Software Puzzle Approach: A Measure to Resource-Inflated Denial-of-Service Attack

Vishal Walunj
Assistant Professor
DYPSOEA,
Pune, Maharashtra

Vinod Pawar
Dept. of Computer Engineering,
DYPSOEA,
SPPU, Pune, Maharashtra

ABSTRACT
In Cyber security Denial-of-service (DoS) and distributed DoS (DDoS) are two major threats, and client puzzle, which demands a consumer to perform computationally dear operations before being granted services from a server, is a well-known countermeasure to them. However, a wrongdoer will inflate its capability of DoS/DDoS attacks with quick puzzle solving package and/or intrinsic graphics process unit (GPU) hardware to considerably weaken the effectiveness of consumer puzzles. This paper shows how to stop DoS/DDoS attackers from inflating their puzzle-solving capabilities. To this end, this paper introduces a new consumer puzzle said as software puzzle. Unlike the existing consumer puzzle schemes, which publish their puzzle algorithms in advance, a puzzle algorithmic program in the gift package puzzle theme is at random generated solely once a consumer request is received at the server aspect and therefore the algorithm is generated specified: 1) Associate in Nursing wrongdoer is unable to arrange Associate in Nursing implementation to unravel the puzzle before and 2) the wrongdoer wants extended effort in translating a central process unit puzzle package to its functionally equivalent GPU version such that the interpretation can't be drained real time. Moreover, the paper shows how to implement package puzzle within the generic server-browser model.

General Terms
GPU, Data Puzzle, Hash-Reversal, Resource-Inflation

Keywords
Software puzzle, Code Obfuscation, GPU programming, Denial of Service (DoS), Distributed Denial of Service (DDoS)

1. INTRODUCTION
Denial of Service (DoS) attacks and Distributed DoS (DDoS) attacks attempt to exhaust an internet service's resources like network information measure, memory and computation power by overwhelming the service with bogus requests [1]. For example, a malicious client sends a massive variety of garbage requests to associate HTTPS bank server. As the server has got to spend lots of time in finishing SSL handshakes, it may not have adequate resources left to handle service requests from its consumers, resulting in lost businesses and name. DoS and DDoS attacks are not solely theoretical, but conjointly realistic, e.g., Pushdo SSL DDoS Attacks [1].

Denial of Service (DoS) and Distributed DDoS are effective if attackers pay abundant fewer resources than the victim server or are abundant a lot of powerful than traditional consumers. In the above example, the attacker spends negligible effort in manufacturing a request, but the server has to pay rather more machine effort in HTTPS handshaking (e.g., for RSA decryption). In this case, conventional cryptological tools do not enhance the supply of the services; if truth be told, they may degrade service quality attributable to high-priced cryptological operations.

In this paper significant interest is in the countermeasures to DoS/DDoS attacks on server computation power. Let γ denote the ratio of resource consumption by a consumer and a server. Obviously, a countermeasure to DoS and DDoS is to increase the quantitative relation γ, i.e., increase the computational price of the consumer or decrease that of the server. Client puzzle [3] is a well-known approach to extend the price of consumers because it forces the consumers to hold out serious operations before being granted services. Generally, a client puzzle theme consists of 3 steps: puzzle generation, puzzle resolution by the consumer and puzzle verification by the server. Hash-reversal is a vital consumer puzzle theme.

Technically, in the puzzle generation step, given a public puzzle function **P** derived from unidirectional 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 ready to get the service from the server. In this hash-reversal puzzle scheme, a client has to pay an explicit quantity of your time t_c in resolving the puzzle (i.e., finding the puzzle solution **y**), and the server has got to spend time t_s in generating the puzzle challenge **x** and validating the puzzle answer **y**. Since the server is able to settle on the challenge specified $t_c >> t_s$ for traditional users, i.e., **γ>>1**, a wrongdoer will not begin DoS attack expeditiously by resolution several puzzles. Alternatively, the attacker will simply reply to the server with associate whimsical variety $\tilde{y}$ thus as to exhaust the server's time for verification. In this case, although **γ < 1** such that defense impact of consumer 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 came back answer are rejected at a high likelihood. Therefore, in either case, a client puzzle will considerably scale back the impact of DoS attack as a result of it permits a server to pay abundant less time in handling the bulk of malicious requests. Of course, optimizing the puzzle verification mechanism is very vital and doing thus can beyond any doubt improve the server's performance [4].

The existing consumer puzzle schemes assume that the malicious client solves the puzzle using CPU resource solely. However, this assumption is not always true. Presently, the many-core GPU (Graphic Processing Unit) part is virtually a regular configuration in trendy desktop computers, laptop computers, and also smartphones. Therefore, a wrongdoer will simply utilize the "free" GPUs or integrated CPU-GPU to inflate his machine capability [5]. This renders the existing

client puzzle schemes ineffective attributable to the considerably reduced machine price quantitative relation γ. For example, an wrongdoer could liquidate one puzzle-solving task to lots of GPU cores if the consumer puzzle perform is parallelizable (e.g., the hash reversal puzzle), or the attacker could at the same time send to the server several requests and raise each GPU core to solve one received puzzle challenge severally if the puzzle function is non-parallelizable (e.g. modular sq. root puzzle [7] and Time-lock puzzle [8]). This parallelism strategy will dramatically scale back the total puzzle-solving time, and hence increase the attack potency. Green et al. [6] examined different GPU-exaggerated DoS attacks, and showed that attackers can use GPUs to exaggerate their ability to solve typical reversal based mostly puzzles by an element of over 600. In order to beat GPU-exaggerated DoS attack to client puzzles, they proposed to track the individual consumer behavior through client's scientific discipline address [9]. However, if IP chase is effective to thwart the GPU inflation, IP filtering will be wont to defense against DoS attacks directly while not utilizing consumer puzzles.

As the present browsers don't expressly support consumer puzzle schemes, Kaiser and Feng [11] developed a web-based consumer puzzle theme that focuses on transparency and backwards compatibility for progressive readying. The scheme dynamically embeds client-specific challenges in webpages, conspicuously delivers server provocations and client responses. However, this scheme is vulnerable to DoS attackers. Technically, associate wrongdoer will rewrite the puzzle function P(•) with a native language like C/C++ specified the price of an wrongdoer is way smaller than that the server expects[3]. Even worse, a GPU-inflated DoS attacker will notice the quick package implementation on the many-core GPU hardware and run the package in all the GPU cores at the same time specified **t** is simple to defeat the web based consumer puzzle theme.

Obviously, if a puzzle is designed supported client's GPU capability, the GPU-inflation DoS does not work at all. However, This paper does not suggest to try to thus as a result of it's difficult for enormous readying attributable to (1) not all the consumers have GPU-enabled devices; and (2) an additional time period surroundings shall be put in so as to run GPU kernel. By exploiting the architectural distinction between CPU and GPU, this paper presents a new kind of client puzzle, called software puzzle, to defend against GPU-inflated DoS and DDoS attacks. Unlike the existing consumer puzzle schemes that publish a puzzle function before, the software puzzle theme dynamically generates the puzzle function **P(•)** in the type of a software core **C** upon receiving a consumer's request. Specifically, by extending DCG technology which produces machine commands at runtime [10], the proposed theme at random chooses a set of basic functions assembles them together into the puzzle core **C**, constructs a software puzzle $\boldsymbol{C0_x}$ with the puzzle core **C** and a random challenge **b**. If the server aims to defeat high-level attackers who are ready to reverse-engineer the software, it will change $\boldsymbol{C0_x}$ 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 consumer puzzle theme will. However, a malicious client could try to offload the puzzle-solving task into its GPU. In this case, the malicious client has to translate the CPU puzzle into its functionally equivalent GPU version as a result of GPU and CPU have completely different instruction sets designed for various applications. This translation cannot be tried in advance since the software puzzle is created dynamically and at random. To demonstrate the applicability of software puzzle, applet is used to implement software puzzles such that the software puzzle implementation has a similar deserves as [11] in terms of simple readying, but overcomes its security weaknesses.

The reminder of this paper is organized as follows. Section 2 provides summary of GPU and its distinction with CPU. Section 3 introduces the software puzzle, the countermeasure to GPU-inflated DoS attacks; associated Section 4 addresses the packing mechanism of thus software puzzle so that the puzzle may be solved at the consumer with an applicable permission. Section 5 analyzes the security of software puzzle. Section 6 evaluates the performance of software puzzle. Section 7 draws conclusions and addresses the future work. Finally, acknowledgement is given.

- **Notations**

For understanding, significant notations used through the paper are listed below.

x: A provocations chosen by server.

m: A message gathered from environment.

y: A solution to the puzzle provocations x.

$(\tilde{x}, \tilde{y})$: A puzzle response returned from client.

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

C: Puzzle core which is the software implementation of P(·).

$C0_x$: Puzzle which embeds the information of x into C.

$C1_x$: Obfuscated $C0_x$.

2. GPU INTRODUCTION

Modern GPUs have many processing cores that can be used for general-purpose computing as well as graphics processing. Additionally, nVidia and AMD, the major GPU vendors, provide convenient programming libraries to use their GPUs for intensive computation applications. Without loss of generality, nVidia GPU will be used to present techniques in the following. For self-contained, this Section briefly introduces nVidia GPU [12], its application on the basic GPU-inflated DoS attacks, and its difference from CPU which will be exploited to defeat against the GPU-inflated DoS attack.

2.1 Overview of NVidia GPU

In nVidia architecture, a GPU has many SMs consisting of many identical processing cores. For example, the nVidia GeForce GTX 680 consists of 1,536 cores. A GPU processor has fast but small shared memory. Besides, it has access to the host's global memory which is large but slow. CUDA, the major programming language [4] for nVidia GPU, extends ANSI-standard C99 language by allowing a developer to define C functions, or kernels. For example, the client puzzle function **P(·)** can be implemented as a GPU kernel. At any one time, a GPU device is committed to a single application which may include various kernels. When a kernel is loaded into GPU and invoked, it is executed by multiple identical threads in parallel for maximum efficiency.

2.2 Difference between CPU and GPU

Unlike modern CPUs [5], which are designed to efficiently optimize the execution of single-thread programs using complex out-of-order execution strategies, a modern GPU executes massively data-parallel programs in almost predictable way. Hence, GPU does not explicitly support branch instructions. Although both CPU software and GPU

software can be implemented using the same high-level language such as C, their low-level instruction sets are totally different. Particularly, some instruction operations are not supported in GPU software. As all the GPU cores share the same kernel, if one thread modifies the kernel, the final software output is hard to predict on account of the independence of threads. A CPU processor is usually much slower than a GPU processor as a whole, 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 the registers and caches. 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. This paper will exploit the above difference between CPU and GPU to prevent GPU from being used to accelerate the puzzle-solving process.

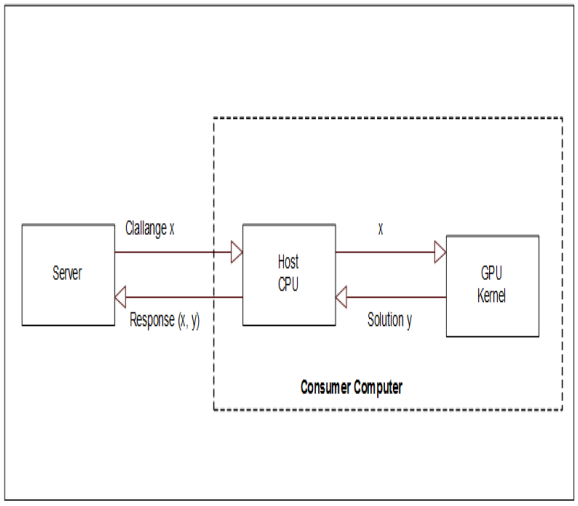

Fig 1: GPU-exaggerated DoS attack against data puzzle

3. SOTWARE PUZZLE

Software puzzle is classified into 2 sorts. If a puzzle function **P**, as all the existing consumer puzzle schemes [13][14], is fixed and disclosed in advance, the puzzle is called a knowledge or data puzzle; otherwise, it is said as a software puzzle. Data puzzle aims to enforce the client's computation delay of the inverse operate $\boldsymbol{P^{-1}(x)}$ for a random input **x**; whereas software puzzle aims to discourage associate soul from understanding/translating the implementation of random puzzle function **P(·)**. That is to mention, unlike a knowledge puzzle challenge which incorporates a challenge knowledge solely, a software puzzle challenge includes a dynamically generated software **C(·)** that as well as a knowledge puzzle operate as a part. Although a software puzzle theme will not publish the puzzle function earlier, it also follows the Kerckhoffs's Principle [15] as a result of associate soul is aware of the algorithmic program for constructing software puzzles, and is able to "reverse-engineer" the software puzzle $C1_x$ to understand the puzzle function **P(·)** many minutes later once receiving the software puzzle.

3.1 Basic GPU-Inflated DoS Attack

In order to explain software puzzle, this research recap its rival GPU exaggerated DoS attack earlier. When a consumer desires to get a service, she/he sends a request to the server. After receiving the consumer request, the server responds with puzzle challenge **x**. If the consumer is real, she/he will find the puzzle answer **y** directly on the host CPU, and sends the response **(x, y)** to the server. However, as shown in Fig 1, by using the similar mechanism in fast calculation with GPU [16], a malicious user who controls the host can send the challenge **x** to GPU and accomplish the GPU resource to speed up the puzzle-solving method.

3.2 Framework of Software Puzzle

In order to breakdown the GPU-inflated DoS attack described in Subsection 3.1, data puzzle to software puzzle as shown in Fig 2. At the server, the software puzzle scheme has a code block warehouse **W** gathering various software instruction blocks. Inside, it contains two modules: generating the puzzle $C0_x$ by randomly assembling code blocks extracted from the warehouse; and obfuscating the puzzle $C0_x$ for high security puzzle $C1_x$.

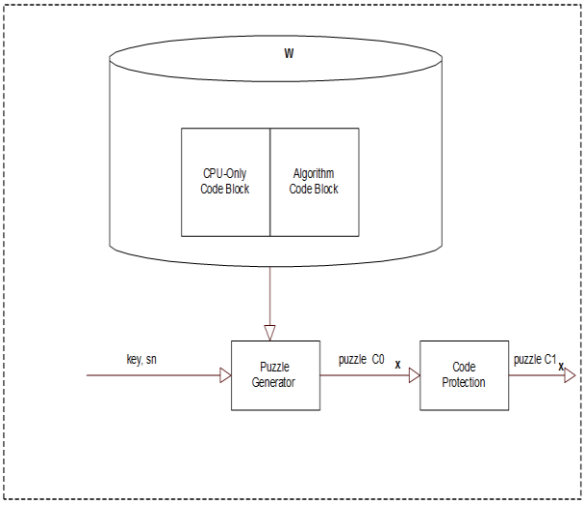

Fig 2: Diagram of software puzzle generated with secret key and nonce sn

3.3 Code Block Warehouse Construction

The code block warehouse **W** has compiled instruction blocks, e.g., in Java byte code, or C binary code. The purpose to store compiled codes instead of source codes is to avoid wasting server's time; otherwise, the server has to take time beyond regulation to compile source codes into compiled codes within the method of software puzzle generation. The necessary requirements for every block are:

- So as to assemble the code blocks along (see section 3.4), each block has well-defined input parameters and output parameters such that the output from one block will be used because the input of the subsequent blocks.
- The size of every code block is set by the safety parameter **κ**. Given that the dimensions of software puzzle is constant, if the block size is smaller, there are more blocks on average specified more puzzles will be created. Thus smaller block size implies higher security level as a result of associate wrongdoer has to pay a lot of effort to figure out a puzzle in question. The shortcoming of tiny block size is that the server has to pay longer in extracting the essential blocks and collecting the extracted blocks into computer software puzzle.

Preferably, the warehouse stores both Java byte code and the corresponding C code. Because the former is applicable to completely different OS platforms however slow, it is suitable to deliver the computer code puzzle to the consumer within the format of Java byte code. As a result, this Java-C hybrid

scheme ensures that the server has advantage over the client/adversary in terms of resource consumption, as well because the support of cross-platform deployment.

3.4 Software Puzzle Generation

In order to construct a software puzzle, the server has to execute three modules: puzzle core generation, puzzle challenge generation, software puzzle encrypting/obfuscating, as shown in Fig. 2.

3.4.1 Puzzle Core Generation

From the code block warehouse, the server first chooses n code blocks based on hash functions and a secret, e.g., the j^{th} instruction block b_{ij}, where **ij = H_1 (y, j),** and **y = H_2 (key, sn)**, with one-way functions $H_1(\cdot)$ and $H_2(\cdot)$, key is the server's secret, and sn is a present or timestamp. All the chosen blocks are assembled into a puzzle core, denoted as **C(·) = (b_{i1}; b_{i2};⋯; b_{in})**.

3.4.2 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 information processing addresses, port numbers and cookies, and produces a challenge **x =C(y, m)**, similar to encrypting plaintext m with key y to produce cipher text **x**. As the attacker doesn't grasp the puzzle core **C(·)** (or equivalently the puzzle operate **P(·)**) earlier, it cannot exploit GPU to unravel the puzzle $C0_x$ in real time mistreatment the fundamental GPU-inflated DoS attack addressed in subdivision 3.1. However, if the puzzle is merely made as on top of, it is possible for Associate in Nursing assaulter to get the GPU kernel by mapping the CPU commands in $C0_x$ to the GPU directions one by one, i.e., to automatically translate the CPU code puzzle $C0_x$ into its functionally equivalent GPU version.

3.4.3 Puzzle Protection

Intuitively, code obfuscation is able to thwart on the top of translation threat to some extent. So there are no generic obfuscation techniques which may stop a consumer and skilled hacker from understanding a program in theory [17], results in [18] show that obfuscation does increase the price of reverse-engineering. Thus, although code obfuscation could be not satisfactory in long code defense against hacking, it is suitable for invigorating code puzzles that demand a protection amount of many seconds solely.

As a popular obfuscation technology, code encryption technology treats code as knowledge string and encrypts each quantity and opCode. Concretely, given the code $C0_x$, the server generates an encrypted puzzle $C1_x$ **= ε(y,** $C0_x$**),** where **ε(·)** is a cipher like AES, and **y** is used because the encryption key. In practice, there are several industrial code obfuscation tools for C/C++ code such as VMprotect which will be wont to defend the code puzzle from hacking. In all, there are two layer encryptions. The outer layer is used to encrypt the code puzzle $C0_x$, and the inner layer uses the puzzle software to write the challenge as knowledge puzzle will. Therefore, after receiving $C1_x$, the client has to attempt $\tilde{y}$. If and only if $\tilde{y}$**= y**, the original software puzzle $C0_x$ is recovered and any wont to solve the challenge.

4. SOFTWARE PUZZLE PACKING

Once a software puzzle $C1_x$ is created at the server aspect and compiled into the Java category file $C1_x$.class, it will be delivered to the consumer requests for services over an insecure channel like web, and run at the client's side. Applet is an appropriate delivery suggests that as a result of it is run in browsers on several platforms like Windows, Mac and UNIX operating system [19], accept not applicable to some mobile browsers until jail breaking the OS like iOS [20]. Usually, Applet is embedded into a hypertext markup language page that is embedded with an archive as well as the software puzzle category $C1_x$.class and a Java category init.class for activating the puzzle package $C1_x$.class.

<APPLET CODE=''init.class''

ARCHIVE = ''init.class, $C1_x$.class''

WIDTH=''200'' HEIGHT=''40''>

</APPLET>

However, not all Applets can be run at the consumer's browser with the default access policy such that the design for software puzzle varies with the browser's configurations at the consumer side. In the following, this paper describes an option for packing software puzzle based on the configuration at the consumer side.

1. Read the $C1_x$.class
2. Repeat
3. Randomly choose a small $\tilde{y}$
4. Decrypt $C1_x$.class with key $\tilde{y}$ into $\widetilde{C0_x}$.class
5. Load class$\widetilde{C0_x}$.class
6. Invoke $C0_x$.class to obtain $\tilde{m}$ and further $\tilde{x}$ = $\widetilde{C0}(\tilde{y},\tilde{m})$
7. Until $\tilde{x}$=x
8. Output ($\tilde{x}$. $\tilde{y}$)

In above init.class structure for reloading puzzles class on JVM. If a correct solution **y** is found, $C0_x$.class shall be the same as the original puzzle $C0_x$.class, where z = x ⊕ y is calculated in advanced and hard-coded into at the server side.

1. Read the $C1_x$.class
2. Load class $C1_x$.class
3. Repeat
4. Randomly choose a small $\tilde{y}$
5. Decrypt $C1_x$.class with key $\tilde{y}$ into $\widetilde{C0_x}$.class
6. Invoke $C0_x$.class to obtain $\tilde{m}$ and further $\tilde{x}$ = $\widetilde{C0}(\tilde{y},\tilde{m})$
7. Until $\tilde{x}$=x
8. Output ($\tilde{x}$. $\tilde{y}$)

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

4.1 Class Reloading in Java Sandbox

The instructions in $C1_x$.class will not be directly run at client's JVM as a result of the software puzzle commands got to be decrypted then replaced with the decrypted one on the fly. However, a Java class will not decision the new command generated by itself. Nonetheless, it is legal in JVM to exchange a complete class by reloading a new/recovered version. To this end, the server will generate another category file init.class as in above snippet for managing the puzzle category $C1_x$.class. At the client aspect, init.class is used to decode $C1_x$.class into a temporary category $\widetilde{C0_x}$.class and reload the category $\widetilde{C0_x}$.class for one answer trial.

5. SECURITY ANALYSIS

Software puzzle aims to avoid GPU from being used in the puzzle-solving process based mostly on totally different instruction sets and real-time environments between GPU and CPU. Conversely, a soul could try to deface the software system puzzle theme by simulating the host on GPU (Subsection 5.1), cracking puzzle algorithm (Subsection 5.2), re-producing GPU-version puzzle (Subsections 5.3 ∼ 5.5), or abusing the access priority in puzzle-solving (Subsection 5.6).

5.1 Employing Host Machine on GPU

If an assaulter is in a position to run a CPU simulator over GPU surroundings, the software puzzle will be run on GPU directly. However, this simulator-based attack may be impractical in fast the puzzle-solving method as a result of the whole hardware resources must be emulated by VM software system, problems will arise if the properties of hardware resources significantly totally different in the host and therefore the visitor [22]. Of course, it is not trivial to develop a full-functional CPU simulator on GPU as a result of the CPU surroundings together with OS, and all the foreign Java libraries (and their imported libraries then on) should be simulated. If only a portion of machine functions is enforced, the GPU kernel may have to communicate with the host for the non-simulated functions. In this case, the GPU-exaggerate function is reduced significantly as a result of it will not run in a very parallel approach and therefore the GPU-CPU line is way slower than its internal memory access; A software system running over a machine is way slower than over its guest environment directly because there are additional process steps to execute the software commands.

5.2 Cracking Data Puzzle Algorithm

According to Section 4, a soul obtains the puzzle answer $(\tilde{x},\tilde{y})$ to the software puzzle $C1_x$, such that $x = \tilde{x} = C0_x(\tilde{y},\tilde{m})$, where range **x** is hard-coded in the software system puzzle and $\tilde{m}$ springs on the fly. Since the software puzzle is encrypted with the commonplace cipher, a soul has to recover the puzzle software system by brute force. Moreover, for the inner-layer encryption, as $C(\cdot)$ is an encoding perform, theoretically, an soul will not find a legitimate answer $(\tilde{x}, \tilde{y})$ in a better approach than brute force provided that y is over a tiny low interval. Hence, the practical strategy of the assaulter is to accelerate the brute force method by exploiting the parallel computation capability of GPU cores.

5.3 Replaying Data Puzzle

When a software system puzzle is designed upon an information/data puzzle, the number of software system puzzles is needed to be terribly giant specified associate wrongdoer is unable to re-construct the GPU-version software system puzzles earlier and re-use them. Indeed, this requirement will be simply satisfied. For instance, even though a service provider merely adds one AES spherical transformations between 2 AES transformations within the customary ten rounds, the number of AES variants is up to 49×4+3 =278. Moreover, a software system will have several polymorphic codes such that the quantity of software puzzles is even larger. Unfortunately, a smart human might collect all the code blocks within the warehouse **W**, and rebuild the GPU version code block warehouse WGPU in advance. Once a new software puzzle is delivered to the human, he will reconstruct the GPU-version puzzle by matching the puzzle code blocks against the software system puzzle. In this case, the adversary is in a position to extend the attack performance. However, as the server encrypts the puzzle software $C0_x$ into $C1_x$, the adversary has to recover $C0_x$ by brute force, and hence will not successful to re-construct the GPU-version puzzle by matching code patterns.

5.4 De-Obfuscating Software Code

In order to rewrite the GPU kernel, a wrongdoer might confirm the instruction flow on the fly by debugging the software system puzzle. Generally, dynamic translation can accelerate the offensive speed, but it is not terribly useful to the GPU-inflated DoS wrongdoer as a result of

- Dynamic translation is sometimes a human-machine interactive method. If human interference is required, the DoS attack is very ineffective;
- In order to hold on the dynamic translation, the attacker desires a simulation atmosphere for "debugging" the software system puzzle. In the translation process, the decryption key $\tilde{y}$ has to be tested by brute force. Because it is not possible to come to a decision whether or not a tested key's right supported the recovered opCode worth owing to the instruction permutation in subdivision 3.4.3, the attacker has to run the puzzle $C0_x$ for each key check to form the choice.
- If the simulation environment is running on host processor, the host cannot generate the GPU kernel till the answer is found. Therefore, this translation time is longer than the time used to directly solve software puzzle by processor host.

Once the translated code has an error, the attacker fails to recover the software system puzzle $C0_x$ to find out the correct response specified, he can't launch DoS attack. Therefore, it is tough for a wrongdoer to develop GPU kernel for determination of the initial software system puzzle by deobfuscating software system puzzle.

5.5 Exploiting Instruction Compliance

Code obfuscation can give sensible security or ad-hoc security by increasing the attacker's effort. In order to supply a theoretical security, cryptographic protection technique shall be used. Nonetheless, the method can't use in a very simple manner. According to Java syntax [23], all the opCode values are at intervals the interval [00,0C9](Hexadecimal) in the Java instructions. Additionally, for some instruction codes opCode, their operands have additional interval restrictions. If the adversary tries to decipher the software system with a trial key $\tilde{y}$ and finds a non-compliant instruction in terms of opCode or opCode-operand combination, the adversary will discard that trial worth $\tilde{y}$ right away such that the puzzle-solving method is accelerated dramatically. To overcome this instruction compliance weakness, the server can adopt the cipher over finite domain [24]. Specifically, the server divides the instruction set into subsets. In each set, all the opCodes are of the same length, and their operands are in the same interval. Then, the server permutes the instructions over the set solely in the code cryptography method or code self-modifying method. If the index of the instruction opCode is permuted, a valid and encrypted instruction is obtained. Therefore, the adversary fails to accelerate puzzle solving by exploiting the instruction compliance.

5.6 Abusing Access Priority

All the consumer puzzle schemes assume that there is no secure channel between the client and also the server till puzzle verification completion. Otherwise, the client puzzle theme is redundant. Thus, a wrongdoer will intercept all the traffic between the consumer and the server machine, and start

man-in-the-middle attack, says, sending malicious software system puzzles to the consumer browser therefore as to launch attacks to the consumers. However, an access policy needs to be defined therefore as to change the software system puzzle to decision some special category generation functions. Hence, the attacker might have additional right to produce new categories to form troubles to the consumers. Luckily, this "flaw" does not extremely incur any new threat to the consumer host. As any new class created from the wrongdoer has the same priority because the original one, i.e., the same as normal category except category generation permission, it cannot access the other additional resources within the host platform. Nonetheless, this class generation permission allows the wrongdoer to run through the memory resource of the native host by making infinite variety of categories. But this memory DoS attack to native host conjointly exists in the "legal" application program that requests for an outsized quantity of memory. Hence, the adversary is unable to incur new threat to the host by abusing the additional priority.

6. EXPERIMENTAL EVALUATION

In the experiment, an Apache-Tomcat Server 7.0.30 is started to response to consumer requests on Dell Precision T3600 (Intel computer hardware E5-1607,3.0GHZ, RAM 8GB) installed with Windows 8.1 64 bit. When a consumer sends a request to the server, a servlet will produce the code puzzle. Microsoft Internet soul, installed with Java VM 1.7.0.67, is run over Dell T3600. Here associate experimental server (servlet) is designed that includes a code block warehouse for CPU-only directions and AES spherical operations (see section 3.3), a module for puzzle generation and a module for instruction-compliant code encryption (see section 5.5). Besides, we conjointly developed associate application program for the code puzzle package delivery.

6.1 Experiment Results

SSL/TLS protocol is the most well-liked on-line transaction protocol, associated an SSL/TLS server performs dearly-won RSA decipherment operation for every consumer affiliation request, thus it is at risk of DoS attack. Main objective is to protect SSL/TLS server with code puzzle against process DoS attacks, particularly GPU-inflated DoS attack. As a complete SSL/TLS protocol includes several rounds, system uses RSA decipherment step to appraise the defense effectiveness in terms of the server's time value for simplicity.

Assume the time to perform one RSA decryption be t_0, and the time to get and verify one software puzzle be t_s (Note that $t_0 > t_s$, otherwise, software puzzle is useless). Suppose the number of attacker's requests be atomic number n_a, and the number of real consumer requests be n_c, the server's computational time needed for replying all the requests is $\tau_1 = (n_a + n_c) \times t_0$ if there is no code puzzle; otherwise, $\tau_2 = (n_a + n_c) \times t_s + n_c \times t_0$ given that the adversary doesn't come back valid solutions to the puzzles. Thus, software puzzle defense is effective if

$$\tau_1 \geq \tau_2, \text{ i.e., } n_a \geq \frac{t_s}{t_0 - t_s} n_c. \quad (1)$$

That is, when the range of malicious requests atomic number n_a is bigger than $\frac{t_s}{t_0 - t_s} n_c$, the genuine consumers pay less time in looking forward to the services. Hence, a good strategy is to initiate the code puzzle defense if the quantity of requests is on the far side a threshold, otherwise, no defense is required as a result of quality of service is satisfactory for all consumers. To demonstrate the effectiveness of software puzzle, let's see the cost of the participants.

6.1.1 Server Cost

If the server-client system adopts software puzzle, the CPU time spent in the server is

1. time t_1 for getting ready the initial puzzle C0x;
2. time t_2 for changing C0x into code puzzle C1x;
3. time t_3 for puzzle package generation;
4. time t_4 for validating the consumer answer.

Thus the server time $t_s = t_1 + t_{12} + t_3 + t_4 \approx t_1 + t_2 + t_3$, where the approximation holds as a result of the puzzle verification time t_4 is terribly little. In this experiments, t_1 = 1.7μs, t_2 = 1.5μs and t_3 = 1.2μs on average, or $t_s \approx t_1 + t_2 + t_3$ = 4.4μs in total. On the other hand, it will take the server t_0 =1476μs for activity one RSA2048 decipherment with OpenSSL package 1.0.1f. Therefore $t_s \leq t_0$. It means that the code puzzle may be a sensible defense. More exactly, according to Eq.(1), if $n_a \geq \frac{t_s n_c}{t_0 - t_s} = \frac{4.4 n_c}{1476 - 4.4} =$ 0.003nc, the software puzzle defense is effective. For example, suppose an SSL server receives n_c = 600 and atomic number n_a =20,000 requests per second, since τ_2 =(20000+600)×4.4+600×1476=976,240μs < 1s, all the genuine shoppers (i.e., 600 clients) can be served if code puzzle is employed, otherwise, only $\frac{1000}{t_0} \times \frac{n_c}{n_c + n_a} \approx$ 19 real consumers on average (or 3.3% of total real clients) will be served per second. Fig. 5 illustrates that the code puzzle will increase the service quality significantly in terms of the proportion of served customers. In the countermeasure, the server has to transfer the software puzzle package (i.e., webpage including the Applet) to the consumer. The package is merely a 12,000 bits on average, hence, the server is able to serve $12 \times 10^9 / 120000 = 10^5$ users presumptuous the network bandwidth is 12Gbps. Indeed, the server capacity will be raised if the puzzle core is built from random and light weight function. Thus, the bandwidth DoS attack threat is little. In other words, the present theme will increase the defense capability against time-DoS attack, without sacrificing the defense capability against space-DoS attack. In order to verify the response $(\tilde{x}, \tilde{y})$, the server has to store the corresponding (x, y) into the storage **S**, which is concerning 128+16=144 bits, or 18 bytes. In order to get rid of long-time open request so on stop memory exhaustion, each result is unbroken for a few times solely, e.g., 1 minute. Thus, given that there are 15,000 requests per second, the storage for the server is merely 18×15,000×60=1.62×107 bytes, or about 16M bytes, which is terribly little for a server.

6.1.2 Client Cost

In order to be served by the server, a client has to solve the code puzzle by trial and error. For each trial, the client has to run the code puzzle. In the experiments, the client takes a pair of seconds to strive solely 2000 keys for finding the solution **y** as a result of a fresh loaded category has got to run the loadClass(), getMethod() and invoke() which are terribly slow in the current JVM. To enable larger search area, the new class is reconstructed with a batch of trial solutions so as to liquidate the re-loading time, e.g., when the new class includes puzzle code for 36 trials, the client is able to take a look at 11,918 keys within a pair of seconds, while the communication value is simply raised half-hour with jar package. To increase the space additional for prime security, JNI programming will be utilized.

6.1.3 Attacker Cost

The assailant has 2 decisions to solve the code puzzle. One is to solve the puzzle as a traditional client will. Obviously, the attacker has no advantage over the traditional consumer

during this case. In other words, the software puzzle achieves its goal. A second choice is that the attacker's host simulates the code puzzle and converts the code puzzle into the GPU version. In this case, GPU can quickly solve the puzzle in parallel, but the conversion method takes nearly the same time because the first alternative. This gives the assailant no incentive to perform the conversion.

6.2 Revere-Engineering Results

Given an encrypted byte code, the output of the well-known disassembler jad 1.5.8g is almost orthogonal to the initial byte code (except the multibyte instruction like loadCalss) though each output directions are valid. Reverse-Engineering confirm that its task is to dis-assemble the protected byte codes, in particular to those byte codes that are created knavishly. Naturally, it is even hard to translate one Java byte code to a GPU kernel.

7. CONCLUSION AND FEATURE WORK

In this paper, software puzzle theme is planned for defeating GPU-inflated DoS attack. It adopts software protection technologies to guarantee challenge knowledge confidentiality associate degreed code security for an acceptable fundamental measure, e.g., 1-2 seconds. Hence, it has different security demand from the standard cipher that demands semi permanent confidentiality solely, and code protection which focuses on semi-permanent strength against the reverse engineering solely. Since the software puzzle might be designed upon a knowledge puzzle, it can be integrated with any existing server-side knowledge puzzle theme, and easily deployed because the current consumer puzzle schemes do. Although this paper focuses on GPU-inflation attack, its idea will be extended to thwart DoS attackers that exploit different inflation resources like Cloud Computing. For example, suppose the server inserts some anti-debugging codes for detecting Cloud platform into software system puzzle, when the puzzle is running, the software puzzle can reject to carry on the puzzle-solving process on Cloud atmosphere specified the Cloud-inflated DoS attack fails. In the present software system puzzle, the server has to spend time in constructing the puzzle. In other words, the present puzzle is generated at the server facet. An open drawback is however to construct the client-side software system puzzle thus on save the server time for higher defense performance. Another work is how to value the impact of code de-obfuscation, which is connected to the technology advance of code obfuscation.

8. ACKNOWLEDGMENTS

We are thankful to faculty of Computer Engineering Department, DYPSOEA, SPPU for their support. The product of this research paper would not be possible without all of them.

9. REFERENCES

[1] J. Larimer. (Oct. 28, 2014). *Pushdo SSL DDoS Attacks*. [Online]. Available: http://www.iss.net/threats/pushdoSSLDDoS.html

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

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

[4] T. J. McNevin, J.-M. Park, and R. Marchany, "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..

[5] R. Shankesi, O. Fatemieh, and C. A. Gunter, "Resource inflation threats to denial of service countermeasures," Dept. Comput. Sci., UIUC, Champaign, IL, USA, Tech. Rep., Oct. 2010. [Online]. Available: http://hdl.handle.net/2142/17372

[6] J. Green, J. Juen, O. Fatemieh, R. Shankesi, D. Jin, and C. A. Gunter, "Reconstructing Hash Reversal based Proof of Work Schemes," in *Proc. 4th USENIX Workshop Large-Scale Exploits Emergent Threats*, 2011.Brown, L. D., Hua, H., and Gao, C. 2003. A widget framework for augmented interaction in SCAPE.

[7] Y. I. Jerschow and M. Mauve, "Non-parallelizable and non-interactive client puzzles from modular square roots," in *Proc. Int. Conf. Availability, Rel. Secur.*, Aug. 2011, pp. 135–142.

[8] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," Dept. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep. T/LCS/TR-684, Feb. 1996. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.5709

[9] W. C. Feng and E. Kaiser, "The case for public work," in *Proc. IEEE Global Internet Symp.*, May 2007, pp. 43–48.

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

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

[12] NVIDIA CUDA. (Apr. 4, 2012). *NVIDIA CUDA C Programming Guide, Version 4.2*. [Online]. Available: http://developer.download.nvidia.com/

[13] X. Wang and M. K. Reiter, "Mitigating bandwidth-exhaustion attacks using congestion puzzles," in *Proc. 11th ACM Conf. Comput. Commun. Secur.*, 2004, pp. 257–267.

[14] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols," in *Proc. IFIP TC6/TC11 Joint Working Conf. Secure Inf. Netw., Commun. Multimedia Secur.*, 1999, pp. 258–272.

[15] D. Kahn, *The Codebreakers: The Story of Secret Writing*, 2nd ed. New York, NY, USA: Scribners, 1996, p. 235.

[16] K. Iwai, N. Nishikawa, and T. Kurokawa, "Acceleration of AES encryption on CUDA GPU," *Int. J. Netw. Comput.*, vol. 2, no. 1, pp. 131–145,2012.

[17] B. Barak *et al.*, "On the (Im)possibility of obfuscating programs," in *Advances in Cryptology* (Lecture Notes in Computer Science), vol. 2139.Berlin, Germany: Springer-Verlag, 2001, pp. 1–18.

[18] H. Y. Tsai, Y. L. Huang, and D. Wagner, "A graph approach to quantitative analysis of control-flow obfuscating transformations," *IEEE Trans. Inf. Forensics Security*, vol. 4, no. 2, pp. 257–267, Jun. 2009.

[19] S. Wang. (Sep. 18, 2011). *How to Create an Applet & C++*. [Online]. Available: http://www.ehow.com/how_12074039 create-Applet-c. html#ixzz24Lsk0OJQ

[20] J. Bailey. (Oct. 28, 2014). *How to Install Java on an iPhone*, eHow Contributor. [Online]. Available: http://www.ehow.com/how_5659673_install-java-iphone .html#ixzz24jIAyKiM

[21] J. Ansel *et al.*, "Language-independent sandboxing of just-in-time compilation and self-modifying code," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2011, pp. 355–366.

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

[23] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1999, ch. 9. [Online]. Available: http://docs.oracle.com/javase/specs/jvms/se5.0/html/VM SpecTOC.doc.html

[24] J. Black and P. Rogaway, "Ciphers with arbitrary finite domains," in *Topics in Cryptology* (Lecture Notes in Computer Science), vol. 2271. Berlin, Germany: Springer-Verlag, 2002, pp. 114–130.