Spread Spectrum Communication without any Pre-shared Secret

Aldo Cassola, Tao Jin, Guevara Noubir, Bishal Thapa*†

October 5, 2010

Abstract

Spread spectrum (SS) communication relies on the assumption that some secret is shared beforehand among communicating nodes in order to establish the spreading sequence for long-term wireless communication. Strasser et al. identified the circular dependency problem (CDP) that arises, since a method designed to communicate secret messages requires a pre-shared secret itself [22]. The problem is exacerbated in large networks where nodes join and leave the network frequently, and sharing the secret through physical hand-sake is infeasible. Thus, the secret is shared over the air, making the key establishment susceptible to targeted attacks by adversaries. In this work, we introduce Time Reversed Message Extraction and Key Scheduling (TREKS) to enable SS communication without requiring such pre-shared secrets. Based on two novel techniques we call *intractable forward decoding* and *efficient backward decoding*, TREKS is able to outperform previous solutions to the CDP [22, 21, 15] by four orders of magnitude. Our approach can not only be used for the initial round of exchange to carry the challenge of an authentication and key establishment protocol, but also for long-term communication without establishing keys. The energy cost under TREKS is provably optimal with very small storage and computation overhead, and its message decoding cost is at most twice as under traditional SS. We evaluate TREKS through simulation and real-world experimentation with USRP, GNURadio, and GPU-equipped nodes. With TREKS, using a modest setup can sustain a 1Mbps long-term communication spreading by a factor of 100 (i.e., 100 Megachips per sec) over a 200MHz bandwidth.

1 Introduction

Radio-Frequency (RF) wireless communication occurs through the propagation of electro-magnetic waves over a broadcast medium. Such broadcast medium is not only shared between the communicating nodes but is also exposed to adversaries. Adversaries could simply be just eavesdroppers trying to get hold of some important information or attackers trying to prevent the communication from happening by jamming. Regardless, the resiliency to the malicious behavior of these adversaries is obviously of significant importance for military communication in a battle-field. However, it is also rapidly gaining significance in civilian and commercial applications due to the increased reliance on wireless networks for connectivity to the cyber-infrastructure, and applications that will monitor our physical infrastructure such as tunnels, bridges, and buildings. Security for privacy issues in communication have been in the forefront of networks research for a long time. The study of jamming and anti-jamming techniques are, however, more recent. The impact of such attacks in the physical layer of wireless systems supporting mostly voice communication have been extensively studied [19]. But, it is only recently that the popularity of multi-hop data networks with more sophisticated medium sharing, coding, and application protocols have opened the door for more sophisticated attacks and resulted in the exploration of new resilience mechanisms at the link, network, and upper layers.

*A. Cassola, T. Jin, G. Noubir, and B. Thapa are with the College of Computer and Information Science, Northeastern University, Boston, MA, 02115. E-mail: {acassola, taojin, noubir, bthapa}@ccs.neu.edu

†A preliminary version of this material appeared at the 10th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'09) [11]

Emerging attacks include ultra low-power cross-layer attacks that aim at disturbing the operation of networks by targeting control-mechanisms such as packet routing, communication beacons or pilots, carrier sensing mechanism, collision avoidance exponential back-off mechanism, network topology, size of the congestion control window, etc. For example, by transmitting a few pulses at the right frequency, right time and right location, an tremendously energy/computation efficient attack can be implemented with off-the-shelf hardware [13, 5, 26, 12, 6, 25].

1.1 Motivation

Over the last few years many techniques were developed to overcome jamming attacks [13, 12, 4, 3, 2, 26, 23, 12, 7, 6, 24]. Spread Spectrum communication is one of the most efficient mechanisms used for anti-jamming communication. In particular, military systems rely on SS along with antenna nulling, channel coding to counteract malicious attacks. In civilian systems however, SS has been discarded from use mainly because it requires the communicating nodes to pre-share a secret beforehand. This is a limitation common to many other anti-jamming techniques. For example, in the case of SS communication, the shared secret is used to establish a spreading sequence that is used to cryptographically spread the transmitted signal, making it (a) robust against jammers, and (b) private among those who do not know the secret. However, in large and dynamic network of mobile nodes who associate and disassociate themselves from the network frequently, and sharing a secret pre-communication through physical contact is infeasible, sharing of the secret occurs over the open medium, and making it accessible to legitimate users and adversaries alike. This leads to a circular dependency between establishing a shared key to form a resilient communication channel, and such method requiring a resilient communication channel to establish a shared key for long-term communication [22].

1.2 Contribution

In this paper, we extend on our preliminary work [11] to provide the following key properties that, to the best of our knowledge, are not provided by any other system.

- No communication-energy overhead at key establishment phase, in contrast to conventional SS with pre-shared keys
- Undetectable communication until the end of transmission This forces the jammer to become energy-inefficient and channel-oblivious [3].
- A destination-oriented scheme that prevents simultaneous attacks to multiple receivers.
- Computationally efficient end of the message detection and message extraction
- Space overhead of at most twice the cost of conventional spread spectrum decoding
- *Real-time* SS communication. Our prototype can sustain a real-time 1Mbps SS communication spread by a factor of 100 (i.e., 100 mega-chips per second) over a 200MHz bandwidth. This constitutes a *four orders of magnitude* improvement over the existing solution (UFH) of [15] in terms of the communication latency.
- No need for certification authority nor key establishment.
- Equal energy expense as that of traditional SS systems
- No receiver synchronization is required.

This substantial performance improvement over previously known solutions is achieved through a combination of encoding and decoding design optimizations. First, the intractable forward-decoding and efficient backward-decoding mechanisms prove to be lead to a powerful and resilient method for secret sharing. Second, the proposed techniques to implement this method, including block processing, FFT-based message

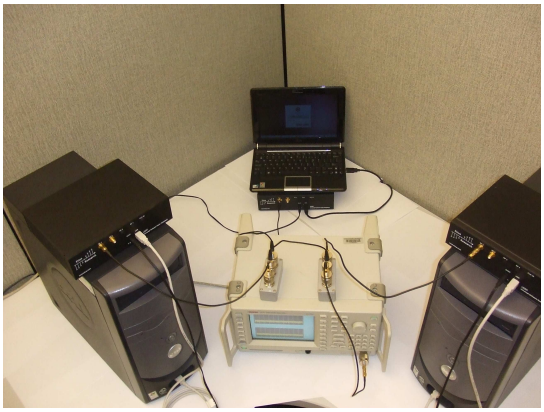

Figure 1: TREKS Test-bed: Two GPU-equipped PCs (Sender and Receiver) and the laptop (Jammer) individually connected to USRPs.

detection, synchronization recovery, and key scheduling during sending, play a vital role in optimizing the communication cost, and the decoding cost making it a fit for long-term and real-time communication.

We test our system both by simulation in MATLAB and experimentally through our test bed consisting of Universal Software Radio Peripherals (USRPs), GNURadio, and a GPU-equipped receiver. Our performance evaluation consists of measuring Bit Error Rate (BER) over several kinds of jamming attacks, computation cost and real time throughput. We also evaluate throughput of our system over 3 different levels of GPU processing power. Our results indicate that the higher-end card (GTX280) can sustain a real time stream of 1000 blocks per second where each block has 1000 bits spread by a factor of 100 (i.e., 100 mega-chips per second).

1.3 Related Work

Anti-jamming techniques were extensively studied for decades [19]. Most of the earlier mechanisms focussed on physical layer protection and made use of spread-spectrum techniques, directional antennas, and coding schemes. At the time, most wireless communication was not packetized, or networked. Furthermore, the small size of the networks then (mostly military), and the way they were deployed allowed for pre-configuration with shared secret keys to be possible.

Reliable communication in the presence of adversaries regained significant interest in the last few years. New attacks and thus, needs for more complex applications and deployment environments have emerged. Several specifically crafted attacks and counter-attacks were proposed for: packetized wireless data networks [13, 12], multiple access resolution in the presence of adversaries [4, 3, 2], multi-hop networks [26, 23, 12], broadcast communication [7, 6, 24], cross-layer attacks [13], and navigation information broadcast [17]. While many recently proposed countermeasure techniques can (and are assumed to) be layered on a SS physical layer, it is usually taken for granted that the communicating nodes pre-share a secret key. Strasser et al. recognized this as a significant impediment to the use of SS, even when the communicating nodes possess public keys and certificates that potentially allow them to setup a shared secret key [22]. They name this phenomenon as the anti-jamming/key establishment circular dependency problem.

To break the dependency, Strasser et al. proposed UFH, a technique for establishing a symmetric secret key in the presence of adversaries. In UFH, the sending node hops at a relatively fast rate (e.g., 1600 hops per second) over n channels. It repeatedly sends fragments of the mutual authentication and key establishment protocol. The receiver hops at a significantly slower rate. Although, the receiver does not know the sender's hopping sequence, statistically, it can receive $1/n$ of the sent packets. The authors show that an adversary has a very low probability of jamming these packets. They build upon this basic mechanism to construct a jamming-resilient mutual authentication and key establishment protocol. Their paper introduced the

first reliable key establishment protocol for SS without a pre-shared secret. However, unlike SS systems with pre-shared keys, the proposed mechanism incurs an energy increase by a factor of n due to the required redundancy in packet transmissions (retransmissions of message fragments that are not received). This is the closest work related to our paper. Our mechanisms retain the main benefits of the original SS communication in terms of communication energy (all transmitted energy is used in the packet decoding process). It does incur a higher computation cost, which we show later is no more than twice the cost of the traditional SS with pre-shared secret.

Later on, Strasser et al. [21], and Slater et al. [20] proposed several coding-based mechanisms for error-detection in fragments and erasures-correction to improve the performance of the UFH. Popper et al. also proposed a generalization to broadcast direct sequence SS [15]. Other countermeasure techniques discard the possibility of using SS because of the narrow RF bands available to ad hoc networks, or because of the absence of a pre-shared key [10, 2]. However, those techniques are much less energy efficient than SS.

This paper is organized as follows: In Section 2, we present the system and the adversary model of TREKS. In Section 3, we present the main scheme. In Section 4, we present algorithms for efficient message decoding, implementation details, and optimizations. In Section 6, we first evaluate TREKS using MATLAB simulation in the presence of different jammer strategies of Section 2, and identify the optimal strategy under our formulation. Then, we evaluate TREKS using an experimental test-bed to demonstrate TREKS's feasibility to operate real-time SS communication. Finally, we conclude in Section 7 and discuss future work.

2 System Model

Our model considers systems that are traditionally capable of performing SS communication, such as mobile ad hoc networks. In addition, we require participating nodes to have moderate computational power for executing FFT operations efficiently. Under our scheme, the more computation power the receiver has, the faster the message extraction becomes.

2.1 Communication

We consider a wireless communication network where several nodes are trying to communicate among each other in pairs in the presence of adversaries. Participants lack any pre-shared secret, which is a prerequisite for traditional SS communication systems. Under our communication model, sender, receiver and adversary share the same Additive White Gaussian Noise (AWGN) channel and any information known to the receiver about the sender, communication protocol, and encoding/decoding scheme is known to the adversary as well. The seed of a Pseudorandom Number (PN) generator that produces sequences to spread signals is private information known only to the sender, and is unknown to both the receiver and the adversary. The goal of the sender is to establish an adversary-resilient, efficient communication channel without divulging the private seed. The goal of an adversary on the other hand, is to prevent communication from happening, as opposed to eavesdropping. We will show later that eavesdropping under our communication model provides no gain to the adversary in terms of its utility.

2.2 Adversary

Under our model, an adversary is within range of the sender and the receiver and can jam, replay previously collected messages or insert/modify bits of messages. The primary goal of the adversary is to prevent successful reception of the message. However, in an attempt to do so, it might just increase the cost of message decoding or cause a Denial of Service attack to delay the message extraction process on the receiver side. The adversary's utility function is a trade-off between the energy cost spent on adversarial attacks versus the Packet Loss Rate (PLR) on the receiver side. We also evaluate the adversary in terms of the delay incurred by its attacks on the receiver's decoding process.

In this work we consider the following kinds of attacks by the adversary:

1. Jamming: The adversary may jam ongoing communication either reactively or as an oblivious jammer sending a high power pulse periodically, continuously or in a memoryless fashion. The impact would be the distortion of few bits of the packet enough to cause packet decoding to fail on the receiver side.

2. Replay Attack: The adversary may insert previously collected messages to cause either a Denial of Service attack or just a delay in the message extraction process on the receiver side.

3. Modification: The adversary may target few bits in the message to modify its contents. However, this kind of attack is not possible under TREKS, because we will see later how the jammer cannot sense the communication until towards the end of the transmission.

4. Insertion: The adversary can insert partial or complete messages following the publicly-known protocol to overwhelm the decoding process on the receiver side.

In Section 6, we only implement protocol specific adversarial strategies that are feasible to implement in real-time within the limitations of the hardware available. We demonstrate the optimal adversarial strategy and establish its cost-efficiency under our scheme.

2.3 Assumptions

We ignore the gain obtained by configuring the physical layer parameters such as coding, and antenna gains, since they can be optimized independently of our mechanism.

We also do not consider the case where the adversary can completely block the propagation of the radio signal from the sender to the receiver. If the adversary has unlimited power and continuously jams the channel with a strong signal, then it can obviously reduce the throughput to zero, just like it would on traditional SS. Our ultimate goal is to devise jammer[1]-resistant SS communication that does not require a pre-shared secret.

We also assume that the adversary cannot relay the brute-forcing of the key to some remote location with supercomputational power, and get the cracked key back before the time (few milliseconds) it takes to complete the transmission.

3 Time-Reversed Message Extraction and Key Scheduling (TREKS) in DSSS

TREKS is a communication approach based on DSSS. We will first present the core idea of zero pre-shared key DSSS and its efficiency against jamming. Then we propose a novel key scheduling scheme, which enables efficient backward-decoding, and thus making TREKS optimal in terms of both communication energy, computation, and storage costs.

3.1 Zero pre-shared key DSSS

Sender S, receiver R, and jammer J all share the same physical channel. Let M denote the message that S wants to transfer to R, and l the length of M in bits. Prior to the start of transmission, S randomly generates a secret key K of k bits. Unlike conventional DSSS, K is known only to S when transmission occurs. S uses K to generate a cryptographically strong PN-sequence, and uses it to spread M. Although, PN-sequences cryptographically generated from keys (e.g., seeding a symmetric encryption algorithm such as AES or DES) are not optimal in terms or orthogonality, they have a very satisfactory performance and have been used in many military spread spectrum communications system [19].

In conventional DSSS, S and R pre-share the secret key. R keeps attempting to despread incoming signals with pre-shared key until she detects the beginning of the message, then she starts forward-decoding the

[1]From here onwards, we will use the terms *jammer* and adversary interchangeably as jamming turns out to be the only feasible and optimal strategy of the adversary under our scheme.

whole message. In zero pre-shared key DSSS, R needs to first identify the key K chosen by S. Without knowing K or when such DSSS communication occurs, R needs to brute force all possible keys on each chip of the incoming signal until she finds a key that could properly decode the incoming signal. Given a key size of k bits, the complexity of exploring the key space is $O(2^k)$. Obviously, this is impractical for real-time communication when no information is available about the start of a packet. In Section 3.3, we show how backward-decoding with a key schedule makes our approach efficient for real-time communication.

3.2 Jamming resiliency

We first demonstrate the fundamental strengths of the proposed approach from the energy efficiency against jammers and key recovery intractability.

3.2.1 Communication energy efficiency

We present the way the packet data bits are spread and how the total energy per packet is preserved. We also show that the cost for the jammer to counter the effect of spreading requires an energy increase by a factor of n. Let us first introduce some terminology:

- $d \in \{+1, -1\}$: data bit being sent, both 0 and 1 are equally probable, otherwise the data can be compressed and might also be used by the adversary.
- $\hat{d} \in \{+1, -1\}$**:** estimated data bit on receiver side.
- n**:** Spreading factor.
- $pn_{i\in\{1,\ldots,n\}} \in \{-1, +1\}$**:** i^{th} chip of cryptographically designed spreading sequence unknown to the adversary.
- E_b**:** energy per transmitted bit (w.l.o.g, assume that we are sending one bit per unit of time).
- $u_i = d\sqrt{\frac{E_b}{n}}pn_i$**:** chip signals transmitted by sender. Note that the energy[2] per bit remains equal to E_b. We consider a Binary Phase Shift Keying modulation, but the results generalize to other modulations.
- J**:** jammer energy per unit of time.
- $I_{i\in\{1,\ldots,n\}}$**:** adversary's transmitted signals indexed at the chip level. The mean square of I_i is $\frac{J}{n}$ which corresponds to J amount of energy per bit.
- v_i**:** received signals indexed at chip level.
- $BER(E_b, J, m)$**:** Bit Error Rate at receiver side when sender is using E_b Joules per bit, adversary J Joules per bit, and transmitter spreading by factor m.

Theorem 1 *Spreading a signal by a factor $n \gg 1$ allows, the communicating nodes to counter an n-times stronger jammer at no extra-energy cost for the sender:*

Proof: Since, we are only interested in the impact of jamming, we normalize the path loss and antenna gains to 1. For simplicity, we ignore thermal (white) noise. The same result still holds in the general case. Let v_i denote the received signal indexed at the chip level:

$$\begin{aligned} v_i &= u_i + I_i \\ &= d\sqrt{\frac{E_b}{n}}pn_i + \sqrt{\frac{J}{n}}r_i \end{aligned}$$

[2]Energy is equal to the signal mean square.

where r_i is the jamming chip with unit mean square.
Consider the following decoding technique[3]:

$$\hat{d} = 1 \text{ iff } \sum_{i=1}^{n} v_i pn_i > 0$$

The Bit Error Rate of the despread signal, $BER(E_b, J, n)$

$$\begin{aligned}
&= Pr\left[\hat{d} = 1 \text{ and } d = -1\right] + Pr\left[\hat{d} = -1 \text{ and } d = 1\right] \\
&= 2Pr\left[\sum_{i=1}^{n} v_i pn_i > 0 \text{ and } d = -1\right] \\
&= 2Pr\left[d\sqrt{\frac{E_b}{n}}\sum_{i=1}^{n} pn_i pn_i + \sqrt{\frac{J}{n}}\sum_{i=1}^{n} r_i pn_i > 0 \text{ and } d = -1\right] \\
&= 2Pr\left[-\sqrt{\frac{E_b}{n}}\sum_{i=1}^{n} pn_i pn_i + \sqrt{\frac{J}{n}}\sum_{i=1}^{n} r_i pn_i > 0 \text{ and } d = -1\right] \\
&= 2Pr\left[-\sqrt{E_b n} + \sqrt{\frac{J}{n}}\sum_{i=1}^{n} r_i pn_i > 0\right] * Pr\left[d = -1\right] \\
&= Pr\left[\sum_{i=1}^{n} r_i pn_i > \sqrt{\frac{E_b}{J}} n\right]
\end{aligned}$$

where pn_i is a random variable independent from the adversary's r_i choices. Therefore, $\sum_{i=1}^{n} r_i pn_i$ is the sum of n random variables of equal probability taking values $\{-1, +1\}$. The distribution of the sum can be derived from the Binomial distribution. For $n \gg 1$, this distribution can be approximated by a Normal distribution of zero mean and variance n: $N(0, n)$. Thus,

$$\begin{aligned}
BER(E_b, J, n) &= \int_{n\sqrt{\frac{E_b}{J}}}^{\infty} \frac{1}{\sqrt{2\pi n}} e^{-\frac{x2}{2n}} dx \\
&= \int_{\sqrt{\frac{E_b n}{J}}}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x2}{2}} dx
\end{aligned} \tag{1}$$

Eq. (1) indicates that when the spreading factor is increased by a factor c, the adversary needs to scale its jamming energy J by a factor c to maintain the same BER. On the transmitter side, since the energy per bit is kept constant, transmitter still spends the same amount of energy while being resilient to c times more jamming. □

3.2.2 Computational infeasibility for jammer

In order to jam in a cost efficient way, the adversary needs to identify the spreading key. As shown above, the complexity of finding the key is $O(2^k)$. If k is designed such that identifying the key takes significantly more time than the packet transmission then even if the jammer eventually finds the key, he will miss the chance to jam the transmission. We call this *intractable forward-decoding*, which is illustrated in Figure 2.

Note that intractable forward-decoding also applies to the message-decoding process at the receiver. Since the receiver needs to try 2^k possibilities for despreading key on each incoming chip signal, it causes a considerably high computation overhead. This is a major limitation of the basic zero pre-shared key DSSS (ZPKS) scheme.

In the following section, we introduce a novel spreading key scheduling scheme, which builds upon ZPKS and enables *efficient backward-decoding*. This drastically reduces the computation overhead for the receiver from $O(2^k)$ to $O(2k)$ while the jamming resiliency remains the same.

[3]Note that we are assuming that the receiver knows the bit synchronization. This is a common assumption in analyzing SS systems. We will see in Section 4 how this is achieved.

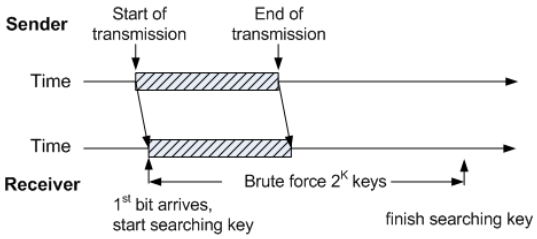

Figure 2: Message delivered before key is brute forced by adversary.

3.3 Key scheduled reverse-time decoding

3.3.1 Key Size vs. jamming resiliency

Before delving into the details of our key scheduling scheme, we first show that reducing key entropy as the transmission gets closer to the end still requires the same effort from an adversary to recover the key.

Theorem 2 *Let $T_{trans}(l)$ denote the transmission time of l bits, $T_s(k)$ the time required to brute force all possible k bit keys. Given a message M and key size k, if it is secure to spread M with a k bits key, it is secure to spread the last $\frac{|M|}{2^i}$ bits with $k-i$ bit key, where $i \leq \log_2(|M|)$.*

Proof: We first show that it is secure to spread the second half of M with $k-1$ bit key Since it is secure to spread M with a k bits key, we have

$$\begin{aligned} T_{trans}(|M|) &\ll T_s(k) \\ T_{trans}(\tfrac{|M|}{2}) &= \frac{1}{2}T_{trans}(|M|) \\ &\ll \frac{1}{2}T_s(k) = T_s(k-1) \end{aligned} \tag{2}$$

Eq. (2) shows that it is secure to encode $\frac{|M|}{2}$ bits with $k-1$ bit key. Therefore, even if we use a 1-bit weaker key to encode the second-half of M, we can guarantee that the whole message can still be delivered before the jammer brute forces all possible keys. By induction, it is easy to get that $T_{trans}(\frac{|M|}{2^i}) \ll T_s(k-i)$. Thus, it is secure to spread the last $\frac{|M|}{2^i}$ bits with $k-i$ bit key. □

Intuitively, Theorem 2 states that as transmission goes on, less time is left for jammer to find out the key, so it is safe to use a slightly weaker key to encode the rest of the message [4].

3.3.2 Spread key scheduling

Based on Theorem 2, we introduce a key scheduling scheme to TREKS. As shown in figure 3, instead of spreading the complete the message with a fixed key, we partition the message into k segments (note that the segments are transmitted in a continuous way), where k is the key size. We call each segment "schedule". The size of ith segment M_i is $\lceil \frac{|M|}{2^i} \rceil$. At the start of spreading process, we use full length key to spread M_1. After we finish encoding a segment, we set the most significant bit of the key to a known value, and resume encoding the next segment with this 1-bit weaker key. We repeat this process until the last schedule, which is encoded with only 1 bit key. Here it is easy to see that the message length l has to be at least 2^k so that the key size k could be decreased to 1 bit as schedule goes on. For simplicity of presentation, we assume that $l = 2^k$. We will show how to loosen this constraint in later section. Algorithm 1 outlines the message segmentation and key scheduling scheme.

We can see from Theorem 2, the key size at each schedule is large enough for the size of the corresponding message segment. Thus, the property of intractable forward-decoding is maintained.

[4] Additional measures can be taken to prevent overlap between weakened key spaces.

Symbol	Definition
M	message to be transferred
K	secret key
l	length of message in bits
k	size of secret key in bits
$K[m \dots n]$	part of the K from m^{th} bit to n^{th} bit
$M[m \dots n]$	part of the M from m^{th} bit to n^{th} bit
K_i	key used in schedule i
M_i	message segment spread using K_i
N_i	Rest of the message left at schedule i

Table 1: Summary of the notation.

1. $N_1 \leftarrow M$;
2. **for** $i = 1 \dots k$ **do**
   $K_i \leftarrow K[i \dots k]$;
   $M_i \leftarrow N_i[1 \dots \lceil \frac{|N_i|}{2} \rceil]$;
   cryptographically generate PN_i from K_i ;
   encode M_i with PN_i ;
   $N_{i+1} \leftarrow N_i[|M_i| + 1 \dots |N_i|]$
end

Algorithm 1: Sender encoding message with key schedule.

Additionally, due to the decreasing key entropy, it becomes easier for the receiver to identify the key as the transmission is closer to the end. Specifically, in order to detect the last message segment, the receiver just needs to attempt two keys. Once the receiver detects the potential end of message, he starts guessing the keys for previously received signals following the key scheduling scheme in reverse time. In order to guess the key for each previous schedule, receiver only needs to try two keys, because the bit difference for two adjacent schedules is only 1 bit. So the receiver needs to try up to $2k$ keys to find out all the k key bits, which is significantly lower then the $O(2^k)$ guesses of the basic scheme. We refer to the above decoding as *efficient backward-decoding*.

3.4 Further improvements

3.4.1 MAC-masked key scheduling

In the key scheduling scheme presented above, the last scheduled key K_k is always either 0 or 1 for any sender/receiver pair. Hence, the jammer could jam all communication within its range with a single PN-sequence generated by 0 or 1 key, which is likely to compromise the last message fragment. Once the end of the message is jammed and the receiver is not able to detect it, the reverse decoding cannot start. In order to tackle this issue, we take the receiver's MAC address to mask the key at each schedule. The revised key scheduling strategy is illustrated in figure 4. The key K_i used to encode M_i is generated by replacing the most significant $i-1$ bits of the receiver's MAC address with the most significant $i-1$ bits of K. It is easy to see that the hardness of the key inferring remains the same. Whereas, the last scheduled key is different across different receivers. Thus, the jammer can only target one receiver at a time. The potential jamming attack mentioned earlier becomes a destination-oriented attack.

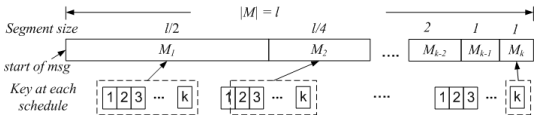

Figure 3: TREKS with key scheduling.

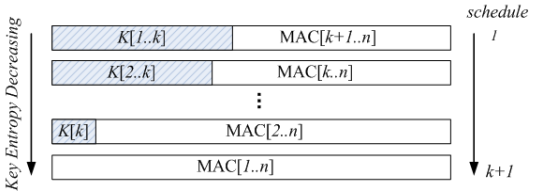

Figure 4: MAC-masked key scheduling.

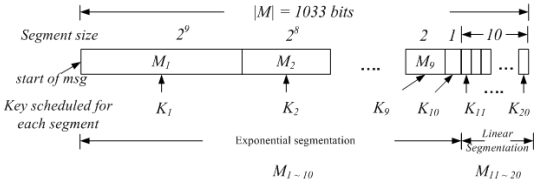

Figure 5: Key scheduling with linear tail.

3.4.2 Key scheduling with linear tail

Consider a key size of $k = 20$, the jammer needs to explore 2^{20} keys. Even for a chip rate of 100Mcps (10ns chip duration), it is infeasible for a field deployed device to brute force the key transmission of a few milliseconds (e.g., 1ms for 1000 bits spread with $n = 100$). However, as mentioned at the end of 3.3.2, we assumed that $l = 2^k$ so that key size can be decreased down to 1 bit by k^{th} schedule, Thus, the total message length l for $k = 20$ should be 1M bits; obviously too large for a message size.

We also observed that if $T_{trans}(|M|) \leq T_\delta$, where T_δ is the radio turn around time of the jammer, it is impossible for the jammer to jam M. In this case, when the jammer detects the transmission and switches to a transmit mode, the message has already been delivered. Take 802.11 as an example, the radio turn around time is $10\mu s$. Consider a spreading factor $n = 100$, chip rate of 100Mcps, then we have $T_{trans}(1) = 1\mu s$. So for the last 10 bits of the message, the sender can weaken the key at a linear rate of 1 key bit per packet bit. Therefore, only the first 10 bits of the key need to be scheduled. Thus, the message size becomes $10 + \sum_{i=0}^{9} 2^i$ = 1033 bits, which is a reasonable size. Note that if T_δ allowed for only the transmission x number of bits, we can linearly weaken the key by $x + 1$ or more key-bits per transmitted bit. This slightly increases the computation cost of key inferring but only on a small number of bits. The revised key scheduling algorithm is illustrated in Figure 5.

Next, we present the efficient backward decoding algorithm in detail, its computation complexity, and briefly discuss the key establishment protocol under TREKS.

4 Efficient Backward-Decoding

4.1 Overview of TREKSDecoding

MAC-masked TREKS enables efficient backward-decoding. The backward-decoding is best described as a two-phase procedure [See Figure 6]:

- ***Phase-I***: Finding End of the Message(EoM) by computing the cross-correlation between the received spread signal and the PN-sequence generated with receiver's MAC address.
- ***Phase-II***: Inferring the key in reverse time, and despreading the message.

Symbol	Definition
m	Sender message consisting of z segments
$Seg[i]$	Message segments, $1 \leq i \leq z$
$K[i]$	Key to generate spreading sequence, $1 \leq i \leq z$
K_i	Possible set of keys, $1 \leq \|K_i\| \leq 2$, that receiver tries to despread Seg[i] with.
$S[i]$	Signal sampled at the receiver side.
$PEoM[i]$	Array of possible EoM indices.
$M[i]$	Array of extracted complete messages.
$GetBuffer(.)$	Gets the next $n * l$ chips from signal stream.
$DotProd(.)$	Dot product of two vectors (correlation).
$FFT(.)$	Fast Fourier Transform.
$IFFT(.)$	Inverse Fast Fourier Transform.
$Fast\_Correlate(.)$	Convolute a short and a long signal.
$Key\_Infer(.)$	Function to infer the key.
$Peak\_Detection(.)$	Function to detect peaks at Seg[i], $1 \leq i \leq z$
$Despread(.)$	Function to despread SS signal.
$Signature\_Verify(.)$	Function to verify the sender.

Table 2: Additional notations

4.2 Finding the EoM (Phase-I)

As shown in Figure 6, Phase-I consists of two steps, (a) sampling and buffering, and (b) FFT EoM detection. When new signal samples arrive, the receiver enqueues them into a FIFO. At any instance, the receiver only has to keep $2nl$ chips in his buffer because after finding the EoM, he will have to traverse at most nl length before he recovers the message. We compute cross-correlation to achieve bit synchronization, a very common practice in SS systems [19]. However, calculating cross correlation is computationally expensive. We optimize this calculation by (a) using FFT, which reduces the cost of computing cross correlation from $O\left(2n^2l\right)$ to $O\left(nl \log\left(nl\right)\right)$, and (b) process a batch of nl chips at once during FFT computation unlike conventional SS systems that process n chips (spread of a bit) at a time.

```
1. Old_Buffer = GetBuffer(S)
2. for each buffer of length (n * l) do
     Current_Buffer = GetBuffer(S)
     Set k = MAC_ADDRESS(Rcvr)
     Corr[1 : n * l]=Fast_Correlate(Current_Buffer,k)
     for each j ∈ {1, ··· , n * l} do
       If Corr[j] > threshold then
         push j into PEoM[]
     end
     If PEoM[] is empty
       Old_Buffer = Current_Buffer
     Else
       Buffer = concat(Old_Buffer,Current_Buffer)
       Key_Infer(Buffer,PEoM)
end

Fast_Correlate(Buff,key){
  Temp_Key[1:n*l] = Zeros
  Temp_Key[1:n] = key
  Input1 = FFT(Buff)
  Input2 = FFT(Temp_Key); //Pre-computed
  Corr[1:n*l] = IFFT(Input1*Input2')
  return Corr}
```

Algorithm 2: Finding the End of the Message (EoM)

As illustrated in Algorithm 2, our EoM detection process iterates over each chip in the buffer. As there may be multiple values of the correlation vector that pass the threshold, we enqueue all possible EoMs into

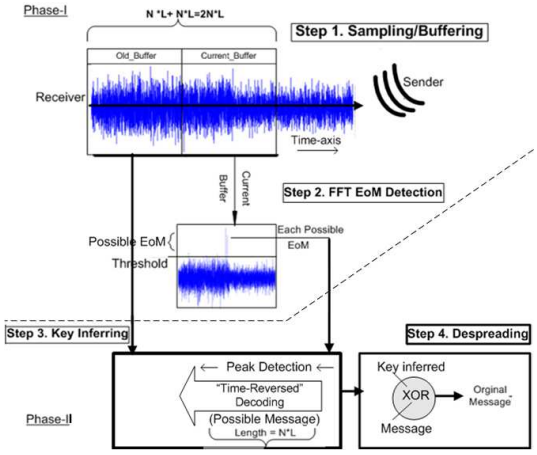

Figure 6: Workflow of TREKS Decoding

PEoM[], and pass it to Phase-II for further processing. We pick our threshold value empirically by observing TREKS performance over a large number of simulation runs, details of which are given in Section 6.

4.3 Message Extraction (Phase-II)

Phase-II consists of Step-3 and Step-4 as shown in Figure 6. In Step-3, we infer the key by finding the legitimate EoM out of all PEoM found in Phase-I. For each PEoM, we begin time-reversed key inferring. At each stage of the process, we try two possible choices for the key-bit. Algorithm 3 illustrates this process. For a certain guess, if more than 50% of the total bits are detected in a segment, then we confirm the value at the corresponding key bit position and move on to the next. Otherwise, we abort the key inferring. From the above we have Theorem 3.

Theorem 3 *The computational cost of TREKS message despreading is at most twice the computational cost of conventional SS systems with a pre-shared key.*

Proof: For each segment, the receiver attempts to despread the bits with two potential keys. Therefore each bits is despread twice. Leading to a computational cost of twice a conventional spread-spectrum. Note, that this cost can be reduced by eliminating one of the two keys after attempting only few bits of a packet. □

The abortion of key inferring implies a packet loss. Otherwise we despread the message using the key inferred [Step-4]. We discuss the choice of the threshold values used in Algorithm 2 in Section 6.

Long-term Communication using TREKS

TREKS can either be used to share the secret required by conventional SS to establish a spreading sequence or to operate long-term SS communication without establishing any keys. The former can be accomplished by choosing any mutual authentication and key establishment protocol, and using TREKS to perform the protocol. The choice for the key establishment protocol is discussed extensively in [11]. For the latter TREKS simply transfers the data between parties. Along with the optimizations discussed above, some implementation optimizations were also necessary. We discuss them in the following section.

5 TREKS Implementation

Besides being computationally and energy efficient secret sharing mechanism, TREKS can also eliminate the need to establish keys pre-communication and operate long-term communication in real-time. In the following, we present a high-level overview of the TREKS architecture, its basic hardware and software components and how they contribute to the design of our system that enables real-time message decoding.

```
1. Key_Infer(Buffer,PEoM){
for each possible EoM j ∈ PEoM do
    PeakPos = n+j; //EoM = Buffer[n+j]
    endIndx = PeakPos-n; //End of Seg[z-1]
    for each p ∈ {1, ··· , z} do
        startIndx = endIndx−|Seg[z]| + 1;
        CntOfSucc = 0;
        for each key candidate k ∈ K_{z−p} do
            succ = Peak_Detection(k,Buffer, startIndx, endIndx);
            CntOfSucc = CntOfSucc + succ;
        end
        If(CntOfSucc==1)
          K[p] = k;
        Else
          Abort Key_Infer(Buffer,PEoM);
        endIndx = startIndx;
    end
    m = Despread(Buffer[j − (n ∗ l) + 1, j], K[]);
    Enqueue m into M[];}
end

2. Signature_Verify(M[]);

Peak_Detection(key, Buf, startIndx, endIndx)
{
  ExpNumofPeaks = (endIndx - startIndx)/n;
  CntOfPeaks = 0;
  for each d ∈ {1, ··· ,ExpNumOfPeaks} do
    P=DotProd(key,Buf[startIndx,startIndx+n]);
    If P > threshold then
       CntOfPeaks = CntOfPeaks+1;
    startIndx = startIndx+(d ∗ n) − 1;
end
  If CntOfPeaks > 50%*ExpNumOfPeaks
     succ = 1;
  Else
     succ = 0;
  return succ;
}
```

Algorithm 3: Message Extraction

5.1 System Layout

Our system consists of two communication nodes: a sender and a receiver sharing a common AWGN channel. Data flows in one direction, starting with the sender constructing a message, spreading it using DSSS, modulating the spread signal, applying signal processing filters in software, and then passing it through the radio board for Digital-to-Analog Conversion (DAC) and Digital-Up-Conversion (DUC) before transmitting the signal over the air through the RF front-end. At the other end, the receiver continuously captures data samples from the air using its RF front-end, passes it through the radio board for Analog-to-Digital conversion (ADC) and digital down-conversion (DDC), enqueues them for applying signal processing filters to demodulate the received signal into streams of bits, employs its GPU for FFT-based End of Message (EoM) detection on the collected bits, and finally passes it onto its CPU for key inferring procedure and message extraction. All the complexity of TREKS lies in the receiver and therefore requires to be carefully designed and implemented.

5.2 Hardware and Software Components

The hardware components of TREKS include two host PCs, two radio boards, one for the sender and one for the receiver. In addition, the receiver node includes a GPU for the FFT-based operations.

The software components of TREKS include: SDR for signal processing, a GPGPU programming platform for computationally intensive but parallelizable tasks, such as FFT, and an implementation of time-reversed message extraction. Software components also include standard interprocess communication such as pipes and shared memory for passing data between processes. A high-level system overview of TREKS with its basic components is depicted in Figure 7.

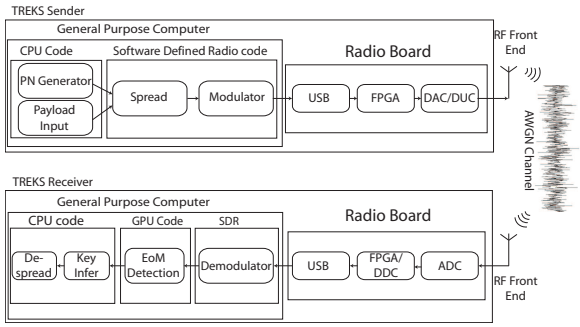

Figure 7: The TREKS system architecture

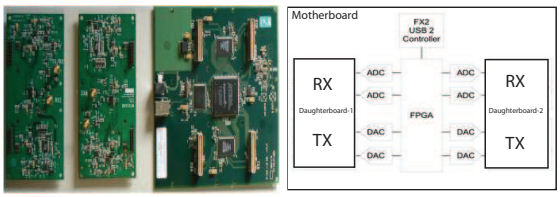

Figure 8: USRP board

5.3 Implementation

In this subsection, we take a look into the details of implementing the Sender and Receiver following the architecture presented above. We begin by describing the choices we made at the hardware and software level in the design of our implementation testbed.

5.3.1 TREKS Testbed

Testbed Hardware: We chose to use the Universal Software Radio Peripheral (USRP) for our radio boards at both the sender and the receiver [18]. For the choice of GPU at the receiver side, we chose NVIDIA graphic cards [14]. We build three TREKS receivers, each with different computational power.

Testbed Software: We chose GNU Radio for Software-Defined Radio (SDR) [16] and Compute Unified Device Architecture (CUDA) for GPGPU programming [1]. In the following we will give a brief background on these four main components of our TREKS testbed.

5.3.2 USRP and GNU Radio

USRP is an integrated radio board which incorporates AD/DA converters, RF front end, and a Field Programmable Gate Array (FPGA) which does some computationally expensive pre-processing of input and output signals. USRP is a low cost, high speed device that is designed to allow general purpose computers with a USB port to function as software radios [18]. It serves as a digital baseband and IF section of the radio communication system. The workload is divided so that all of the waveform-specific processing, like modulation and demodulation, can be done on the host CPU, whereas, the high-speed general purpose operations like digital up and down conversion, decimation and interpolation are done on the FPGA.

A typical setup of USRP boards consists of one motherboard and up to two daughterboards, as shown in Figure 8. Plug-in daughterboards allow the USRP to be used on different radio frequency bands. The USRP has four 128 MS/s 14-bit DACs, four 64 MS/s 12-bit ADCs, four DDCs with programmable decimation rates, and two DUCs with programmable interpolation rates.

GNU Radio is an Open Source Software, and one of the most popular SDR systems. GNU Radio provides a framework for developers to implement almost all of the signal processing blocks in software alone. GNU

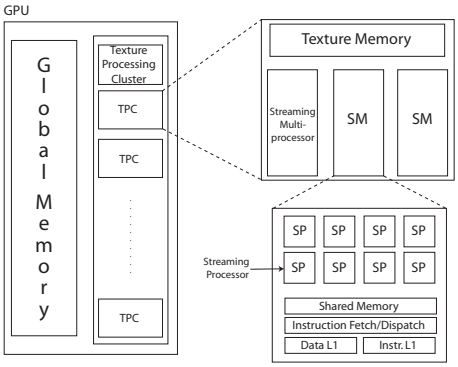

Figure 9: Sketch of GPU NVIDIA GTX280.

Radio needs radio hardware (such as the USRP) to transmit and receive signals to/from the air. GNU Radio takes care of the software side and USRP takes care of the hardware side, forming a state-of-the-art radio communication platform to experiment with digital communication. Inside the framework, GNU Radio is designed as a two layer architecture: a design layer and signal processing layer. In the design layer, Python is used to build and run a graph which represents the DSP blocks and the dataflow between them. In the signal processing layer, C++ is used to implement the performance-critical DSP blocks [16].

5.3.3 NVIDIA GPUs and CUDA

General-Purpose GPU programming aims to utilize the processing power of current graphics processing cards. With the high number of cores available in low-cost processing hardware (112 processors for $110 at the time of writing) it is possible to implement highly parallelizable algorithms such as FFT with considerable speed gains. With the aid of programming frameworks such as Open CL or CUDA, programming is done in terms of common control structures and memory management rather than a GPU-specific language. Applications developed with such frameworks are typically written in C or other high-level languages, and then compiled to CPU- and GPU-specific machine language.

CUDA-supported GPUs (depicted in Figure 9) consist of global memory, and an array of *Texture-Processing clusters* (TPCs). Each TPC contains read-only texture memory, and — depending on the card model— between two and three *Streaming Multiprocessors* (SMs). Each SM in turn contains 8 individual processors, shared memory, shared L1 cache, and an Instruction Fetch-Dispatch controller.

CUDA consists of the tools that make the above translation possible —a compiler, linker, debugger— and a set of run-time, FFT, and Linear Algebra libraries. A CUDA application is written in an extension of C that includes keywords to specify the where (CPU or GPU) functions will run, their running configuration with respect to the problem at hand, and where in memory variables and structures reside.

Functions that run on the GPU are referred to as *kernels*. When invoked, the code of the kernel is executed concurrently by the GPU processors according to the run configuration specified in the call syntax. A run configuration tells the compiler how many copies of the kernel will run, how many copies will run on a given processor, etc.

Kernels are executed as an array of *threads*, all copies of the kernel code. Each thread is assigned an identifier according to the run configuration, as part of the information required to perform its task. For instance, a kernel that applies some function f to each element of a given one-dimensional array and then stores the resulting value in the same array would use a linear configuration. Each thread reads and writes on the position equal to its thread identifier. Most kernels operate in this fashion: whereas traditionally a `for` loop nest would cycle through every element in a data structure, threads would operate independently and concurrently on each element to accelerate the operation.

An array of threads is called a *block*. Threads inside a block are guaranteed to run in the same multiprocessor, sharing high-bandwidth memory, and allowing for inter-thread synchronization and atomic operations

within the block. Threads in different blocks however, run completely separate of each other, and no cooperation mechanisms (synchronization nor atomic operations) are provided.

The run-time library provides GPU memory allocation, akin to C's malloc and free. Data needs to be copied to GPU memory before the kernel is invoked as all operations in kernels are performed therein. Memory copying mechanisms equivalent to the well- known memcpy routine allow copying to occur to and from the GPU.

In addition, newer GPU hardware supports concurrent copy and computation. That is, while kernels and `memcpy`s can be concurrent with CPU code in all hardware revisions, they can be concurrent to each other only in more recent devices.

5.3.4 TREKS Comunication Nodes

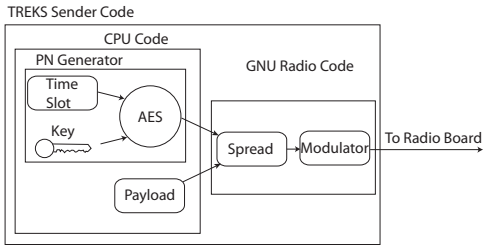

Figure 10: TREKS Sender. The key and a Time Slot counter are input to AES to generate the PN. Message will then be spread and sent to the radio.

TREKS Sender: Prior to any communication, the TREKS sender performs the following steps (also illustrated in Figure 10):

1. *PN Generator:* Sender randomly generates a secret key K of k bits, known only to the sender in the network. Then it generates a cryptographically strong PN-sequence using K and time information as the input to a symmetric encryption algorithm, such as AES.

2. *Spreading:* Sender spreads message M with PN sequence generated in Step 1 following DSSS.

3. *Modulator:* Using the GNU Radio signal processing block for modulation, the sender modulates the spread message (stream of chips) into the waveform (stream of complexes).

4. *USRP processing:* USRP board takes the complex baseband signal and up-converts it to a real passant signal using its DUCs.

5. *RF Front End:* Finally, the modulated upconverted signal is ready to leave the sender for transmission through the RF front end.

TREKS Receiver

The receiver software is composed of several modules or processing blocks that operate as a pipeline, such that every task can run concurrently and independently. The operation of the TREKS receiver follows the phases described in [11], and receives the parameters summarized in Table 6.3.1.

1. First, the demodulator outputs the chips as they become available, storing them in order inside buffers of size nl, and placing them at the tail of a queue.

2. According to our original design (depicted in Figure 11), we obtain the correlation of the signal and the MAC address by taking FFT of signal, FFT of MAC address, multiplying them together, and finally computing the inverse FFT of the result. The final vector is then inserted in a second queue, where it will be examined for correlation peaks.

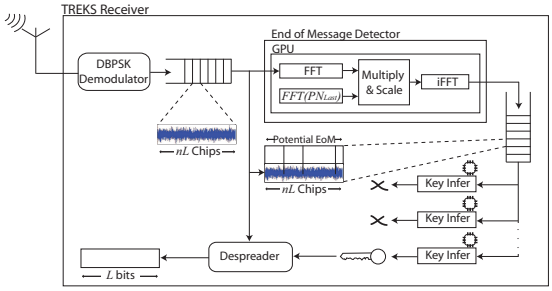

Figure 11: TREKS receiver processing blocks. Arrows show the flow of data from the USRP Radio device to despreading.

3. The correlation vector associated with a particular chip buffer is examined to find correlation peaks. If one is found, a key search starts from such peak and proceeds backwards until the search fails or a key is found. Each of these Key Inferring searches is handled by a single CPU core.

4. Finding a key triggers the despreading module, which will use the key and the PN generator to obtain the schedule of PN sequences necessary to recover the packet bits.

The EoM detection block The design of the EoM detector (Figure 11) reflects the implications of using the GPU as a general-purpose processing unit for our application. GPUs facilitate the task of operating concurrently on a data structure. The hardware available on GPUs consists of multiple parallel processors, each of which can execute a single instance of code and operate on a fraction of the original data structure. The increase in computing power however, is offset by the price of constantly copying data to and from the GPU.

The principal components of the EoM detection block are detailed below.

- *CPU/GPU data copy:* Before data can be operated on by the GPU hardware, it must be transferred to GPU global memory. The underlying operation is analogous to performing a C standard library `memcpy` call. Once the operation is complete, a copy of the original array will reside in GPU memory, and a pointer will be available for subsequent operations. Similarly, a `memcpy` operation of the resulting correlation vector back to CPU space is necessary.

  A further optimization for TREKS could come in the form of asynchronous copies. Memcpy operations can be made to run concurrently along with CPU and GPU code. In our case, transfers off the GPU can be performed while the FFT and product operations of the next chip block are being executed.

- *FFT:* The most common usage of FFT libraries such as FFTW [9] comprises the creation of a transform plan, based mainly on the number of elements, and later the execution of the plan when the data is available. In our design, we choose a fixed size for the FFT transform plan, and compute it during the application start-up only, storing it in GPU memory, ready for the product phase.

- *Point to Point product:* In order to avoid data transfer to CPU memory space, we perform the product of the two vectors within the GPU. Multiple copies of the code multiply the coordinates, scale the result by nl, and store the result back in the signal vector.

Key Inferring Module The EoM module produces a cross-correlation vector associated with the currently examined nl block. A peak in the correlation vector corresponds to a potential End of Message bit in that position, and a Key Search starts from such position in reverse time. The Key Inferring algorithm presented in [11] is the one included in the TREKS receiver design. Namely, the current nl block and the one preceding it are treated as one, and divided in sub-blocks. The ith sub-block starting from the cross-correlation peak has size $2^i n$, with $i = 0, 1, 2, \ldots, k$. Each sub-block is correlated (for details see *Other Components* under

this section.) twice: once for every key value of the ith key bit. The best correlation is chosen to be the one with at least a fraction α of the expected number of peaks. In case of a tie, the number of peaks and the average correlation of the sub-blocks —in that order— are used to decide the winner. The bit that yields the best correlation is set to the key, and the search continues. If both vectors fail to produce at least half of the peaks, or if they are still tied after tie-breaking, the search is abandoned.

We take advantage of CPU parallelization for the Key Inferring module by running key searches independently in each core. Depending on the amount of available CPU cores in the system, a key search task is assigned to each core.

Despreading Module If a key is successfully derived by the inferring module, the despreading process can begin. The despreading operation in TREKS has no particularities, expect for the usage of multiple PN sequences as defined by the key schedule. First, the PN generator (see Section 5.3.4) re-creates the set of PN sequences used by the sender by using the inferred key, and computes the correlation of each n-chip block and the corresponding PN sequence. We obtain the correlation of every n chip block concurrently according to the number of CPU cores available in the system.

Other components Other routines that are important to the TREKS receiver are the PN generator, and the correlation routine used by both the Key Inferer and the Despreader.

The PN generator is a cryptographically secure function $f : \{0,1\}^k \rightarrow \{0,1\}^n$ where k is the Key size, and n is the process gain. The output is the PN sequence of length n induced by the given key. The initialization of the EoM module builds the minimum-entropy PN by calling the PN generator on the receiver's MAC address. Later, during Key Inferring the PN generator receives each of the key guesses as the search progresses. Once a key has been inferred, the PN generator receives the Key Schedule the sender used during transmission to obtain the original PN sequences.

The correlation routine takes a n chip block $b \in \{-1,1\}^n$ and a PN sequence p, multiples them together and adds the result to obtain the correlation. Due to clock skew in the radio platform we multiply using a 3-chip correlation window, and choose the maximum value of the set.

6 Performance Evaluation

In this section, we thoroughly evaluate the performance of TREKS through MATLAB simulation and real-world experimentation using our novel test-bed described above. First, we will discuss the performance metrics of our evaluation, the simulation setup and the hardware/software specification of experimental runs. Then, we will present the types of jammers we consider for TREKS evaluation, and finally the performance results with and without jammers present in the medium.

6.1 Performance Metrics

We evaluate the performance of TREKS in terms of (a) Computation cost (message decoding cost), (b) Storage cost, and (c) Jamming resiliency. The jamming resiliency can be measured in terms of (a) Bit Error Rate (BER), (b) Packet Loss Rate (PLR) and (c) Computation delay sustained by TREKS communication in the presence of jamming.

6.1.1 Computation Cost

The communication cost under TREKS is same as that under the conventional SS. Same amount of energy is spent per bit sent over the air. However, the computation cost under TREKS is at most twice the computation cost under conventional SS. It is the message decoding cost that a receiver must incur when trying to recover the sender message. We will see that it is four orders of magnitude (a couple of milliseconds vs. tens of seconds) better than UFH [22], and under a modest system requirement of our testbed it can actually sustain a 1Mbps long-term SS communication in real-time.

6.1.2 Storage Cost

TREKS receiver maintains a FIFO of size $2nl$ to buffer incoming signal, and storage of size nl in GPU to store the PN sequence generated using receiver MAC address for FFT-based EoM detection. This totals the storage requirement to that of $3nl$ chips (each chip is 4-Byte), which is clearly within the capacity of today's hardware. The use of GPU to process nl chips fast enough to detect the EoMs before another batch of nl chips arrive at the receiver is the key to sustaining a real-time SS communication using TREKS.

6.1.3 Jamming Resiliency

Jamming resiliency is the communication throughput achieved by TREKS in the presence of jammer. BER and PLR directly corresponds to the throughput of a network system. Sometimes, depending on the ability and availability of jamming resources (budget), a jammer might simply choose to increase the delay of message decoding instead of preventing the whole communication. This could occur when the jammer inserts complete or partial messages as discussed in Section 2. Such a strategy increases the number of false positives (FPs) for EoMs during FFT EoM detection process causing the decoding process to take longer than normal before it detects the actual EoM.

6.2 Types of Jammers

We consider two types of jammers during our evaluation:*Continuous* jammer and *Non-continuous* jammer. Continuous jammers are non-budgeted and have the resources to jam continuously. Under such a formulation, we consider four protocol-specific continuous jammers: Gaussian, MAC, Random and BPSK jammers. Non-continuous jammers, on the other hand, are budgeted and are forced to jam selectively at each communication timeslot based on the budget available. Also, since the adversary does not know the beginning of the transmission, non-continuous jammers are resorted to jamming obliviously at some fixed jamming rate λ (a function of the jammer budget). We consider two protocol-specific non-continuous jammers: MAC and Random jammer.

1. Continuous jammers: They are basically the *non-continuous* jammers with $\lambda = 1$. We evaluate four kinds of continuous jammers:

   - Gaussian Jammer: Jammer that produces AWGN signal continuously at the communication frequency. This jammer's energy is reduced by a factor of n when a signal is spread with spreading factor n.
   - MAC Jammer: Jammer that continuously sends a random message spread with the PN sequence generated by receiver's MAC address as the seed. This jammer basically imitates to be the sender by transmitting fake messages to cause DoS attack.
   - Random Jammer: Jammer that continuously sends a random message spread with a random PN sequence.
   - BPSK Jammer: Jammer that sends continuous Binary Phase Shift Keying modulated random 0/1 signal.

2. Non-continuous jammer: We consider a discretized time with timeslots of duration $n * l$ chips. We define two different kinds of jammers that take parameters λ and JSR. λ represents the probability that a jammer sends a jamming message at a given timeslot (this corresponds to discretization of a Poisson memoryless jammer to a Bernoulli jammer), and JSR is the jammer to signal power ratio. The cost of the jammer is $\lambda * JSR$, and its goal is to maximize the PLR and BER for a given budget. Note that because the adversary does not know when transmissions happen, the cost of the jammer should be further scaled by a factor μ which corresponds to the data transmission rate. In the following we consider the best case for the jammer where $\mu = 1$. Also, we only consider the case where the jammer sends complete messages. The jammer could also send partial messages but this can be independently addressed with appropriate interleaving and coding [13]. Hence, we consider following jammers:

- **Random Jammer**: Inserts a random message, each bit spread with a random PN-sequence.
- **MAC Jammer**: Inserts a random message, each bit spread with a PN-sequence generated using MAC address of the receiver as the seed.

We use MATLAB to simulate the non-continuous jammers, and implement only the continuous jammers both inside our test-bed for experimental evaluation and the simulation. We will discuss the reasons for this later in the results subsection.

6.3 Evaluation Setup

In this subsection, we will present the configurable parameters of our evaluation process, hardware and software specifications of our experimental test-bed, and the scenarios we consider for both simulation and experimentation.

6.3.1 Parameter Settings

All the graphs are based on 10K simulation/experimental runs for each possible setting of parameters under our model. The parameter settings of MATLAB simulation and experimentation are as follows:

Spreading Factor, n	100
Packet Size, k	1024 bits
Key Size, n	10
Jammer to Signal Ratio,JSR	[-30dB..20dB]
Frequency	2.5GHz
Modulation	Differential BPSK
Rate	1 Megachips per second
Normalized Signal Power	0 dBW
Noise Power	-20 dBW

Table 3: Parameters for Simulation/Experiment.

EoM Threshold: We use the PLR and the number of FPs observed while running TREKS at a fixed noise level of 0dB, to choose the peak detection threshold used in Algorithm-2.

We define the threshold as $t * avg$ where avg represents the average of the correlation vector produced by function fast_correlate(.) of Algorithm-2, and t represents the constant multiplier. Based on the results from Table 6.3.1 and 6.3.1, we chose t to be 2.5 because of the much smaller FP rate even if we loose about 2dB of jammer resiliency.

SNR (dB)	**t=1.0**	**t = 2.0**	**t = 2.3**	**t = 2.5**	**t = 3.0**
-10	11.79%	1.48%	0.94%	0.58%	0.22%
-5	11.80%	1.48%	0.94%	0.58%	0.22%
0	11.79%	1.47%	0.96%	0.59%	0.22%
5	11.79%	1.51%	0.98%	0.61%	0.23%
10	11.78%	1.57%	1.01%	0.64%	0.25%

Table 4: False Positives (FP)

6.3.2 Software and Hardware Specification

Our initial experimental setup is comprised of system components with following software and hardware specifications:

SNR (dB)	**t=1.0**	**t = 2.1**	**t = 2.3**	**t = 2.5**	**t = 2.9**
−10	19.00%	48.00%	49.50%	47.50%	64.50%
−5	0.00%	0.20%	0.50%	1.50%	4.00%
0	0.00%	0.00%	0.00%	0.00%	0.00%

Table 5: Packet Loss Rate (PLR).

Component Name	Type/Model	Specification
Host Operating Systems	Ubuntu	9.10
Host CPUs	Intel	Core2 Q9300
Receiver GPU	NVidia	GTX280
GPGPU Programming	CUDA	2.2
SDR	GNU Radio	3.2
Radio Board	USRP	1

Table 6: Experimental Test-bed Specifications.

6.3.3 Experimentation Setup

We carried out our experiments in a cabled setup for reproducibility purposes. Three USRPs connected to three separate host computers representing a sender, a receiver and a jammer respectively as seen in Figure 1. The communication nodes are cabled using 50 ohm SMA cables with significant attenuation of about $30dB$ on the receiver side. It is highly recommended to have attenuation above $30dB$ when connecting transmitter directly into the receiver through the use of coaxial cables [18]. All USRP boards are placed inside well-insulated enclosures. For the simplest setup, the jammer acts as the source of white noise in the medium by constantly inducing AWGN signal into the communication frequency. We call this kind of jammer, the gaussian jammer. For each experimental run, sender sends 1 million packets, each of size 1024 bits. Between packets, there is a sleep time of 6 milliseconds introduced to delay the end of the message transmission thread so that the packet transmission are completed before the thread dies.

With GNU Radio v.3.2.2, theoretically you can vary the transmitter amplitude, Tx_{amp} in the range $(0, 1]$, however, experimentally we saw that after Tx_{amp} was set to anything above 0.5, the ADC runs at almost a full scale, thus causing saturation effect. Therefore, we could only use half of the available range $(0, 0.5]$ to reliably tune the transmitter power to different values for sending messages. Same is true with the jamming power setting. We verified this with the manual inspection of power calibration of the USRP/GNU Radio generated signals with varying Tx_{amp} and Jx_{amp} using spectrum analyzer. The worst of all is that the granularity of this kind of power variation controlled by the GNURadio API is very coarse. For an example: Tx_{amp} of value 0.08 and 0.06 differs by at most $1dB$ on average over large number of samples. Hence, we resorted ourselves to using different set of attenuators to induce power variations instead of using GNURadio APIs to change Tx_{amp} and Jx_{amp}.

We measure the performance of our system against varying Signal to Jammer Ratio (SJR). We define SJR (in dB) as,

$$SJR = 20 * log_{10}(\frac{Tx_{amp}}{Jx_{amp}} * m)$$

where m is an empirical constant that represents the ratio between Tx'_{amp} and Jx'_{amp}, such that both Tx'_{amp} and Jx'_{amp} measures the same transmit power in dBm.

Hardware Limitations: To summarize our hardware limitation and its impact, we ran across two major problems while carrying out experiments:

- *Saturation Effect:* Compared to the expected range for varying the amplitude of a sender signal according to the GNU Radio API documentation, we found out that the range without the sender reaching its saturation point is much smaller.

	Recvr 1	Recvr 2	Recvr 3
GPU Model	GTX280	9800GT	8600GT
CPU (Core2)	Q9300	E8400	Q9300
EoM Detect	0.891	0.943	1.37
Key Infer	3.2	3.9	3.11
Despreading	1.6	1.95	1.55

Table 7: Average Task processing time of one a 1024-bit (in ms), 20dB gain.

	GTX280	9800GT	8600GT
Global Memory	1GB	512MB	512MB
Shared Memory	16KB	16KB	16KB
Registers/SM	16384	8192	8192
SMs	30	14	4
SPs	240	112	32
Retail Price	$250	$110	$40

Table 8: GPU specifications.

- *Granularity:* The mapping between the actual signal power (in dBm) and amplitude setting in the GNU Radio application is not precise. The power variation is deterministic only when the step-size is $1/10th$ of the allowed range.

We resolve these issues by using various size attenuators to stay within a power-variation attainable window instead of using GNURadio APIs to acheieve the SJRs needed for our evaluation.

6.4 Performance Results

We first look at the analysis of computation cost under the implementation of TREKS in our carefully designed test-bed.

6.4.1 Computation Cost

To analyze the speed of TREKS, we built 3 receiver systems and tested each in turn by transmitting a 1024-bit packet spread by a factor of 100, using a 10-bit key. Each receiver has distinct CPU computing capabilities, and each uses a different GPU: a mid-range GTX280, a cheaper 9800GT, and a legacy 8600GT. Table 8 contains the hardware specifications of each GPU Model [1].

Before analyzing each component individually, we present the Computation cost for our whole system. The time taken for each task in the receiver is summarized in Table 7.

In order to sustain a 100M chip per second rate, the time for each element in the pipeline must fall below the 1ms mark. The EoM Detection module falls below this mark thanks to the GPGPU implementation. Even if the Inferring and Despreading modules account for most of the computation time, they will still be able to match the rate of the EoM Detector as long as there are enough cores available. If false positives fail on early rounds, the less time is spent searching for a nonexisting key, freeing cores for new data blocks. Note that the Key Infer time on Table 7 corresponds to a successful search of the key after $O(2k)$ steps. Since each key search round correlates exponentially more chips than the last, early search failures will free resources at a fraction of the above time. Section **??** analyzes the false positive rate in our system. In turn, the despreader also benefits from having available cores for execution

In terms of economic cost/benefit, the mid-range 9800GT performs better than the GTX280. Both of these GPUs can be used to obtain the 100Mchip rate, but the GTX offers only 5.5% improvement for twice the cost. Even if the 8600GT falls short of such mark, it is still suitable for rates of over 70Mchips per second.

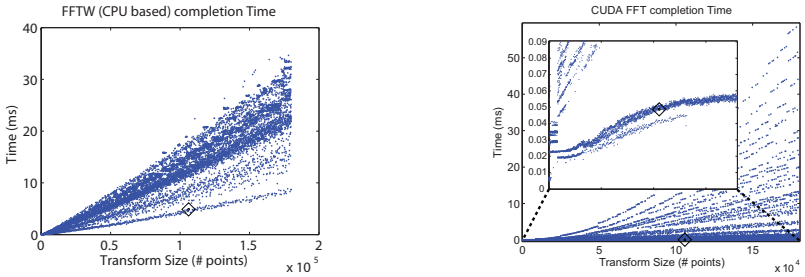

Figure 12: FFT completion time comparison between FFTW (a CPU implementation) and CUDA FFT. The best times for cuda FFT transforms lie below 0.06ms.

We now turn to analyze the performance of the major components: The EoM detector, the Key Inferring module, and the Despreading.

GPU-optimized EoM Detector To analyze the performance of our EoM Detector module, we compare its speed against a EoM module build using FFTW3, a widely-used, fast, and free CPU-based FFT library [9]. We perform 100 runs of each with transforms sizes ranging from 16 to 180 thousand points in 16-point steps. All the plots included in this section are composed of individual points, one for each tested size. Due to the amount of data points collected, no line joins the plot; the patterns emerging from the tests are still apparent, however.

It is critical to the FFT implementation's speed to choose an adequate transform size. Figure 12 compares the time to complete FFT for FFTW3, and CUDA's FFT. The best transform speeds for CUDA lie at the bottom of the plot, and are visible even on the largest transform sizes. There exist great differences between transforms of similar size, however. For instance, 178496 ($2^6 \cdot 2789$) points take 0.053ms while the next size, 178512 ($2^4 \cdot 3 \cdot 3719$) takes 58.02ms. FFTW transform times grow faster and the best transforms lie well above the best CUDA cases, but its scattering is limited. The plots show a clear tendency of the algorithm to assign transform sizes to specific execution paths, specially noticeable on the CUDA case.

Inputs that are powers of small prime numbers seem to give the best performance on both cases. This is consistent to the Cooley-Tukey algorithm used in both FFTW and CUDA implementations for FFT [8, 9].

Another operation that benefits from the high parallelization of GPU code is the transform product. While a CPU-based implementation of the EoM Detector would need to loop through the complex vectors sequentially to obtain the product, the GPU can perform numerous products simultaneously. The sheer number of processors in the GPU accelerates the operation noticeably. The performance for the product of the transforms for both implementations is shown in Figure 13. Note the shape of the CPU plot. Most points lie on the lower line, a linear polynomial fit passes just short of the bottom set of points. The upper points are less frequent, potentially pointing to page faults or cache misses and the associated processing time in the underlying OS. It is also worth noting that the transform size used for this experiment had the negative effect of placing the data point on the upper set of points. Better performance could theoretically be achieved for the product, but nothing close to the speed observed in GPU. The plot shows that for every transform size, the GPU implementation is always faster, even if a bit more scattered. We obtain about two orders of magnitude improvement on this operation alone.

Transfer of data points to the GPU is a step required by the GPU-accelerated EoM Detector (see Section 5.3.3) and does not exist in a CPU-based FFT implementation. Figure 14 shows the average time needed to complete copies in each direction. The GPU to CPU copy is noticeably slower. This fact is explained by the card manufacturer by historic reasons: traditionally data flows towards the display device from the CPU, not the opposite [1].

The data transfers between the CPU and GPU are the most expensive operations assuming a good choice of transform size. A possible improvement for the EoM detector therefore, would be to apply the same pipelining concept driving the system design to the data transfers and operations in the GPU. To accomplish

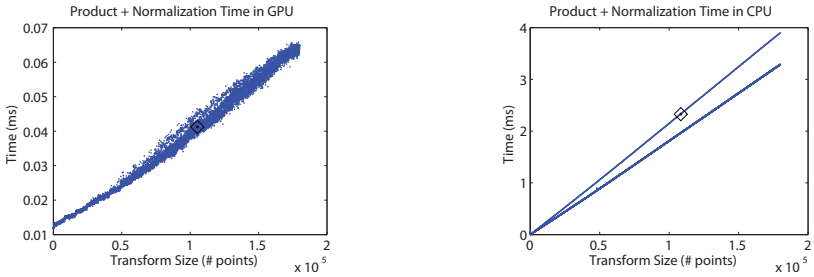

Figure 13: Product and scale completion time. CPU product time shows two sets of points.

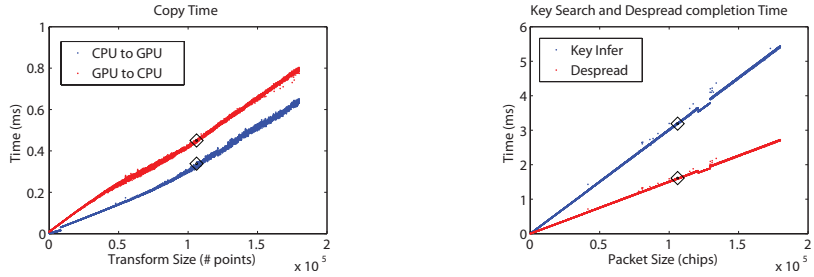

Figure 14: GPU copy time and CPU Key Infer and Despread time. Copy time to GPU is always slower due to hardware optimizations. Despreading is consistently twice as fast as Key Inferring.

this, we can make use of CUDA's asynchronous `memcpy` and concurrent kernel execution available in newer GPU models. The pipeline would start with the `mempcy` into the GPU. At this point, a new `memcpy` into the device can start, running concurrently with the FFT, product, and inverse FFT of the previous data set. Finally, output `memcpy` of the correlation occurs after the previous `memcpy` terminates. To illustrate the strategy, Table 9 lists the time for each EoM component in the 1024-bit packet, 20dB gain experiment, and Figure 15 depicts the mechanism.

Since bandwidth to and from the GPU device is limited, copy operations should not overlap each other. This leads to saving the FFT and product operations. A gain of 25% on the 8600 GPU, although its hardware does not support concurrent kernel and copy operations. The other two cards would sport 14% and 17% savings.

Key Inferring and Despreading Full Key inferring and despreading times on one CPU core in Figures 14, show that the TREKS receiver bottleneck resides on these two operations when using the accelerated EoM block. Since our design contemplates distributing each key search to every CPU core available, the time in average can at least be cut in half in multi-core systems.

Other considerations An obvious optimization would have the Key Inferring and Despreading modules running on the GPU to benefit from the clear time savings. In turn, even the software radio platform could run on the GPU.

Model	In Copy	FFT	Prod.	iFFT	Out Copy
GTX280	325	40.3	43.1	39.0	443
9800GT	287	49.1	67.2	45.4	494
8600GT	478	42.9	239	40.8	567

Table 9: FFT, product, and copy times in μs for a 102400 size transform for three GPUs.

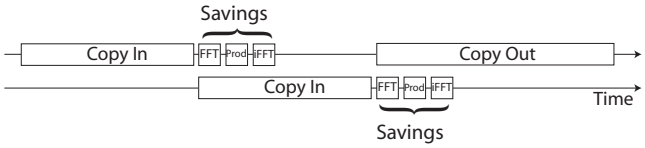

Figure 15: A possible improvement to the GPU-accelerated EoM detector.

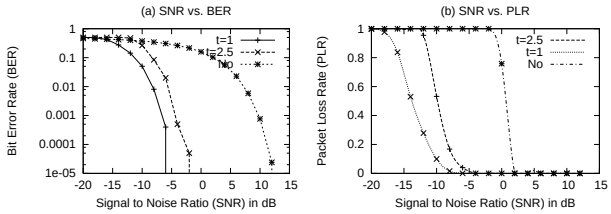

Figure 16: SNR vs. (a) BER (b) PLR with (Threshold = t*avg, t=1, t=2.5) and without (No) TREKS.

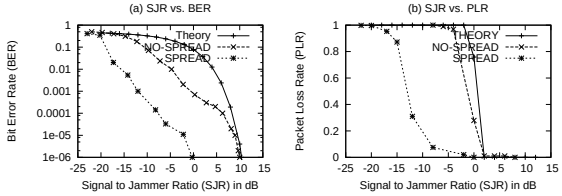

Figure 17: (a) SJR vs. BER (b) SJR vs. PLR with gaussian jammer

Not every algorithm translates well and easily to a massively parallel device, though. For instance the correlation operation multiplies the elements of the arrays point to point (just as in the EoM detector), but the addition operation acts as a barrier for parallelization, and the variable that contains it must be updated atomically. Because of this, threads residing on different blocks, which have no way of synchronizing or performing atomic operations, would need to reside in the same block, and thus forcing execution to occur in only one multiprocessor. It would not be too hard to design a Key Inferring procedure that executes on the GPU, but it is not clear that it would yield the savings seen on an FFT operation. As seen in Figure 12, even GPU implementations have border cases that run *slower* than in a CPU implementation.

We decided to run the Key Inferring and Despreading operations on the CPU mainly because the savings gained by the GPU-accelerated EoM Detector were enough to make our Radio Hardware the bottleneck of the system. Under a different choice of hardware, we may have needed to look into pushing more work into the GPU. For our choice of hardware, the decision was sound.

6.4.2 Jamming Resiliency

Now, we look at the analysis of TREKS performance in terms of jamming resiliency against various types of jammers as described above.

1. Continuous Jammer: First, we present the performance results from running TREKS in the presence of continuous jammers. We implemented continuous jammers, both using MATLAB for simulation and in real-time inside our test-bed for experimental evaluation.

    - Gaussian Jammer: Figures 16 and 17 show the PLR and the BER under TREKS in the presence of gaussian jammers as a function of an increasing SNR/SJR and two detection thresholds (for Simulation and one detection threshold of 2.5 for Experimentation). Note that due to the

limitations in implementing TREKS, the synchronization and EoM recovery process is imperfect, thus causing the experimental and simulation graph to not precisely match the theoretical graph even without spreading. However, the graphs show that the imperfect gain of $15 - 17$ dB (i.e., 20 to 50 resiliency gain, instead of 100) is consistent across simulation and experimentation. In terms of FPs, Figures 18 (Simulation) and 19 (Experimentation) show that both in simulation and experimentation, most of the FPs among potential EoMs are detected by the first two stages of key inferring. Thus FP does not impact TREKS computationally by much.

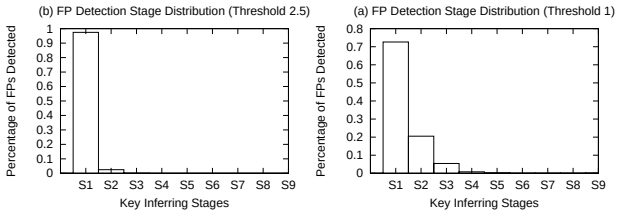

Figure 18: Distribution of the FP detection stage.

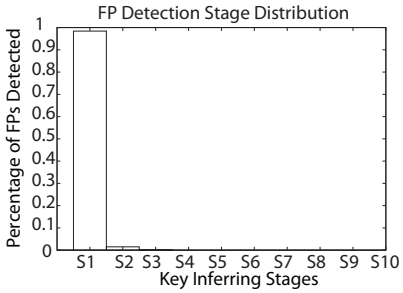

Figure 19: Distribution of False Positives

- MAC Jammer vs. BPSK Jammer: Figures 20 21 shows the performance of TREKS against MAC and BPSK jammer. Similar results were obtained through Simulation, however, due to space limitation, we choose to only present the experimental results for the performance of TREKS against MAC jammer and BPSK continuous jammer. We see that the the gain between MAC jammer with spreading and BPSK jammer without spreading is almost 20dB as expected theoretically with 100 spreading factor.

2. Non-continuous Jammer: For the evaluation of a non-continuous jammer who jams obliviously at a fixed rate λ, consider a time-slotted communication time as shown in Figure 21. Lets assume a sender message overlaps two consecutive timeslots, TS1 and TS2. Now, there are four possible scenarios for the occurrence of jammer message:

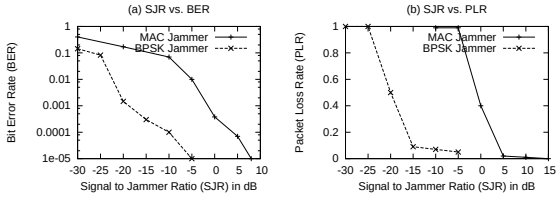

Figure 20: (a) SJR vs. BER (b) SJR vs. PLR with MAC jammer

Figure 21: Interleaved Timeslots

- **Scenario-1**: Only the first TS is jammed.
  Impact: Key inferring.
- **Scenario-2**: Only the second TS is jammed.
  Impact: EoM detection.
- **Scenario-3**: Both TS are jammed.
  Impact: Key inferring and EoM detection.
- **Scenario-4**: None of the TS are jammed.

Note that Scenario-4 implies no packet loss since there is no overlapping between the jammer and the data packet. Hence, we only show the performance results for Scenarios-1,2, and 3 in Figure 22(a). We see that there is hardly much difference between the MAC jammer and the Random jammer in terms of PLR. Scenario-1 has the most impact in terms of affecting the key inferring process, thus the PLR. Hence, we see that scenario-1 outperforms scenarios-2 and 3 for both jammers. In Figure 22(b), we compare Scenarios-3 and 4 and show that there is no incentive for the jammer to increase its JSR if its only objective is to increase the FPs, regardless of the jammer type.

Now, given all possible scenarios and a fixed λ (jamming rate), we calculate the expected PLR for a non-continuous jammer as follows:

$$E[PLR] = E_1 * \lambda(1-\lambda) + E_2 * \lambda(1-\lambda) + E_3 * \lambda^2 + E_4 * (1-\lambda)^2$$

where E_1, E_2, E_3,and $E_4(=0)$ are the expected PLR for above defined Scenarios-1,2,3,4 respectively.

Figure 23 shows the expected PLR for Scenarios-1,2,3 and 4. Depending on the budget, the jammer maximizes its impact on the receiver (PLR). We observe that MAC jammer and Random jammer attain their optimum approximately when $10 \leq JSR \leq 15$. Even in the best case for the jammer ($\mu = 1$), the jammer needs to spend 10 times more energy to reduce the throughput to 30%. For $\mu = 0.1$, the jammer would need to spend 100 more energy then the communicating to reduce the throughput to 30%. This shows that TREKS is pretty resilient even against the strongest of the jammers.

Lastly, any jammer strategy that does not use the destination MAC address as the seed for its spreading sequence, its jamming message gets reduced to a noise under TREKS with impact of a $10 \log n$dB less JSR [See Theorem 3]. This is why we model our jamming strategies under two kinds of jammers, one using the MAC address and another using random sequence to spread the packet. It is important to note that a jammer may send only portions of the message. This may cause a BER that is correctable on the receiver side depending on the type of error correction code used. If the portion of the message is large enough to cause 15% or more BER, then the jammer's cost may be reduced by 1/3. However, under

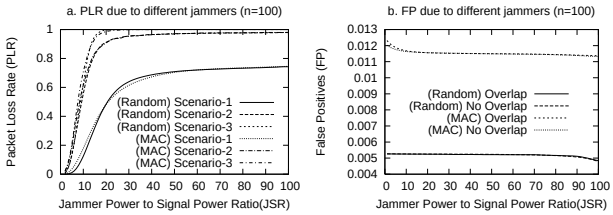

Figure 22: Jammer performance comparison.

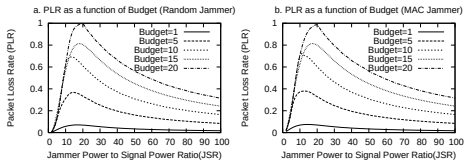

Figure 23: Jammer performance under fixed budget.

TREKS it is hard for the jammer to detect the communication, thus it cannot possibly synchronize its jamming with the sender message.

7 Conclusion and Future Work

In this paper, we propose new mechanisms, design and a full hardware and software implementation of a real-time spread spectrum direct sequence system that does not require any pre-shared secret between the communicating nodes. We make use of the USRP/GNU Radio platform and GPU acceleration to build our demonstrator. We demonstrate four orders of magnitude improvement of the computation cost in comparison with existing schemes. Our demonstrator can sustain (in terms of computation) a 1Mbps bit-rate spread by a factor of a 100 (i.e., 100 mega-chips per second) spread over 200Mhz bandwidth. We evaluate both the computation cost and the achieved jamming resiliency.

References

[1] *Nvidia cuda programming guide v.2.2*, 2009.

[2] Baruch Awerbuch, Andrea Richa, and Christian Scheideler, *A jamming-resistant mac protocol for single-hop wireless networks*, ACM PODC, 2008.

[3] E. Bayraktaroglu, C. King, X. Liu, G. Noubir, R. Rajaraman, and B. Thapa, *On the performance of ieee 802.11 under jamming*, Infocom, 2008.

[4] Michael A. Bender, Martin Farach-Colton, Simai He, Bradley C. Kuszmaul, and Charles E. Leiserson, *Adversarial contention resolution for simple channels*, SPAA, 2005.

[5] T. Brown, J. James, and A. Sethi, *Jamming and sensing of encrypted wireless ad hoc networks*, ACM MobiHoc, 2006.

[6] Agnes Chan, Xin Liu, Guevara Noubir, and Bishal Thapa, *Control channel jamming: Resilience and identification of traitors*, IEEE ISIT, 2007.

[7] J.T. Chiang and Y.-C. Hu, *Cross-layer jamming detection and mitigation in wireless broadcast networks*, MobiCom, 2007.

[8] James W. Cooley and John W. Tukey, *An algorithm for the machine calculation of complex fourier series*, Mathematics of Computation **19** (1965), no. 90, 297–301.

[9] M. Frigo and S.G. Johnson, *The design and implementation of fftw3*, Proceedings of the IEEE **93** (2005), no. 2, 216–231.

[10] S. Gilbert, R. Guerraoui, and C. Newport, *Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks*, OPODIS, 2006.

[11] Tao Jin, Guevara Noubir, and Bishal Thapa, *Zero pre-shared secret key establishment in the presence of jammers*, MobiHoc '09: Proceedings of the tenth ACM international symposium on Mobile ad hoc networking and computing (New York, NY, USA), ACM, 2009, pp. 219–228.

[12] Mingyan Li, Iordanis Koutsopoulos, and Radha Poovendran, *Optimal jamming attacks and network defense policies in wireless sensor networks*, INFOCOM, 2007.

[13] Guolong Lin and Guevara Noubir, *On link layer denial of service in data wireless lans*, Wirel. Commun. Mob. Comput. **5** (2005), no. 3, 273–284.

[14] NVIDIA, *Compute unified device architecture*.

[15] C. Popper and M. Strasser andS. Capkun, *Jamming-resistant broadcast communication without shared keys*, USENIX Security Symposium, 2009.

[16] GNU Radio, *Gnu radio*.

[17] Kasper Bonne Rasmussen, Srdjan Capkun, and Mario Cagalj, *Secnav: secure broadcast localization and time synchronization in wireless networks*, MobiCom, 2007.

[18] Ettus Research, *Universal software radio peripheral*.

[19] Marvin K. Simon, Jim K. Omura, Robert A. Scholtz, and Barry K. Levitt, *Spread spectrum communications; vols. 1-3*, Computer Science Press, Inc., NY, 1986.

[20] D. Slater, P. Tague, R. Poovendran, and B. Matt, *A coding-theoretic approach for efficient message verification over insecure channels*, 2nd ACM Conference on Wireless Network Security (WiSec), 2009.

[21] M. Strasser, C. Popper, and S. Capkun, *Efficient uncoordinated fhss anti-jamming communication*, MobiHoc, 2009.

[22] M. Strasser, C. Popper, S. Capkun, and M. Cagalj, *Jamming-resistant key establishment using uncoordinated frequency hopping*, ISSP, 2008.

[23] P. Tague, D. Slater, G. Noubir, and R. Poovendran, *Linear programming models for jamming attacks on network traffic flows*, WiOpt, 2008.

[24] Patrick Tague, Mingyan Li, and Radha Poovendran, *Probabilistic mitigation of control channel jamming via random key distribution*, PIMRC, 2007.

[25] Patrick Tague, Sidharth Nabar, Jim Ritcey, David Slater, and Radha Poovendran, *Throughput optimization for multipath unicast routing under probabilistic jamming*, PIMRC, 2008.

[26] W. Xu, K. Ma, W. Trappe, and Y. Zhang, *Jamming sensor networks: attack and defense strategies*, IEEE Network **20** (2006), no. 3, 41–47.