Randomly Failed! The State of Randomness in Current Java Implementations

Kai Michaelis, Christopher Meyer, and Jörg Schwenk
{kai.michaelis, christopher.meyer, joerg.schwenk}@rub.de

Horst Görtz Institute for IT-Security, Ruhr-University Bochum

Abstract. This paper investigates the Randomness of several Java Runtime Libraries by inspecting the integrated Pseudo Random Number Generators. Significant weaknesses in different libraries including Android, are uncovered.

1 Introduction

With a market share of 33-50% [1], [2], [3] Android is currently the most popular mobile OS. Each of the 331-400 million sold devices (cf. [3]) is able to run Java applications written in Java. Java provides interfaces for different Pseudo Random Number Generators (PRNGs), such as `SecureRandom`, which is intended for use by cryptographic schemes. But, only the API is specified - each library implements own algorithms. Designing secure and reliable PRNGs is a hard and complicated task [4], as well as implementing these algorithms correctly. One of the worst ideas is to implement own unproved PRNG constructs.

This paper examines the quality of the random numbers generated by common Java libraries. In detail, the PRNGs, intended for use in cryptographic environments, of Apache Harmony[1], GNU Classpath[2], OpenJDK[3] and BouncyCastle[4] are inspected. It is shown that the over-all entropy of the Android PRNG can be reduced to 64 bits. Beyond this, multiple weaknesses of entropy collectors are revealed. However, some of these weaknesses only occur under special conditions (e.g., unavailable `/dev/{u}random` device). We clearly point out that we are not going to discuss the quality of random numbers generated by PRNGs shipped with Operating Systems (OS). Discussions of OS provided (P)RNG facilities can be found at e.g., [5], [6], [7].

[1] `http://harmony.apache.org/`
[2] `http://www.gnu.org/software/classpath/`
[3] `http://openjdk.java.net/`
[4] `http://www.bouncycastle.org/`

2 Related Work

Problems related to weak pseudo random number generators (PRNGs) have been topic of several previously published papers. In 2012 Argyros and Kiayias investigated in [8] the state of PRNGs in PHP[5] and outlined flaws leading to attack vectors. Their results are based on insecure constructions of PRNGs introduced by custom algorithms.

Kopf outlined in [9] multiple cryptographic and implementational flaws in widespread Content Management Systems (CMS). These observations focused weak cryptographic constructs and pecularities of PHP. The resulting bugs are not caused by weak PRNGs, but by vulnerable custom algorithms in combination with implementational flaws.

Meyer and Somorovsky [10] uncovered a vulnerable use of `SecureRandom` in WSS4J[6]. A function responsible for nonce generation, used at various places in the framework, suffered from weak PRNG seeding.

Problems related to PRNGs are also topic of multiple CWEs (Common Weakness Enumeration)[7] that deal with the misuse or use of weak pseudo random number generators (cf. CWE 330-343).

In [11]) Lenstra et al. inspected millions of public keys of different types (RSA, DSA, ElGamal and ECDSA) and found keys violating basic principles for secure cryptographic parameters. According to the authors these weak keys could be a result of poorly seeded PRNGs.

More alarming results concerning key quality are presented by Heninger et al. in [7] pointing out that *"Randomness is essential for modern cryptography"*. Missing entropy was identified as a root cause for many weak and vulnerable keys. Additionally, the authors identified weak entropy problems under certain conditions in the Linux RNG.

A more theory based cryptanalytic set of attacks on PRNGs can be found at e.g., [12].

2.1 Contribution

The results of this paper focus on PRNG implementations of Java core libraries. Even thus all libraries implement the same functionality - generating (pseudo) *random* numbers - based on the same API, the algorithms for random number generation differ from library to library.

[5] `http://www.php.net`
[6] `http://ws.apache.org/wss4j/`
[7] `http://cwe.mitre.org`

The main contribution is an analysis of algorithms implemented in most commonly used `SecureRandom` generators. For this analysis tests provided by the Dieharder [13] and STS [14] testsuites are used to check for cryptographic weaknesses. Additionally manual code analysis uncovered algorithmical and implementational vulnerabilities.

3 Implementations & Algorithms

In Java, random numbers are derived from an initial seed. Two instances of a PRNG seeded with equal values always generate equal *random* sequences. The *cryptographic strong* PRNGs are accessed via the `SecureRandom` interface which is part of the Java Cryptography Architecture.

3.1 Apache Harmony - Android's version of `SecureRandom`

Apache Harmony, introduced in 2005 as an open source implementation of the Java Core Libraries published under the Apache License[8], became obsolete with the publication of SUN Microsystems' reference implementation in 2006. Although discontinued since 2011, the project is further devleoped as part of Google's Android platform.

```
1 // require cnt: counter >= 0, state: seed bytes and iv: previous output
2 iv = sha1(iv,concat(state,cnt));
3 cnt = cnt + 1;
4 return iv;
```

Listing 1.1. Apache Harmony's SecureRandom

The PRNG shipped with Android uses the SHA-1 [15] hash algorithm to generate pseudo random sequences and is ideally seeded by the random devices provided by the OS. Random numbers are generated by calculating hash sums of an internal state concatenated with a 64 bit integer counter and a padding (algorithm in Listing 1.1). The counter is, starting at zero, incremented each run of the algorithm. Additionally, a padding is required to fill remaining bits of a 85 bytes buffer. This padding follows the SHA-1 padding scheme: The last 64 bits hold the length *len* of the values to be hashed in bits. Any space between the end of the values and the length field is filled with a single '1' bit followed by zeros. The resulting hash sums are returned as pseudo random sequence. Figure 1 illustrates the state buffer.

[8] `http://www.apache.org/licenses/`

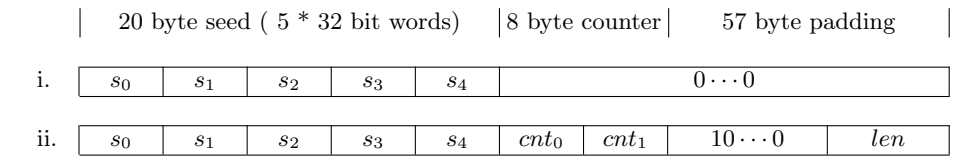


Fig. 1. The seed bytes $s_0, .., s_4$ in row i concatenated with a 64 bit counter c_0, c_1 (two 32bit words), padding bits and the length *len* as in row ii are hashed repeatedly to generate a stream of pseudorandom bytes.

3.2 GNU Classpath

The GNU Classpath project started in 1998 as the first open source implementation of Java's core libraries. The library is licensed under GPL [9] and is e.g., partly used by the IcedTea [10] project.

```
1 // require state: <= 512 bit buffer, iv: current Initialzation Vector
2 byte[] output = sha1(iv,state);
3 state = concat(state,output);
4 if(state.length > 512) {                  // in bits
5         iv = sha(iv,state [0:512]) ;      // first 512 bits
6         state = state[512:-1];            // rest
7         output = sha1(iv,state);
8 }
9 return output;
```

Listing 1.2. GNU Classpath's SecureRandom

The `SecureRandom` implementation (algorithm in Listing 1.2) of GNU Classpath is powered by a class `MDGenerator` that is parameterized with an arbitrary hash algorithm. A new `SecureRandom` instance is seeded with 32 bytes yielding an internal state as shown in row i of Figure 2. Based on this start value a digest is computed. The resulting hash (e.g. 160 bit in case of SHA-1) is returned as pseudo random value r_0. r_0 in turn is concatenated with the former seed forming the new state in row ii.
These bytes are hashed again yielding the second output r_1. Finally, the seed concatenated with the previous two hash values form the new state (c.f. row iii) whose digest is the third output value r_3. r_3 is again appended to the previous state resulting in the state illustrated in row iv. Each fourth r_i values a block overflow happens causing the implementation to hash the full block and use this hash as initialization vector (IV) for the new block. The only unknown value is the 32 byte long initial seed. All other information are known (as they are parts of former r_i value).

[9] `http://www.gnu.org/licenses/`
[10] `http://icedtea.classpath.org/wiki/Main_Page`

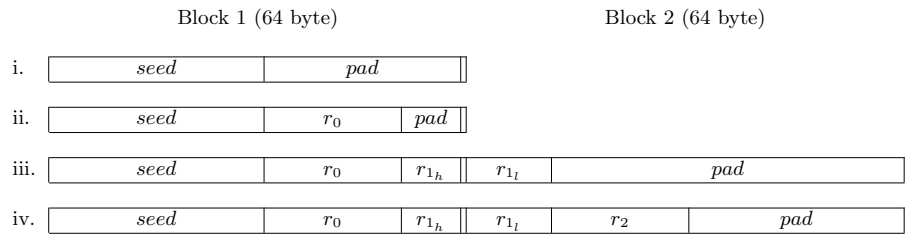


Fig. 2. Hashing the previous pseudo random bytes concatenated with the fomer seed produces the next output value.

GNU Classpath includes a "backup" seeding facility (algorithm in Listing 1.3) for Unix-like operating systems. The `VMSecureRandom` class is able to harvest entropy from the OS' process scheduler. Every call to the seeding facility starts 8 threads, each one incrementing a looped one byte counter. The parent thread waits until at least one of the 8 threads is picked up by the scheduler. The seed value is calculated by XORing each of the one byte counters. This forms the first seed byte. Accordingly, the next seed byte is generated the same way. When the requested amount of seeding material is gathered the threads are stopped.

```
1  int n = 0
2  byte[] S = new byte[8];
3  byte[] output = new byte[32];
4
5  for(int i = 0; i < 8; i++) {
6          S[i] = start_thread(n); // ''spinner'' incrementing a counter starting at n
7          n = (2*n) % pow(2,32);
8  }
9
10 while(!spinners_running())
11         wait();
12
13 for(int i = 0; i < 32; i++) {
14         output[i] = 0;
15
16         for(int j = 0; j < 8; j++)
17                 output[i] = output[i] ^ counter(S[j]);
18 }
19
20 for(int i = 0; i < 8; i++)
21         stop_thread(S[i]);
22
23 return output;
```

Listing 1.3. GNU Classpath's entropy collector

3.3 OpenJDK

As the direct successor of the Java Development Kit (JDK), OpenJDK [11] provides not only Java core libraries, but additionally a Java compiler and a virtual machine. OpenJDK is the official reference implementation of Java and open source licensed under GPL. The project is supported by major vendors such as IBM or SAP.

```
1  // require state: seed bytes, iv: SHA-1 standard IV
2  byte[] output = sha1(iv,state);
3  byte[] new_state = (state + output + 1) % pow(2,160);
4
5  if(state == new_state)
6          state = (new_state + 1) % pow(2,160);
7  else
8          state = new_state;
9
10 return output;
```

Listing 1.4. OpenJDK's SecureRandom

OpenJDK uses the OS' PRNG in conjunction with a similar scheme as the previously mentioned libraries (algorithm in Listing 1.4). An initial 160 bit seed value is hashed using SHA-1. The result is XORed with the output of the OS specific PRNG and returned as pseudo random bytes. This output is added to the initial seed plus 1 modulo 2^{160}. An additional check compares the new seed to the old one preventing the function from getting trapped in a fixed state where $s_i + \mathit{SHA\text{-}1}(s_i) + 1 \equiv s_i \bmod 2^{160}$.

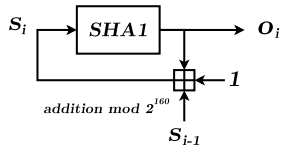


Fig. 3. SecureRandom of OpenJDK. In each step the current state (seed) s_i is compressed yielding output o_i. The sum of o_i, s_{i-1} and 1 give new state s_{i+1}.

The integrated entropy collector (cf. Listing 1.5) uses multiple threads to gather randomness. In contrast to GNU Classpath, only one thread increments a counter. Subsequently, new threads are started suspending five times for 50ms to keep the scheduler busy. Before continuing the lower 8 bits of the current counter value pass an S-Box. The XOR sum of all 5 counters is returned as random byte. The entropy collector enforces mandatory runtime (250ms) and counter value (64000). Even after enough

[11] `http://openjdk.java.net/`

seed is produced the entropy collector continues to run. The seed bytes are hashed together with entries of `System.properties` and the result is used as seed.

```
1  counter = 0;
2  quanta = 0;
3  v = 0;
4
5  while(counter < 64000 && quanta < 6) {
6          start_thread()   // loops 5 times, sleeping 50ms each
7          latch = 0;
8          t = time();
9
10         while(time() - t < 250ms)       // repeat for 250ms
11                 latch = latch + 1;
12
13         counter = counter + latch;
14         v = v ^ SBox[latch % 255];
15         quanta = quanta + 1;
16 }
17
18 return v;
```

Listing 1.5. OpenJDK's entropy collector

3.4 The Legion of Bouncy Castle

Bouncy Castle is not a complete core library, but a security framework for the Java platform. This includes a `SecurityProvider` as well as cryptographic APIs for standalone usage. The project provides enhanced functionality, as well as support for a broader range of cryptographic algorithms compared to default OpenJDK. Bouncy Castle implements various pseudo random generators as well as a threaded entropy collector akin of the one in OpenJDK.

The `DigestRandomGenerator` (cf. Listing 1.6) uses a cryptographic hash algorithm to generate a pseudorandom sequence by hashing an internal secret state of 160 bits with a 64 bit *state counter* producing 160 bits of pseudorandom output at once. After each hash operation the state counter is incremented. The initial secret state is received from a seeding facility. Every tenth hash operation on the state, a 64 bit *seed counter* is incremented and a new secret state is generated by hashing the current state concatenated with the seed counter. This new secret state replaces the old one, where the state counter remains at its previous value. When another pseudorandom value is requested from the `DigestRandomGenerator` instance, this new secret state is hashed with the state counter, producing a cryptographic checksum to be returned to the caller as random byte array.

```
1  // require seedBuffer: 160bit seed, stateBuffer: 160bit array, seedCounter and
       stateCounter: 64bit integers
2
3  if(stateCounter % 10 == 0) {
4          stateBuffer = sha1(iv,concat(seedBuffer,seedCounter));
5          seedCounter += 1;
6  }
7
8  byte[] output = sha1(iv,concat(stateBuffer,stateCounter));
9  stateCounter++;
10
11 return output;
```

Listing 1.6. DigestRandomGenerator

The `VMPCRandomGenerator` is based on Bartosz Zoltak's Variable Modified Permutation Composition one-way function [16].

The `ThreadedSeedGenerator` implements a threaded entropy collector scheme. Only two threads are used: one thread increments a counter in a loop, whereas the other waits 1ms until the counter has changed. The new value is appended to an output array. The incrementing thread is teared down after all random bytes are collected. The generator offers two modes of operation: a) "slow" mode where only the least significant bit of every byte is used and b) "fast" mode where the whole byte is used.

```
1  // require count: number of seed bytes needed, fast: enable "fast" mode
2  byte[] output = byte[count];
3  t = start_thread() // increments a counter in a loop
4  int last = 0;
5
6  // use bits in "slow" mode
7  if (! fast )
8          count *= 8;
9
10 for(int i = 0; i < count; i++) {
11         while(counter(t) == last)
12                 sleep(1);
13
14         last = counter(t);
15         if( fast )
16                 output[i] = (byte)last;
17         else
18                 output[i/8] = (last % 2) | (output[i/8] << 1);
19
20 }
21 stop_thread(t);
22 return output;
```

Listing 1.7. ThreadedSeedGenerator

4 Methodology

Manual code review was performed for each of the introduced PRNGs. During code review the code was checked for implementation flaws and obvious bugs. Aside from code review blackbox tests on the output were preformed to grade the entropy. For this the Dieharder test suite [13] for (pseudo) random number generators was used, as well as Monobit, Runs, and Serial tests from the STS [14] suite.

While these tests can not replace cryptanalysis they still uncover bias and dependency in the pseudo random sequence. For every test exists an expected distribution of outcomes. Test runs produce a value that is compared to the theoretical outcome. A p-value, describing the probability that a real RNG would produce this outcome, between 0 and 1 is computed. A p-value below a fixed significance level $\alpha = 0.001$ indicates a failure of the PRNG with probability $1 - \alpha$. Dieharder differs from this methodology as it relies on multiple p-values to evaluate the quality of a PRNG. It is possible (and expected) for a good RNG to produce "failing" p-values. Instead of grading a single outcome, 100 p-values are computed and the distribution of these values is compared to an uniform one. This generates a final p-value grading the quality of the PRNG.

No cryptanalysis was performed - we analyzed the algorithms and evaluated the quality of randomness by using special purpose testsuites.

5 Results

This section highlights prospective weaknesses of the implementations and evaluates the quality of generated randomness. Due to space limitations, only conditions targeting the observed weaknesses are regarded - the statistical graphs can be found in the Appendix in Section 7.

Striking about all implementations of `SecureRandom` is the limited state size. In OpenJDK and GNU Classpath adding more entropy ($>$ 160bit) to an instance will not enhance security. This limitation is alarming, since it renders the PRNGs useless for key generation $>$ 160 bit (as e.g., in AES' case). Only Apache Harmony relies on a 512 bit buffer.

5.1 Apache Harmony

Apache Harmony revealed multiple weaknesses caused by implementation bugs. As a part of Android a plethora of cryptographic functions [17] rely on this PRNG. One of the bugs addresses directly the Android platform, where as the second one only targets Apache Harmony.

Weaknesses. *FIRST* - When creating a *self seeding* `SecureRandom` instance (by calling the constructor without arguments and subsequent `setSeed()` call), the code fails to adjust the byte offset (a pointer into the state buffer) after inserting a start value. This causes the 64 bit counter and the beginning of the padding (a 32 bit word) to overwrite parts of the seed instead of appending to it. The remaining 64 bits of entropy render the PRNG useless for cryptographic applications[12].

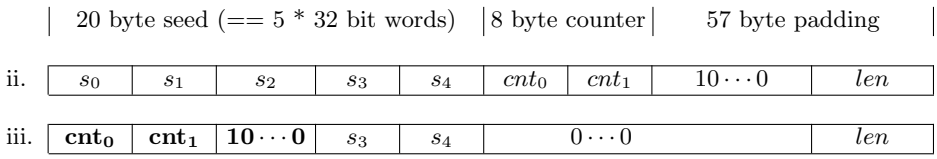


Fig. 4. Instead of appending (c.f. row ii), the counter and the succeeding padding overwrite a portion of the seed, yielding row iii.

SECOND[13] - When running under a Unix-like OS a new `SecureRandom` instance is seeded with 20 bytes from the *urandom* or *random* device. If both are unaccessible the implementation provides a fall-back seeding facility (cf. Listing 1.8): seed is gathered from the `random()` PRNG of the GNU C library, which is seeded it via `srandom()` with the UNIX-time, processor time and the pointer value of a heap-allocated buffer. After seeding, `random()` is used to generate seed bytes for `SecureRandom`. Before these bytes are returned the most significant bit is set to zero (mod 128) - this behavior is neither documented, expected nor explained.

```
1 char *seed = malloc(20);
2 srandom(clock() * time() * malloc()) % pow(2,31));
3 for(int i = 0; i < 20; i++)
4         seed[i] = random() % 128;
5
6 return seed;
```

Listing 1.8. Apache Harmony's `getUnixSystemRandom`

The missing entropy is not compensated (e.g., by requesting > 20 bytes). As a consequence, the effective seed of a `SecureRandom` instance is only 7/8 for each requested byte, degrading security (of only 64 bits due to the *first* bug) by another 8 bits to 56 bits (s_3 and s_4 are 2 * 32 bit words == 8 byte). Even worse, the argument of `srandom()` in the GNU C library is of type `unsigned int` while Harmony reduces the argument modulo `INT_MAX` (defined in *limits.h*) - the maximum value for *signed int*s. This limits the entropy of a single call to the seeding facility to 31 bits.

[12] The bug was communicated to the Google Security Team.
[13] This bug is not part of the Android Source

Quality of entropy collectors. Generating 10MiB of seed - two consecutive bytes are interpreted as a single point - lead to the chart in Figure 9. It shows a lack of values above 127 in each direction. This test targets the *second* bug. The *first* bug limits the security to 64/56 bit, depending on the seeding source. This *seed space* of only 2^{64} elements is within reach of Brute-Force attacks utilizing GPGPUs/FPGAs (cf. [15]).

5.2 GNU Classpath

The library's entropy collector revealed inconsistencies regarding normal distribution of the output bits which could result in vulnerabilities.

Weaknesses. - The implementation of GNU Classpath contains a significant weakness related to internal states. As long as a new generated random value concatenated with the previous state does not overflow the current block, all hash computations are done with the same IV. States in rows i and ii from Figure 2 are both hashed with the SHA-1 standard IV. The obtained state in row iii overflows the first block - the hash value of this first block is used as input for the hash of the second block in row iii and as IV for the succeeding computation of $r_{1_l}|r_2$ concatenated with r_3. The state is reduced from 32 bytes seed to only 20 unkown IV bytes.

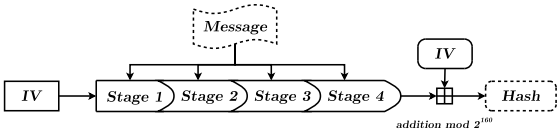


Fig. 5. Schematic view of a single SHA-1 iteration. Values in the dotted boxes are known when used in the SecureRandom class from GNU Classpath.

The IV is the only unknown value. To break the algorithm an attacker has to discover the IV for a known message producing a known checksum. While no such attack on SHA-1 has been published yet, this scenario is different from a preimage attack (c.f., [18], [19]). The compression function in SHA-1, ignoring the final 32 bit additions is invertible(c.f., [20]) for a given message. Thus, the addition of the secret IV (see Figure 5) remains the only hurdle in breaking the implementation.

Quality of entropy collectors. The seeding facility harvests entropy from multiple threads competing for CPU time. While the behaviour of the scheduler is difficult to predict it is possible to influence it. Only one of eight threads is expected to be scheduled. During seed extraction this precondition is not checked, enabling an attacker to fill (parts) of the output array with identical values by preventing threads to run (e.g., by creating high process load).

To test this construction under worst conditions, 11GiB of seed values were generated. To simulate high load 8 processes were run simultaneously. Each process queried for 16384 bytes while iterating in a loop. At first inspection, the resulting random seed revealed large (up to 2800 bytes) "holes" where random bytes were equal. As a result, the algorithm did not pass any blackbox tests. While the test conditions were extreme, they still expose a weakness in this entropy harvester.

The first 10MiB of seed are sampled on a graph (c.f., Figures 10). As can be seen, Classpath was unable to fill the whole space, leaving 64 by 64 large patches in the second and forth quadrant, as well as 32 by 32 along the diagonal when running under heavy load. In contrast, under normal conditions the entropy collector produced a well-balanced pane.

5.3 OpenJDK

The overall impression of `SecureRandom`'s reference implementation suggests a thoughtful and mature implementation.

Weaknesses. Code review bared no obvious weaknesses. Still, if an attacker is able to learn any internal state (s_i in Figure 3) all following states $s_j \quad \forall j > i$ and outputs $o_j = \mathit{SHA\text{-}1}(s_j)$ can be predicted if the OS PRNG (*/dev/{u,}random*) is unavailable[14].

Quality of entropy collectors. OpenJDK's strategy for seed generation in abstance of OS support is similar to `VMSecureRandom` of GNU Classpath. OpenJDK supplements the threaded entropy collector by enforcing minimal limits on runtime before extracting bytes and adding a substitution box. The unhashed[15] seed bytes were evaluated. The implementation revealed to be magnitudes slower, resulting in only 20MiB of seed generated by 8 processes running simultaneously. From the 114 blackbox tests only 30 were passed, whereas 12 failed with a p-value < 0.05 and 72 with p-value $< 10^{-6}$. The random bytes had grave difficulties with the STS tests, failing Monobit, Runs and the first eight Serial tests. This indicates poor variance in single bits and tuples up to eight bits. Nevermind, the resulting graph is filled very balanced (cf. Figure 11).

5.4 The Legion of Bouncy Castle

Bouncy Castle's implementation is also a hash-based algorithm (`DigestRandomGenerator`). No obvious bugs were found during code review. In contrast, the entropy collector `ThreadedSeedGenerator` revealed difficulties to generate sufficient random bytes.

[14] until the user manually reseeds the SecureRandom instance
[15] before hashing with additional `System.properties` values

Weaknesses. In `DigestRandomGenerator` the seed is modified every tenth call (cf. Section 3.4). This may increase the period of the PRNG and hinders an attacker aware of the secret state to calculate previous outputs. Predicting all succeeding outputs is still possible as the counter values can be guessed by observing the amount of values produced yet.

The VMPC function used in `VMPCRandomGenerator` is known to be vulnerable to multiple distinguishing attacks (cf. [21], [22], [23]).

Quality of entropy collectors. The entropy collector checks if a counter incremented in another thread has changed. Under heavy load the counter often differs only about 1 incrementation.

The seed generator running in "fast" mode passed most of the STS tests including Monobit, Runs and Serial up to eight bit tuple size. In "slow" mode, long sequences of ones and zeros caused to fail the Runs test, as well as the Serial test for 2-bit blocks. Both modes were still able to fill the pane after 10MiB of one byte samples (c.f., Figure 12).

5.5 Vulnerabilities Summary

The Tables depitcted in Figures 6, 7, 8 finally summarize our results[16].

Library	**Security when seeded from**	
	OS PRNG	integrated source
Apache Harmony	64 bit	31 bit
GNU Classpath	160 bit	dubious$^{\text{(see text)}}$
OpenJDK	160 bit	dubious$^{\text{(see text)}}$

Fig. 6. Summary of the results from SecureRandom audits.

Library	**Source of entropy**	**Passed Blackbox tests**
Apache Harmony	UNIX-time, processor time, heap pointer	0/114
GNU Classpath	Thread scheduler	0/114$^{\text{(see text)}}$
OpenJDK	Thread scheduler, `System.properties`	30/114

Fig. 7. Summary of the integrated seed generators.

6 Conclusion

The `SecureRandom` PRNG is the primary source of randomness for Java and is used e.g., by cryptographic operations. This underlines its importance regarding security. Some of fallback solutions of the investigated

[16] Bouncy Castle is partly missing since it ships with replacements for `SecureRandom` and configurable entropy collectors. And is thus not directly comparable

Library	Component	Vulnerability	only if OS PRNG unavail.
Apache Harmony	`SecureRandom`	Entropy limited to 64bit.	
Apache Harmony	`SecureRandom`	Entropy limited to 31bit.	X
Gnu Classpath	`SecureRandom`	Possibly predictable if later IVs are known.	
Gnu Classpath	Entropy Collector	Suggestible by other threads.	X
Bouncy Castle	`VMPCRandomGenerator`	Vulnerable to distinguishing attacks.	

Fig. 8. Overall summary of all uncovered vulnerabilities.

implementations revealed to be weak and predict- or capable of being influenced. Very alarming are the defects found in Apache Harmony, since it is partly used by Android. Long update intervals and missing pre-OS-release patching of device manufacturers in the past (c.f., [24]) may cause this bug to remain in the Android ecosystem for months or even years. Although, some libraries provide acceptable randomness, the use of hardware RNGs instead of PRNGs is recommended for critical purposes.

References

1. Gupta, A., Cozza, R., Nguyen, T.H., Milanesi, C., Shen, S., Vergne, H.J.D.L., Zimmermann, A., Lu, C., Sato, A., Glenn, D.: Market Share: Mobile Devices, Worldwide, 1Q12. Technical report, Garnter, Inc. (May 2012)
2. Nielsen: Two Thirds of New Mobile Buyers Now Opting For Smartphones. Technical report, The Nielsen Company (June 2012)
3. Bennett, J.: Android Smartphone Activations Reached 331 Million in Q1'2012 Reveals New Device Tracking Database from Signals and Systems Telecom . Technical report, Signals and Systems Telecom (May 2012)
4. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. J. ACM (August 1986)
5. Gutterman, Z., Pinkas, B., Reinman, T.: Analysis of the Linux Random Number Generator. In: IEEE Symposium on Security and Privacy. (2006)
6. Dorrendorf, L., Gutterman, Z., Pinkas, B.: Cryptanalysis of the windows random number generator. In: ACM Conference on Computer and Communications Security. (2007)
7. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In: Proceedings of the 21st USENIX Security Symposium. (August 2012)
8. Agyros, G., Kiayias, A.: I forgot your password: Randomness attacks against PHP applications. In: Proceedings of the 21st USENIX Security Symposium, USENIX Association (2012)
9. Kopf, G.: Non-Obvious Bugs by Example. `http://gregorkopf.de/slides_berlinsides_2010.pdf` (2010)
10. Meyer, C., Somorovsky, J.: Why seeding with System.currentTimeMillis() is not a good idea... `http://armoredbarista.blogspot.de/2012/01/why-seeding-with-systemcurrenttimemilli.html` (January 2012)

11. Lenstra, A., Hughes, J., Augier, M., Bos, J., Kleinjung, T., Wachter, C.: Public Keys. In: Advances in Cryptology – CRYPTO 2012. Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012)
12. Kelsey, J., Schneier, B., Wagner, D., Hall, C.: Cryptanalytic Attacks on Pseudorandom Number Generators. In: FSE. Lecture Notes in Computer Science, Springer (1998)
13. Brown, R.G., Eddelbuettel, D., Bauer, D.: Dieharder: A Random Number Test Suite. Technical report, Duke University (2012)
14. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert1, A., Dray, J., Vo, S., III, L.E.B.: A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications. Technical report, National Institute of Standards and Technology (NIST) (April 2010)
15. Lee, E.H., Lee, J.H., Park, I.H., Cho, K.R.: Implementation of high-speed SHA-1 architecture. IEICE Electronics Express (2009)
16. Zoltak, B.: VMPC One-Way Function and Stream Cipher. In: Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers. Lecture Notes in Computer Science, Springer (2004)
17. Google Inc.: javax.crypto — Android Developers. (July 2012)
18. McDonald, C., Hawkes, P., Pieprzyk, J.: Differential Path for SHA-1 with complexity O(2ˆ52). IACR Cryptology ePrint Archive (2009)
19. Wikipedia: Preimage attack — Wikipedia, The Free Encyclopedia (2012) [Online; accessed 24-August-2012].
20. Handschuh, H., Knudsen, L., Robshaw, M.: Analysis of SHA-1 in Encryption Mode. In: Topics in Cryptology — CT-RSA 2001. Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2001)
21. Tsunoo, Y., Saito, T., Kubo, H., Shigeri, M., Suzaki, T., Kawabata, T.: The Most Efficient Distinguishing Attack on VMPC and RC4A (2005)
22. Maximov, A.: Two Linear Distinguishing Attacks on VMPC and RC4A and Weakness of RC4 Family of Stream Ciphers (Corrected). (2007)
23. Shunbo Li, Yupu Hu, Y.Z.Y.W.: Improved cryptanalysis of the vmpc stream cipher. Jurnal of Computational Information Systems (2012)
24. Sverdlove, H., Brown, D., Cilley, J., Munro, K.: Orphan Android: Top Vulnerable Smartphones 2011. Technical report, Bit9, Inc. (November 2011)

7 Appendix

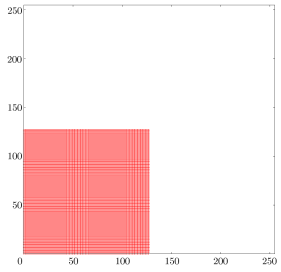
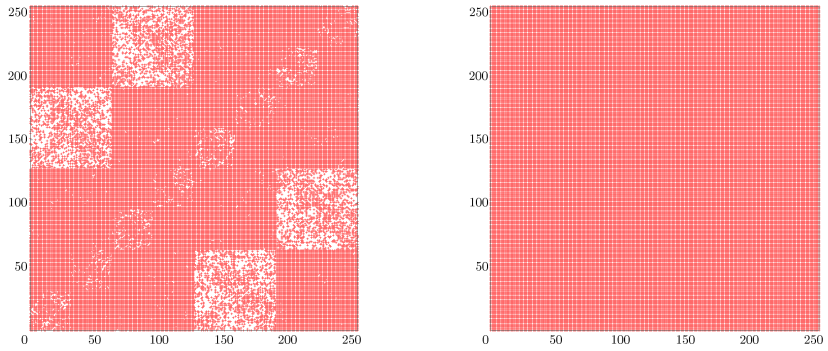


Fig. 9. Distribution of 2-tuples from Apache Harmonys integrated seeding facility.



(a) heavy (b) normal

Fig. 10. Distribution of 2-tuples from `VMSecureRandom` under heavy (a) and normal (b) workload as implemented in GNU Classpath.

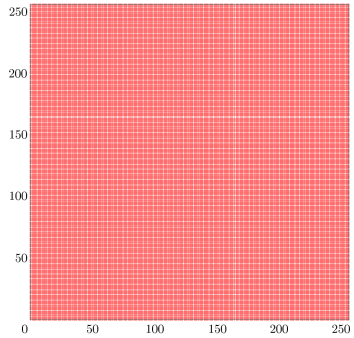
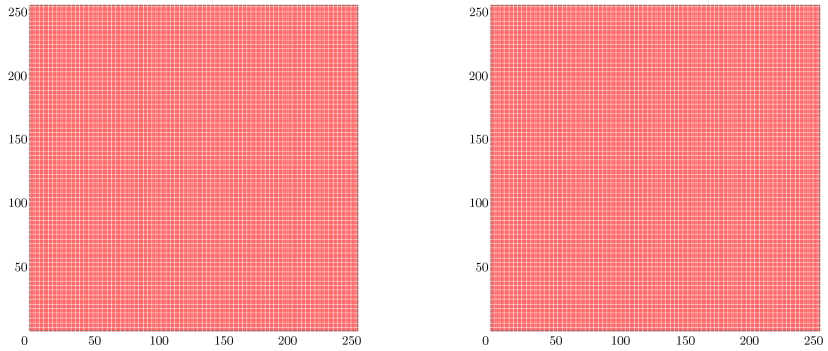


Fig. 11. Distribution of 2-tuples from the entropy collector in OpenJDK.



(a) slow (b) fast

Fig. 12. Distribution of 2-tuples from the `ThreadedSeedGenerator` in Bouncy Castle's lightweight crypto library for Java running on "slow" (a) and "fast" (b)