using the basic GPU-inflated DoS attack. Nevertheless, if the puzzle is generated as above, it is possible for an attacker to generate the GPU kernel by mapping the CPU instructions in C_{Ox} to the GPU instructions one by one, i.e., to automatically translate the CPU software puzzle C_{Ox} into its functionally equivalent GPU version.

CODE PROTECTION:

Code obfuscation is able to thwart the puzzle translation threat to some extent. Though, there are no generic obfuscation techniques which can prevent a patient and a hacker from understanding a program in theory , results in show that the obfuscation does increase the cost of reverse-engineering. Thus, although code obfuscation may not be satisfactory in long-term software defense against hacking, it is suitable for fortifying software puzzles which demand a protection period of several seconds only.

A software puzzle consists of instructions and each instruction has a form (opCode, [operands]), where opCode indicates which operation such as addition, shift, jump etc, while the operands , varying with opCode, are the parameters like target address of jump instruction to complete the operations. As a popular obfuscation technology, code encryption technology treats software code as data string and encrypts both operand and opCode. Concretely, given the code C_{Ox}, the server generates an encrypted puzzle $C1_x = E(y, C_{Ox})$, where E(.) is a cipher such as AES and y is used as the encryption key.

In all, there are two-layer encryptions. The outer layer is used to encrypt the software puzzle C_{Ox}, and the inner layer uses the puzzle software to encrypt the challenge as like data puzzle. Therefore, after receiving C_{1x}, the client has to try $y' = y$, the original software puzzle C_{Ox} can be recovered and further used to solve the challenge.

IV. PACKAGING SOFTWARE PUZZLE

Once a software puzzle C_{1x} is created at the server side and compiled into a Java class file C_{1x}.class, it will be delivered to the client who requests for services over an insecure channel such as Internet, and run at the clients side. To demonstrate the applicability of software puzzle, Applet is used to implement software puzzles such that the software puzzle implementation has the same merits as easy deployment.

ALGORITHM 1:

init.class structure for reloading puzzle class on JVM. If a correct solution y is found, $C1_x$.class shall be the same as the original puzzle $C0_x$.class.

Read C_{1x}.class

Repeat

Randomly chose a small y'

Decrypt C_{1x}.class with key y' into class C_{0x}.class

Load class C_{0x}.class

Invoke C_{0x}.class to obtain m' and further $x' = C_0(y', m')$

Until $x' = x$

Output (x', y')

ALGORITHM 2:

init.class structure for activating puzzle class on dedicated sandbox.

Read C_{1x}.class

Load class C_{1x}.class

Repeat

Randomly chose a small y'

Decrypt C_{1x}.class with key y' into class C_{0x}.class

Invoke C_{0x}.class to obtain m' and further x' = C_0(y' , m')

Until x' = x

Output (x' , y')

V. SECURITY ANALYSIS

Software puzzle aims to prevent GPU from being used in the puzzle solving process based on different instruction sets and real-time environments between GPU and CPU. Conversely, an adversary may attempt to deface the software puzzle scheme by simulating the host on GPU, cracking puzzle algorithm, re-producing GPU-version puzzle, abusing the access priority in puzzle solving which are discussed below.

EMPLOYING HOST SIMULATOR ON GPU

If an attacker is able to run a CPU simulator over GPU environment, the software puzzle can be executed on GPU directly. However, this simulator based attack can be impractical in accelerating the puzzle solving process due to the following reasons.

"VM software must emulate the entire hardware environment, so that problems can arise if the properties of hardware resources are significantly different in the host and the guest" [9]. To the best of knowledge, there is no host simulator on GPU at present. Indeed, it is not trivial to develop a full-functional CPU simulator on GPU because the CPU environment including Operating System, and all the imported Java libraries (and their imported libraries and so on) must be simulated. If only a portion of simulator functions is implemented, the GPU kernel may have to communicate with the host for the non-simulated functions. In this case, the GPU-inflation function is reduced significantly because it cannot run in a parallel way and the GPU-CPU communication channel is much slower than its internal memory access.

A software running over a simulator is much slower than over its guest environment directly because there are more processing steps to execute the software instructions.

CRACKING DATA PUZZLE ALGORITHM

According to Algorithm 1 or Algorithm 2, an adversary obtains the puzzle solution (x' , y') to the software puzzle C_{1x} , such that x = x' = C_0x (y' ,m'), where number x is hard-coded in the software puzzle and m˜ is derived on the fly. Since the software puzzle is encrypted with the standard cipher, an adversary has to recover the puzzle software by brute force. Moreover, for the inner-layer encryption, as C(·) is an encryption function, theoretically, an adversary cannot find a valid solution (x', y') in a better way than brute force given that y is over a small interval. Hence, the practical strategy of the attacker is to accelerate the brute force process by exploiting the parallel computation capability of GPU cores. Even the code blocks (e.g., AES round transformations) are cryptographic primitives, their combination may be not as secure as the original ones for the basic software puzzle. But in software puzzle, by randomly adding some round transformations into the existing AES code. As the new added transformation will increase the diffusion effect, the AES variants have at least the same security level as standard AES.

REPLAYING DATA PUZZLE

When a software puzzle is built upon a data puzzle, the number of software puzzles is required to be very large such that an attacker is unable to re-construct the GPU-version software puzzles in advance and re-use them. Indeed, this requirement can be easily satisfied. For instance, even though a service provider merely adds one AES round transformations between two AES transformations in the standard 10 rounds, the number of AES variants is up to 49×4+3 = 278. Moreover, a software can have many polymorphic

codes such that the number of software puzzles is even larger. Unfortunately, a smart adversary may collect all the code blocks in the warehouse W, and rebuild the GPU version code block warehouse gpu in advance. Once a new software puzzle is delivered to the adversary, he will reconstruct the GPU-version puzzle by matching the puzzle code blocks against the software puzzle. In this case, the adversary is able to increase the attack performance. However, as the server encrypts the puzzle software C_0x into C_{1x} , the adversary has to recover C_0x by brute force, and hence cannot successfully re-construct the GPU-version puzzle by matching code patterns.

VI. EXPERIMENTAL EVALUATION

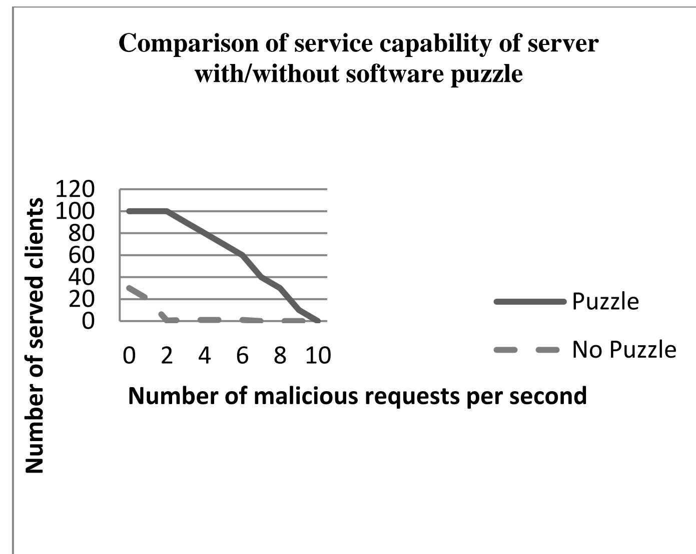
In this experiment, Apache Tomcat server is used to respond for clients requests. The server, on receiving a request from the client, the servlet will generate software puzzle. An experimental server is built which included CPU only instructions and AES round operations a module for puzzle generation and a module for code encryption.

EXPERIMENT RESULTS

The most popular on-line transaction protocol is SSL/TLS and the corresponding server performs an expensive RSA decryption operation for each client-connection request, thus the generated software puzzle is vulnerable to DoS attack. For decryption RSA is used to evaluate the defense effectiveness.

Assume the time to perform one RSA decryption be t_0, and the time to generate and verify one software puzzle be t_s . Suppose the number of attacker's requests be n_a, and the number of genuine client requests be n_c, the server's computational time required for replying all the requests is $\tau_1 = (n_a + n_c)\times t_0$ if there is no software puzzle; otherwise, $\tau_2 = (n_a + n_c) \times t_s + n_c \times t_0$ given that the adversary does not return valid solutions to the puzzles. Thus, software puzzle defense is effective if

$$\tau_1 \geq \tau_2, \text{ i.e., } n_a \geq \frac{ts}{t0-ts} n_c$$



CONCLUSION AND FUTURE WORK

Software puzzle scheme is implemented to overcome the drawbacks of existing client puzzle scheme which generates a client puzzle in advance and also to overcome GPU-inflated DoS attack. The implemented software puzzle scheme adopts software protection technologies to ensure challenge data confidentiality and code security for an appropriate time period. Hence, it has different security requirement from the conventional cipher which demands long-term confidentiality only. Since the software puzzle may be built upon a data puzzle, it can be integrated with any existing server-side data puzzle scheme, and easily deployed as the present client puzzle schemes do. Although this scheme focuses on GPU-inflation attack, its idea can be extended to thwart DoS attackers which exploit other inflation resources such as Cloud Computing. For example, suppose the server inserts some anti-debugging codes for detecting Cloud platform into software puzzle, when the puzzle is running, the software puzzle will reject to carry on the puzzle-solving processing on Cloud environment such that the Cloud-inflated DoS attack fails. In the implemented software puzzle, the server has to spend time in constructing the puzzle.

As a future enhancement, the server time has to be saved to construct the client-side software puzzle for better performance.

REFERENCES

[1] Yongdong Wu, Zhigang Zhao, Feng Bao and Robert. H. Deng (2015): "Software Puzzle-A countermeasure to Resource inflated Denial-of Service attacks". IEEE Transaction Information Forensics and Security, vol 10, pp 168-177

[2] NVIDIA CUDA (2012): " NVIDIA CUDA C Programming Guide" Version 4.2.

[3] E. Kaiser and W.-C. Feng (2007): "mod_kaPoW: Mitigating DoS with transparent proof-of-work," in Proc. ACM CoNEXT Conf., 2007, p. 74.

[4] D. Keppel, S. J. Eggers, and R. R. Henry (1991): "A case for runtime code generation" Dept. Comput. Sci. Eng, Univ. Washington, Seattle, WA, USA, Tech. Rep. CSE-11-04-91.

[5] R. Shankesi, O. Fatemieh, and C. A. Gunter (2010): "Resource inflation threats to denial of service countermeasures," Dept. Comput. Sci, UIUC, Champaign, IL, USA, Tech. Rep, Oct. 2010.

[6] C. Douligeris and A. Mitrokotsa (2004): "DDoS attacks and defense mechanisms: Classification and state-of-the-art" Comput. Netw., vol. 44, no. 5, pp. 643–666.

[7] A. Juels and J. Brainard (1999): "Client puzzles: A cryptographic countermeasure against connection depletion attacks," in Proc. Netw. Distrib. Syst. Secur. Symp.,pp.151–165.

[8] T. J. McNevin, J.-M. Park, and R. Marchany (2004): "pTCP: A client puzzle protocol for defending against resource exhaustion denial of service attacks," Virginia Tech Univ., Dept. Elect. Comput. Eng., Blacksburg, VA, USA, Tech. Rep. TR-ECE-04-10, Oct. 2004.

[9] J. E. Smith and R. Nair, Virtual Machines (2005): "Versatile Platforms for Systems and Processes". San Mateo, CA, USA: Morgan Kaufmann, , p. 19.

[10] R. L. Rivest, A. Shamir, and D. A. Wagner (1996): "Time-lock puzzles and timed-release crypto," Dept. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep. MIT/LCS/TR-684,Feb 1996.