Software Puzzle: A Countermeasure to Resource-Inflated Denial-of-Service Attacks

Yongdong Wu, Zhigang Zhao, Feng Bao, and Robert H. Deng

***Abstract*— Denial-of-service (DoS) and distributed DoS (DDoS) are among the major threats to cyber-security, and client puzzle, which demands a client to perform computationally expensive operations before being granted services from a server, is a well-known countermeasure to them. However, an attacker can inflate its 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 this paper, we study how to prevent DoS/DDoS attackers from inflating their puzzle-solving capabilities. To this end, we introduce a new client puzzle referred to as software puzzle. Unlike the existing client puzzle schemes, which publish their puzzle algorithms in advance, a puzzle algorithm in the present software puzzle scheme is randomly generated only after a client request is received at the server side and the algorithm is generated such that: 1) an attacker is unable to prepare an implementation to solve the puzzle in advance and 2) 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. Moreover, we show how to implement software puzzle in the generic server-browser model.**

***Index Terms*— Software puzzle, code obfuscation, GPU programming, distributed denial of service (DDoS).**

I. Introduction

DENIAL of Service (DoS) attacks and Distributed DoS (DDoS) attacks attempt to deplete an online service's resources such as network bandwidth, memory and computation power by overwhelming 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, e.g., *Pushdo SSL DDoS Attacks* [1].

DoS and DDoS are effective if attackers spend much less resources than the victim server or are much more powerful than normal users. In the example 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 [2]. In this paper, we are particularly interested in the countermeasures to DoS/DDoS attacks on server computation power. 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 [3] 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: puzzle generation,[2] puzzle solving by the client and 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 $\mathcal{P}$ 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 $x = \mathcal{P}(y)$, 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 $t_c \gg t_s$ for normal users, *i.e.*, $\gamma \gg 1$, an attacker can not start DoS attack efficiently by solving many puzzles. Alternatively, the attacker can merely reply to the server with an arbitrary number $\tilde{y}$ 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

Manuscript received July 8, 2014; revised September 9, 2014 and October 24, 2014; accepted October 27, 2014. Date of publication October 30, 2014; date of current version December 17, 2014. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. C.-C. Jay Kuo.

Y. Wu and Z. Zhao are with the Department of Infocomm Security, Institute for Infocomm Research, Agency for Science, Technology and Research, Singapore 138632 (e-mail: wydong@i2r.a-star.edu.sg; zzhao@i2r.a-star.edu.sg).

F. Bao is with the Shield Laboratory, Central Research Institute, Huawei International Pte. Ltd., Singapore 486035 (e-mail: bao.feng@huawei.com).

R. H. Deng is with the School of Information Systems, Singapore Management University, Singapore 188065 (e-mail: robertdeng@smu.edu.sg).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2014.2366293

[1]Note that the DoS attack is different from the "normal" congestion case where a server receives the overwhelming number of requests in peak hours, *e.g.*, some booking systems often crash in the period of festival eve due to sudden increase of ticket purchase requests. The later is usually predictable, but the former is not.

[2]There are two methods to generate client puzzles. One is that the server fully generates the puzzle, while another is the server gives the client partial input, and asks the client to solve for both puzzle input and output. In this paper, we focus on the first one only.

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 [4].

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 880M in Lenovo Thinkpad W510), and even smartphones (*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 [7] and Time-lock puzzle [8]). This parallelism strategy can dramatically reduce the total puzzle-solving time, and hence increase the attack efficiency. Green et al. [6] 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, they proposed to track the individual client behavior through client's IP address [9]. 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.

As the present browsers such as Microsoft Internet Explorer and Firefox do not explicitly support client puzzle schemes, Kaiser and Feng [11] developed a web-based client puzzle scheme which focuses on transparency and backwards compatibility for incremental deployment. The scheme dynamically embeds client-specific challenges in webpages, 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 $\mathcal{P}(\cdot)$ with a native language such as C/C++ such that the cost of an attacker is much smaller than that the server expects.[3] 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 based on client's GPU capability, the GPU-inflation DoS does not work at all. However, we do 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, this paper 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 $\mathcal{P}(\cdot)$ in the form of a software core $\mathcal{C}$ upon receiving a client's request. Specifically, by extending DCG technology which produces machine instructions at runtime [10], the proposed scheme randomly chooses a set of basic functions, assembles them together into the puzzle core $\mathcal{C}$, constructs a software puzzle $\mathcal{C}0_x$ with the puzzle core $\mathcal{C}$ and a random challenge x. If the server aims to defeat high-level attackers who are able to reverse-engineer software, it will obfuscate $\mathcal{C}0_x$ 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 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 can not 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. To demonstrate the applicability of software puzzle, we use Applet to implement software puzzles such that the software puzzle implementation has the same merits as [11] in terms of easy deployment, but overcomes its security weaknesses.

The reminder of this paper is organized as follows. Section II provides an overview of GPU and its difference with CPU. Section III introduces the software puzzle, the countermeasure to GPU-inflated DoS attacks, and Section IV addresses the packing mechanism of software puzzle so that the puzzle can be solved at the client with an appropriate permission. Section V analyzes the security of software puzzle. Section VI evaluates the performance of software puzzle. Section VII draws conclusions and addresses the future work. Finally, an appendix gives an example of randomly generated software puzzle, and its resistance to the hacking attack.

A. Notations

For ease of reference, important notations used throughout the paper are listed below.

x: A challenge chosen by server.

m: A message collected from environment.

[3]In our experiments, a native code is about 20 times faster than a Java bytecode for the same function.

y: A solution to the puzzle challenge x.
$(\tilde{x}, \tilde{y})$: A puzzle response returned from client.
$\mathcal{P}(\cdot)$: Puzzle algorithm such that $x = \mathcal{P}(y, m)$.
$\mathcal{C}$: Puzzle core which is the software implementation of $\mathcal{P}(\cdot)$.
$\mathcal{C}0_x$: Puzzle which embeds the information of x into $\mathcal{C}$.
$\mathcal{C}1_x$: Obfuscated $\mathcal{C}0_x$.

II. 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 our 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.

A. nVidia GPU Overview

In the nVidia architecture, a GPU has many Streaming Multiprocessors (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 programmer to define `C` functions, or kernels. For instance, the client puzzle function $\mathcal{P}(\cdot)$ can be implemented as a GPU kernel. At any one time, a GPU device is dedicated to a single application which may include multiple kernels. When a kernel is loaded into GPU and invoked, it is executed by multiple identical threads in parallel for maximum efficiency.

B. 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. 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 on account of the independence of threads.

[4]Another GPU programming language is OpenCL `C`, the dominant open general-purpose GPU computing language (www.khronos.org/opencl/), which can be used too.

[5]There are multiple-core CPUs in the market. However, in comparison with GPU, the number of cores in a multiple-core CPU is too small. Hence, we omit multiple-core CPU in this paper without loss of generality.

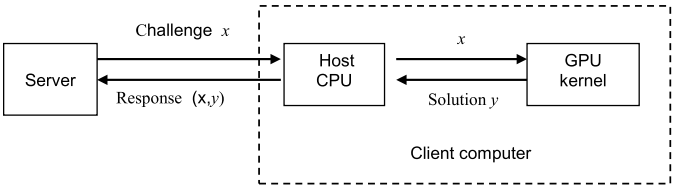

Fig. 1. GPU-inflated DoS attack against data puzzle.

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.

III. Software Puzzle

We classify client puzzles into two types. If a puzzle function $\mathcal{P}$, as all the existing client puzzle schemes (see [13], [14]), is fixed and disclosed in advance, the puzzle is called a data puzzle; otherwise, it is referred to as a software puzzle. Data puzzle aims to enforce the client's computation delay of the inverse function $\mathcal{P}^{-1}(x)$ for a random input x; while software puzzle aims to deter an adversary from understanding/translating the implementation of a random puzzle function $\mathcal{P}(\cdot)$. That is to say, unlike a data puzzle challenge which includes a challenge data only, a software puzzle challenge includes a dynamically generated software $\mathcal{C}(\cdot)$ which including a data puzzle function as a component. Although a software puzzle scheme does not publish the puzzle function in advance, it also follows the Kerckhoffs's Principle [15] because an adversary knows the algorithm for constructing software puzzles, and is able to "reverse-engineer" the software puzzle $\mathcal{C}1_x$ to know the puzzle function $\mathcal{P}(\cdot)$ several minutes later after receiving the software puzzle.

A. Basic GPU-Inflated DoS Attack

In order to elaborate software puzzle, we recap its rival GPU-inflated DoS attack in advance. When a client wants to obtain a service, she sends a request to the server. After receiving the client request, the server responds with a puzzle challenge x. If the client is genuine, she will find the puzzle solution y directly on the host CPU, and send the response (x, y) to the server. However, as shown in Fig. 1, by using the similar mechanism in accelerating calculation with GPU [16], a malicious user who controls the host will send the challenge x to GPU and exploit the GPU resource to accelerate the puzzle-solving process.

B. Framework of Software Puzzle

In order to defeat the GPU-inflated DoS attack described in Subsection III-A, we extend data puzzle to software puzzle as shown in Fig. 2. At the server, the software puzzle scheme has

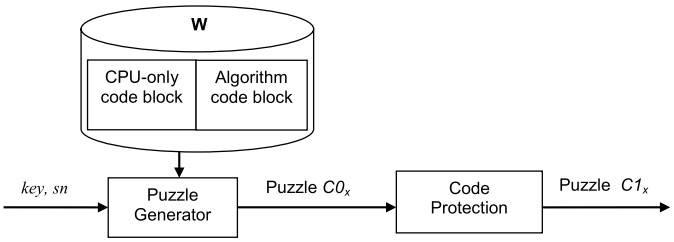

Fig. 2. Diagram of software puzzle generated with secret key and nonce sn.

a code block warehouse **W** storing various software instruction blocks. Besides, it includes two modules: generating the puzzle $\mathcal{C}0_x$ by randomly assemblying code blocks extracted from the warehouse; and obfuscating the puzzle $\mathcal{C}0_x$ for high security puzzle $\mathcal{C}1_x$.

C. Code Block Warehouse Construction

The code block warehouse **W** stores compiled instruction blocks $\{b_i\}$, *e.g.*, in Java bytecode, or C binary code. The purpose to store compiled codes rather than source codes is to save server's time; otherwise, the server has to take extra time to compile source codes into compiled codes in the process of software puzzle generation. The intuitive requirements for each block are

- In order to assemble the code blocks together (see Subsection III-D), each block has well-defined input parameters and output parameters such that the output from one block can be used as the input of the following blocks.
- The size of each code block is decided by the security parameter κ. 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 constructed. Thus smaller block size implies higher security level because an attacker has to spend more effort to figure out a puzzle in question. The shortcoming 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 corresponding 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/adversary in terms of resource consumption, as well as the support of cross-platform deployment. In general, code blocks can be classified into two categories: CPU-only instruction block and data puzzle algorithm block.

1) 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 (e.g., `try-catch-finally`, `goto`) are inherently non-predictable and are non-parallelable, executing them in GPU is slow such that the major merit of GPU can not be exploited by the attacker; Secondly, some hardware-related operations such as reading hardware input and surfing network, can not be performed on GPU; Thirdly, the state-of-the-art GPUs do not support dynamic thread generation; Fourthly, 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.

Therefore, we can exploit the instructions which are different between GPU and CPU as components to design software puzzle. Table I lists some of these instructions.

TABLE I
EXAMPLE CPU-ONLY INSTRUCTIONS

Instruction	Difference exploited
1.Read local cookie	GPU can not direcly read CPU storage
2.Allocate large memory	GPU has much smaller memory than CPU
3.Try-catch	GPU does not support except handling
4.Goto (address)	GPU does not support branch
5.Network interface	GPU cann't support networking function
6.Human-machine interface	GPU cann't support
7.Create new class	GPU does not support dynamic code
8.Create new thread	GPU does not support child thread

2) Data Puzzle Algorithm Block: Similar to the blocks in data puzzle, algorithm blocks perform the mathematical operations only. For example, in an AES round, *ShiftRows* code block outputs a transformed message matrix (or state), 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: puzzle core generation, puzzle challenge generation, software puzzle encrypting/obfuscating, as shown in Fig. 2.

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 jth instruction block b_{i_j}, where $i_j = \mathcal{H}_1(y, j)$, and $y = \mathcal{H}_2(key, sn)$, with one-way functions $\mathcal{H}_1(\cdot)$ and $\mathcal{H}_2(\cdot)$, key is the server's secret, and sn is a nonce or timestamp. All the chosen blocks are assembled into a puzzle core, denoted as $\mathcal{C}(\cdot) = (b_{i_1}; b_{i_2}; \cdots; b_{i_n})$. As an illustrative example, Table III in the appendix shows an example puzzle core $\mathcal{C}$ generated from AES operation blocks stored in warehouse **S**.

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 IP addresses, port numbers and cookies, and produces a challenge $x = \mathcal{C}(y, m)$, smiliar to encrypting plaintext m with key y to produce ciphertext x.

As the attacker does not know the puzzle core $\mathcal{C}(\cdot)$ (or equivalently the puzzle function $\mathcal{P}(\cdot)$) in advance, it can not exploit GPU to solve the puzzle $\mathcal{C}0_x$ in real time using the basic GPU-inflated DoS attack addressed in Subsection III-A. Nonetheless, if the puzzle is merely constructed as above,

it is possible for an attacker to generate the GPU kernel by mapping the CPU instructions in $\mathcal{C}0_x$ to the GPU instructions one by one, i.e., to automatically translate the CPU software puzzle $\mathcal{C}0_x$ into its functionally equivalent GPU version.

3) Code Protection: Intuitively, code obfuscation is able to thwart the above translation threat to some extent. Though there are no generic obfuscation techniques which can prevent a patient and advanced hacker from understanding a program in theory [17], results in [18] show that obfuscation does increase the cost of reverse-engineering. Thus, although code obfuscation may be not 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 instructus, and each instruction has a form (*opCode*, [*operands*]), where *opCode* indicates which operation (*e.g.*, *addition, shift, jump*) is, while the *operands*, varying with *opCode*, are the parameters (*e.g.*, 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 $\mathcal{C}0_x$, the server generates an encrypted puzzle $\mathcal{C}1_x = \mathcal{E}(y, \mathcal{C}0_x)$, where $\mathcal{E}(\cdot)$ is a cipher such as AES, and y is used as the encryption key. In practice, there are many commercial code obfuscation tool for C/C++ software such as VMprotect (*http://vmpsoft.com/*) which can be used to protect the software puzzle from hacking.

In all, there are two-layer encryptions. The outer layer is used to encrypt the software puzzle $\mathcal{C}0_x$, and the inner layer uses the puzzle software to encrypt the challenge as data puzzle does. Therefore, after receiving $\mathcal{C}1_x$, the client has to try $\tilde{y}$. If and only if $\tilde{y} = y$, the original software puzzle $\mathcal{C}0_x$ can be recovered and further used to solve the challenge.

IV. Software Puzzle Packing

Once a software puzzle $\mathcal{C}1_x$ is created at the server side and compiled into the Java class file $C1x.class$, it will be delivered to the client who requests for services over an insecure channel such as Internet, and run at the client's side. Applet is a suitable delivery means because it can be run in browsers on many platforms such as Windows, Unix, Mac and Linux [19], despite not applicable to some mobile browsers without jailbreaking the operating system such as iOS [20].

Usually, an Applet is embedded into an HTML page which is embedded with an archive including the software puzzle class $C1x.class$ and a Java class *init.class* for activating the puzzle software $C1x.class$

```
1: <APPLET CODE=''init.class''
          ARCHIVE = ''init.class, C1x.class''
          WIDTH=''200'' HEIGHT=''40''>
2: </APPLET>
```

However, not all Applets can be run at the client's browser with the default access policy such that the design for software puzzle varies with the browser's configurations at the client side. In the following, we describe two options for packing software puzzle based on the configuration at the client side.

```
1: Read the C1x.class
2: Repeat
3:  Randomly choose a small ỹ
4:  Decrypt C1x.class with key ỹ into
      class C0x.class
5:  Load class C0x.class
6:  Invoke C0x.class to obtain m̃
      and further x̃ = C0(ỹ, m̃)
7: until x̃ = x
8: Output (x̃, ỹ)
```

Fig. 3. *init.class* structure for reloading puzzle class on JVM. If a correct solution y is found, $\widetilde{C0x.class}$ shall be the same as the original puzzle $C0x.class$, where $z = x \oplus y$ is calculated in advanced and hard-coded into at the server side.

```
1: Read the C1x.class
2:  Load class C1x.class
3: Repeat
4:  Randomly choose a small ỹ
5:  Decrypt C1x.class with key ỹ into
      class C0x.class
6:  Invoke C0x.class to obtain m̃
      and further x̃ = C0(ỹ, m̃)
7: until x̃ = x
8: Output (x̃, ỹ)
```

Fig. 4. *init.class* structure for activating puzzle class on dedicated sandbox.

A. Class Reloading in Java Sandbox

The instructions in $C1x.class$ can not be directly executed at client's JVM because the software puzzle instructions have to be decryped and then replaced with the decrypted one on the fly. However, a Java class can not call the new instructions generated by itself. Nonetheless, it is legal in JVM to replace an entire class by reloading a new/recovered version. To this end, the server will generate another class file $init.class$ as in Fig. 3 for managing the puzzle class $C1x.class$. At the client side, $init.class$ is used to decrypt $C1x.class$ into a temporary class $\widetilde{C0x.class}$ and reload the class $\widetilde{C0x.class}$ for one solution trial.

B. JNI in Dedicated Sandbox

Java Native Interface (JNI) provides Java programs easy access to native shared libraries with native language such as C/C++. In comparison with the reloadable method in Subsection IV-A, the software puzzle implemented with native codes can achieve better obfuscation performance. However, JNI programming requires a dedicated execution client platform.

As the dedicated sandbox is able to run the puzzle directly, without re-loading the class, the structure of $init.class$ can be simplified as Fig. 4, where the puzzle software can be executed directly after it is loaded. In comparison with Fig. 3, this scheme loads $C1x.class$ only once and does not load the decrypted $\widetilde{C0x.class}$. Its weakness is that a dedicated client

platform mechanism (see [21]) shall be deployed for safely and efficiently sandboxing software.

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 (Subsection V-A), cracking puzzle algorithm (Subsection V-B), re-producing GPU-version puzzle (Subsections V-C $\sim$ V-E), or abusing the access priority in puzzle-solving (Subsection V-F).

A. 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 may be impractical in accelerating the puzzle-solving process because

- "*VM software must emulate the entire hardware environment. $\cdots$, problems can arise if the properties of hardware resources are significantly different in the host and the guest*" [22]. To our best 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 can not 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.

B. Cracking Data Puzzle Algorithm

According to Fig. 3 or Fig. 4, an adversary obtains the puzzle solution $(\tilde{x}, \tilde{y})$ to the software puzzle $\mathcal{C}1_x$, such that $x = \tilde{x} = \widetilde{\mathcal{C}0}_x(\tilde{y}, \tilde{m})$, where number x is hard-coded in the software puzzle and $\tilde{m}$ 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 $\mathcal{C}(\cdot)$ is an encryption function, theoretically, an adversary can not find a valid solution $(\tilde{x}, \tilde{y})$ 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.

Remark: 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, this problem can be easily overcome, as shown in the Table II, 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.[6]

TABLE II
Instruction Subset for Branching

Index	Mnemonic	*opCode* (in Hex)	Target address
0	ifeq	99	2 branch bytes
1	ifne	9a	2 branch bytes
2	iflt	9b	2 branch bytes
3	ifge	9c	2 branch bytes
4	ifgt	9d	2 branch bytes
5	ifle	9e	2 branch bytes
6	if_icmpeq	9f	2 branch bytes
7	if_icmpne	a0	2 branch bytes
8	if_icmplt	a1	2 branch bytes
9	if_icmpge	a2	2 branch bytes
10	if_icmpgt	a3	2 branch bytes
11	if_icmple	a4	2 branch bytes
12	if_acmpeq	a5	2 branch bytes
13	if_acmpne	a6	2 branch bytes
14	goto	a7	2 branch bytes
15	jsr	a8	2 branch bytes
16	ifnull	c6	2 branch bytes
17	ifnonnull	c7	2 branch bytes

C. 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 $4^{9\times4+3} = 2^{78}$. 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 $\mathbf{W}$, and rebulid the GPU version code block warehouse $\mathbf{W}_{gpu}$ 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 $\mathcal{C}0_x$ into $\mathcal{C}1_x$, the adversary has to recover $\mathcal{C}0_x$ by brute force, and hence can not successfully re-construct the GPU-version puzzle by matching code patterns.

D. Deobfuscating Software Code

In order to rewrite the GPU kernel, an attacker may determine the instruction flow on the fly by debugging the software puzzle. Generally, dynamic translation can accelerate the attacking speed, but it is not very helpful to the GPU-inflated DoS attacker because

- Dynamic translation is usually a human-machine interactive process. If human interference is required, the DoS attack is very ineffective;

[6]The iteration in the key expansion process shall be adapted to meet the required number of *AddRoundKey* transformation.

- In order to carry on the dynamic translation, the attacker needs a simulation environment for "debugging" the software puzzle. In the translation process, the decryption key $\tilde{y}$ has to be tested by brute force. Because it is impossible to decide whether a tested key is right based on the recovered *opCode* value due to the instruction permutation in Subsection III-D.3, the attacker has to run the puzzle $\widetilde{\mathcal{C}0}_x$ for every key test to make the decision.
  - If the simulation environment is run on CPU host, the host can not generate the GPU kernel until the solution is found. Therefore, this translation time is longer than the time used to directly solve software puzzle by CPU host. In other words, the GPU is useless for accelerating puzzle-solving in this case.
  - If the simulator is run on GPU, the attacker has to face the troubles stated in Subsection V-A besides the trouble existing in the above CPU simulation environment.

Once the translated code has one error, the attacker fails to recover the software puzzle $\mathcal{C}0_x$ to find the correct response such that he can not launch DoS attack. Therefore, it is not easy for an attacker to develop a GPU kernel for solving the original software puzzle by deobfuscating/analyzing software puzzle.

E. Exploiting Instruction Compliance

Code obfuscation can provide practical security or ad-hoc security by increasing the attacker's effort. In order to offer a therotical security, cryptographic protection method shall be used. Nonetheless, the method can not be employed in a straightforward way. According to Java syntax [23], all the *opCode* values are within the interval [00, 0*C*9] (Hexadecimal) in the Java instructions. Additionally, for some instruction codes *opCode*, their *oprand*s have additional interval restrictions. If the adversary tries to decrypt the software with a trial key $\tilde{y}$ and finds a non-compliant instruction in terms of *opCode* or *opCode-oprand* combination, the adversary can discard that trial value $\tilde{y}$ immediately such that the puzzle-solving process 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 subset, all the *opCode*s are of the same length, and their *operand*s are in the same interval. Then, the server permutes the instructions over the subset only in the code encryption process or code self-modifying process. For example, Table II is an instruction subset for branching,[7] where each instruction has one *opCode* and two bytes for target address. If the index of the instruction *opCode* is permuted, a valid and encrypted instruction is obtained. Generally, given an ordered t-entry instruction subset $\mathbf{O} = \{o_0, o_1, \cdots, o_{t-1}\}$ according to *opCode* values, to encrypt the ith instruction *opCode* o_i, the server calculates $i' = i + s_j \mod t$ with a key sequence element s_j, and replaces the instruction o_i with $o_{i'}$. At the client side, the decryption step is $i = i' - \tilde{s}_j \mod t$ and hence the decrypted instruction is always valid no matter what the key is. Therefore, the adversary fails to accelerate puzzle-solving by exploiting the instruction compliance.

[7]Besides the subset for branch instructions, single-byte Java instructions can be used to form another subset which has 145 elements.

F. Abusing Access Priority

All the client puzzle schemes assume that there is no secure channel between the client and the server until puzzle verification completion. Otherwise, the client puzzle scheme is redundant. Thus, an attacker can intercept all the traffic between the client and the server, and start man-in-the-middle attack, say, sending malicious software puzzles to the client browser so as to launch attacks to the clients. However, an access policy should be defined so as to enable the software puzzle to call some special class generation functions. Hence, the attacker may have extra right to create new classes to make troubles to the clients.

Luckily, this "flaw" does not really incur any new threat to the client host. As any new class created from the attacker has the same priority as the original one, *i.e.*, the same as normal class except class generation permission, it can not access any other extra resources in the host platform. Nonetheless, this class generation permission enables the attacker to deplete the memory resource of the local host by creating infinite number of classes. But this memory DoS attack to local host also exists in the "legal" Applet which requests for a large amount of memory. Hence, the adversary is unable to incur new threat to the host by abusing the extra priority.

VI. Experimental Evaluation

In the experiment, an Apache-Tomcat Server 7.0.30 is started to response to client requests on Dell Precision T3600 (Intel Xeon CPU E5-1607, 3.0GHZ, RAM 8GB) installed with Ubuntu 14.04.1 LTS 64 bit. When a client sends a request to the server, a servlet will create the software puzzle. Microsoft Internet Explorer, installed with Java VM 1.7.0_67, is run over Dell T3600.

We built an experimental server (servlet) which includes a codeblock warehouse for CPU-only instructions and AES round operations (see Subsection III-C), a module for puzzle generation and a module for instruction-compliant code encryption (see Subsection V-E). Besides, we also developed an applet for the software puzzle package delivery (see appendix for an example puzzle).

A. Experiment Results

SSL/TLS protocol is the most popular on-line transaction protocol, and an SSL/TLS server performs an expensive RSA decryption operation for each client connection request, thus it is vulnerable to DoS attack. Our objective is to protect SSL/TLS server with software puzzle against computational DoS attacks, particularly GPU-inflated DoS attack. As a complete SSL/TLS protocol include many rounds, we use RSA decryption step to evaluate the defense effectiveness in terms of the server's time cost for simplicity.[8]

[8]In fact, RSA decryption time is only a portion of server response time in SSL/TLS protocol. As a conservative estimation method, only RSA decryption time is used in evaluating the performance of software puzzle in this Section.

Assume the time to perform one RSA decryption be t_0, and the time to generate 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 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

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

That is, when the number of malicious requests n_a is greater than $\frac{t_s}{t_0 - t_s} n_c$, the genuine clients spend less time in waiting for the services. Hence, a good strategy is to initiate the software puzzle defense if the number of requests is beyond a threshold, otherwise, no defense is required because quality of service is satisfactory for all clients. To demonstrate the effectiveness of software puzzle, let's compare the cost of the participants.

1) Server Cost: If the server-client system adopts software puzzle, the CPU time spent in the server is

- time t_1 for preparing the initial puzzle $\mathcal{C}0_x$;
- time t_2 for converting $\mathcal{C}0_x$ into software puzzle $\mathcal{C}1_x$;
- time t_3 for puzzle package generation;
- time t_4 for verifying the client answer.

Thus the server time $t_s = t_1 + t_2 + t_3 + t_4 \approx t_1 + t_2 + t_3$, where the approximation holds because the puzzle verification time t_4 is very small. In our experiments, $t_1 = 1.7\mu s$, $t_2 = 1.5\mu s$ and $t_3 = 1.2\mu s$ on average, or $t_s \approx t_1 + t_2 + t_3 = 4.4\mu s$ in total. On the other hand, it will take the server $t_0 = 1476\mu s$ for performing one RSA2048 decryption with `OpenSSL` package 1.0.1f. Therefore $t_s \ll t_0$. It means that the software puzzle is a practical defense. More precisely, according to Eq.(1), if $n_a \geq \frac{t_s n_c}{t_0 - t_s} = \frac{4.4 n_c}{1476 - 4.4} = 0.003 n_c$, the software puzzle defense is effective. For example, suppose an SSL server receives $n_c = 600$ and $n_a = 20,000$ requests per second, since $\tau_2 = (20000 + 600) \times 4.4 + 600 \times 1476 = 976,240\mu s < 1s$, all the genuine clients (i.e., 600 clients) can be served if software puzzle is used, otherwise, only $\frac{1000}{t_0} \times \frac{n_c}{n_c + n_a} \approx 19$ genuine clients on average (or 3.3% of total genuine clients) can be served per second. Fig. 5 illustrates that the software puzzle can increase the service quality significantly in terms of the percentage of served customers.

In the countermeasure, the server has to send the software puzzle package (i.e., webpage including the Applet) to the client. The package is merely 120,000 bits on average, hence, the server is able to serve $12 \times 10^9 / 120000 = 10^5$ users assuming the network bandwidth is 12Gbps. Indeed, the servce capacity can be increased if the puzzle core is constructed from random and lightweight function. Thus, the bandwidth DoS attack threat is small. In other words, the present scheme can 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 $\mathbf{S}$, which is about $128 + 16 = 144$ bits, or 18 bytes. In order to remove a long-time open request so as to prevent memory exhaustion,

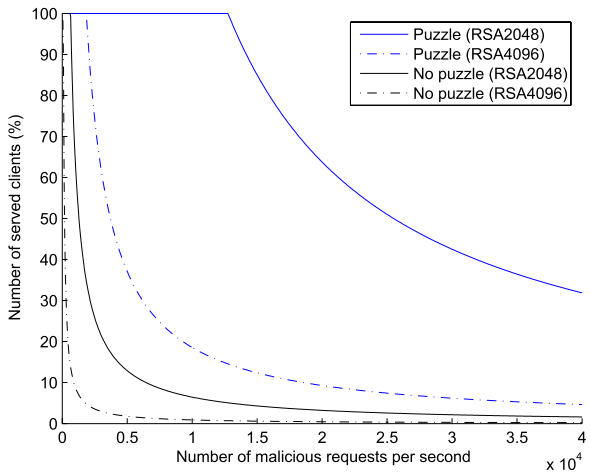

Fig. 5. Service capability comparison of server with/without software puzzle for RSA2048/RSA4096, assume that the request rate of the attacker is 20 times of that of honest clients.

each result is kept for some time only, *e.g.*, 1 minute. Thus, given that there are 15,000 request per second, the storage for the server is merely $18 \times 15,000 \times 60 = 1.62 \times 10^7$ bytes, or about 16M bytes, which is very small for a server.

2) Client Cost: In order to be served by the server, a client has to solve the software puzzle by trial and error. For each trial, the client has to run the software puzzle. In the experiments, the client takes 2 seconds to try only 2,000 keys for finding the solution y because a newly loaded class has to run the `loadClass()`, `getMethod()` and `invoke()` which are very slow in the present JVM. To enable bigger search space, the new class is reconstructed with a batch of trial solutions so as to amortize the re-loading time, e.g., when the new class includes puzzle code for 36 trials, the client is able to test 11,918 keys within 2 seconds, while the communication cost is merely increased 30% with jar package. To increase the space further for high security, JNI programming (See Subsection IV-B) can be employed.

3) Attacker Cost: The attacker has two choices to solve the software puzzle. One is to solve the puzzle as a normal client does. Obviously, the attacker has no advantage over the normal client in this case. In other words, the software puzzle achieves its goal.

A second choice is that the attacker's host simulates the software puzzle and converts the software puzzle into the GPU version. In this case, GPU can quickly solve the puzzle in parallel, but the conversion process takes almost the same time as the first choice. This gives the attacker no incentive to perform the conversion.

B. Reverse-Engineering Result

Given an encrypted bytecode, the output of the well-known disassembler `jad` 1.5.8g (http://www.varaneckas.com/jad/) is almost orthogonal to the original bytecode (except the multi-byte instruction such as `loadCalss`) although every output instructions are valid. Thus, we can confirm that it is not easy to dis-assemble the protected bytecodes, in particular to those bytecodes which are created craftily. Naturally, it is even hard to translate one Java bytecode to a GPU kernel.

VII. Conclusions and Future Work

In this paper, software puzzle scheme is proposed for defeating GPU-inflated DoS attack. It adopts software protection technologies to ensure challenge data confidentiality and code security for an appropriate time period, *e.g.*, 1-2 seconds. Hence, it has different security requirement from the conventional cipher which demands long-term confidentiality only, and code protection which focuses on long-term robustness against reverse-engineering 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 paper 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 present software puzzle, the server has to spend time in constructing the puzzle. In other words, the present puzzle is generated at the server side. An open problem is how to construct the client-side software puzzle so as to save the server time for better defense performance. Another work is how to evaluate the effect of code de-obfuscation, which is related to the technology advance of code obfuscation.

Appendix

This appendix introduces an example software puzzle, and its webpage design so as to have a concrete view on the software puzzle.In order to run the software puzzle, the client must install JVM, otherwise, the response is browser-dependent.

A. Example Software Puzzle

Based on the puzzle generation process elaborated in Subsection III-D, the example software puzzle core includes two modules (see example in Table II). One module is method `Msg_generator()` for constructing message m with CPU-only instructions.

Another module is to construct the data puzzle from algorithm code blocks and CPU-only code blocks. For example, if the AES round operations are used as algorithm code blocks, the server can randomly add more AES round operations (e.g., step 1a in module 2 of Table III), or even insert new operations (e.g., step 3a in module 2 of Table III) into an AES round.

B. Webpage

To deliver the software puzzle, the webpage is embedded with an archive including files for *init.class* and $C1x.class$.

TABLE III
Example Puzzle Core $\mathcal{P}(\cdot)$

1	Msg_generator()	01: n= read_a_local_cookie();
		02: try {
		03: buf= new bytes[$(100000+n)$]
		04: cls=ClassLoader(ClassName);
		05: method = cls.getMethod("nextInt");
		06: m = method.invoke(inData);
		07: }
		08: catch (exception (e)) {
		09: $m = m$+return_from_new_thread(e)
		10: }

2	Round 1	1: SubBytes 1a: **MixColumns** 2: ShiftRows 3: MixColumns 3a:**Add coded pseudo-random number** 4: AddRoundKey
	⋮	⋮
	Round 10	...

The $C1x.class$ comprises the encrypted Java class for the software puzzle in Table II with code protection elaborated in Subsection III-D.3, while the class *init.class* is similar to Fig. 4. An example webpage is as follows.

```
1: <HTML>
2: <BODY>
3: <H1>Software puzzle</H1>
4: <APPLET CODE=''init.class''
      ARCHIVE = ''init.class, C1x.class''
        WIDTH = ''200'' HEIGHT=''40''>
5:  Software puzzle is running ···
6: </APPLET>
7: </BODY>
8: </HTML>
```

C. Permission Grant

As the reloadable method has to access local host resource, it violates the default policy of Java VM, thus an authorization from the user is required. Specifically, as the example software puzzle needs to create and load a new class, the user shall assign class creation permission for the puzzle, otherwise, the new class can not be executed in JVM. To this end, the authorization file *diskname:\users\username\.java.policy* (for Windows 7) should be modified with tool *policytool.exe* by the user as

```
grant CODEBASE "www.server.com" {
  permission java.lang.RuntimePermission
"createClassLoader";
}
```

so as to enable the software puzzle sent from `www.server.com` to create a new loadable Java class $\widetilde{C0x.class}$ in Fig. 3.

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.

[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. MIT/LCS/TR-684, Feb. 1996. [Online]. 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/VMSpecTOC.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.

Yongdong Wu received the B.A. and M.S. degrees from Beihang University, Beijing, China, the Ph.D. degree from the Institute of Automation, Chinese Academy of Sciences, Beijing, and the master's degree in management of technology from the National University of Singapore, Singapore. He is currently a Senior Scientist with the Department of Infocomm Security, Institute of Infocomm Research, Agency for Science, Technology and Research, Singapore. He is also an Adjunct Associate Professor with Singapore Management University, Singapore. His research interests include multimedia security, e-Business, digital right management, and network security. He has authored over 100 papers, and holds seven patents. His research results and proposals were incorporated in the ISO/IEC JPEG 2000 security standard 15444-8 in 2007. He was a recipient of the Best Paper Award at the 13th Joint IFIP TC6 and TC11 Conference on Communications and Multimedia Security (2012).

Zhigang Zhao received the B.A. degree from the University of Science and Technology Beijing, Beijing, China, and the M.S. degree from the Institute of Software, Chinese Academy of Sciences, Beijing. He is currently a Senior Research Engineer with the Department of Infocomm Security, Institute of Infocomm Research, Agency for Science, Technology and Research, Singapore. His research interests include digital right management, software protection, network security, and multimedia security.

Feng Bao received the B.S. degree in mathematics and the M.S. degree in computer science from Peking University, Beijing, China, and the Ph.D. degree in computer science from Gunma University, Maebashi, Japan. He was a Researcher with the Chinese Academy of Sciences, Beijing, and a Visiting Scientist with the University of Hamburg, Hamburg, Germany. From 1996 to 2012, he was with the Institute of Infocomm Research, Agency for Science, Technology and Research, Singapore, and was the Principal Scientist and Head of the Department of Cryptography and Security. He is currently the Director of the Security Laboratory with Huawei International Pte. Ltd., Singapore. His research interests are mainly in cryptography and information security. He has authored over 200 papers in international conferences and journals, which have over 5000 citations. He holds 16 patents and has been involved in the management of dozens of industry projects and international collaborations. He is a member of the Asiacrypt Steering Committee and an Editorial Member of two international journals. He has chaired over 20 international conferences in security.

Robert H. Deng received the B.Eng. degree from the National University of Defense Technology, Changsha, China, in 1981, and the M.Sc. and Ph.D. degrees from the Illinois Institute of Technology, Chicago, IL, USA, in 1983 and 1985, respectively. He has been a Professor with the School of Information Systems, Singapore Management University, Singapore, since 2004. He was the Principal Scientist and Manager of the Department of Infocomm Security at the Institute of Infocomm Research, Agency for Science, Technology and Research, Singapore. His research interests include data security and privacy, multimedia security, and network and system security. He is the Cochair of the Steering Committee of the ACM Symposium on Information, Computer and Communications Security. He was a recipient of the University Outstanding Researcher Award from the National University of Singapore, Singapore, in 1999, and the Lee Kuan Yew Fellow Award for Research Excellence from the Singapore Management University in 2006. He was named as Community Service Star and Showcased Senior Information Security Professional by $(ISC)^2$ under its Asia-Pacific Information Security Leadership Achievements Program in 2010. He received the Distinguished Paper Award of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012) and the Best Paper Award of CMS 2012.