On Secure Shared Key Establishment for Mobile Devices using Contextual Information

Ala' Altaweel, Radu Stoleru and Subhajit Mandal
Department of Computer Science and Engineering, Texas A&M University
{altaweel, stoleru, subhajitm89}@cse.tamu.edu

***Abstract*—In this paper we first show that the Wi-Fi Protected Setup (WPS) protocol (used by Wi-Fi Direct, the de facto adhoc communication mechanism for smartphones and mobile devices) is vulnerable to a brute-force or dictionary attack. To defend against these attacks, we propose the idea of using contextual information (i.e., data obtained from mobile device's sensors) to establish a long (128 bits) secure session key between two Wi-Fi Direct enabled devices, instead of using the keypad. Our solution, Session Key Generated from Sensors (SekGens) employs three phases. In the *Quantization Phase*, the key is iteratively generated based on different sensors' data. In the *Reconciliation Phase*, the two devices eliminate minor differences in the bits of their keys by using the Cascade reconciliation mechanism. In the *Privacy-Amplification-and-Hashing Phase*, the two devices omit all bits exposed during the reconciliation phase and apply hashing to the remaining secret bits. SekGens is implemented and evaluated by modifying the Android kernel code responsible for WPS in Google Nexus 5 and Samsung Galaxy S2 smartphones. The results show that SekGens generates keys with low mismatch ratio (less than 3%), at a fast rate (∼20 bits/sec), and with high entropy (∼92%).**

I. Introduction

Smartphones are increasingly equipped with a new emerging wireless standard that allows users to securely establish an adhoc wireless network (i.e., without the assistance of a wireless router) and directly exchange data between each other. The new standard, the Wi-Fi Direct protocol, developed by the Wi-Fi Alliance [13], is built upon the IEEE 802.11 infrastructure and it enables the devices to negotiate and decide which one will rule the AP-like functionalities. As a result, it inherits all mechanisms that have been developed for Wi-Fi infrastructure, including security, QoS and power saving.

Wi-Fi Direct, like other direct device-to-device wireless technologies, is vulnerable to many security attacks against confidentiality, authentication and integrity. In order to establish a secure key between two devices, Wi-Fi Direct implements the Wi-Fi Protected Setup (WPS) protocol [7] to achieve a secure connection with lower user involvement. There are two major modes of operation of WPS: in-band configuration and out-of-band configuration and both of them depend on secure key exchange between nodes. With out-of-band configuration, the WLAN credentials are encrypted and sent across an out-of-band channel, e.g., USB flash drive, to the Enrollee. With in-band configuration, which is our focus in this paper, a *device password*, which is obtained from the Enrollee and entered into the Registrar manually using a keypad, is used to perform a Diffie-Hellman key exchange to guarantee the authentication between the two devices.

One major security concern and research challenge for the Wi-Fi Protected Setup (WPS) [7] protocol is its vulnerability to a brute-force or a dictionary attack [22]. This vulnerability depends upon the device password's length. Even though it is recommended to use an eight digit numeric PIN, this length does not guarantee the large amount of entropy that is needed for strong mutual authentication. That is why it is recommended to use a fresh PIN at each running time of the registration protocol. Entering a fresh and long PIN each time is not convenient from the user's perspective in terms of memorizing it and retrying according to wrong entries. Nevertheless, if the user uses the same keys many times or enters short keys, the attacker's task of compromising the connection becomes easier. From another perspective, it was shown [16] that human adversaries, even without a recording device, can be effective at eavesdropping (i.e., shoulder surfing a victim entering a PIN as a numeric password in smartphones) by employing cognitive strategies and by training themselves.

To address the aforementioned research challenge, we propose to use contextual information, obtained from on-board sensors. Our proposed solution, SekGens has three phases. Quantization: the draft key is generated from different sensors data. Reconciliation: the two devices eliminate any minor differences in the bits of their draft keys by using the Cascade reconciliation mechanism. Privacy-Amplification-and-Hashing: the two devices omit all bits exposed during the reconciliation phase and apply MD5 hashing to the remaining secret bits to generate the final secret 128 bits key. The contributions of this paper are as follows: a) we demonstrate a successful attack against Wi-Fi Direct; b) We design a new algorithm, Session Key Generated from Sensors (SekGens) to establish a secure key (128 bits) from the contextual sensors data for two mobile devices; c) We demonstrate the feasibility of our proposed solution through a real implementation of SekGens on Google Nexus 5; d) We demonstrate the robustness of SekGens when it runs on two different smartphones (i.e., Google Nexus 5 and Samsung Galaxy S2); e) We prove the effectiveness of SekGens by showing that it generates keys with high agreement, at a fast rate, and with high Shannon entropy.

II. Background & Motivation

In Wi-Fi network, the role of each device in the network is either a client or an AP. Different sets of functionalities are assigned to each role. A major feature of Wi-Fi Direct [13] is that these roles in it are dynamic. Any Wi-Fi Direct device has to implement the role of client and the role of AP (soft AP). These, are just logical roles that could be executed simultaneously by a single device either by using different frequencies or time-sharing the channel via virtualization techniques.

Wi-Fi Direct devices agree on these roles by forming P2P groups, which are equivalent in term of functionalities to

978-1-4673-8590-9/15/$31.00 ©2015 IEEE

the Wi-Fi infrastructure networks. The Wi-Fi Direct devices, which are also known as P2P devices, are classified as: P2P Group Owner (P2P GO) and P2P clients. P2P GO is the device that implements the AP-like functionality in the P2P group. Other devices, which act as clients, are called P2P clients. There are many mechanisms for the two devices to establish a P2P group. The selected mechanism depends on either the negotiation of P2P GO's role, or if there is any pre-shared security information available. These group formation mechanisms are: Standard formation (the most complex way) and other two simple mechanisms (Autonomous and Persistent).

The group formation mechanisms consist of different phases, which are shared and executed by the two devices: Discovery, Wireless Protected Setup (WPS) Provisioning [7], and Address Configuration. The Standard formation mechanism has an additional phase, the GO Negotiation. WPS, which is implemented by all mechanisms, aims at supporting a secure connection between devices. WPS is developed by the Wi-Fi Alliance as an optional certification program to simplify the setup of security-enabled Wi-Fi networks in small areas like offices. WPS has two modes of operation: in-band and out-of-band configurations. In case of in-band mode, a Diffie-Hellman key exchange is accomplished and authenticated via a shared secret password. This password is obtained from the Enrollee and it is entered into the Registrar using a keypad entry. In case of out-of-band mode, WLAN credentials are sent across an out-of-band channel, e.g., USB flash drive, to the Enrollee.

In this paper, we focus on the in-band configuration mode of operation using a shared secret key that is entered manually using the keypad. The goal of registration Protocol messages (M3-M7, described in the Appendix) is to incrementally prove the mutual knowledge of the device password between the two devices. When the Registrar and Enrollee devices have proven knowledge of the password, the encrypted configuration data is exchanged. Cryptographic protection for the messages depends on a key derivation key (KDK) [7], which is computed from the nonces, Diffie- Hellman secret, and MAC address of Enrollee.

A. Motivation: Brute Force Attack against Wi-Fi Direct

We motivate our research by demonstrating a brute-force attack against Wi-Fi Direct. An inherent flaw in the WPS protocol makes the key discovery easier [22]. There are two ways of cracking the authentication key. Tools like Reaver [5] can crack the PIN in less than 4 hours. On average an attacker can succeed in around 2 hours, this is real-time online way of cracking the WPS PIN. Another method is where the attacker takes a passive strategy by sniffing the packet transmissions M3-M7 that take place between the two devices. Then run an offline tool like aircrack-ng [4] that can look up a dictionary and recreate the PIN used by the Enrolee and Registrar devices. Depending on the configuration of the machine used to run the penetration tool and the complexity of the PIN it may take between 4-10 hours to discover the PIN.

We used an offline method to discover the PIN used by the Registrar to authenticate the Enrolee. On a low end notebook (i.e., without sophisticated hardware capabilities), we configured the wireless interface in monitor mode. Then we used a packet capturing tool, Wireshark to capture the packet transmission between the Registrar and Enrolee, run on two LG Optimus Black smartphones. We generated an exhaustive list of possible PINs and stored it in a dictionary. We used airodump-ng to find the BSSID of our Wi-Fi Direct group. Just providing our prepared PIN dictionary and the BSSID we are interested in, the aircrack-ng tool was able to find out the PIN in just over 3 hours. With a more powerful system the time can be reduced further.

The WPS protocol was introduced by Wi-Fi Alliance in 2007 to enable secure pairing of Wi-Fi devices with compatible APs. The security flaw in its design, which causes the PIN brute force vulnerability, was discovered by S. Viehböck [22] in 2011. At that time, the WPS protocol was used in the the wireless routers. S. Viehböck recommended to disable the WPS protocol on the APs.

The WPS PIN brute force vulnerability has been studied by some security researchers. A. Sanatinia et al. [21] studied the spreading of Wi-Fi APs infections using WPS flaws. They proposed new designs and frameworks for APs to allow more scalable and flexible administration to avoid the spreading of APs infections. D. Zisiadis et al. [23] proposed an enhancement for the WPS protocol by introducing the ViDPSec, a user-based device that runs a paring protocol relying on human visual out-of-band verification. ViDPSec requires from the AP to be equipped with a small LED to display the PIN and SUM values and a button for SUM acknowledgement. Through six steps, ViDPSec requires the users to acknowledge a small number through the visual channel, instead of typing the PIN.

Due to the unavailability of wireless network administrators in adhoc networks, the proposed solution in [21] can not solve the WPS brute-force attacks for Wi-Fi Direct devices. The ViDPSec device [23] is designed for the wireless routers and it is not convenient for smartphone users due to the six steps needed and the need to carry this additional device. Indeed, many intrusion detection systems such as Waidps [6] can detect WPS brute-force attacks. However, these intrusion detection systems are not always available for Wi-Fi Direct devices.

SekGens is a convenient and usable approach for smartphone users, which aims at solving the PIN brute force vulnerability in the traditional WPS, by establishing a 128 bits secure session key between two Wi-Fi Direct enabled smartphones.

III. SekGens: Session Key Generated from Sensors

This section presents our system model and assumptions, our attacker model, and SekGens design.

A. System Model and Assumptions

Our system contains Wi-Fi enabled mobile devices that are in the same radio range. Each device has various sensors, which are continually sensing and generating data for acceleration, sound, etc. There is a pair of devices (i.e., the Enrollee and Registrar) interested in establishing a secure P2P Wi-Fi Direct channel between them. In order to establish this secure channel, these devices implement the WPS [7] protocol and use the in-band configuration mode of operation. The eight registration protocol messages of the WPS protocol are shown in the Appendix. The two devices use M_3-M_7 messages to verify each other. If the verification of one part of the device password fails, the receiving node has to send a failure indication acknowledgment. Accordingly, both devices stop the protocol and discard all nonces and keys related to that session. If the two devices successfully exchange the M_8 message, this

is considered as a proof-of-possession of the device password and they can securely exchange data messages.

The users in the traditional WPS protocol need to enter the same PIN (maximum of eight numerical digits) at both devices to establish the secure connection. Messages M_3 and M_4, shown in the Appendix, contain the two hashed halves of the PIN. For example, if PIN value is "1234", PSK1 and PSK2 at the Enrollee and Registrar devices will result from the HMAC of "12" and "34", respectively. However, SekGens changes the WPS protocol by replacing the two halves that are used to calculate PSK1 and PSK2 with the two halves of the KEY_{Final}'s (i.e., the final key resulted from SekGens on both devices). Therefore, PSK1 and PSK2 are obtained by both Enrollee and Registrar devices as:

PSK1 = first 128 bits of $\text{HMAC}_{AuthKey}$(1st half of KEY_{Final})

PSK2 = first 128 bits of $\text{HMAC}_{AuthKey}$(2nd half of KEY_{Final})

SekGens runs before the WPS protocol. Apart from changing the way of obtaining PSK1 and PSK2, the remaining steps and messages of the registration protocol are not changed.

An important assumption of SekGens is the existence of a collaboration phase, which has to be executed before SekGens. During this phase, the two devices agree on the sensors involved in the calculation and their sampling rate. After that, SekGens will be ready to run on the two devices using the same number and type of sensors. Moreover, the sampling rate sr_i and the time periods $[start_i, end_i]$ for any sensor (denoted as S_i) running at each device will be the same.

B. Attacker Model

Our attacker model assumes that a computationally bounded malicious attacker, Eve, is curious to discover the device password that is used between the Enrollee and Registrar devices. The most attractive areas for Eve are those crowded ones (e.g., restaurants, cafes, airports lounges, or student community areas), where it is difficult to notice her. Eve is able to sniff and capture the exchanged messages M_3 - M_7 that contain the two encrypted halves of the device password. We assume that Eve knows our SekGens algorithm and the value of the nonce exchanged at the beginning of the algorithm. We also assume that Eve has no access to the data generated from the sensors of the Enrollee and Registrar devices. However, Eve is able to compromise the collaboration phase and discover the number and type of involved sensors, their sampling rate, and their time periods. Moreover, we assume that Eve can neither jam the communication channel between the two devices, nor can she modify any messages exchanged between them. Essentially, Eve is not interested in disrupting the key establishment between the two devices. However, she is just interesting in discovering it.

Based on the assumptions described above, Eve can mount the following two types of attacks.

1) **Brute-force attack**: Eve can stimulate the Registrar to start the Diffie-Hellman exchange messages with it. Eve is able to capture the M_4 message, extract R-Hash1 & ENC(R-S1), and use brute-force to discover PSK1 (assuming that the 1st half of the device password is relatively short). By running the second round of the protocol without changing the key, the 2nd half of the device password can be discovered in the same way.

2) **Imitation attack**: Eve tries to generate the same sensors data of the Enrollee or Registrar devices and run SekGens to discover the device password. Firstly, Eve adjusts the sampling rate and the time periods of its sensors to the same values used by the sensors of the pairing devices. Secondly, based on the sensor type, Eve strives to generate the same sensors data. For example, Eve can observe and imitate the gesture made by the victims while they generate data from their acceleration sensors. In case of sound sensors, Eve can (physically) get closer to the victims to have the same readings.

Algorithm 1 Quantization.

1: $\text{KEY}_Q = \Phi$
2: **for** i = 1 to J **do**
3: $\text{Key}_j = \Phi$
4: **for** i = 1 to N **do**
5: find $H(S_i[start_i, end_i])$
6: P = Norm $(H(S_1[start_1, end_1]),\ H(S_2[start_2, end_2]),\ H(S_3[start_3, end_3])$ .... $H(S_N[start_N, end_N]))$
7: Label(P[0,100], $S_1, S_2 ... S_N$)
8: **for** j = 1 to J **do**
9: r = GenNum(NC)
10: Locate r in P, then select the corresponding sensor (say S_k)
11: $[d_{start_k}, d_{end_k}]$ = GenRandData($[start_k, end_k], NC, sr_k$)
12: coef = $\rho(S_k[start_k, end_k], [d_{start_k}, d_{end_k}])$
13: Key_j = GenKey(coef)
14: $\text{KEY}_Q = \text{KEY}_Q \parallel \text{Key}_j$

C. SekGens Design

The SekGens algorithm has three phases: Quantization, Reconciliation, and Privacy-Amplification-and-Hashing. Both Enrollee and Registrar run quantization and privacy-amplification-and-hashing phases independently. The reconciliation phase, however, involves exchanging of specific messages between the two devices. Table I shows the notations we use in describing these phases. In the following, we give a detailed description of each phase of SekGens.

1) Quantization Phase: The quantization phase aims at generating the raw key KEY_Q at the Enrollee and Registrar devices independently. This phase starts when the Enrollee sends a random nonce *NC* to the Registrar before the eight messages of WPS protocol described in the Appendix. In SekGens, *NC* is used as a seed input for the uniform distribution function (*GenNum()*). Each time *NC* is used, it is incremented by one.

Algorithm 1 shows the pseudo code of the quantization phase. Lines 1-3 set the KEY_Q and all other keys Key_j's, which are generated from each iteration *j*, to null. This initialization is necessary to delete the previous session keys each time SekGens is initiated. The entropy $H(S_i[start_i, end_i])$ of each sensor *i* is calculated as shown in line 5. These entropy values are used to decide which sensor is selected to generate Key_j in each iteration. The sensor, which has a higher entropy for its data, has a higher probability to be involved in generating the KEY_Q. Line 6 normalizes all entropy values to be in the period *P*. Depending on the calculated entropy values, each sensor labels (line 7) a sub-period of *P* as it is illustrated in Table I. Assuming that there is a high level of similarities between the sensors data of both devices, the labeling of the *P* period on both devices is likely the same. Based on the normalization principle, the sensor with high entropy has a larger sub-period. As a result, sensors with high entropy values are more likely to be selected in each iteration for generating the partial key Key_j as a part of KEY_Q. Involving more sensors with high entropy values increases the

randomness of the generated bits of KEY_Q.

Lines 9-14 are responsible for generating the KEY_Q by creating Key_j in each iteration *j*. Firstly, a specific sensor, which is supposed to be the same on both devices, is selected. By using the same *NC* as an input to *GenNum()* function in line 9, both devices will generate the same random number *r*. In line 10, the two devices find the sub-period of *P* that contains *r*, then they select the corresponding sensor S_k, which labels that sub-period. Generating random data (line 11) in the period $[start_k, end_k]$ using the shared nonce *NC* and based on the sampling rate sr_k, will be the same on both devices.

The correlation coefficient, *coef*, is calculated in line 12 after performing the correlation between the random data $[d_{start_k}, d_{end_k}]$ and the sensor data $S_k[start_k, end_k]$ in the period $[start_k, end_k]$ on both devices. We assume that both devices have already agreed upon the same time interval $[start_k, end_k]$ in the collaboration phase and they share the same random data $[d_{start_k}, d_{end_k}]$ (line 11). Then if their sensors data $S_k[start_k, end_k]$ for sensor S_k is likely the same, then both devices compute the same *coef*. At each iteration in line 13, Key_j is generated using the *GenKey(coef)* function seeded with *coef*. The length of this key depends on the number of iterations *J*, SekGens produces 320 bits from the quantization phase. To increase the chance of involving more sensors in the quantization phase, we set *J* to 160 with the length of each Key_j = 2 bits. We justify in Section V why we need to generate 320 bits. Line 14 cumulatively concatenates all Key_j's to form the 320 bits of KEY_Q.

2) Reconciliation Phase: Reconciliation mechanisms can be used by any communicating devices to eliminate any minor bit-differences in their data using the Cascade protocol [12]. For our SekGens, the reconciliation is the second phase. Similar to the concept of cyclic redundancy check, the two devices have to exchange meta-data to identify any mismatching bits and reconcile their bit-strings. Meanwhile, they attempt to minimize the potential leakage of information about their bit-strings to an eavesdropper. When any mismatching bits are found, the two devices correct them accordingly.

The Cascade protocol aims at fixing the different bits in two n bit-strings ($n > 2$) by performing multiple parity checks and shuffling the bit sequence before each pass. The main part of the Cascade is the Binary protocol, which can be applied to any two equal length bit-strings ($s_1 \& s_2$) with different parities. By recursively applying the divide-and-conquer strategy, the Binary protocol keeps comparing the parities of the two bit-strings and is able to detect and correct one erroneous bit.

Before starting the Binary protocol, both devices have to find out if there are any differences in their KEY_Q's. Each device divides its KEY_Q into a number of blocks and calculates the even parity of each block, the size of each block is called the initial block size (denoted as k_0). The Enrollee sends the even parity of its first block and the Registrar compares that to the even parity of its first block, if they are different then the same process is applied to the successive halves of the block at both devices until the location of the erroneous bit is narrowed down by a binary search. If the parity bits are equal, the registrar sends the parity of its second block. The same process is repeated on all blocks, which we call the first pass of the reconciliation. For any two blocks (each of size n) that have different parity bits, the total number of exchanged messages needed to correct one erroneous bit is $\log_2(n) + 1$.

TABLE I: NOTATIONS AND DEFINITIONS

E: The Enrollee device.
R: The Registrar device .
N: Total number of shared sensors between the two devices.
J: Number of iterations in the quantization phase.
P: The labeled period ([0,100]). Each sensor labels a sub-period of P based on its entropy value.
NC: Nonce sent from Enrollee to Registrar.
$\mathbf{S_i}$: Sensor number *i*, $1 \leq \text{i} \leq N$.
$\boldsymbol{[start_i, end_i]}$: Time period of sensor S_i (in sec).
$\mathbf{S_i}\boldsymbol{[start_i, end_i]}$: Data of sensor number *i* during the time period: $[start_i, end_i]$.
$\mathbf{KEY_Q}$: The raw key generated from the quantization phase (320 bits).
$\mathbf{Key_j}$: Key generated from iteration number *j*. Its size depends on J (i.e., concatenating all Key_j's creates KEY_Q).
$\mathbf{KEY_{Rec}}$: The reconciled key resulted after the reconciliation phase (320 bits).
$\mathbf{KEY_{Final}}$: The final key resulted after the privacy-amplification-and-hashing phase (128 bits).
[1]$\boldsymbol{\rho(X, Y)}$: Pearson's Correlation Coefficient between two sets of data X and Y.
[2]$\boldsymbol{H(S_i[start_i, end_i])}$: Shannon Entropy of sensor S_i during the time period: $[start_i, end_i]$.
[3]$\boldsymbol{Norm(x_1, x_2, x_3 ... x_n)}$: Normalize a set of numbers $(x_1, x_2, x_3, ...x_n)$ into the period P.
[4]$\boldsymbol{Label(P, x_1 ... x_n)}$: Divide the period P into different sub-periods and label each one with a corresponding number x_i.
A $\|$ **B**: Concatenation A and B bit-strings.
r = GenNum(s): Generate a random number $r : 0 \leq r \leq 100$ using the uniform distribution function with seed = *s*.
$\mathbf{sr_i}$: Sample rate of sensor number *i* (in samples/sec).
GenRandData($\boldsymbol{[start_i, end_i]}$**, s,** $\boldsymbol{sr_i}$**)**: Generate random data in the $[start_i, end_i]$ period using the uniform distribution function with seed = *s* and sample rate = sr_i.
$\boldsymbol{[d_{start_i}, d_{end_i}]}$: Random data in the $[start_i, end_i]$ period.
GenKey(s): Generate a key for specific iteration (say *j*, Key_j) randomly using the uniform distribution function with seed = *s*. Each bit is generated independently.

• [1] The Pearson's correlation coefficient between X and Y sets of data, where $\overline{X}$ and $\overline{Y}$ are the mean values of X and Y respectively, is defined as:

$$\rho(X,Y) = \frac{\sum_{i=1}^{n} \left[(X_i - \overline{X}) \times (Y_i - \overline{Y})\right]}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2} \times \sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}} \quad (1)$$

• [2] The Shannon Entropy of sensor S_i during the $[start_i, end_i]$ time period, where *n* is the total number of sampled data and $p(y_i)$ is the probability of sample y_i, is defined as:

$$H(S_i[start_i, end_i]) = -\sum_{i=1}^{n} p(y_i) \times log_2 p(y_i) \quad (2)$$

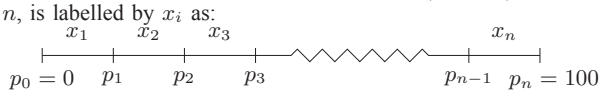
• [3] To normalize a set of numbers: $(x_1, x_2, x_3, ... x_n)$ into the period *P*, each x_i is normalized to a sub-period of *P* as:

$x_1 \xrightarrow{normalized\ to} [p_0, p_1]$, $x_2 \xrightarrow{normalized\ to} [p_1, p_2]$, $x_3 \xrightarrow{normalized\ to} [p_2, p_3]$, ... $x_n \xrightarrow{normalized\ to} [p_{n-1}, p_n]$.

Where p_i's are calculated as, let $\Lambda = \frac{100}{\sum_{i=1}^{n} x_i}$,:

$p_0 = 0, p_1 = p_0 + x_1 \times \Lambda, p_2 = p_1 + x_2 \times \Lambda$,... $p_n = p_{n-1} + x_n \times \Lambda$

• [4] After the normalization, each sub-period (p_{i-1}, p_i), s.t. $1 \leq i \leq n$, is labelled by x_i as:



Algorithm 2 Reconciliation of one pass at the Enrollee.

1: $blk\_num = 0$
2: $K = total\ number\ of\ blocks$
3: $current\_block \longrightarrow block_{blk\_num}$
4: $blk\_ord \longrightarrow position(current\_block)$
5: send $\langle pb\ of\ current\_block,\ blk\_num,\ blk\_ord\rangle$ to R
6: **while** $blk\_num < K$ **do**
7: wait for $\langle rcvd\_pb,\ rcvd\_blk\_num,\ rcvd\_blk\_ord\rangle$ from R
8: **switch** $(position(rcvd\_blk\_ord))$
9: **case** $\langle 1\ bit\rangle$**:**
10: correct the erroneous bit
11: $blk\_num = blk\_num + 1$
12: $current\_block \longrightarrow block_{blk\_num}$
13: **case** $\langle block_{blk\_num+1}\rangle$**:**
14: $blk\_num = blk\_num + 1$
15: $current\_block \longrightarrow block_{blk\_num}$
16: **if** error in $current\_block$ **then**
17: $current\_block \longrightarrow LSH\ of\ current\_block$
18: **else**
19: $blk\_num = blk\_num + 1$
20: $current\_block \longrightarrow block_{blk\_num}$
21: **case** $otherwise$**:**
22: $current\_block \longrightarrow position(rcvd\_blk\_ord)$
23: **if** error in $LSH\ of\ current\_block$ **then**
24: $current\_block \longrightarrow LSH\ of\ current\_block$
25: **else**
26: $current\_block \longrightarrow MSH\ of\ current\_block$
27: **end switch**
28: $blk\_ord \longrightarrow position(current\_block)$
29: send $\langle pb\ of\ current\_block,\ blk\_num,\ blk\_ord\rangle$ to R

In order to solve the masked error cases (i.e., two different blocks with equal even parity bits), the Enrollee and Registrar devices run one additional pass after randomly shuffling their KEY_Q's using the same random shuffling function seeded with *NC*. The initial block size for SekGens (i.e., k_0) is 8 bits and the block size for the second pass is 16 bits.

Algorithms 2 and 3 show the reconciliation steps of one pass at the Enrollee and Registrar devices, respectively. K represents the total number of blocks (i.e. if k_0 = 8 bits, $K = (320/8)$ blocks). blk_num variable contains the number of main blocks being reconciled by both devices (for our last example, $0 \leq blk\_num < 40$). Moreover, each device has a pointer $(current\_block)$ to the current block/sub-block being reconciled. blk_ord contains the first and last bit-index of $current\_block$, $position$ function returns these indexes.

The Enrollee device starts the protocol (line 5) by sending the parity bit, block number and block order of $block_0$ ($\langle pb\ of\ current\_block,\ blk\_num,\ blk\_ord\rangle$) to the Registrar device. The Registrar checks (line 13) the parity bit of its first block ($block_0$) and finds out if it is different from the received parity ($rcvd\_pb$), if so, the Registrar sends the parity bit of the left significant half (LSH) of its $current\_block$. Otherwise, it increments the block number to point to the next block and it sends its parity bit to the Enrollee.

If two blocks have different parities, both devices narrow down the position of the erroneous bit by using binary search (**case** *otherwise*), until it is located and corrected (**case** $\langle 1\ bit\rangle$). Both devices run each pass of the protocol on all blocks (until $blk\_num = K$). When moving to next block (**case** $\langle block_{blk\_num+1}\rangle$) and in case of erroneous bit, the receiving device can arbitrary select to send the parity bit of either the left significant half (LSH) or most significant half (MSH) of that block, we choose to send the (LSH) of the block (lines: 17 & 22 in Algorithms 2 & 3, respectively).

Algorithm 3 Reconciliation of one pass at the Registrar.

1: $blk\_num = 0$
2: $K = total\ number\ of\ blocks$
3: $current\_block \longrightarrow block_{blk\_num}$
4: $blk\_ord \longrightarrow position(current\_block)$
5: **while** $blk\_num < K$ **do**
6: wait for $\langle rcvd\_pb,\ rcvd\_blk\_num,\ rcvd\_blk\_ord\rangle$ from E
7: **switch** $(position(rcvd\_blk\_ord))$
8: **case** $\langle 1\ bit\rangle$**:**
9: correct the erroneous bit
10: $blk\_num = blk\_num + 1$
11: $current\_block \longrightarrow block_{blk\_num}$
12: **case** $\langle block_{blk\_num}\rangle$**:**
13: **if** error in $current\_block$ **then**
14: $current\_block \longrightarrow LSH\ of\ current\_block$
15: **else**
16: $blk\_num = blk\_num + 1$
17: $current\_block \longrightarrow block_{blk\_num}$
18: **case** $\langle block_{blk\_num+1}\rangle$**:**
19: $blk\_num = blk\_num + 1$
20: $current\_block \longrightarrow block_{blk\_num}$
21: **if** error in $current\_block$ **then**
22: $current\_block \longrightarrow LSH\ of\ current\_block$
23: **else**
24: $blk\_num = blk\_num + 1$
25: $current\_block \longrightarrow block_{blk\_num}$
26: **case** $otherwise$**:**
27: $current\_block \longrightarrow position(rcvd\_blk\_ord)$
28: **if** error in $LSH\ of\ current\_block$ **then**
29: $current\_block \longrightarrow LSH\ of\ current\_block$
30: **else**
31: $current\_block \longrightarrow MSH\ of\ current\_block$
32: **end switch**
33: $blk\_ord \longrightarrow position(current\_block)$
34: send $\langle pb\ of\ current\_block,\ blk\_num,\ blk\_ord\rangle$ to E

Algorithm 4 Error-Estimation at the Enrollee & Registrar.

if Enrollee **then**
send $\langle POS,\ E\_subsets \subseteq KEY_{Rec}\rangle$ to R
wait for $R\_subsets$ from R
calculate e_{obs}
if $e_{obs} > 0.0$ **then**
abort
else
continue to Privacy-Amplification-and-Hashing phase
if Registrar **then**
wait for $\langle POS,\ E\_subsets \subseteq KEY_{Rec}\rangle$ from E
send $\langle R\_subsets \subseteq KEY_Q\rangle$ to E
calculate e_{obs}
if $e_{obs} > 0.0$ **then**
abort
else
continue to Privacy-Amplification-and-Hashing phase

When both devices finish the two passes, they have to find out whether their KEY_{Rec}'s are equal or not. Therefore, they run the last sub-phase of the reconciliation (Error-Estimation). The estimating of the discrepancy between their KEY_{Rec}'s is done by exchanging random subsets of bits picked randomly from their KEY_{Rec}'s. The Enrollee starts the error-estimation sub-phase (Algorithm 4) by sending a random subsets ($E\_subsets$) of its KEY_{Rec} and their positions (POS) to the Registrar. The Registrar replies by sending its subsets

($R\_subsets$) from the same positions of its KEY_{Rec}. The size of each subset is 32 bits. After that, both devices calculate the observed error rate (e_{obs}). If e_{obs} is greater than zero, then both devices stop establishing the secure channel. Else, they pursue by starting the privacy-amplification-and-hashing phase.

3) Privacy-Amplification-and-Hashing Phase: This phase aims at improving the privacy of the KEY_{Rec}'s in the Enrollee and Registrar devices by omitting all corrected bits during the reconciliation phase and all random subsets (32 bits) of the KEY_{Rec}'s exposed during the error-estimation sub-phase. This omission is needed for removing eavesdropper's knowledge of the KEY_{Rec}'s. Finally, both devices apply the MD5 hashing on the remaining secret bits to form the final 128 bits KEY_{Final}.

IV. Implementation & Evaluation

A. Implementation

In order to evaluate SekGens algorithm, we implemented it on Google Nexus 5 and Samsung Galaxy S2 smartphones. On these devices, the Android kernel code, that is responsible for creating the Wi-Fi Direct connection between any two devices, implements the WPS protocol. We downloaded, modified and built the CyanogenMod [2] Android kernel code for Google Nexus 5 and Samsung Galaxy S2 smartphones.

In order to generate sensors data on the smartphones, we installed the "Accelerometer Monitor" [3], which is an Android app that uses the vibration sensor (accelerometer). This app generates three sensors data, the device acceleration data for each space axis: X, Y and Z (m/s^2). SekGens uses the value of $\sqrt{X^2+Y^2+Z^2}$ at each reading. We also installed the "Sound Meter" [3], which is Android app that uses the microphone to measure sound levels in dB (decibels). Each app stores its data in a log file that is saved on the SD card.

The quantization phase of SekGens is initiated when the user navigates to Wi-Fi Direct screen and searches for nearby available Wi-Fi Direct devices. We modified the WPS protocol such that the first message sent from the Enrollee to the Registrar contains the *NC*. Then, both devices start the quantization phase after reading the sensors data from the SD card.

For the reconciliation phase, in order to enable both devices to exchange parity bit messages, block number, block order of the current block being reconciled, and the random subsets of the reconciled KEY_Q, we defined new message types to the existing WPS protocol. Also, to enable the two devices to exchange reconciliation messages, we set the value of "EAP_MAX_AUTH_ROUNDS" variable to 200. After compiling the modified Android kernel code, we downloaded it on smartphones using the Android Debug Bridge [1] tool.

B. Performance Evaluation

We evaluated our SekGens when the Enrollee and Registrar are smartphones of either the same or different hardware components using the following metrics, commonly used to evaluate the performance of secret key generation schemes:

1) **Pairwise Key Mismatch:** the fraction of different bits of the generated key at both ends, ideally 0%.

2) **Key Entropy:** this metric (shown in Equation 2) measures the randomness of the generated keys. A value close to 1.0 indicates high entropy with high random keys.

3) **SekGens Execution Time (msec):** the time delay resulted from running SekGens in smartphones.

TABLE II: TEST CASES

Test Case Number	Movement Shape	Distance Between Smartphones (m)	During Lunch Hour
TC_1	Bump	0	No
TC_2	Bump	0	Yes
TC_3	Bump	1	No
TC_4	Bump	1	Yes
TC_5	Bump	2	No
TC_6	Bump	2	Yes
TC_7	Handshake	0	No
TC_8	Handshake	0	Yes
TC_9	Handshake	1	No
TC_{10}	Handshake	1	Yes
TC_{11}	Handshake	2	No
TC_{12}	Handshake	2	Yes

4) **Secret Bit Rate:** the average number of secret bits extracted from sensors per second. This depends on sensors sampling rate (i.e., 50 and 5 sample(s)/sec for "Accelerometer Monitor" and "Sound Meter", respectively) and SekGens execution time.

5) **Key Generation Diversity:** the key mismatch ratio among different successful runs of SekGens. Even if SekGens successfully generates keys with 0% mismatch ratio, it is still important to show that SekGens is able to generate very random keys for each run.

We are interested in the impact of the following on the performance of SekGens: a) the amount of sensor data collected for key generation; b) the type of sensor data used for key generation. For this, we performed the 12 test cases shown in Table II. In each test case, the users either bump or handshake their smartphones together for 2 to 3 seconds and then they keep their smartphones either close (i.e., 0 m distance) or distant (i.e., 1 m or 2 m distances). These test cases were conducted in a cafeteria during and after a busy lunch hour (12:00 PM - 1:00 PM).

For each test case, we repeated the experiment five times and we averaged these five runs per each test case. The users start each experiment together by starting the "Accelerometer Monitor" app on their smartphones at the same time. However, the "Sound Meter" app was always running on each device. SekGens starts reading its output from the time when the users started the experiment.

1) Impact of Experiment Time Duration on KEY_Q Mismatch: In order to investigate the best time duration for each experiment, we performed all experiments for all test cases for different time duration using two Google Nexus 5 smartphones. Figures 1(a) and 1(b) show the mismatch ratio of KEY_Q's for all test cases. We found that the mismatch ratio of KEY_Q is decreasing when we increase the time duration of each experiment, however, this mismatch ratio does not improve after 6 seconds. We aim at not increasing the experiment time duration because this lowers the secret bit rate of SekGens. We set the time duration for each experiment to 6 seconds and we guarantee that by reading the first 300 and 30 sensors data from the log files of "Accelerometer Monitor" and "Sound Meter", respectively.

2) SekGens on Same Smartphones: To perform the experiments on smartphones of the same hardware, we used two Google Nexus 5 smartphones to find out the impact of the type of data Enrollee and Registrar use to establish the KEY_{Final}

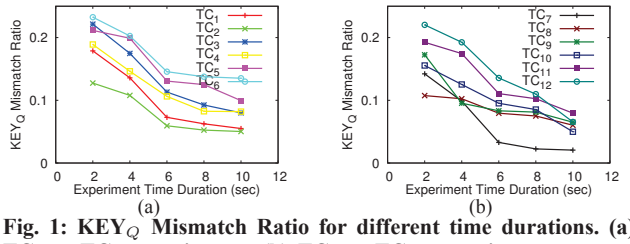


Fig. 1: KEY_Q Mismatch Ratio for different time durations. (a) TC_1 to TC_6 experiments. (b) TC_7 to TC_{12} experiments.

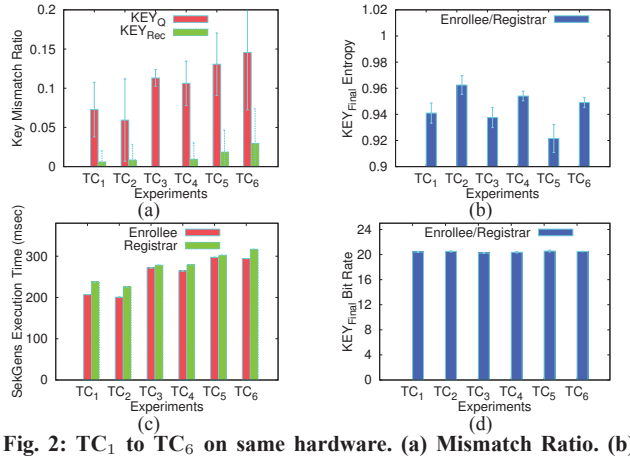


Fig. 2: TC_1 to TC_6 on same hardware. (a) Mismatch Ratio. (b) KEY_{Final} Entropy. (c) Execution Time. (d) KEY_{Final} Bit Rate.

for their devices. We compare the performance of SekGens in terms of the metrics we mentioned above for all test cases. Figures 2 and 3 show the evaluation results of our metrics for all test cases, with error bars showing standard deviation.

Figures 2(a) and 3(a) show the *key mismatch ratio* for both KEY_Q's and KEY_{Rec}'s. This represents the mismatch ratio after the quantization phase (KEY_Q) and the effect of the reconciliation phase on improving the key agreement between the Enrollee and Registrar devices (KEY_{Rec}). The mismatch ratio for all experiments after the reconciliation phase is less than 2%. Out of 60 experiments for all test cases, there were 53 experiments in which SekGens could generate the same KEY_{Rec} on both devices (i.e., the success rate is 88%).

The *key mismatch ratio* increases for 1 m and 2 m distances between the two smartphones. This increase in the mismatch ratio is due to the fact that the readings of the "Sound Meter" app depend mainly on the location of the smartphones (i.e., the readings are different for distinct locations). However, this mismatch ratio is reduced after the reconciliation phase.

Figures 2(b) and 3(b) show the *entropy* of KEY_{Final} resulted from the successful experiments of all test cases. The generated KEY_{Final}'s entropy is higher than 92% for all experiments, which indicates a high level of randomness of KEY_{Final}. The KEY_{Final} entropy values resulted from the handshake experiments (Figure 3(b)) are higher compared to the bump experiments (Figure 2(b)). This is due to the higher entropy of "Accelerometer Monitor" readings when the users handshake their smartphones rather than bump them. Also, the KEY_{Final} entropy values are higher for the experiments that were conducted during the lunch hour. This is due to the higher entropy of "Sound Meter" readings for these experiments.

Figures 2(c) and 3(c) show the *execution time* of SekGens needed to generate the KEY_{Final} for the successful experiments of all test cases. The execution time is higher

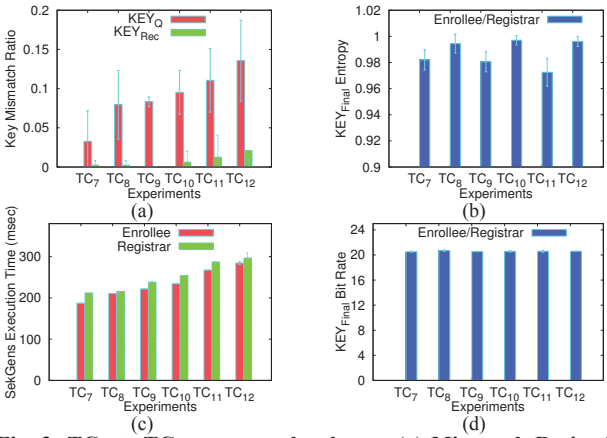


Fig. 3: TC_7 to TC_{12} on same hardware. (a) Mismatch Ratio. (b) KEY_{Final} Entropy. (c) Execution Time. (d) KEY_{Final} Bit Rate.

for the test cases with higher KEY_Q mismatch ratio due to the additional reconciliation messages needed to reconcile the KEY_Q. In all experiments, the execution time for the Registrar is higher than the execution time for the Enrollee. This is due to an implementation choice of SekGens in which the Enrollee is the terminator of the last phase of SekGens that is the privacy-amplification-and-hashing phase. The Enrollee finishes SekGens when it applies the MD5 hashing to form its KEY_{Final} and then it sends message M_1 of WPS protocol to the Registrar as an indication of pursuing the communication. However, the Registrar still needs to apply the MD5 hashing to form its KEY_{Final} before finishing SekGens, then it replies with message M_2 of WPS protocol.

The KEY_{Final} *bit rate* for the successful experiments of all test cases are shown in Figures 2(d) and 3(d). Because it is longer than the Enrollee, we use the execution time of the Registrar in calculating the KEY_{Final} bit rate. Even though this execution time is less than 1 sec, the time needed to generate sensors data (i.e., 6 seconds) dominates the bit rate and lowers it to be around 20 bits/sec. This KEY_{Final} bit rate might be increased with sensors with higher sampling rate.

3) SekGens on Different Smartphones: To investigate the robustness of SekGens on smartphones with different hardware components, we repeated the 12 test cases but on different smartphones (i.e., Google Nexus 5 and Samsung Galaxy S2). Figures 4 and 5 show the evaluation results of these experiments, with error bars showing standard deviation.

Figures 4(a) and 5(a) show the *key mismatch ratio* for both KEY_Q's and KEY_{Rec}'s for all test cases. The mismatch ratio for all experiments after the reconciliation phase is less than 3%, which proves that SekGens is robust when running on different smartphones. Out of 60 experiments for all test cases, there were 45 experiments in which SekGens could successfully generate the same KEY_{Rec} on both smartphones.

The *entropy* of KEY_{Final} for the successful experiments of all test cases are shown in Figures 4(b) and 5(b). SekGens creates KEY_{Final} with high entropy (i.e., $>$ 92%). The *execution time* of SekGens for the successful experiments is between 200 and 350 msec as shown in Figures 4(c) and 5(c). And the KEY_{Final} *bit rate* for the same experiments is around 20 bits/sec as shown in Figures 4(d) and 5(d).

By comparing the performance of SekGens to the most related previous works [18] and [11], SekGens is better in

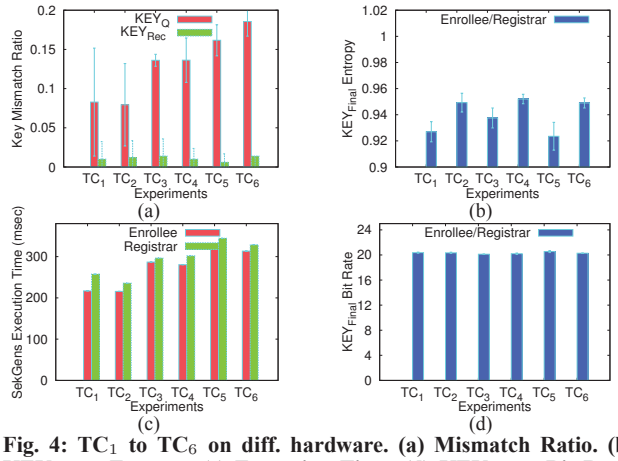


Fig. 4: TC_1 to TC_6 on diff. hardware. (a) Mismatch Ratio. (b) KEY_{Final} Entropy. (c) Execution Time. (d) KEY_{Final} Bit Rate.

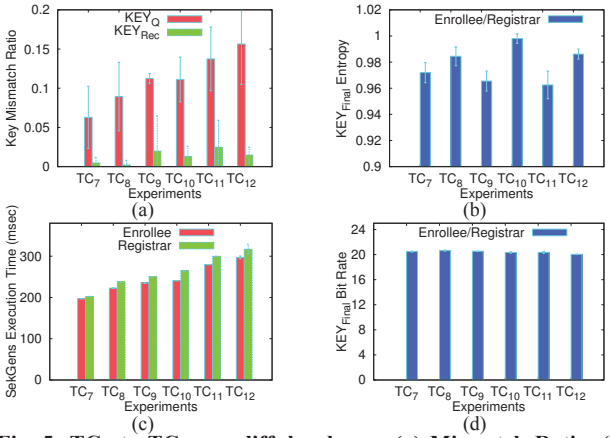


Fig. 5: TC_7 to TC_{12} on diff. hardware. (a) Mismatch Ratio. (b) KEY_{Final} Entropy. (c) Execution Time. (d) KEY_{Final} Bit Rate.

terms of success rate, key bit rate, execution time, security against offline attacks and robustness on different hardware sensors. We will discuss that in more details in Section VI.

4) Key Diversity Validation for SekGens: To ensure the randomness of the generated KEY_{Final} bit streams by SekGens, we run the randomness tests available in the NIST test suite [8]. Even though there are 15 different statistical tests in the NIST test suite, we run only 8 tests due to the fact that the KEY_{Final} bit streams that we obtain from SekGens meet the input size recommendations of the 8 NIST tests only. We find that the SekGens generated KEY_{Final} bit streams pass all the 8 tests as shown in Table III. The remaining 7 tests require a very large input bit stream ($\approx 10^6$ bits).

V. Security Analysis

A. Defending against the brute-force attack

Claim 1. *The attacker, Eve, has to perform 2^{128} guesses in order to mount a brute-force attack against the KEY_{Final} generated from SekGens.*

Proof. To prove Claim 1., we have to show that if Eve tries to mount a brute-force attack against any key generated from any phase of SekGens, the difficulty of her guessing is at least equal to the difficulty of guessing a 128 bits key.

• **The quantization phase** generates the KEY_Q that is 320 bits. There are 160 iterations, at each iteration j, Key_j with length 2 bits is generated using the correlation coefficient, *coef*, which can take any value in the interval [-1.0, 1.0]. In our implementation, we multiply *coef* by 10 and round it to the nearest integer before we use it as a seed input to the *GenKey(coef)* function. So, Eve has two ways to brute force KEY_Q. Either by guessing different seed inputs, in this case she has to try 20^{160} guesses. Or by guessing each Key_j separately, in this case she has to try $(2^2)^{160}$ guesses.

TABLE III: NIST statistical test suite results. The *p* value from each test for the successful experiments of all test cases is shown below. To pass a test, the *p* value for that test has to be > 0.01.

Statistical Test	Section IV-B2 Experiments	Section IV-B3 Experiments
Frequency	0.03	0.04
Block Frequency	0.28	0.04
Cumulative Sums (Fwd)	0.28	0.04
Cumulative Sums (Rev)	0.04	0.02
Runs	0.05	0.04
Longest Run of Ones	0.06	0.20
Approximate Entropy	0.13	0.70
Serial	0.04, 0.25	0.02, 0.04

• **The reconciliation phase** might make Eve's task easier, when considering the exposed parity bit messages and the 32 bits that are needed for the error-estimation sub-phase. If Eve is able to discover the secret KEY_{Rec}, she can apply the MD5 hashing function to generate KEY_{Final}. Eve can sniff the parity bit messages ($\langle pb\ of\ current\_block, blk\_num, blk\_ord\rangle$), the subsets ($E\_subsets$ and $R\_subsets$) of KEY_{Rec}, as well as the random nonce *NC* before she starts guessing the KEY_{Final}.

For each captured reconciliation message, Eve can omit half of the guessing space of the block being reconciled for both Enrollee and Registrar devices. After exchanging 4 messages ($log_2(8) + 1$) for each 8 bit block, Eve ends up with 2^4 guesses for the block being reconciled, and, thus, she has to try 2^4 guesses for each 8 bits block. The ideal scenario for Eve is when the Enrollee and Registrar devices have different parities for all their 8 bit blocks. In that case, Eve can reduce its guessing space for all 8 bit blocks. Moreover, Eve knows that there are 32 bits that are needed for error-estimation.

We have to find the length of KEY_{Rec} such that after Eve narrows its guessing space based on the captured parity bits messages, and omits the 32 error-estimation bits, so that she still needs to make at least 2^{128} guesses. Finding the value of *Number_Of_Blocks* such that $\frac{(2^4)^{Number\_Of\_Blocks}}{2^{32}} \geq 2^{128}$ is satisfied guarantees that the difficulty of guessing the remaining secret bits after the reconciliation phase is at least equivalent to the difficulty of guessing a 128 bits key. The numerator in the above equation represents the number of guesses for each 8 bit block, when we have *Number_Of_Blocks* blocks. And the denominator represents the 32 error-estimation bits which are known by Eve. When solving the above equation, *Number_Of_Blocks* = 40, we found that the length of KEY_{Rec} (*Number_Of_Blocks* $\times$ *block size*) has to be at least 320 bits (40×8). That is why the length of KEY_Q is 320 bits.

B. Defending against the imitation attack

In order to accomplish the imitation attack, Eve tries to generate the same sensors data of the Enrollee or Registrar devices. We used two sensors in Section IV-B. For the sound sensor, the sound level (chapter 2 of [10]) in the free space environments (i.e., the open air or the anechoic rooms) follows

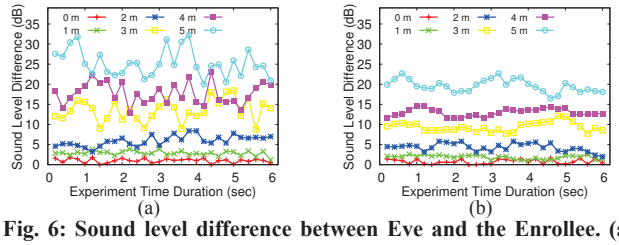


Fig. 6: Sound level difference between Eve and the Enrollee. (a) During Lunch Hour. (b) After Lunch Hour.

the inverse square law (i.e., the sound level will decrease 6.02 dB each time the distance from the source is doubled). In such environments, it might be easy for Eve to predict the sound level at the Enrollee location and fake its sound sensor data to pretend as the legal Enrollee.

In real-world, Eve prefers to attack the victims in crowded indoor areas (e.g., restaurants, cafes, airports lounges, or student community areas), where it is difficult to be noticed. In such crowded indoor areas, there are many sound sources (i.e., music, talking people and airport announcements) and the sound wave for each source is reflected and diffracted (chapter 10 of [20]) many times due to the walls, doors, tables and moving objects (i.e., customers and waiters in cafes and restaurants, passengers in airports lounges, students in student community areas). As a result, the inverse square law is not strictly applicable and the sound level is very random. Therefore, it is difficult for Eve to predict the sound level values unless she (physically) gets closer to the victims to have the same readings.

In order to investigate the distance that enables Eve to generate the same sound readings as the Enrollee, we performed six experiments shown in Figure 6 at different distances between Eve and the Enrollee to calculate the differences between their sound readings. For each distance, we repeated the experiment five times and we averaged these five runs per distance. These experiments were conducted in a cafeteria during and after a busy lunch hour.

As shown in Figures 6(a) and 6(b), in order to make the sound readings for Eve $\approx$ the sound readings generated at the Enrollee location, Eve has to be close within 1 m distance to the Enrollee. Eve must depend on the reconciliation phase to eliminate the minor differences in the bits of its KEY_Q and the Enrollee KEY_Q. However, Eve can not get so close (i.e., $\sim$1 m) to the victims, otherwise, the victims will notice and discover her especially if she tries to imitate the same gesture (i.e., bump, handshake, etc.) made by the victims to generate data from the acceleration sensors. For more than 1 m distances, the difference in the sound readings increases and this makes Eve's task of discovering KEY_Q more difficult.

Accordingly, we recommend the users to be within 1 m distance when they use SekGens. Indeed, 93 out of the total 98 successful experiments in Sections IV-B2 and IV-B3 are conducted when the Enrollee and Registrar devices are within 1 m distance. If Eve conceals herself in the environment and she is a skilful attacker such that she is able to imitate the exact gesture made by the victims, it is difficult for her to generate the same sound readings generated from the victims devices.

VI. State of the Art

Some prior work that is related to ours is "Smart its friends" [14] and "Are you with me" [17], which use the accelerometer sensors in device pairing. "Smart its friends" uses sensor data to connect smart-artefact devices when a user shakes them together. In [17], accelerometer data are used to check if two ubiquitous devices are carried by the same person.

The extended work from [14] and [17] that has the closest relation to ours is the Candidate Key Protocol (CKP) [18]. In CKP, the devices generate an encryption key from acceleration data by extracting feature vectors from the inputs. Then each feature vector is hashed using a standard hash function and sent to the other device. Depending on the similarity of the received and local hashed vectors, the devices determine if they are shaken together, and they use the identical hashed vectors to create the encryption key. The key bit rate of CKP is 7 bits/sec, less than ours (i.e., 20 bits/sec). A major drawback of CKP is that both devices have to exchange their own derived key parts, which makes it susceptible to offline brute-force attacks. The security analysis is missing in [18]. CKP has no reconciliation phase and it requires very low environmental noises to produce enough identical vectors between the devices. Also, there is no clear answer to the question of how large hashed vectors can be ignored due to unequal values without endangering the security and without decreasing the key bit rate.

Another work presented in [11] aims at generating a cryptographic key by applying appropriate signal processing methods on the acceleration data of small hand-held devices. Compared to [18], the key generation algorithm in [11] does not require the two devices to exchange their acceleration characteristics. However, this algorithm aims at creating a symmetric key that is equally as strong as the typical Bluetooth PIN (i.e., three to four digits that range from 0 to 9). The proposed algorithm generates an equivalent key with 13 bits (i.e., the entropy of Bluetooth PIN of three to four digits $(10^3 \sim 10^4) \approx$ the entropy of 13 bits key (2^{13})). Even though the success rate of generating this 13 bits key is 80%, this key is vulnerable to brute-force attacks in Wi-Fi Direct. Also, the longest achievable key in this approach, which is 140 bits, has a low success rate (i.e., 1/88).

S. Jane et al. [15] evaluated the effectiveness of secret key extraction from the received signal strength (RSS) variations in wireless channels using real world measurements in static and dynamic environments. Their results showed that the static environments are unsuitable for generating a secret key (i.e., the mismatch ratio is around 50%). However, the key bits extracted in dynamic environments showed a higher secret bit rate with a very low mismatch ratio. The dynamic experiments require either a normal speed walking for 10 to 25 feet, riding a bike on a city street, or connect the devices in a crowded cafeteria or across a busy road. These dynamic requirements might not be available or convenient for Wi-Fi Direct users.

S. Ali et al. [9] presented a method for generating shared secret keys for body-worn health monitoring devices based on the motion of these devices. This method has no reconciliation phase and it depends on the received signal strength indicator (RSSI), a measure of signal power in logarithmic units, to quantize the key. This method takes 15 to 35 minutes to generate a 128 bits key with a 75% chance of perfect agreement between endpoints. This long time to extract the key might be acceptable in health monitoring devices, but it is very long in case of Wi-Fi Direct communications.

M. Miettinen et al. [19] presented an approach for secure zero-interaction pairing suitable for Internet-of-Things and

wearable devices. Their scheme uses sensed context fingerprints to evolve the pairing key periodically in a way that is only possible for devices co-present over extended periods of time. They use readily available context sensor modalities like audio and luminosity. Their approach is not suitable for Wi-Fi Direct because it requires long time periods (i.e., hours).

VII. Conclusions

We presented a novel algorithm Session Key Generated from Sensors (SekGens) for generating shared secret keys using contextual information in smartphones. Our main contribution has been to exploit the contextual sensors data on two communicating smartphones to generate a 128 bits secure session key in a short time, which mainly depends on the time needed to generate sensors data. We used SekGens to solve the WPS PIN brute force vulnerability. SekGens has three phases: Quantization, Reconciliation and Privacy-Amplification-and-Hashing. We implemented these phases on WPS Android kernel code, which is responsible for creating Wi-Fi Direct connections, for Google Nexus 5 and Samsung Galaxy S2 smartphones. The evaluation results prove that SekGens is efficient in generating keys with low mismatch ratio, at a fast bit rate, and with high Shannon entropy.

Acknowledgment

This work was funded by the German Jordanian University.

References

[1] Android debug bridge, http://developer.android.com/tools/help/adb.html.

[2] Building cyanogenmod android code, http://wiki.cyanogenmod.org/.

[3] Google play store, https://play.google.com/store.

[4] Penetration tools for network security, http://www.diva-portal.org/smash/get/diva2:694738/fulltext02.

[5] Reaver open source, https://code.google.com/p/reaver-wps/.

[6] Waidps, wireless auditing intrusion detection and prevention system.

[7] Wi-fi protected setup specification version 1.0h. 2006.

[8] Nist. a statistical test suite for random and pseudorandom number generators for cryptographic applications. 2010.

[9] S. Ali, V. Sivaraman, and D. Ostry. Zero reconciliation secret key generation for body-worn health monitoring devices. *WiSec*, 2012.

[10] G. Ballou. Handbook for sound engineers, 4th edition. 2008.

[11] D. Bichler, G. Stromberg, M. Huemer, and M. Low. Key generation based on acceleration data of shaking processes. *UbiComp*, 2007.

[12] G. Brassard and L. Salvail. Secret-key reconciliation by public discussion. *EUROCRYPT*, 1994.

[13] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano. Device-to-device communications with wi-fi direct: overview and experimentation. *IEEE Wireless Communications Magazine*, 20(3), 2013.

[14] L. E. Holmquist, F. Mattern, B. Schiele, P. Alahuhta, M. Beigl, and H.-W. Gellersen. Smart-its friends: A technique for users to easily establish connections between smart artefacts. *UbiComp*, 2001.

[15] S. Jana, S. N. Premnath, M. Clark, S. K. Kasera, N. Patwari, and S. V. Krishnamurthy. On the effectiveness of secret key extraction from wireless signal strength in real environments. *MobiCom*, 2009.

[16] T. Kwon, S. Shin, and S. Na. Covert attentional shoulder surfing: Human adversaries are more powerful than expected. *IEEE Transactions on Systems, Man, and Cybernetics*, 44(6), 2013.

[17] J. Lester, B. Hannaford, and G. Borriello. Are you with me? using accelerometers to determine if two devices are carried by the same person. *Pervasive*, 2004.

[18] R. Mayrhofer and H. Gellersen. Shake well before use: authentication based on accelerometer data. *Pervasive*, 2007.

[19] M. Miettinen, N. Asokan, T. D. Nguyen, A.-R. Sadeghi, and M. Sobhani. Context-based zero-interaction pairing and key evolution for advanced personal devices. *CCS*, 2014.

[20] G. Mller and M. Mser. Handbook of engineering acoustics. 2013.

[21] A. Sanatinia, S. Narain, and G. Noubir. Wireless spreading of wifi aps infections using wps flaws: An epidemiological and experimental study. *CNS*, 2013.

[22] S. Viehbck. Wifi protected setup (wps) pin brute force vulnerability, cert vulnerability note vu#723755. 2011.

[23] D. Zisiadis, S. Kopsidas, A. Varalis, and L. Tassiulas. Enhancing wps security. *WD*, 2012.

Appendix

The WPS messages (page 33 of [7]) between the Registrar (R) and the Enrollee (E) are:

1) **E→R:** M_1=Version||N_1||Description||PK_E

2) **E←R:** M_2 = Version || N_1 || N_2 || Description || PK_R [|| ConfigData] || $HMAC_{AuthKey}(M_1 \parallel M_2^*)$

3) **E→R:** M_3 = Version || N_2 || E-Hash1 || E-Hash2 || $HMAC_{AuthKey}(M_2 \parallel M_3^*)$

4) **E←R:** M_4 = Version || N_1 || R-Hash1 || R-Hash2 || $ENC_{KeyWrapKey}$(R-S1) || $HMAC_{AuthKey}(M_3 \parallel M_4^*)$

5) **E→R:** M_5 = Version || N_2 ||$ENC_{KeyWrapKey}$(E-S1) || $HMAC_{AuthKey}(M_4 \parallel M_5^*)$

6) **E←R:** M_6 = Version || N_1 || $ENC_{KeyWrapKey}$(R-S2) || $HMAC_{AuthKey}(M_5 \parallel M_6^*)$

7) **E→R:** M_7 = Version || N_2 || $ENC_{KeyWrapKey}$(E-S2 [|| ConfigData]) || $HMAC_{AuthKey}(M_6 \parallel M_7^*)$

8) **E←R:** M_8 = Version || N_1 || [$ENC_{KeyWrapKey}$ (ConfigData)] || $HMAC_{AuthKey}(M_7 \parallel M_8^*)$

Such that:

- ||: concatenation of parameters to form a message.
- **Subscript**: identifies the key used by a cryptographic function.
- $\mathbf{M_n^*}$: message M_n excluding the HMAC-SHA-256 value.
- **Version**: identifies the type of the Registration Protocol message.
- $\mathbf{N_1}$, $\mathbf{N_2}$: 128 bits random nonces from E and R, respectively.
- **Description**: description and capabilities of the sending device.
- $\mathbf{PK}_E$ & $\mathbf{PK}_R$: Diffie-Hellman public keys of E and R, respectively.
- **AuthKey**: an authentication key derived from the Diffie-Hellman secret g^{AB} mod p, MAC address of E, and N_1 and N_2 nonces.
- **E-Hash1**, **E-Hash2**, **R-Hash1**, **R-Hash2**: commitments of E & R.
- $\mathbf{ENC_{KeyWrapKey}(...)}$: encrypted message using KeyWrapKey.
- **R-S1**, **R-S2**, **E-S1**, **E-S2**: 128 bits secret nonces.
- $\mathbf{HMAC_{AuthKey}(...)}$: HMAC hashed message using AuthKey.
- **ConfigData**: WLAN settings and credentials for the Enrollee.

E-Hash1 and E-Hash2 in M_3 are computed by the Enrollee after creating two secret nonces E-S1 and E-S2:

E-Hash1 = $HMAC_{AuthKey}$(E-S1 || PSK1 || PK_E || PK_R).

E-Hash2 = $HMAC_{AuthKey}$(E-S2 || PSK2 || PK_E || PK_R).

Also, R-Hash1 and R-Hash2 in M_4 are computed by the Registrar after creating two secret nonces R-S1 and R-S2:

R-Hash1 = $HMAC_{AuthKey}$(R-S1 || PSK1 || PK_E || PK_R).

R-Hash2 = $HMAC_{AuthKey}$(R-S2 || PSK2 || PK_E || PK_R).

PSK1 and PSK2 are calculated by converting the device password into two 128 bits values as follows:

PSK1 = first 128 bits of $HMAC_{AuthKey}$(1st half of Device Password), **(i.e., the 1st half of the key that SekGens will create)**.

PSK2 = first 128 bits of $HMAC_{AuthKey}$(2nd half of Device Password), **(i.e., the 2nd half of the key that SekGens will create)**.