PUFSec: Device Fingerprint-based Security Architecture for Internet of Things

So-Yeon Park[†], Sunil Lim[§], Dahee Jeong[†], Jungjin Lee[§], Joon-Sung Yang[§], and HyungJune Lee[†]
[†]Ewha Womans University, South Korea
[§]Sungkyunkwan University, South Korea
Email: hyungjune.lee@ewha.ac.kr

***Abstract*—A low-end embedded platform for Internet of Things (IoT) often suffers from a critical trade-off dilemma between security enhancement and computation overhead. We propose *PUFSec*, a new device fingerprint-based security architecture for IoT devices. By leveraging intrinsic hardware characteristics, we aim to design a computationally lightweight security software system architecture so that complex cryptography computation can dramatically be prohibited. We exploit the innovative idea of Public Physical Unclonable Functions (PPUFs) that fundamentally protects attackers from recovering the secret key from public gate delay information. We implement its hardware logic in a real-world FPGA board. On top of the PPUF fingerprint hardware, we present an adaptive security control mechanism consisting of adaptive key generation and key exchange protocol, which adjusts security strength depending on system load dynamics. We demonstrate that our PPUF FPGA implementation embeds distinctive variability enough to distinguish between two different PPUFs with high fidelity. We validate our *PUFSec* architecture by implementing necessary algorithms and protocols in a real-world IoT platform, and performing empirical evaluation in terms of computation and memory usages, proving its practical feasibility.**

I. Introduction

Embedded systems for the Internet of Things (IoT) are proliferated to broad areas thanks to the recent open-source hardware technology. Wearable devices such as smart wristband and health kit have widely been used to monitor health, physical training, and medical status. Although the innovative devices are expected to create disruptive services, protecting privacy and security issues on personal information is a key to the success of Internet of Things where 20 billion IoT devices are predicted to be connected each other by 2020.

To tackle the essential security problem, there has been much research effort on devising lightweight yet practically feasible software-based authentication mechanisms for low-end hardware platforms. Due to their hardware resource constraints of the low-end CPU running at lower clock speeds, smaller memory and storage sizes, and limited power supply, computationally less intensive cryptography algorithms have been proposed in both symmetric-key and asymmetric-key cryptography in the sensor network research community. LEAP [13], SPINS [8], and TinySec [3] based on symmetric-key cryptography that takes less computation cost are proven to be feasible in small embedded sensor platforms, while resulting in a large communication overhead for key distribution in return. As for asymmetric-key cryptography, TinyECC [5] and TinyPBC [7] have been proposed to overcome the expensive asymmetric key computation cost by reducing key size.

Despite their substantial contributions to embedded security systems, previous works are not free from a critical dilemma of the performance trade-off issue between security level and computation overhead in delay and power consumption. This is due to the fact that those algorithms are designed purely in the software base, out of consideration with hardware and software co-optimization under the inherent resource-constrained environment. Furthermore, the purely software-based cryptography approach has demonstrated a critical vulnerability under hardware cloning. Accordingly, a hardware-software co-design methodology is a necessary approach to address the fundamental problem and design an efficient and reliable embedded security system.

Recently, a new methodology of exploiting characteristic hardware fingerprints as the uniqueness of a single device for secure authentication has been introduced in the integrated circuit (IC) hardware research community, e.g., *Physical Unclonable Functions (PUFs)* [2], [10], [12]. Uncontrollable process variation during the manufacturing process leads to sensitive operation discrepancy in a certain situation even for identically designed IC chips. This innovative approach opens a whole new opportunity to design a highly resilient security system, which is clone-proof, ultrafast, and power-efficient, against side-channel and various physical attacks.

In this paper, we present a novel security system architecture that exploits a hardware fingerprint approach based on PUF, suitable for small IoT platforms. By borrowing the innovative idea of Public Physical Unclonable Functions (PPUFs) enabling public-key security, a variation of PUF [1], [4], [6], [9] from the IC research, we implement a PPUF-based practical security control mechanism with adaptive key generation in a real-world IoT platform. To address the inherent computation overhead issue in embedded security, we dynamically control security strength by altering the key size to generate, depending on the current load status of the system.

To the best of our knowledge, this paper is the first to implement the PPUF security primitive in a real-world system based on hardware-software co-optimization and measure empirical performance in both computation and storage, along with theoretical security analysis. Our main contributions can be summarized as:

- We implement the real-world PPUF security hardware

logic in FPGA, employing a simple counter-based delay measurement scheme.

- We leverage the trade-off relationship between security level and computation overhead to design a load-aware key generation scheme compatible with PPUF, which adapts security strength depending on the available system resources.
- We empirically quantify computation running time and memory footprint for running our PPUF-based system in a real-world embedded platform.

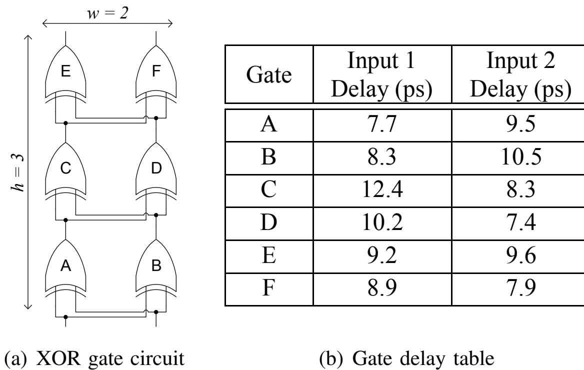
The remainder of this paper is organized as follows: After reviewing background on PUF in Sec. II, we describe our high-level system architecture in Sec. III. We present our PPUF-based fingerprint hardware in Sec. IV and load-aware key generation and exchange schemes in Sec. V. After demonstrating real-world validation of our system architecture in Sec. VI, we conclude this paper in Sec. VII.

II. Background on PUF

Security keys for cryptography are traditionally embedded on-chip by fuse or EEPROM. However, they can be hampered by a number of reverse engineering attacks. To overcome the vulnerability issue, PUF [2], [12] that exploits random process variations in manufacturing and provides uniqueness is proposed. Each manufactured chip can utilize the inherent hardware fingerprint feature for improving security.

In PUF, instead of relying on secret keys, cryptographic public keys are extracted from PUF's element characteristics. To extract those keys, a certain input (referred as *challenge*) is applied to PUF, and its corresponding output (referred as *response*) is generated from the PUF. Given the same challenge, each chip has a unique response originated from manufacturing variations. Based on this distinctive feature, we can utilize PUFs for security such as key generation and authentication [10].

Recently, PPUF [1], [9], [11], which is a variation of PUF, is introduced such that public-key cryptography is enabled by using PPUF characteristics as public information. The PPUF exploits an execution-simulation gap (ESG). PPUF *simulation* at an attacker side takes an extremely longer time than PPUF *execution* at the legitimate receiver who physically possesses its authentic PPUF circuit.



Gate	Input 1 Delay (ps)	Input 2 Delay (ps)
A	7.7	9.5
B	8.3	10.5
C	12.4	8.3
D	10.2	7.4
E	9.2	9.6
F	8.9	7.9

(a) XOR gate circuit (b) Gate delay table

Fig. 1. PPUF example

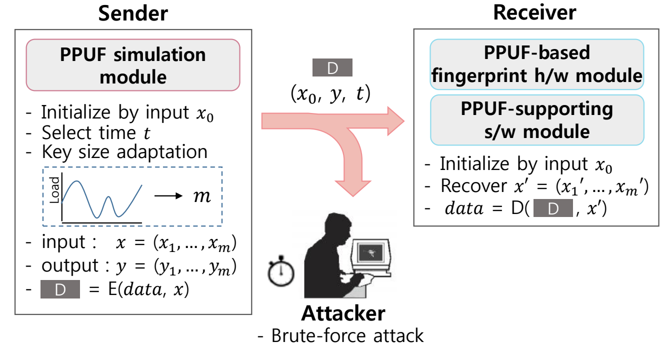


Fig. 2. *PUFSec* system architecture with PPUF security control algorithm and protocol based on PPUF fingerprint hardware

One example of PPUF is cascading XOR gates depicted in Fig. 1(a). An output of XOR gate changes whenever one of two inputs toggles. In this way, the number of transitions exponentially increases as the number of the cascaded XOR gates h increases. If there are h cascaded XOR gates, 2×2^h transitions are generated (where the width of input signals w is 2). With the increased number of transitions, EGS exponentially increases.

Fig. 1(b) shows an example of gate delay table from the circuit in Fig. 1(a). Assuming that the initial input of gates A and B is "10", the output of gate E and F becomes "00" after some amount of time (i.e., reaching a steady-state). If the input changes to "01" at $t = 0$, the output of gate A changes from "1" to "0" at $t = 7.7$, and then to "1" at t = 9.5 according to the gate delay table. In the same fashion, the output of gate B makes transitions at $t \in \{8.3, 10.5\}$. The transition continues over its subsequent row of the circuit. On the second row (the output of gate C and D), the number of transitions of each gate output is 4 since each input toggles twice (e.g., the output of gate C toggles at $t \in \{16.6, 18.8, 20.1, 21.9\}$).

The PPUF-based secret key exchange protocol can be established as follows. The gate delay table is public information in the PPUF-based public-key security. A sender selects x_0, x_1, and t to calculate a PPUF output y_1; x_0 is an input that makes the PPUF reach a steady-state; x_1 is an input arriving at time 0; y_1 is a simulated output using public information at a selected time t. After obtaining the PPUF output y_1, the sender sends x_0, y_1, and t to an intended user. In order to recover the secret key x_1, the legitimate user executes the PPUF logic with all possible input values. Note that the simulation time exponentially increases as the number of the transitions doubles at each row. Therefore, only the legitimate receiver can obtain the result by *executing* the PPUF within a much shorter amount of time than the attacker who requires a tremendous amount of *simulation* time.

III. System Architecture

We present *PUFSec*, a fingerprint-based security architecture for the Internet of Things, as illustrated in Fig. 2. By leveraging inherent hardware characteristics as the uniqueness of a

device itself, we aim to design a computationally lightweight security software system architecture with which an otherwise software-wise complex cryptography computation can be prohibited as much as possible.

We consider a scenario of the end-to-end authenticated data delivery between two IoT devices. We let a sender deliver an encrypted data to a user. We want only the legitimate user to successfully decrypt the received data, while preventing any attacker from doing so within a reasonable time limit.

We define three main components in our *PUFSec* architecture: 1) PPUF-based fingerprint hardware module, 2) PPUF-supporting software module, and 3) PPUF simulation module.

A. PPUF-based fingerprint hardware module

This module is executed at the user side by running it to reconstruct its secret key that has been used to encrypt the received data from a sender. By exploiting unique delay characteristics as the fingerprint of a given hardware, only the legitimate user installed with the universally-unique hardware module can successfully recover the *hidden* challenge bits (which are the input of this module) from the given *public* response bits (which are the output of this module) within a limited timeframe.

B. PPUF-supporting software module

This module is located at the user side together with the PPUF-based fingerprint hardware module. It is physically connected to the above hardware module via I/O ports. The software module queries to the hardware module with a challenge and a specific timing (to measure its output as response) by writing it to output ports, and receives its response from the hardware module by reading it from input ports.

The PPUF-supporting software module is launched when it receives (x_0, y, t) information with the encrypted data from the sender where $y = (y_1, \ldots, y_m)$ and m is the key concatenation length. In case of the concatenation length $m = 1$, in order to find the hidden challenge x'_1, which is the input of the hardware module that results in y_1, it tries to iterate all possible challenges as input. It sends each input candidate x and the timing t to the hardware module, leading to the worst case of comparing with 2^w input trials. It should be noted that gate delay characteristics (e.g., Fig. 1(b)) of each end-user and (x_0, y, t) are *public* information available anywhere.

C. PPUF simulation module

This module is executed at the sender side to generate a response given a selected challenge in a software manner given the public delay characteristics of the designated receiver. Using the gate delay table, it performs software-wise simulations of every possible output at each output measurement time.

After choosing a specific challenge x_1 (which will be used as a secret key with the concatenation length of 1) and a specific time t, it obtains the simulated output result y_1 as the corresponding response. This module encrypts a data to deliver with the secret key, and sends a designated user the encrypted data together with (x_0, y_1, t).

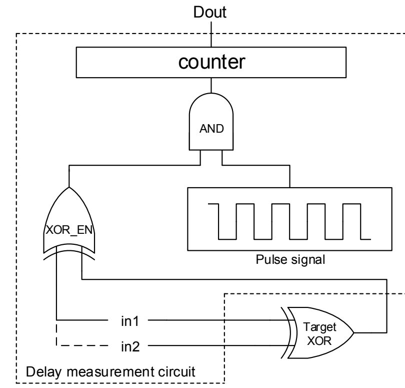


Fig. 3. Delay measurement circuit with targeted XOR gate

The software-wise simulation of all possible output at each different timing can be somewhat computationally intensive. To tackle the computation overhead that low-end devices often suffer from, we dynamically control security strength depending on the current load status of a system by changing the concatenation length of a secret key to generate. In this way, we construct a load-aware adaptive key generation mechanism, making it practically feasible in real-world low-end IoT devices.

IV. PPUF-based Fingerprint Hardware

This section describes PPUF hardware architecture and implementation details for resource-constrained IoT devices.

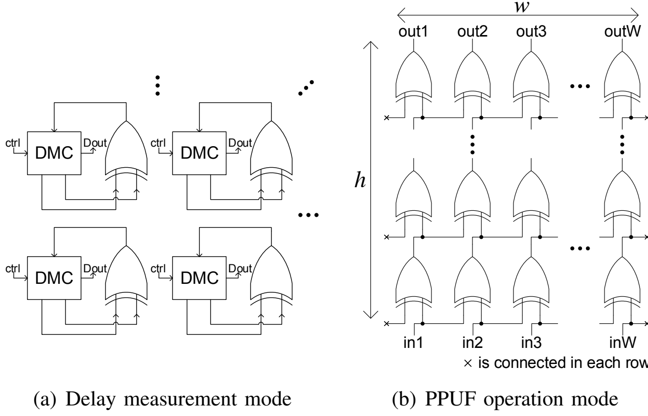
A. Counter-based delay measurement circuit (DMC)

In PPUF, the gate delay measurement is essential and important to set up public data for public-key security. Measuring gate delays is a cumbersome process that requires a complex circuit and an expensive equipment. To construct a PPUF hardware on resource-constrained IoT devices, a lightweight delay measurement method is required. This subsection presents a simple counter-based gate delay measuring implementation. The counter can be considered as an on-chip delay-meter. Fig. 3 depicts the proposed *DMC* (delay measurement circuit) that consists of a counter, AND gate, and XOR gate.

To measure a delay, we initialize the *DMC* by setting in1 = 0 and in2 = 0. Then, in1 = 1 is applied to measure an in1 propagation delay of the targeted XOR gate. However, for a short amount of time, the targeted XOR output does not change to 1 even after in1 = 1 owing to the propagation delay. This makes the output of XOR_EN 1, and the counter increases since the pulse signal such as system clock reaches to the counter. The counter is only incremented until the targeted XOR output becomes 1 after the propagation delay. This sets the output of XOR_EN to 0, and consequently, the counter stops increasing. As a result, we obtain the targeted XOR gate delay by observing the counter value through Dout.

B. Hardware architecture

The PPUF architecture needs to support not only PPUF operation, but also delay measurement for public data construction. To serve the two operations, we propose a PPUF



(a) Delay measurement mode (b) PPUF operation mode

Fig. 4. PPUF hardware architecture

hardware architecture given in Fig. 4. It has two modes: delay measurement mode and PPUF operation mode.

1) Delay measurement mode: Fig. 4(a) shows a configuration of delay measurement mode. In this mode, each delay of XOR gates in PPUF is measured. As addressed in Sec. IV-A, each DMC is controlled to measure the delay of XOR gate.

2) PPUF operation mode: Fig. 4(b) illustrates a configuration of PPUF operation mode. It builds a canonical form of PPUF [1] whose size is h(height) × w(width). There are w-bit input and w-bit output. To recover the secret key, the hardware receives the data (x_0, y_1, t) from the software module and starts iteration. After it finds the matched x'_1 that makes the output y_1 at time t, it sends x'_1 back to our PPUF-supporting software module.

V. Adaptive Security Control

In this section, we present our key exchange procedure and adaptive key generation scheme in the proposed *PUFSec* architecture. Two parties of a sender and a receiver aim to establish a secure connection without exposing significant security credentials in the middle of communication.

A sender executes the *PPUF simulation module* (presented in Sec. III-C) that generates a secret key (as challenge) based on the unique delay characteristics of the receiver to which it wants to send data. It sends only its response with which the original secret key can be recovered not by any unintended receiver including attackers, but by only the intended receiver running its own PPUF system (presented in Secs. III-A and III-B). We present the procedure of how key-related credentials are exchanged in the architecture in Sec. V-A.

To effectively use available resources usually limited in IoT devices, it is important to perform adaptive resource allocation over time depending on resource usage dynamics. When the given resource becomes scarce due to the main job processing, the device may decrease the security level only during some interval and increase back when the resource is available. We apply the idea of adaptive security control to the PPUF architecture through dynamic key concatenation in Sec. V-B.

The resource itself drastically affects PPUF performance in terms of computation and memory usage. Given the limited computation and memory resources, the design parameters of the height and the width of PPUF, and the concatenation length are intertwined, and should accordingly be determined to protect from attackers. We perform theoretical security analyses on computation complexity and memory usage with respect to those design parameters in PPUF in Sec. V-C.

A. Key Exchange Procedure

We consider two parties of a sender and a receiver to exchange public key information (i.e., *PPUF response*) and encrypted data using the original secret key (i.e., *PPUF challenge*). The sender executes the *PPUF simulation module*, while the receiver runs its own *PPUF-based fingerprint hardware module* to recover the original secret key from the received public key information.

We implement a key exchange protocol similar to [1]. To let the sender control its suitable encryption security strength, we use a key concatenation method to change the secret key size. We denote the concatenation length as m.

In the sender side, *PPUF simulation module* is run to simulate a pair of (challenge, response) based on the public delay information of the intended receiver. It first selects a specific time t at which the output will be measured for the input fed at t = 0. Then, it randomly selects a steady-state input x_0 among all possible input cases and makes the PPUF simulation module reach a steady-state. After this, it randomly selects the first challenge x_1 from $[0, 2^w)$, except for x_0, and runs the PPUF simulation to get its resulting response y_1. Using the concatenation length of m, the module continues this procedure until we obtain a whole response set (y_1, y_2, $\ldots$, y_m) for the challenge set (x_1, x_2, $\ldots$, x_m). We use the challenge set (x_1, x_2, $\ldots$, x_m) as a secret key and encrypt data with it. The sender delivers the encrypted data with only the *symptom* (x_0, y_1, y_2, $\ldots$, y_m, t) to its intended receiver, while suppressing the original challenge set.

When the intended receiver receives those information from the sender, it first initializes its PPUF fingerprint hardware by feeding the received steady-state input x_0. We construct a set of (challenge, response) pairs by iterating over all possible input cases and measuring each corresponding response measured at t via the PPUF fingerprint hardware. Once the whole pair table is obtained, we start finding x'_1 that matches the received y_1. We continue to do so until fully recovering its original challenge set (x'_1, x'_2, $\ldots$, x'_m), which is the original secret key used for the encryption at the sender. Finally, the receiver decrypts the received encrypted data with the recovered secret key.

B. Load-Aware Adaptive Key Generation

We suppose that there exists the maximum permitted energy budget E_{max} to use during a certain interval in an individual IoT device. A device has tasks to perform, and its system load status fluctuates over time. In low-end embedded platforms, keeping a fixed strong security level all the time irrespective of system's load status may severely interrupt some concurrent task execution other than the security operation. A desirable approach is that if the system is currently heavy-loaded,

Algorithm 1 Adaptive PPUF Key Simulation at Sender

```
1: Input: Gate-level delays of receiver
2: Output: Encrypted data & Yvalue
3: Monitor the current system load;
4: Calculate the available computational energy budget;
5: KeyLength = select-key-length(availablePower);
   // Make the PPUF reach a steady-state by feeding initialValue
6: Randomly choose initialValue among [0, 2^width);
7: Invoke ppuf-simulate(initialValue);
8: Select a selectedTime;
9: SecretKey = 0, Yvalue = 0;
10: while ( length(SecretKey) < KeyLength )
11:     Randomly choose Xvalue among [0, 2^width);
12:     SecretKey = (SecretKey ≪ width) | Xvalue;
13:     Response = ppuf-simulate(Xvalue, selectedTime);
14:     Yvalue = (Yvalue ≪ width) | Response;
15: endwhile
16: Encrypt data with SecretKey;
17: Send the Encrypted data with (initialValue, Yvalue, selectedTime);
```

Algorithm 2 Recover PPUF Key Using Fingerprint H/W at Receiver

```
1: Input: Encrypted data & (initialValue, Yvalue, selectedTime)
2: Output: Decrypted data
3: if ( have enough resource to decrypt the data )
       // Make the PPUF reach a steady-state by feeding initialValue
4:     Invoke ppuf-execute(initialValue);
5:     for ( Challenge = 0 to 2^width − 1 )
6:         Response = ppuf-execute(Challenge, selectedTime);
7:         Store (Challenge, Response) pair;
8:     endfor
9:     SecretKey′ = 0;
10:    while ( length(SecretKey′) < KeyLength )
11:        Response = low-width-bits(Yvalue ≫ (width×(concatLength −1)));
           // Find Challenge by looking up (Challenge, Response) pairs
12:        Challenge = find-by-lookup(Response);
13:        SecretKey′ = (SecretKey′ ≪ width) | Challenge;
14:        concatLength = concatLength −1;
15:    endwhile
16:    Decrypt the data with SecretKey′;
17: else
18:    delay the decryption process;
19: end if
```

it autonomously decreases its security level, reducing the required computation overhead for security. After completing a large portion of tasks, the system may invest some more resources for security by increasing the security level in an adaptive manner.

We design an adaptive key generation mechanism based on an understanding of the trade-off between computation overhead and security level. We monitor the current energy consumption E_{load} during an interval made by main tasks. We can calculate the remaining available energy budget that can be invested for security, i.e., $E_{rem} = E_{max} - E_{load}$. Our goal is to fully allocate the remaining energy budget E_{rem} for increasing its security level as strong as possible.

To achieve this goal, we first construct an empirical model of energy consumption for PPUF key generation and encryption (at the sender side) with respect to key size. We run the PPUF simulation module (as described in Sec. III-C) in Raspberry Pi 2 Model B (which is our system environment of implementation and evaluation in Sec. VI) to measure the incurred energy consumption. After the key generation, the sender side should perform an encryption process before sending data. For encryption and decryption, we use a modified AES algorithm that has been ported from AES in TinyOS and has been extended with various key size. In case of the data size of 20 Kbytes, we measure the incurred energy consumption for running the encryption with respect to key size and record them in a lookup table as in Table I. We use the lookup tables (with respect to various data size) to determine the key size to generate within the available energy budget.

Key Size (Byte)	Key Simul. (J)	Encryption (J)	Total (J)
16	3.184	0.183	3.368
32	5.955	0.256	6.210
48	8.963	0.325	9.287
64	11.751	0.390	12.140
80	14.704	0.468	15.172
96	17.346	0.528	17.874
112	20.702	0.603	21.305
128	23.213	0.685	23.898
144	26.138	0.753	26.891

TABLE I
EMPIRICAL ENERGY CONSUMPTION MEASUREMENTS FOR RUNNING PPUF KEY SIMULATION AND MODIFIED AES ENCRYPTION

Once the available energy budget E_{rem} is given, we find the maximum key size k_{max} that can fit into E_{rem} by referring to the lookup table. Then, we calculate the concatenation length $m = \lfloor k_{max}/w \rfloor$.

To the end, we choose a key size as strong as possible to maintain in a certain interval and continue to refresh the key size in a subsequent interval, while reflecting real-time load dynamics in IoT devices.

C. Security Analysis

We perform the theoretical analyses of security performance in a PPUF system. Since IoT devices face the fundamental constraints of relatively low-end computation power and small storage, it is important to analyze the qualitative system performance of a PPUF system under various system configurations. Depending on the amount of resource and a given reliability requirement, feasible hardware and software design parameters (e.g., the height and the width of PPUF) should be determined.

We first define the *transition delay*, which specifies the incurred latency from challenge to response at PPUF. The overall PPUF delay characteristic results from a composite of gate delay characteristics over the selected path.

Definition 1 (Transition Delay) *The transition delay T_h from stage 1 up to h in PPUF is defined as $\sum_{i=1}^{h} \tau_i$ where τ_i is the gate delay from a selected input to its resulting output at stage i.*

We define the *collision*, which is is the result of two signal paths indistinguishable due to the inherent hardware require-

ment from the measurable time resolution. We categorize it into two types: *intra-collision* and *inter-collision*.

Definition 2 (Intra-Collision) *Given that there exists the minimum detectable glitch duration δ required by hardware, PPUF has an "intra-transition" ambiguity between inputs 1 and 2 within a gate, called "intra-collision" in case of $|\tau_i^{(1)} - \tau_i^{(2)}| < \delta$.*

Definition 3 (Inter-Collision) *Given that there exists the minimum detectable glitch duration δ required by hardware, PPUF has an "inter-transition" ambiguity between two different paths up to stage h, called "inter-collision" in case of $|T_h - T_h'| < \delta$, excluding any intra-collision case in parts.*

Theorem 1 (Identification Failure Due to Intra-Collision) *PPUF has the identification failure probability due to at least one intra-collision that increases as h, w, or δ does, respectively.*

Proof: The probability of no intra-collision over $h \times w$ PPUF gate configuration is given by $[P(|\tau_i^{(1)} - \tau_i^{(2)}| \geq \delta)]^{h \cdot w}$. The signal paths from each input to output are uniquely characterized due to the manufacturing variability. Accordingly, $\tau_i^{(1)}$ and $\tau_i^{(2)}$ can be considered as independent random variables of the gate delay. Given the probability density function $g(\tau)$ of the gate delay τ, the probability can be computed as follows:

$$P(|\tau_i^{(1)} - \tau_i^{(2)}| \geq \delta) = 2 \cdot \int_0^\infty \int_{x_2+\delta}^\infty g(x_1) g(x_2) dx_1 dx_2$$
$$= 2 \cdot \int_0^\infty \bar{G}(x_2 + \delta) g(x_2) dx_2$$

where the probability is a non-increasing function of δ.

Consequently, there exists an identification failure due to the intra-collision with the probability of $1 - [P(|\tau_i^{(1)} - \tau_i^{(2)}| \geq \delta)]^{h \cdot w}$, increasing as h, w, or δ does, respectively. ■

Theorem 2 (Identification Failure Due to Inter-Collision) *PPUF has the identification failure probability due to an inter-collision, dependent upon h and δ.*

Proof: The probability with no inter-collision over $h \times w$ PPUF gate configuration is given by $P(|T_h - T_h'| \geq \delta)$. As aforementioned, $\tau_1, \tau_2, \ldots$ can be assumed to be independent and identically distributed with mean μ and variance σ^2. Applying the central limit theorem, T_h (= $\sum_{i=1}^h \tau_i$) can be approximately normally distributed with mean $h\mu$ and variance $h\sigma^2$, i.e., $T_h \sim N(h\mu, h\sigma^2)$. Denoting $l(t)$ as the probability density function of the transition delay t, the probability of no intra-collision can be calculated as follows:

$$P(|T_h - T_h'| \geq \delta) = 2 \cdot \int_0^\infty \int_{t_2+\delta}^\infty l(t_1) l(t_2) dt_1 dt_2$$
$$= 2 \cdot \int_0^\infty \bar{L}(t_2 + \delta) l(t_2) dt_2$$

where $l(T_h)$ is reduced to be the probability density function of a normal distribution with $N(h\mu, h\sigma^2)$.

Accordingly, there exists an identification failure due to an inter-collision with the probability of $1 - P(|T_h - T_h'| \geq \delta)$, dependent upon h and δ. ■

Theorem 3 (Identification Reliability) *PPUF has the identification reliability that decreases as h, w, or δ increases, respectively.*

Proof: The probability of the successful identification free from both intra-collision and inter-collision is given by $[P(|\tau_i^{(1)} - \tau_i^{(2)}| \geq \delta)]^{h \cdot w} + P(|T_h - T_h'| \geq \delta)$. As the first term is the dominant factor with respect to h, it is straightforward to conclude that the probability decreases as h, w, or δ increases, respectively. ■

We now measure the qualitative complexity of computation and storage by varying design parameters of a PPUF system.

Theorem 4 (Computation Complexity) *Computation complexity with the height of h, the width of w, and the concatenation length of m in PPUF is given by $O(w \cdot 2^h \cdot m)$ at the sender side that simulates keys, $O(2^w)$ at the legitimate receiver side, and $O(w \cdot 2^w \cdot 2^h)$ at the attacker side.*

Proof: The sender side needs to compute the transition time for each gate at each stage by iterating over the total $w \cdot 2^h$ number of transitions up to stage h (as also noted in [1]). We repeat this procedure with m times for its subsequent concatenations, thereby taking the total computation complexity of $O(w \cdot 2^h \cdot m)$. The legitimate receiver side seeks an input that matches the given output by iterating over all possible input cases, i.e., 2^w cases. It continues with m concatenations by just referring to the iterated input-output pairs, leading to the computation complexity of $O(2^w)$. The attacker side, on the other hand, should iterate over all possible input cases as well as all possible transition cases up to stage h. For the subsequent concatenations, it can just refer to the generated input-output pairs, resulting in the computation complexity of $O(2^w \cdot w \cdot 2^h)$. ■

This theorem implies that increasing the height of PPUF is a very effective way of forcing an exponentially increasing computation burden to attackers. In return, however, there exists a side effect on the sender side with the exponentially increasing computation time for simulation, making practical operations infeasible particularly for IoT devices.

It should be noted that although increasing the concatenation m strengthens a cryptography algorithm's security for encryption and decryption, it may cause some computation burden for simulating a key at the sender side. This is due to the fact that the computation complexity of the other parties (i.e., the user side and the attacker side) is irrespective of the concatenation length m. Therefore, there should exist a maximum allowable value m_{max} such that the computation complexity of the attacker side is significantly larger than that of the sender with the concatenation length of [1, m_{max}].

Theorem 5 (Storage Complexity) *Storage complexity with the height of h, the width of w, and the concatenation length of m in PPUF is given by $O(w \cdot 2^h)$ at the sender side that simulates keys, $O(2^w)$ at the legitimate receiver side, and $O(2^w + w \cdot 2^h)$ at the attacker side.*

Proof: The sender side needs to store all possible transition times with the memory complexity of $O(w \cdot 2^h)$. The legitimate receiver side should record each input and its corresponding output until it finds the matched input for the given output, requiring the memory complexity of $O(2^w)$. On the other hand, the attacker side needs to store all the input-output pairs, and also all possible transition times, leading to the memory complexity of $O(2^w + w \cdot 2^h)$. Note that the concatenation procedure does not require any further memory space at all parties since the already-assigned memory space is reusable for the next concatenation. ■

Taking a holistic consideration in designing a practically feasible PPUF system, there is an inevitable trade-off between identification reliability, and computation and storage overhead for increasing the height and the width of the PPUF system.

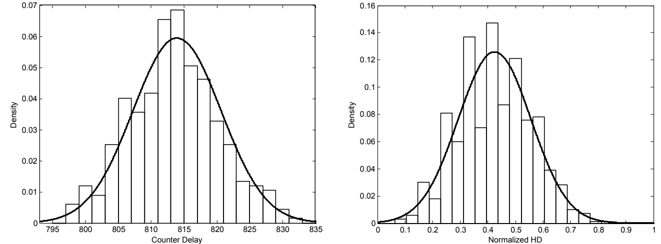
VI. Evaluation

We validate our device fingerprint-based security architecture *PUFSec* by implementing a PPUF hardware logic and a PPUF-compatible software module in real-world platforms. We implemented our PPUF fingerprint hardware consisting of a simple counter-based delay measurement logic and an XOR array network of up to 7 × 24 (where $h = 7$ and $w = 24$) for the PPUF operation, in Xilinx Zynq-7000 FPGAs. We implemented our PPUF-based software system in Raspberry Pi 2 Model B with Raspbian operating system based on Linux Debian.

A. PPUF Hardware Performance

First, we evaluate our PPUF fingerprint hardware performance with respect to a gate delay distribution and board uniqueness. We measured delays for 336 gates at a normal operating temperature from an FPGA board. As in Fig. 5(a), the delay counter value is ranged with the mean value of 813.9 and the standard deviation of 6.7, where one counter granularity is approximately 2.5 ns. One interesting observation is that the gate delay distribution of the PPUF implementation is a high goodness-of-fit with a normal distribution based on both Kolmogorov-Smirnov test and χ^2 test with the p-values of 0.89 and 0.69, respectively under a significance level of 5%.

More importantly, we quantify the inherent uniqueness of each different PPUF FPGA implementation. We use the board-to-board Hamming distance (HD) as a uniqueness measure, showing what percentage of response bits are different from each other between two FPGA boards for a given identical challenge (x_0, x_1, t). We repeat this procedure with 1,000 different challenge sets. Fig. 5(b) shows the distribution of board-to-board HD measurements at a normal operating temperature. The mean value of the normalized HD is 0.425, which is close to the ideal value 0.5. This implies that our PPUF FPGA implementation embeds distinctive variability enough to distinguish between two different PPUF hardware security modules.



(a) Gate delay measurements from our PPUF FPGA implementation

(b) Board-to-board Hamming distance as a uniqueness measure between two different PPUF FPGA implementations

Fig. 5. PPUF FPGA performance with respect to gate delay and uniqueness

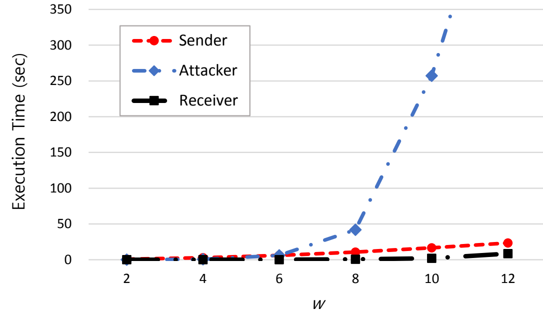
B. PPUF System Performance

We evaluate the authentication execution performance of *PUFSec* under various system configurations in the Raspberry Pi 2 platform. We measure the execution time for key generation at the simulation module of a sender, and the execution time for key recovery at a legitimate receiver, and an attacker. The attacker attempts to retrieve its original secret key based on a brute-force attack at a third party by simulating the PPUF logic over all possible challenge inputs to find the exactly matched response output. We vary the design parameters of w, h, and m. Since the sender and the attacker side have been implemented purely based on software, their execution time is determined by their simulation runtime. The receiver side consists of the PPUF-supporting software module and the PPUF-based fingerprint hardware module. To quantify the total execution time at the receiver, we measure the software execution time and the FPGA access time through I/O ports. The FPGA access time is calculated as the sum of two parts: the GPIO access time to write for x_0, x_1, x_2, ..., x_m, t and read for y_1, y_2, ..., y_m toward Raspberry Pi 2, and the maximum PPUF gate logic runtime inside the FPGA board (obtained from the worst case logic latency from FPGA measurements).

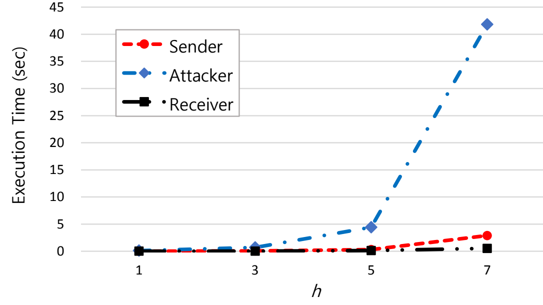
To validate the storage complexity, which is of a great practical interest to resource-constrained embedded IoT systems, we quantify the memory footprint for running a PPUF system at each party of a sender, a receiver, or an attacker, respectively.

In our experiments, the parameters of $w = 8$, $h = 7$, and $m = 16$ in the PPUF hardware and software are used, unless otherwise noted.

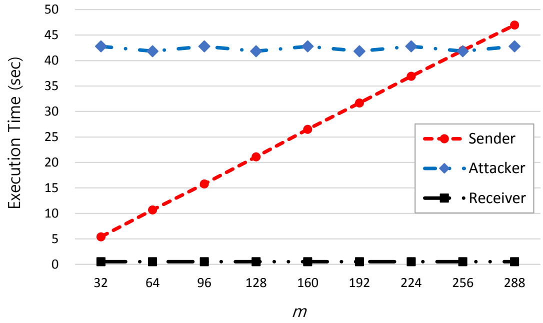
We investigate computation complexity in terms of execution time as in Fig. 6. As the width w of PPUF increases in Fig. 6(a), the attacker has an exponentially increasing execution time. In a Raspberry Pi 2 board, the attacker has indeed failed to finish its simulation module to recover the secret key in a brute-force manner beyond $w = 10$. On the other hand, the intended receiver in possession of its own



(a) Execution time with respect to w ($h = 7$, $m = 16$)



(b) Execution time with respect to h ($w = 8$, $m = 16$)



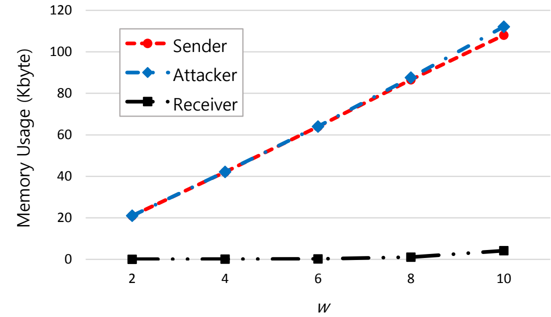
(c) Execution time with respect to m ($w = 8$, $h = 7$)

Fig. 6. Authentication execution performance comparison at each party of sender, intended receiver, and attacker

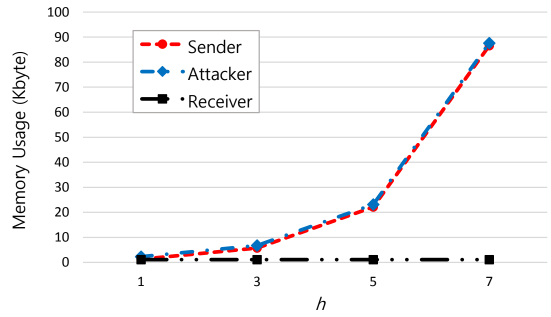
PPUF fingerprint hardware and software is very fast to recover the secret key, while the sender spends only slightly more time than the receiver. The time gap results from whether the key generation is made by fingerprint hardware or pure software. Regarding the effect of the height h, the measured execution time shows a similar trend with varying w as shown in Fig. 6(b).

Given our current PPUF implementation scale up to 7×24 array network (where $h = 7$ and $w = 24$), it turns out that increasing w is a very effective way to protect against a brute-force attack, severely penalizing its simulation execution time.

Next, we explore the effect of the concatenation length m in Fig. 6(c). To show how the computation complexity is affected by m for all of the parties, we have intentionally run experiments in a small PPUF scale with $w = 8$. The receiver and the attacker have run their execution irrespective of the concatenation length m, whereas the execution time at the sender increases in a linear fashion (as also analyzed in Theorem 4). It is shown that there can exist a maximum allowable value of m where the execution time of the sender is



(a) Memory usage with respect to w ($h = 7$, $m = 16$)



(b) Memory usage with respect to h ($w = 8$, $m = 16$)

Fig. 7. Memory footprint measurement at each party of sender, intended receiver, and attacker

even larger than that of the attacker. This means that increasing the concatenation length for improving the encryption security may harm its precedent key generation at the sender, leading to high computation burden at some point in return. Thus, it is important to choose the proper design parameters of w, h, and m to implement a PPUF system with high efficiency and security by taking into account their relative performance relationship.

We have observed that the empirical results are consistent with the theoretical analysis of Theorem 4. It should be also noted that the execution time at the receiver in possession of PPUF hardware and software can further be reduced if the PPUF hardware logic is implemented in application specific integrated circuit (ASIC).

We also examine how our PPUF system consumes memory resource by varying the design parameters of w and h in Fig. 7. As the width w increases, both the sender and the attacker require linearly increasing memory resource, whereas the receiver uses relatively very small memory space as shown in Fig. 7(a). As the height h increases, on the other hand, both the sender and the attacker consume exponentially increasing memory space, whereas the memory usage at the receiver is almost constant, irrespective of h. This means that both the sender and the attacker need a large amount of memory space for maintaining all possible transitions, whereas the receiver can record only each input and its corresponding output by running the PPUF hardware module. Note that these empirical results are also consistent with the theoretical analysis of Theorem 5.

We discuss two ways to make brute-force attacks practically infeasible from the perspectives of storage complexity and

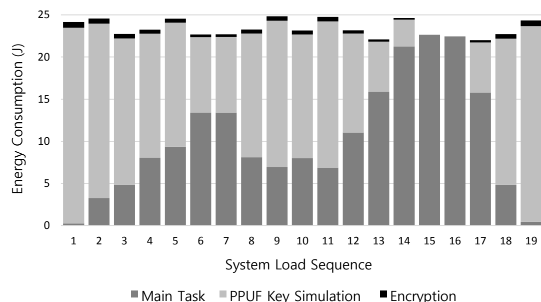


Fig. 8. Adaptive security level control performance under system load dynamics

computation complexity. If we want to disable the brute-force attack by forcing the attacker to consume too much memory, we need to increase either the height h or the width w of PPUF. However, this approach also degrades the performance at the sender side, damaging the entire application scenario itself. From the perspective of computation complexity, on the other hand, choosing either large w or h can be a way to make an attacker to run its simulation with a tremendous amount of time. Thus, we have to choose a certain range of w and h values that can make the attacker enough to run forever, while not causing too much memory resources penalty to the sender.

We also discuss which parameter between w and h is a more critical one to severely affect the PPUF security. Although increasing both parameters results in exponentially increasing computation time at the attacker, the memory usage exponentially increases with h, but only linearly increases with w at the attacker and also the sender. This indicates that increasing the width w of PPUF is a more practical approach to improve both security and feasibility.

Lastly, we validate our load-aware security control mechanism under various load dynamics. To simulate our adaptive security control feature, we execute a different virtual main task with a periodic manner in Raspbian Linux, while consuming processing energy, as shown under 'Main Task' in Fig. 8. Under a system operation requirement that Raspberry Pi 2 should last for one month without battery replacement using its initial battery amount of 6000 mAh, we let our system be limited by a maximum energy budget of 25 J within every 10 minute window. By monitoring the current processing energy consumption for the main task, our security control mechanism successfully finds the strongest security level, but within its remaining energy resource, as system load fluctuates over time. This demonstrates that our adaptive security control based on PPUF contributes to reaching a great balance between computation and security in real-world IoT platforms.

VII. Conclusion

We have presented a PPUF-based security system architecture for a real-world Internet of Things based on unclonable hardware characteristics. We design a load-aware adaptive key generation mechanism compatible with PPUF, which is implemented in FPGA by leveraging a simple counter-based delay measurement scheme. It dynamically adjusts security level for authenticated data delivery between two parties by changing the secret key size to reflect the available computation power of a sender side. Therefore, our cryptography system enhances the security strength with allowable computation overhead in IoT devices.

Our experiments based on PPUF-implemented hardware and software indicate that our system architecture inherently becomes more invulnerable to attacks from a third party as increasing the width or the height of PPUF. In addition, we discover a crucial observation that there exists a maximum allowable key size since a larger key size beyond it puts significant computation burden, disabling the PPUF key simulation in the sender side, through both theoretical analysis and empirical measurements.

For future work, we would conduct extensive experiments to investigate the effect of environment factors such as temperature and interference. Also, we may implement a closed-loop *PUFSec* prototype by more tightly coupling the PPUF-based fingerprint hardware module with the PPUF-supporting software module. It would also be interesting to design a lightweight communication algorithm for security negotiation to determine proper key size and key renewal time optimized by system load at both ends.

Acknowledgment

This work was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education (NRF-2015R1D1A1A01057902 and NRF-2015R1D1A1A01058856).

References

[1] N. Beckmann and M. Potkonjak. Hardware-based public-key cryptography with public physically unclonable functions. In *International Workshop on Information Hiding*, pages 206–220. Springer, 2009.

[2] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Silicon physical random functions. In *ACM CCS*, 2002.

[3] C. Karlof, N. Sastry, and D. Wagner. TinySec: a link layer security architecture for wireless sensor networks. In *ACM SenSys*, 2004.

[4] M. Li, J. Miao, K. Zhong, and D. Z. Pan. Practical public PUF enabled by solving max-flow problem on chip. In *ACM DAC*, 2016.

[5] A. Liu and P. Ning. TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. In *ACM/IEEE IPSN*, 2008.

[6] M. Majzoobi and F. Koushanfar. Time-bounded authentication of fpgas. *IEEE Transactions on Information Forensics and Security*, 6(3):1123–1135, 2011.

[7] L. B. Oliveira, D. F. Aranha, C. P. Gouvêa, M. Scott, D. F. Câmara, J. López, and R. Dahab. TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks. *Computer Communications*, 34(3):485–493, 2011.

[8] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. SPINS: Security protocols for sensor networks. *Wireless networks*, 8(5):521–534, 2002.

[9] M. Potkonjak and V. Goudar. Public physical unclonable functions. *Proceedings of the IEEE*, 102(8):1142–1156, 2014.

[10] M. Rostami, J. B. Wendt, M. Potkonjak, and F. Koushanfar. Quo vadis, PUF?: trends and challenges of emerging physical-disorder based security. In *Proceedings of DATE conference*, 2014.

[11] U. Rührmair. Simpl systems: On a public key variant of physical unclonable functions. *IACR Cryptology ePrint Archive*, 2009:255, 2009.

[12] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, pages 9–14. ACM, 2007.

[13] S. Zhu, S. Setia, and S. Jajodia. LEAP+: Efficient security mechanisms for large-scale distributed sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 2(4):500–528, 2006.