Analysis of Pseudo-Random Number Generators on UDOO Hardware

Jonathan Tan, Site Mao

April 30, 2014

Abstract

This paper details the implementation of software pseudo-random number generators (PRNG) based on the RSA and Rabin key cryptosystems on the mini PC platform, UDOO. The implementation of the modular arithmetic operations and the algorithms of the cryptosystem are discussed, with the end result being a high performance PRNG running on the UDOO ARM hardware. The performance metrics between our implementation and the native GNU/Linux PRNGs will be explored to determine the viability of the UDOO platform for cryptographically secure bit generation.

1 Introduction

RSA and Rabin are asymmetric public-key cryptosystems which can be used to encode input into a corresponding ciphertext. We will implement these cryptosystems and the arithmetic operations they rely on. From these cryptosystems, we can create bit generators by producing ciphertext, taking a number of the least significant bits of the output, using the ciphertext to produce more ciphertext, and repeating the process. We will then gather absolute and relative performance metrics of our bit generation on our UDOO platform.

2 RSA Cryptosystem

RSA is an asymmetric cryptosystem which is commonly used for secure data transmission. RSA involves a public key and a private key. The public key is given to everyone and is used to encrypt messages. Messages encrypted by the public key can only be decrypted in a reasonable amount of time with the private key.

After key generation, with \((n,e)\) as the public key, we can encrypt a message \(m\) by computing:

\[
c = m^e \pmod{n}
\]

With \((d)\) as the private key exponent, we can also decrypt the ciphertext \(c\) by computing:

\[
m = c^d \pmod{n}
\]

3 Rabin Cryptosystem

Rabin is an asymmetric cryptographic technique which, like RSA, is based upon the difficulty of factorization. Similarly, it uses a public and private key pair.
With the public key \( (n) \), we can encrypt the ciphertext \( c \) from message \( m \) by computing:
\[
c = m^2 \pmod{n}
\]

With the private key \( (c,r) \), we can decrypt the message \( m \) from the ciphertext \( c \) by computing:
\[
m \in \{0, ..., n - 1\} \text{ with } m^2 = c \pmod{r}
\]

### 4 Deterministic Random Number Generation

We create the random number generator by first starting with a seed message and then applying the RSA encryption protocol to it. From the output ciphertext, we then take a number of the least significant bits to be our randomly generated bit(s). To generate more bits, we simply repeat this protocol using the output ciphertext previously generated and applying the encryption protocol to it. With each generated ciphertext, we once again use a number of the least significant bits as our randomly generated bits and use the ciphertext as the input for the next computation.

We expect slow performance compared to other more optimized random number generators.