Pseudo-Random Number Generator Verification: A Case Study

Felix Dörre and Vladimir Klebanov

Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, 76131 Karlsruhe, Germany
felix.doerre@student.kit.edu
klebanov@kit.edu

Abstract. In 2013, a monetarily moderate but widely noted bitcoin theft drew attention to a flaw in Android's pseudo random number generator (PRNG). A programming error affecting the information flow in the seeding code of the generator has weakened the security of the cryptographic protocol behind bitcoin transactions.
We demonstrate that logic-based verification can be efficiently applied to safeguard against this particular class of vulnerabilities, which are very difficult to detect otherwise. As a technological vehicle, we use the KeY verification system for Java. We show how to specify PRNG seeding with information flow contracts from the KeY's extension to the Java Modeling Language (JML) and report our experiences in verifying the actual implementation.

1 Introduction

In 2013 a security incident [3] resulting in theft of bitcoin gained significant public attention. While the total monetary damage was at $5700 relatively modest, the ease and low risk of attack were notable. The perpetrators were never identified, and the exact circumstances of the attack remain to a degree speculation. Yet, the attack promptly raised public awareness of a vulnerability in the implementation of the pseudo-random number generator (PRNG) in Android [13]. Soon thereafter, Google replaced the Android PRNG.

The vulnerability in question is an instance of the "squandered entropy" problem, where entropy (i.e., information difficult to guess for an attacker) flows from a source to a destination, and some or all of it is lost (i.e., replaced by a constant or predictable value) underway due to a programming error. Concretely, out of 20 byte of entropy requested from the OS kernel to seed the PRNG, 12 did not reach the generator's internal state, significantly diminishing the quality of the PRNG output. This kind of problem is difficult to detect (as explained later on) and reoccurs periodically. Other notable instances include the Debian weak key disaster [5] (PRNG broken for two years), or the recent FreeBSD-current PRNG incident [7] (PRNG broken for four months), but there are also many others.

So far, entropy squandering is typically detected by manual code inspection, as, e.g., in [13]. With this paper, we present the first, to our knowledge, case study on formally verifying the implementation of a real-world PRNG.[1] We show that absence of entropy squandering can be efficiently specified (in terms of information flow) and practically verified with current deductive verification technology. In fact, we argue that formal verification is the tool of choice for addressing the problem.

We have chosen the above-mentioned Android PRNG as the subject of the case study as it allows us to illustrate how code verification can protect against bugs that have indeed occurred in the wild. On the technical side, the PRNG is implemented in Java, while we have experience in verification of Java programs.

The bitcoin theft incident. The presumed genesis of the attack is as follows. Bitcoin operates a public database of all transactions, the *block chain*. Each transaction is cryptographically signed by its initiator using the ECDSA scheme [8]. Creating ECDSA signatures requires a per-transaction nonce. Partial predictability of nonces allows for attacks like [14], but using the same nonce for two transactions signed by the same key—which is what probably happened—constitutes a catastrophic security failure. Anyone can easily identify this case from the information recorded in the block chain, reconstruct the victim's private key, and divert their money to a bitcoin address of choice. No intrusion into the victim's system is necessary. A loss of seed entropy in the PRNG used for generating nonces increases the probability of the breach.

2 Inner Workings of the Android PRNG

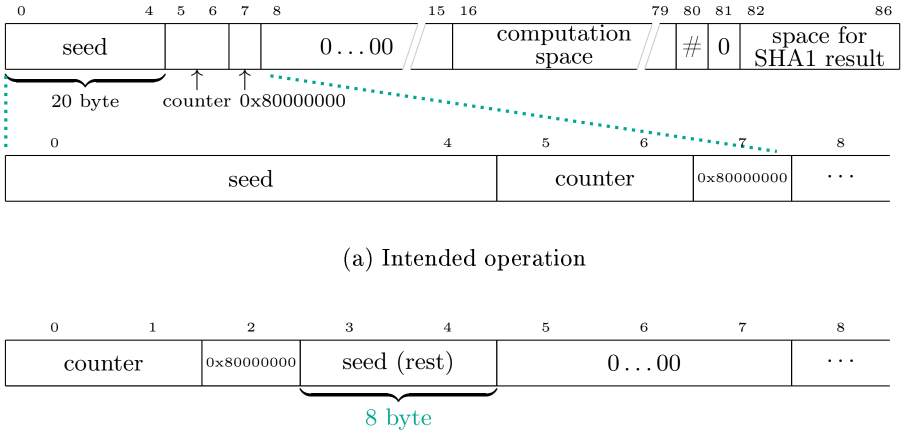
The origin of the Android PRNG lies in the Apache Harmony project, a clean room reimplementation of the Java Core Libraries under the Apache License. The PRNG was part of the Android platform up to and including Android 4.1. The PRNG consists of the main class `org.apache.harmony.security.provider.crypto.SHA1PRNG_SecureRandomImpl` and the auxiliary class `SHA1Impl`.

Overall size of the PRNG is slightly over 300 LOC, though not all functionality was exercised in this case study. The code is monolithic, dense, and hard to follow. The are many comments, but these use jittering terminology, are not always clear, and are at times inconsistent with the code. The description in [13] was instrumental in facilitating our understanding of the implementation.

The main PRNG method `engineNextBytes(byte[] bytes)`, shown schematically in Listing 1.1, fills the caller-supplied array `bytes` with pseudo-random bytes. The PRNG operates in cycles, each cycle generating 20 pseudo-random bytes. If the caller requests more bytes, several cycles are performed; if the caller requests fewer bytes, the surplus generated bytes are stored for later usage.

The main component of the PRNG state is an `int[]` array of length 87, somewhat inappropriately named `seed` (Figure 1). The front part of this array is populated with the externally-provided entropy (i.e., the actual seed). The PRNG can either be seeded manually by calling `setSeed()` or automatically. In

[1] Artifacts available at `http://formal.iti.kit.edu/~klebanov/pubs/vstte2015/`.



(a) Intended operation

(b) Effect of the bug

Fig. 1: Structure of the Android PRNG's main array (1 word = 1 `int` = 4 bytes)

the latter case, the PRNG is seeded with 20 byte of entropy requested from the OS kernel on first invocation of `engineNextBytes()` (Listing 1.1, line 6). This so-called *self-seeding* mode was typically considered preferable as less error-prone, and it is indeed the scenario we are considering here.[2]

In cycle k, the pseudo-random bytes are computed as the pseudo-SHA-1 hash of the seed (words 0–4 in Figure 1a) concatenated with the cycle counter k (as a 64-bit integer in words 5–6). The computation (Listing 1.1, line 17) makes use of the scratch space in words 16–79, and its result is stored in words 82–86. The latter are subsequently unpacked into bytes that form the output of the cycle. The computed hash is not quite the standard SHA-1 hash, as only in cycle zero, the initialization vector defined in the SHA-1 standard is used. In a cycle $k > 0$, the initialization vector is formed by the 20 pseudo-random bytes generated in cycle $k - 1$.

To compute the hash, the seed and the cycle counter have to be suffixed by a standard-defined SHA-1 padding. Now, the PRNG keeps track of the length of the seed in word 80. The essence of the vulnerability is that a stale value of this length (i.e., zero) is used after initializing the seed in the self-seeding mode.[3] As a consequence, the cycle counter and the SHA-1 padding constant overwrite words 0–2, leaving only two words of the original seed (Figure 1b). The effective inflow of entropy into the PRNG amounts thus to 8 instead of 20 bytes.[4]

[2] Google changed its stance on this matter several times, as the PRNG implementation was updated. As far as we are aware, self-seeding is the recommended mode again.

[3] There are more irregularities in the padding code, but they are irrelevant here.

[4] The PRNG also contains a native backup component in case the kernel does not provide an entropy source. Incidentally, this component contained two more instances of entropy squandering, though these were much simpler technically.

```
⟨contract⟩        ::= determines ⟨determinandum⟩ \by ⟨determinans⟩ ;
⟨determinandum⟩   ::= ⟨expr_seq⟩ | \pre( ⟨expr_seq⟩ ) | \post( ⟨expr_seq⟩ )
⟨determinans⟩     ::= ⟨expr_seq⟩ | \pre( ⟨expr_seq⟩ ) | \post( ⟨expr_seq⟩ )
⟨expr_seq⟩        ::= \nothing | ⟨expression⟩ | ⟨expression⟩ , ⟨expr_seq⟩
```

where ⟨*expression*⟩ is an arbitrary JML expression (i.e., term or formula)

Fig. 2: Concrete grammar for information flow contracts in JML*

3 Information Flow Verification with the KeY System

The KeY verification system. The case study has been carried out using the KeY deductive verification system for Java [1, 9]. The reasons for choosing KeY are our familiarity with it due to our involvement with its development, good programming language support (KeY supports, for instance, 100% of the Java Card standard), as well as a frontend for specifying information flow in programs. On the other hand, the approach that we apply is not tool-specific and could be reenacted with another deductive verification system.

The frontend of KeY takes as input a Java program annotated in the Java Modeling Language (JML) [12]. The backend is a theorem prover for *Dynamic Logic* (DL), which can be seen as a generalization of Hoare logic. Reasoning about programs is based on symbolic execution. Proof construction is guided by the user via program annotations and/or interacting with the prover GUI. All proof steps are recorded and can be inspected via an explicit proof object.

For loop- and recursion-free programs, symbolic execution is performed in a fully automated manner. Loops can either be unrolled or abstracted by a user-provided loop invariant. Similarly, method calls can be handled either by inlining the method body or by abstracting with a user-provided method specification. All user-provided abstractions are machine-checked for soundness.

Going beyond functional properties, KeY supports a language for specifying information flow in programs as part of its JML* extension of JML. The language was originally published in [16] though we refer the interested reader to the more up-to-date information source [15] for details.

Specifying information flow with JML*. The main instrument for specifying information flow in JML* is an *information flow contract.* The contract can be attached—among other things—to method declarations, and its grammar is shown in Figure 2.

Definition 1 (Semantics of information flow contracts). *Let m be a terminating sequential method with an attached information flow contract. Let (ds_i) and (dm_j) be the expression sequences of the determinans and the determinandum of the contract respectively. Let (s^a_{pre}, s^a_{post}) and (s^b_{pre}, s^b_{post}) be a pair of runs of the method m, where s^a_{pre} and s^b_{pre} are the initial (or pre) states and s^a_{post} and s^b_{post} are the final (or post) states respectively. The method m satisfies* *the attached information flow contract, iff for each such pair of runs, the coinciding evaluation of the determinans in both runs implies the coinciding evaluation of the determinandum:*

$$\bigwedge\nolimits_i \big((ds_i \ in \ s^a_x) = (ds_i \ in \ s^b_x)\big) \rightarrow \bigwedge\nolimits_j \big((dm_j \ in \ s^a_y) = (dm_j \ in \ s^b_y)\big) \ ,$$

where $x, y \in \{pre, post\}$ according to the state designators wrapping the determinans and determinandum respectively. In absence of explicit state designators, the defaults $x = pre$ and $y = post$ are used.

For example, the specification

```
//@ determines \result \by l1, l2;
int f(int h, int l1, int l2) { ... }
```

says that the return value of the method `f` is completely determined by the method parameters `l1` and `l2`. This means that no information flows from the method parameter `h` (or other data on the heap) to the return value of `f`. Note that since it is not stated otherwise, the determinans $l1, l2$ is evaluated in the initial state, while the determinandum is evaluated in the final state. This convention follows the original design goal of JML* in specifying *absence* of undesired information flow in programs.

More interesting for our purposes is the specification

```
//@ determines \pre(h) \by \post(\result);                (∗)
int f(int h) { ... }
```

describing, in a sense, the opposite situation. It is fulfilled when knowing the result of `f` is sufficient to reconstruct the (initial) value of the parameter `h`. Mathematically, this case amounts to injectivity of `f` and means intuitively that the complete information contained in `h` flows to the return value. Contrary to the JML* defaults, the explicit state designators `\pre()` and `\post()` force the determinans to be evaluated in the final state and the determinandum in the initial state. We have extended JML* with these designators specifically on occasion of this case study.

In case one needs to speak about array content in contracts, the finite sequence comprehensions of JML* allow this easily. For example, the JML* comprehension expression `(\seq_def int i; 0; a.length; a[i])` is essentially a shorthand for the expression sequence `a[0]`,...,`a[a.length-1]` (for presentation in this paper, we also use the notation `a[*]` for this particular sequence).

Proof obligations for information flow. To prove information flow contracts, KeY formalizes the condition of Definition 1 in Dynamic Logic. The formalization follows self-composition style and is straight-forward. The (schematic) proof obligation for a contract like $(*)$ is

$$\forall h^a, h^b. \, f(h^a) = f(h^b) \rightarrow h^a = h^b \ .$$

We refer the interested reader to [15, 17] for details of the formalization in Dynamic Logic. The important fact is that information flow contracts of the callee method can be used—just like functional contracts—when verifying the caller method.

Listing 1.1: The main PRNG method (schematic)

```
1  void
2  engineNextBytes(byte[] bytes) {
3    ...
4    if (state == UNDEFINED) {
5      // entropy source
6      updateSeed(
7        RandomBitsSupplier
8          .getRandomBits(20));
9      ...
10   } else { ... }
11
12   ...
13
14   for (;;) {
15     ...
16     // entropy target
17     SHA1Impl.computeHash(seed);
18     ...
19   }
20 }
```

Listing 1.2: Modified source with top-level requirement specification (excerpt)

```
/*@
  requires counter == 0;
  requires state == UNDEFINED;

  requires bytes.length == 20;
  requires extSource.length == 20;

  determines \pre (extSource[*])
         \by \post(bytes[*]);
 */
void
engineNextBytes(byte[] bytes,
                byte[] extSource) {
  ...
  if (state == UNDEFINED) {
    updateSeed(extSource);
    ...
  } else ...
  ...
}
```

Listing 1.3: Specification of the pseudo-SHA1 method

```
/*@ public normal_behavior
    requires arrW.length==87;
    assignable arrW[16..79],arrW[82..86];
    determines \pre ((\seq_def int i;  0;  5; arrW[i]))
           \by \post((\seq_def int i; 82; 87; arrW[i]));
*/
static void computeHash(int[] arrW) {...}
```

4 PRNG Specification and Correctness Proof

4.1 The Specification and Problems Attaching It

To show full flow of entropy (i.e., absence of squandering), we are instantiating the specification pattern ($*$) for the main PRNG method shown in Listing 1.1.

Our original intent was to show that the entropy returned by the call to the `RandomBitsSupplier.getRandomBits()` method in line 7 of Listing 1.1 (the source) is preserved at least until the call to the `SHA1Impl.computeHash()` method in line 17 (the target). The problem is that the source is nested within another method call expression that is itself nested within an if-statement, while the target occurs in the middle of a loop body. Specification languages like JML are, in contrast, designed to specify programs in a mostly block-structured way, i.e., pre- and postconditions can only be attached to complete blocks, loops, method declarations, etc. Facilities for point-to-point specification are less developed. To overcome this obstacle, we resorted to a minor source code modification as well as to extending the verified property as outlined in the following.

The source. We removed the call to `RandomBitsSupplier.getRandomBits()` in line 7 and replaced it by an extra parameter `extSource`, which allows us to speak about the inflowing entropy in the method specification. The modified source is shown in Listing 1.2. The precondition `state == UNDEFINED` states that the PRNG is indeed in self-seeding mode. For the sake of clarity, we are not showing a few more trivial preconditions stating that the PRNG object is initially in a consistent state (fields are initialized with default values, etc.). These preconditions stem from (separate) symbolic execution of the object constructor.

The target. We solve the problem with the inaccessible entropy target by stating a postcondition on the *whole* method. In other words, we are specifying not only that the 20 byte of entropy in `extSource` are safely transferred into the internal state of the PRNG but that they are contained in the 20 byte of output returned to the caller, which is a stronger property.

The hash. The above strengthening also causes a complication: the call to `SHA1Impl.computeHash()` is now in the code path. Due to the (intended) computational complexity of SHA-1, it is not practicable to reason about this method either by inlining its code or stating a faithful functional specification. In contrast, it is possible to give an information flow specification, which can be used for the proof of `engineNextBytes()`.

We assume (but do not prove) the specification of `SHA1Impl.computeHash()` shown in Listing 1.3, stating that the method transfers all information (i.e., is injective) from the first five words of the main array into the last five words. While we do not know if this assumption is true (as disproving it would amount to finding a collision in SHA-1), it constitutes a fundamental proviso for the security of the PRNG. Unsurprisingly, proof inspection showed that it was indeed not disproved. A similar, if more obviously justifiable, contract was used for the sole standard library method used by the PRNG, `System.arraycopy()`.

4.2 The Proof

The vulnerability is unmissable when attempting the proof, so the following remarks apply to the fixed implementation incorporating the official patch.

The main proof consists of 21 882 proof steps, of which 95 were interactive. The majority of the latter are carrying out case distinctions, splitting the equality of sequences into five equalities over words and 20 over bytes. The rest are for weakening the proof goal to eliminate irrelevant information and reduce the search space, as well as applications of rules for byte packing and unpacking (see below). The automated proof search took altogether 45 minutes to complete the proof. All loops in the main code were unrolled (thus also establishing termination), no invariants or auxiliary annotations were necessary. Trivial invariants were used to prove termination and assignable clause of `SHA1Impl.computeHash()`.

A significant portion of proof complexity stems from the code packing bytes into words and a later converse unpacking. Figure 3 shows the code factored for exposition purposes as synthetic methods. For the proof, we have defined two custom rules that express the injectivity of these code fragments. The soundness

```
int pack(byte[] b) { return
  ((b[0]&0xFF)<<24) | ((b[1]&0xFF)<<16) | ((b[2]&0xFF)<<8) | (b[3]&0xFF);}

byte[] unpack(int i) { return new byte[] {
  (byte)(i>>>24), (byte)(i>>>16), (byte)(i>>>8), (byte)i };}
```

Fig. 3: Packing and unpacking code (illustration)

of the rules has been proven using KeY's rule justification mechanism and the KeY's SMT bridge to Z3/CVC4 (the only place where an external SMT solver was used). Each rule was applied five times, once for each word of the seed.

The KeY logic is based on the theory of integers and not bitvectors. To achieve soundness, proof rules either generate proof obligations showing absence of overflow, or perform operations modulo machine integer range. The former option was used for the majority of the code, while the latter option was necessary to handle the packing and unpacking code.

5 Alternatives and Related Work

Functional verification and testing. Of course, it is possible to state and verify a functional specification of the methods involved without resorting to the concept of information flow. However, such a specification would have to closely mimic the implementation and thus be complex and tedious to write (the same reasoning also applies to functional testing). It would be difficult to understand it and ascertain its adequacy; neither would it be possible to reuse it for another PRNG. It would also be challenging to write down such a specification in existing languages due to the structure of the code (see Section 4.1). The information flow specification, on the other hand, directly expresses the desired property, is compact and easy to understand, and is nearly independent of the PRNG implementation in question.

Statistical testing. Several statistical test suites exist for assessing the quality of random numbers. Among the most popular are DIEHARD with its open source counterpart DIEHARDER and the NIST test suite. The suites scan a stream of pseudo-random numbers for certain predefined distribution anomalies. At the same time, we are not aware of recommendations on how the stream is to be produced. In practice, it appears customary to derive the stream from a single seed. The tests are repeated multiple times (with different seeds) to increase the degree of confidence but the results between individual runs are not cross-correlated. In any case, distinguishing a PRNG seeded with 8 byte of entropy from a PRNG seeded with 20 byte of entropy would likely require a prohibitively high number of tests.

Quantitative Information Flow analysis (QIF). Detecting entropy squandering can be seen as an instance of the Quantitative Information Flow problem

(QIF) concerned with measuring leakage of secret information to an observer of the program output. Several methods and tools for QIF exist, including our own work [10, 11]. Yet, the landscape of available QIF analyses is not well-suited for the specifics of the problem we face. Some techniques are only practicable for small leakage, or small/simple programs. Some are not implemented or do not support real-world programming languages. Some only establish upper bounds on the leakage, while we need lower bounds, as our observer is not an adversary. Given these limitations, the prospects of using current QIF techniques for practical PRNG verification remain unclear at best.

High-level PRNG analysis. Apart from the above-mentioned [13], "modern" PRNGs have been studied in, e.g., [2, 4, 6]. The perspective taken in the latter works is based on elaborate attack models, where the attacker, for instance, can control the distribution of the inputs used to seed the PRNG, view or even corrupt the internal PRNG state. The analysis focuses primarily on design and high-level implementation aspects w.r.t. these models and is not mechanized. In contrast, we do not consider attackers with advanced capabilities, but our work closes the gap concerning low-level implementation aspects with mechanized reasoning.

6 Conclusions

A good design document and a high-level analysis are indispensable for a correct PRNG, but so is low-level verification. The problem of squandered entropy due to subtle code bugs is real and relevant, yet very difficult to detect by conventional means. At the same time, a concise and uniform specification of correctness can be given in terms of information flow. The JML* specification language proved its convenience in this regard.

Logic-based information flow reasoning is the tool of choice for PRNG verification, as other techniques (e.g., type systems, PDGs, etc.) inherently incorporate overapproximations that make them unsuitable. The correctness proofs are conceptually quite simple, and do not require ingenuity, but the complexity and monolithic nature of the code tax the verification system to a significant degree.

A large part of our effort went to understanding the details of the implementation. Besides referring to higher-level descriptions such as [4, 13], we found verification technology in general (for establishing data footprints of code segments) and symbolic execution in particular (for identifying dead code on a given path) very helpful in this regard. While it is hard to quantify the total effort spent on the case study due to a learning process that occurred over a longer period of time, we conjecture that we could now verify a comparable PRNG within one or a few days.[5]

[5] This work was in part supported by the German National Science Foundation (DFG) under the priority programme 1496 "Reliably Secure Software Systems – RS3." The authors would like to thank Christoph Scheben for help with the proof system, and Bernhard Beckert, Mattias Ulbrich, and Sylvain Ruhault for comments on the topic.

References

1. W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich. The KeY platform for verification and analysis of Java programs. In *Proceedings, 6th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, LNCS. Springer, 2014.
2. B. Barak and S. Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 203–212. ACM, 2005.
3. Bitcoin.org. Android security vulnerability. `https://bitcoin.org/en/alert/2013-08-11-android`, 2013.
4. M. Cornejo and S. Ruhault. Characterization of real-life PRNGs under partial state corruption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1004–1015. ACM, 2014.
5. Debian weak key vulnerability. CVE-2008-0166. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166`, 2008.
6. Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 647–658. ACM, 2013.
7. J.-M. Gurney. URGENT: RNG broken for last 4 months. `https://lists.freebsd.org/pipermail/freebsd-current/2015-February/054580.html`, 2015.
8. D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.
9. The KeY tool. Website at `www.key-project.org`.
10. V. Klebanov. Precise quantitative information flow analysis – a symbolic approach. *Theoretical Computer Science*, 538(0):124–139, 2014.
11. V. Klebanov, N. Manthey, and C. Muise. SAT-based analysis and quantification of information flow in programs. In *Proceedings, International Conference on Quantitative Evaluation of Systems*, pages 156–171. Springer, 2013.
12. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
13. K. Michaelis, C. Meyer, and J. Schwenk. Randomly failed! the state of randomness in current Java implementations. In *Proceedings, 13th International Conference on Topics in Cryptology*, CT-RSA'13, pages 129–144. Springer-Verlag, 2013.
14. P. Q. Nguyen and I. E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Des. Codes Cryptography*, 30(2):201–217, Sept. 2003.
15. C. Scheben. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014.
16. C. Scheben and P. H. Schmitt. Verification of information flow properties of Java programs without approximations. In *Proceedings, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS)*, pages 232–249. Springer, 2011.
17. C. Scheben and P. H. Schmitt. Efficient self-composition for weakest precondition calculi. In *Proceedings, Formal Methods (FM), 19th International Symposium*, pages 579–594. Springer, 2014.

A Source Code of the Android PRNG (Excerpt)

Source code below has been slightly edited for presentation purposes. Comments are removed. Constant declarations are elided or inlined. Code unreachable in the verification scenario presented in the paper is elided.

```
public class SHA1PRNG_SecureRandomImpl implements SHA1_Data {

    private transient int[] seed;
    private transient byte[] nextBytes;
    private transient int nextBIndex;
    private transient long counter;
    private transient int state;

    public SHA1PRNG_SecureRandomImpl() { ... }

    protected synchronized void engineNextBytes(byte[] bytes) {

        int i, n;
        long bits;
        int nextByteToReturn;
        int lastWord;
        final int extrabytes = 7;

        if (bytes == null) throw new NullPointerException("bytes␣==␣null");

        lastWord = seed[81] == 0 ? 0 : (seed[81] + extrabytes) >> 3 - 1;

        if (state == UNDEFINED) {

            updateSeed(RandomBitsSupplier.getRandomBits(20));
            nextBIndex = 20;

            // official patch for the vulnerability
            lastWord = seed[81] == 0 ? 0 : (seed[81] + extrabytes) » 3 - 1;

        } else if (state == SET_SEED) { ... }
        state = NEXT_BYTES;

        if (bytes.length == 0) return;

        nextByteToReturn = 0;

        n = (20 - nextBIndex) < (bytes.length - nextByteToReturn) ?
                20 - nextBIndex :
                bytes.length - nextByteToReturn;
        if (n > 0) { ... }

        if (nextByteToReturn >= bytes.length) return;

        n = seed[81] & 0x03;
        for (;;) {
            if (n == 0) {

                // the problem occurs here
                seed[lastWord    ] = (int) (counter >>> 32);
                seed[lastWord + 1] = (int) (counter & 0xFFFFFFFF);
                seed[lastWord + 2] = END_FLAGS[0];

            } else { ... }
            if (seed[81] > 48) { ... }

            SHA1Impl.computeHash(seed);

            if (seed[81] > 48) { ... }
            counter++;

            int j = 0;
            for (i = 0; i < 5; i++) {
                int k = seed[82 + i];
                nextBytes[j    ] = (byte) (k >>> 24);
                nextBytes[j + 1] = (byte) (k >>> 16);
                nextBytes[j + 2] = (byte) (k >>> 8);
                nextBytes[j + 3] = (byte) (k);
                j += 4;
            }

            nextBIndex = 0;
            j = 20 < (bytes.length - nextByteToReturn) ?
                20 : bytes.length - nextByteToReturn;

            if (j > 0) {
                System.arraycopy(nextBytes, 0, bytes, nextByteToReturn, j);
                nextByteToReturn += j;
                nextBIndex += j;
            }

            if (nextByteToReturn >= bytes.length) break;
        }
    }

    private void updateSeed(byte[] bytes) {
        SHA1Impl.updateHash(seed, bytes, 0, bytes.length - 1);
        seedLength += bytes.length;
    }
}

public class SHA1Impl implements SHA1_Data {

    static void computeHash(int[] arrW) { /* elided for brevity */ }

    static void updateHash(int[] intArray, byte[] byteInput, int fromByte, int toByte) {

        int index = intArray[81];
        int i = fromByte;
        int maxWord;
        int nBytes;

        int wordIndex = index >>2;
        int byteIndex = index & 0x03;

        intArray[81] = ( index + toByte - fromByte + 1 ) & 077 ;

        if ( byteIndex != 0 ) { ... }

        maxWord = (toByte - i + 1) >> 2;

        for ( int k = 0; k < maxWord ; k++ ) {

            intArray[wordIndex] = (((int) byteInput[i    ] & 0xFF) <<24 ) |
                                  (((int) byteInput[i + 1] & 0xFF) <<16 ) |
                                  (((int) byteInput[i + 2] & 0xFF) <<8  ) |
                                  (((int) byteInput[i + 3] & 0xFF)      ) ;
            i += 4;
            wordIndex++;

            if ( wordIndex >= 16 ) { ... }
        }

        nBytes = toByte - i +1;
        if ( nBytes != 0 ) { ... }
    }
}
```