Verification of Random Number Generators

Bachelor Thesis of

Felix Dörre

30. September 2015

Department of Informatics
Institute for Theoretical Informatics (ITI)
Application-oriented Formal Verification

Responsible Advisor: Prof. Dr. Bernhard Beckert
Advisor: Dr. Vladimir Klebanov

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 30.09.2015

.....

(Felix Dörre)

Abstract

The goal of this bachelor thesis is to analyze the information flow in the seeding code of pseudo-random number generators (PRNGs). PRNGs are used for cryptographic applications and are often crucial for their correct function. But PRNGs are often complicated and therefore error prone. There is a long history of defects that were caused by failed seeding and that were fatal for cryptographic applications. For example a bug in the "SHA1PRNG" of Apache Harmony (i.e. the PRNG used in earlier Android versions) reduced the entropy that is used for seeding from 20 bytes to 8 bytes. Such bugs are hard to detect, because they cannot be easily tested for and might stay unnoticed for a longer time. They are mainly found in code reviews that are laborious due to the complexity of cryptographic code.

We describe this class of defects in term of reduced information flow. We specify this property formally and develop techniques and tools to verify it. We show that several PRNGs do not have reduced information flow from the seed into the PRNGs internal state. For Java-based PRNGs we specify our information flow goal using KeY's extension to the Java Modeling Language (JML) and use KeY to prove those contracts and additional helper contracts for sub functions. For C-based PRNGs we propose a solution based on the bounded model checker CBMC.

Contents

1 Introduction

1.1 Motivation

Debian-OpenSSL vulnerability, what it is and how bad it is. In 2008 a major flaw in the pseudo random number generator (PRNG) of the Debian package of OpenSSL was discovered [7]. The bug was introduced two years before by the package maintainer. The maintainer wanted to improve the software quality of the package by removing all accesses to uninitialized memory. The maintainer found some accesses and asked on the mailing list if removing those function calls would significantly harm the PRNG. Fatally the question didn't get answered correctly and he accidentally severely harmed the PRNG: the OpenSSL library on Debian was only able to generate at most $2^{16} = 65536$ different keys per given key size. This problem made all generated keys easily breakable and it remained undetected for two years.

Apache Harmony `SHA1PRNG`, problem with Bitcoin applications. In 2013 another PRNG flaw gained public attention. Several Bitcoin wallets on Android were insecure [4]. Bitcoin requires random data for two different actions. Firstly for generating the keys representing the wallet, and secondly for a nonce (number used once) for the ECDSA [12] (elliptic curve digital signature algorithm) signature on every transaction. When the nonce is partly known attacks like the ones shown in [19] are possible, but when a nonce is used multiple times with the same key pair, the private key can be easily reconstructed from those multiple signatures. Because all signatures are publicly visible, an attacker can search for such signatures, reconstruct the private key for the wallet and then transfer all Bitcoins away. A problem in the PRNG severely reduced the quality of its output and made such an attack possible. All wallets that were used on an Android device while the bug still existed (up to and including Android 4.1) should be considered compromised.

The need for a better approach to detect such problems. Such problems, when the quality of a PRNG drops dramatically due to a programming error, reoccur from time to time. Just recently (until 17.02.2015) the system PRNG in FreeBSD was broken for four months [10]. Keys generated with that broken PRNG might be predictable. All these problems didn't make the PRNGs output completely deterministic but only reduced its randomness to a value that is high enough to not be noticed when using the PRNG but low enough to allow serious attacks. Hence these problems are normally only detected in manual code reviews. But code reviews are time consuming and therefore not regularly done, especially in free and open source projects. Consequently such problems

can remain undetected for longer periods. In order to prevent such incidents from reoccurring or at least reduce them, a better means than manual review is needed. In this bachelor thesis we present such a means: we show how to specify the necessary correctness property in terms of information flow and verify that specification for three real world PRNGs.

1.2 PRNG Problems: Background and Commonalities

In order to understand what went wrong in the incidents presented above we need to understand what PRNGs do in principle and then look at the incidents again.

What is a PRNG in principle? PRNGs take a small amount of input entropy (i.e. randomness) and construct a stream of data that is indistinguishable from a stream of random data for an attacker with limited computational resources. The process of adding initial entropy is information theoretically a bottleneck for the total amount of entropy available to the PRNG and the total amount of entropy in its output stream. Therefore getting enough input entropy is crucial for the PRNG to be able to operate correctly.

Debian-OpenSSL. The OpenSSL PRNG acquires randomness from many different sources in order to not depend on a single source. In normal operation these sources are:

- current process ID
- current user ID
- current group ID
- uninitialized memory buffers
- an OS-specific way to collect random data.

The only source that contains significant amounts of entropy is the OS-specific source. The maintainer's patch accidentally removed not only the uninitialized memory buffers but also all other sources except for the current process ID. Yet such an ID is normally not assigned randomly and is only a two byte integer that has at most $2^{16} = 65536$ different values. This incident was caused by reduced input entropy.

Apache Harmony `SHA1PRNG`. The input entropy of `SHA1PRNG` are 20 bytes from an OS-specific source. These bytes are copied to the PRNGs internal state. Accidentally 12 of these 20 bytes got overwritten with deterministic data (i.e. 0-bytes or a counter), because the pointer that indicated where to put more data was not updated. Consequently the PRNG had only $256^8 = 2^{64}$ remaining possible output streams. This number of possible output streams is higher than for the previous incident. Therefore breaking keys generated with this PRNG might be a lot harder, but it was enough to get some

applications use the same nonces more than once. This security failure was also caused by reduced input entropy.

The next step is to formally describe this problem and on the other hand the correctness property for the PRNG that is violated. In order to do so, we need to focus on entropy/information flow and understand how random data moves through the PRNG.

1.3 PRNG Principles and Terminology

1.3.1 Entropy, Injectivity and Information Flow

What is entropy? *Entropy* is a measure for uncertainty. It can be used to describe the degree of randomness of data and thereby the quality of random data. There are different ways to measure entropy. The *min entropy* measures the expected probability of an attacker to guess the random data in one guess, whereas the *Shannon entropy* measures how long a message has to be on average to be transmitted reliably. In this bachelor thesis we focus on evenly distributed random variables and transformations that keep the distribution even. Also we do not need to quantify reduced entropy. Our focus lies on weather *all* entropy is preserved or entropy is reduced. For such variables and in such cases the above mentioned entropy measures are equal so the selection of the entropy measure does not matter. For simplicity we also use the term "entropy" below when we mean random data that contains this specific amount of entropy.

What is information/entropy flow? When we say that information (i.e. entropy) flows from a source variable (typically before execution of a program) to a destination variable, this means an attacker with knowledge of the destination variable can learn something about the source variable. For a formal definition, let $f : ins \to outs$ be the function that calculates the output of a deterministic and terminating program from its input. This function induces an equivalence relation on ins with $in_1 \sim in_2 \Leftrightarrow f(in_1) = f(in_2)$. $\sim$ is the information flow through f. The amount of information flowing corresponds to the degree of injectivity of f (i.e. the number of equivalence classes of $\sim$). Maximal information flow means a maximal number of equivalence classes, that is $|ins|$ assuming $|ins| \leq |outs|$.

As some examples, consider the 3 Java functions in Figure 1.1with their information flow from `source` to their return value.

`f1` and `f2` have information flow, but in `f3` no information flows. For PRNGs we need to quantify the information flow, especially we need to identify, if *all* information from the source variables flows to the target variables. This is the case in `f1` due to overflow and the Java-semantics of additions. The property of full information flow through a function f is equivalent to its injectivity. So while `f1` is injective, neither `f2` nor `f3` is.

Equipped with that understanding of information flow, let's look at how information flows through a PRNG normally.

```
int f1(int source){
        return source + 10;
}
int f2(int source){
        if(source > 0){
                return 1;
        } else {
                return 0;
        }
}
int f3(int source){
        return 10 + source - source;
}
```

Figure 1.1: 3 Java functions as an example for information flow

1.3.2 Basic PRNG Structure

PNRGs typically store the entropy that is initially provided to them in an internal state. For most PRNGs this state initially has a fixed value, so there is no entropy in it. Some PRNGs however may start with their internal state with e.g. uninitialized memory. We want to look at such PRNGs like they would be starting with a fixed internal state and then add the entropy contained in the uninitialized memory to this internal state afterwards.

Abstract description of PRNG seeding. In order to acquire a random internal state, we need entropy to be added to the internal state. This is done by a function we call *addEntropy*. When $\mathbb{B}$ is the set of all byte values, the signature of the *addEntropy* function would look like this:

$$addEntropy : \mathbb{B}^n \times \bigcup_{m=0}^{\infty} \mathbb{B}^m \rightarrow \mathbb{B}^n$$

Using this function, one can add an arbitrary amount of m seeding bytes to the internal state of size n. The process of using *addEntropy* initially to make a known internal state random, i.e. inserting the initial amount of entropy into the PRNG, is called *seeding*. Some PRNGs don't require the user[1] to seed them manually, probably in order to prevent them from using an unseeded PRNG. They seed themselves by requesting random data from an OS-specific source, when they are used the first time. This process is called *self-seeding*. PRNGs designers have made different decisions about how a PRNG can be operated and thereby how *addEntropy* may be used:

SEEDING.1 The PRNG can only be self-seeded

[1]The user of a PRNG is a programmer using its API.

SEEDING.2 The PRNG has to be seeded manually in the beginning otherwise it outputs an error

SEEDING.3 The PRNG has to be seeded manually in the beginning otherwise it produces bad random data.

SEEDING.4 The PRNG may be seeded manually and the PRNG seeds itself if it is not seeded manually before the first use.

SEEDING.5 The PRNG may be seeded manually but the PRNG (additionally) seeds itself regardless of being seeded manually (either on the first use or on initialization)

SEEDING.6 The PRNG may be seeded manually and the user may initiate (or enable/disable) self-seeding.

Regardless of what is chosen here, there is another decision that is free in most of the scenarios above

ADD_ENTROPY.1 The user can call *addEntropy* at any point when the PRNG is operated.

ADD_ENTROPY.2 The user can call *addEntropy* only in the beginning (or never depending on choice above).

ADD_ENTROPY.3 The user can call *addEntropy* at any time, but this resets the PRNG before *addEntropy* is executed.

Abstract description of PRNG output generation. Independently from getting external entropy, PRNGs need to generate a longer stream of data from a short amount of data. In order to achieve that the PRNGs normally operate in cycles. Generating output in cycles can be described using the two functions *permute* and $output_k$:

$$permute : \mathbb{B}^n \to \mathbb{B}^n$$

$$output : \mathbb{B}^n \to \mathbb{B}^k$$

where k is the number of bytes emitted in one cycle. Typically such blocks are 20 bytes long. Let's have a look how these functions are used together to build a PRNG. We consider a PRNG that does seeding initially and not between the iterations. Let

- $s_i \in \mathbb{B}^n$ be the internal state at beginning of the i-th cycle
- $o_i \in \mathbb{B}^k$ be the i-th output block
- $input \in \mathbb{B}^m$ be the seed.
- $s_\perp \in \mathbb{B}^n$ be an initial state

then the PRNG can be described with the following equations.

$$s_0 = addEntropy(s_\perp, input)$$

$$s_i = permute(s_{i-1}) \quad \forall i = 1, \ldots$$

$$o_i = output(s_i) \quad \forall i = 1, \ldots$$

A typical instantiation of such a PRNG would have $n \geq 20$, $k = 20$ and $m = 20$. Often *permute* and *output* are implemented together in one function.

Our goal is to prove that a given PRNG uses the full information supplied. To show that, we prove that all entropy contained in a certain amount n of random seeding bytes is fully used in (the first) n bytes of output. This is equivalent to the function f

$$f(data) = output(permute(addEntropy(c, data)))$$

being injective (assuming $m = k = n = 20$).

1.4 Information Flow through PRNGs

In most PRNGs all 3 functions (*addEntropy*, *permute*, *output*) are designed using cryptographic hash functions. When entropy flows through such PRNGs it needs to pass the hash function. To understand what needs to be proven and specified here, we take a look on how hash functions are used in PRNGs.

1.4.1 Use of Hash Functions in PRNGs

Cryptographic hash functions are used in PRNGs to lower the possibility of using output from different cycles to calculate (parts of) the internal state and by that predict further output of the PRNG. Mathematically cryptographic hash functions h have the signature

$$h : \bigcup_{m=0}^{\infty} \mathbb{B}^m \to \mathbb{B}^n$$

with a fixed value of n. n is called the *hash size* (i.e. digest size) and the result of the function is called a *hash*. Hash functions are typically assumed to have the following properties:

1. It is computationally hard to reverse h, i.e. calculating x from a given y with $h(x) = y$.

2. It is computationally hard to find collisions in h, i.e. calculating x' from a given x with $h(x) = h(x')$ where parts of x' may also be given. While most real-world hash functions used in PRNGs have this property, it's unclear whether it is required for secure operation of the PRNG.

3. The entropy in a hash is not significantly lower than the entropy of the input of h given the input is not larger than n bytes. When H is an entropy measure, then $y = h(x) \implies H(y) \not\ll \min(H(x), n)$.

4. h distributes the entropy "evenly" throughout the result, so e.g., the first half of a hash has half the entropy of the total hash.

Typically, because such functions are often used to calculate hashes of large amounts of data, hash functions operate in cycles, like PRNGs do. They split this input data in blocks, which they incorporate into their internal state individually. Finally the function does some padding and calculates a hash from the internal state. This principle can also be seen in most interfaces for cryptographic hash functions. In the example interface shown here `context` denotes the internal state of the hash function.

1. `void init(context)`: initialize the context of the hash function object.

2. `void update(context, data)`: add the given data to the context and begin with the calculation of the hash function.

3. `hash doFinal(context)`: do final actions like padding, and calculate the result of the hash function (i.e. the *hash*).

Calling conventions for `doFinal`. Calling `doFinal` often leaves the context in an undefined state. Calling `doFinal` or `update` directly after this call would be not allowed. A call of `init` is required to use the context again. Sometimes however this function already resets the context (i.e. calls `init`) so that the hash function context can be instantly used again. Care has do be taken of this two conventions when using or specifying the interface as this distinction cannot be seen in the API.

Distinction of calculations against calls. Regardless of this different calling conventions we still invoke multiple function in order to calculate on hash. To be able to talk about the hash function represented by such interfaces as one, we refer to one calculation of a hash as *calculation* of the hash function that consists of multiple *calls* to this interface.

1.4.2 Formulating Assumptions about Information Flow

In order to prove full information flow through a PRNG, we need to specify that the hash function preserves all input entropy. This can only be true if not more entropy than the hash size is in the hash function's input. When we seed the PRNG with exactly this much entropy, we can be sure that this is always the case (as there is no additional entropy anywhere in the PRNG [2]). But when we conduct a proof over an unmodified

[2] We need this to be true, because for our proof we need to specify injectivity from all entropy. Real-world PRNGs might not have this property. They might use uninitialized memory, the current time, PID, scheduling decisions in multi-threaded programs and other source of non-determinism. These

PRNG, we need to prevent the proof system from analyzing the hash function, because such functions are purposely hard to analyze.

For proving injectivity of the PRNG, we consider 2 possibilities:

1. We assume injectivity of the hash function in a logical specification. When doing so, we need to make sure, that our proof system of choice does not see a contradiction in the hash function assumed injective. Or:

2. We replace the hash function with our own function, which we called *dummy hash function*, that is injective from the seed-bytes of its input to its output.

Below we describe the 2 options in more detail:

1.4.3 Alternative 1: Assuming Injectivity as a Logical Specification

This alternative means specifying the injectivity of the hash function so that the proof system (e.g. KeY [13, 1]) does not have to look at the implementation and can just use the specification. We might be able specify that if more data than the hash size is supplied, then only the entropy is returned. But that is a very complex task, because the hash function might be used in different locations when the entropy that needs to be returned is at different positions in the input data. So a specification would need to reflect the hash function's usage and therefore be very large and complex.

We specify the injectivity of the hash function and ensure that our proof system doesn't see a contradiction by manual proof inspection. This way this alternative is easier to carry out. On the other hand, this alternative requires a proof system that allows the usage of method contracts and the specification of information flow in such contracts.

1.4.4 Alternative 2: Substituting the Hash Function Implementation

When using a proof system that does not allow using method specifications or specifying information flow in contracts as above, substituting the hash function might be the only possibility.

Soundness criteria for the hash function. Because computations can only reduce entropy one can do almost anything in the dummy function. In order to fit our proof, the dummy function must be injective from the seed-part of the input to its output. The following set of properties prevent the function from adding entropy from any other source to its output:

1. The function may not keep any entropy (that came from the seed) between different calculations.

additional entropy sources need to be removed. This is allowed because we do not need to consider these non-significant amounts of entropy. In OpenSSL we had to remove such sources. See Section 4.3 for more details.

2. The function may not access any global state (that contains seed entropy) from the PRNG.

3. However the function may keep counters to distinguish its different calculations and act different in each calculation as they typically do not contain seed entropy.

With that comes the limitation that we can normally only proof so much information flow through the PRNG as fits into the hash. Any more information is definitely lost and can not be preserved by any hash function.

Soundness criteria for the code calling the hash function. Another assumption usually done about hash functions is that they mix all incoming information and distribute it uniformly across the hash. Because our dummy hash function cannot fulfill this property we need to be careful about what the surrounding code does. Consider the following code example with `seed` being 20 bytes of entropy, and `h()` being a hash function:

```
state = h(seed);
state2 = h(state);

output = state[0..9] | state2[0..9];
```

The operation that calculates output takes 10 bytes from each `state` and `state2` and concatenates them to a 20 byte output (`state[0..9]` is denoting the first half of `state`). For injective functions `h()` the entropy of both `state` and `state2` is both 20 bytes and the shared entropy between them is also 20 bytes. So knowing `state` leads to knowing `state2` (and the other way round). In order to understand what the entropy of `output` is, we look at different possibilities for `h()`:

When we assume `h()` to be the identity function, `state[0..9]` and `state2[0..9]` are equal and the resulting entropy of `output` is only 10 bytes. But when we assume `h()` to be a function that reverses its input, `state[0..9]` and `state2[0..9]` are independent from each other, so there is no shared entropy between them and therefore `output` has 20 bytes of entropy.

But when we assume `h()` to be a cryptographic hash function, inverting becomes computationally hard (i.e. getting to know `state` from knowing `state2`). When we take the first 10 bytes of each `state` and `state2`, we would say that they now contain 10 bytes of entropy each and that on average 5 bytes of entropy are shared between those two parts (because we assume that the entropy is evenly distributed). So the entropy of `output` would be estimated with only 15 bytes. Therefore one could enumerate all expectedly 256^{15} values for `output`, but doing so might (because of the hash function's complexity) involve calculating the output for all possible 256^{20} `seeds` and so this limitation might not be that helpful for an attacker.

The example shows that we need to take extra care when the surrounding code treats a hash non-uniformly. If e.g. it takes different code paths based on what a specific byte looks like or a loop count depends on the value of the hash. Also splitting and re-uniting different hashes needs to be watched with care as seen in the example above.

1.5 Basics of the Used Formal Verification Systems

1.5.1 KeY: a Deductive Verification System for Java

This section is based on [1].

The core of the KeY system consists of a theorem prover for a program logic that combines a variety of automated reasoning techniques. The KeY prover differs from many other deductive verification systems in that symbolic execution of programs, first order reasoning, arithmetic simplification, external decision procedures, and symbolic state simplification are interleaved.[3] For loop and recursion free programs, symbolic execution is performed in a fully automated manner.

The program logic supported by KeY is *Dynamic Logic* (DL) [11], a first order multi-modal logic. DL extends first order logic (FOL) with two families of modal operators: $\langle p \rangle$ ('diamond') and $[p]$ ('box') where p is a program fragment. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds, while $[p]\phi$ does not demand termination and expresses that *if* p terminates, then ϕ holds in the final state.[4]

To enable formal arguments about soundness and completeness, the KeY prover employs a *sequent calculus* for reasoning about Java DL formulas. Each proof node is a sequent of the form $\Gamma \Rightarrow \Delta$, where Γ and Δ are sets of formulas, with the intuitive meaning that the conjunction of the assumptions Γ implies at least one of the formulas in Δ.

A proof in KeY consists of logical rule applications on DL sequents. To reason efficiently in a rich program logic for a target language like Java, a large number of sequent calculus rules are needed (over 1500 in the standard configuration). To implement these efficiently and to permit external validation of the rules, we use so-called *taclets*, described in [3, Chap. 4]. Method calls can be handled either by inlining the method body or by replacing a method invocation by the method's specification.

1.5.2 CBMC: a Bounded Model Checker for ANSI-C

This section is based on [16].

The C Bounded Model Checker (CBMC) [5] demonstrates the violation of assertions in C programs, or proves safety of the assertions under a given bound. CBMC implements a bit-precise translation of an input C program, annotated with assertions and with loops unrolled to a given depth, into a formula. If the formula is satisfiable, then an execution path leading to a violated assertion exists.

CBMC performs symbolic execution by eagerly unwinding loops up to a fixed bound, which can be specified by the user on a per-loop basis or globally, for all loops. Constant propagation and expression simplification are key to efficiency, and prevent exploration of certain infeasible branches. At the end of this process the program is represented as a system of equations over renamed program variables in guarded statements. The guards

[3] The prover closest to KeY in this regard is KIV [23].

[4] This formulation assumes a deterministic programming language, like sequential Java in the context of KeY.

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

Figure 1.2: A sample information flow contract

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

Figure 1.3: schematic proof obligation for a information flow contract like Figure 1.2

determine whether an assignment is actually performed in a given concrete program execution.

This system of equations is solved with a SAT solver. A model computed by the SAT solver corresponds to a path violating at least one of the assertions in the program under scrutiny, and the model is translated back to a sequence of assignments to provide a human-readable counterexample. Conversely, if the formula is unsatisfiable, no assertion can be violated within the given unwinding bounds.

1.6 General Approach to Verifying new PRNGs

For a given PRNG we start by identifying how to operate the PRNG: How to seed it, how to obtain output, how to disable initial self-seeding with OS-specific inputs, how does the PRNG interact with the hash function (described in "Structure of the PRNG"). Then we specify the information flow through the hash function by replacing it, or by assuming its injectivity ("Information Flow Through the Hash Function"). Finally we specify our original proof goal and prove that with the prover of our choice.

1.6.1 Proving Injectivity using Information Flow Contracts in KeY

Example for a specification of maximal information flow. Let's have a look how assuming injectivity works in practice using the Java Modeling Language (JML) [17] with KeY-specific extensions. Consider the JML clause in Figure 1.2. This clause is an "information flow contract". It describes the claim that the result of `f` after execution of the function (indicated by `\post(\result)`) is enough to reconstruct the input parameter `h` before execution of the function (indicated by `\pre(h)`). Effectively this means that `f` is injective. We will use clauses of this structure to specify injectivity for functions later. More explanation on the syntax and semantics of these specific JML clauses can be found in [9].

Proof obligations for information flow contracts. This paragraph is based on [9]. To prove information flow contracts, KeY formalizes the contract in Dynamic Logic. The formalization follows self-composition style and is straight-forward. The (schematic) proof obligation for a contract like $(*)$ is shown in Figure 1.3. We refer the interested

reader to [22, 21] 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.

Using information flow contracts. Let's consider the following use of the function `f` with the information flow contract from Figure 1.2.

```
int x1 = ...;
int x2 = f(x1);
```

When evaluating such a function symbolically, KeY will now apply the information flow contract for the function `f` without looking at `f`'s implementation. This will be visible in the proof as a new predicate expressing the relation between `x1` and `x2`. In KeY this predicate's name starts with `RELATED_BY_f` and is dependent on `x1` and `x2`. For simplicity of this bachelor thesis we ignore dependence on e.g. the Heap and other technical things and just write the predicate as `RELATED_BY_f(x1, x2)`. As these predicates mark the functional dependence between variables, we call them markers.

Additionally KeY introduces new rules for treating those predicates. Those rules are generated when the information flow contract is used and the markers are constructed. These rules allow when there are two markers e.g.

$$\texttt{RELATED\_BY\_f(x1\_a, x2\_a)} \land \texttt{RELATED\_BY\_f(x1\_b, x2\_b)}$$

available as assumptions, to add

$$\texttt{x2\_a == x2\_b} \rightarrow \texttt{x1\_a == x1\_b}$$

as a new assumption. This represents the injectivity claimed in Figure 1.2.

Conducting a proof with information flow contracts. When information flow through a function is to be proven, KeY generates a formula with two invocations of the function. When we specify information flow as in Figure 1.2 we get the two invocations together with the obligation to show that if the outputs of the functions are equal, the inputs are equal as well. Such a proof is normally performed using these steps:

1. Simplify the formulas involving the two invocations. In order to not do the simplification work twice, use the macro "Auxilliary calculation for self-compositioning proofs". This macro application generates a new subproof in which we simplify one of the invocations by finishing the symbolic execution and transformation into a DL formula. The subproof is then used create a rule that contains all the simplification steps and that can be applied in the original proof.

Now we have two invocations with "RELATED_BY" markers in the main proof.

2. Simplify the current proof goal so that all "RELATED_BY" markers available as individual assumptions.

```
(\seq_def int i; 0; a.length; a[i])
```

Figure 1.4: Using JML* `seq_def` to transform an array `a` into a JML* sequence

3. Use the special rules on each pair of markers from the two invocations manually. Because the automatic matching of the "RELATED_BY" markers doesn't always get the correct pairs. For our proofs we disabled that part of KeY.
4. Give manual proof guidance specific to the PRNG to be verified.
5. Apply auto-mode to close all remaining goals

Reasoning about arrays, sequence comprehensions. In order to apply this method on the following PRNGs we will need to speak about array contents in the contracts. In order to do so, arrays need to be transformed into finite sequences. Those are usually defined with the `seq_def` syntax:

```
(\seq_def iterationVariableDefinition; from; to; value)
```

`from` denotes the index of the first element and `to` denotes the index of last (exclusive) element of the sequence (and not the length of the sequence). This syntax can be used to create a finite sequence from an array as shown in Figure 1.4. This construct defines the sequence of the whole array contents. Such sequences can be used in `determines` clauses, which is exactly what we will do.

1.6.2 Proving Injectivity using CBMC

To prove injectivity of a function in C, we used CBMC. The procedure of proving this property looks different from the one for proving injectivity of a Java function with KeY. CBMC is mainly used for proving assertions. As already outlined in Subsection 1.5.2 proving with CBMC works in two steps.

1. Use CBMC to transform a finite unwind of the code and assertions into a propositional formula in CNF that is satisfiable if and only if an assertion may fail. This transformation is only possible if all loops (and recursions) are finite with small and obvious bounds (so CBMC's constant propagation can resolve them).
2. Use a SAT solver to check the satisfiability of the CNF.

The found variable assignment will then represent the execution path that led to the failed assertion.

The verification driver. Let's consider the extract from our verification driver for the OpenSSL PRNG in Figure 1.5. When CBMC is used on this code, the resulting formula is satisfiable and each assignment representss a possible execution path that leads to the `assert(0)`. To verify injectivity, we perform the following steps with this code:

```
20 int main(){

             ...

29           for(i = 0; i < INPUT_AMOUNT; i++){
30             input = in[i];
31           }

             ...
             <CODE UNDER VERIFICATION TO BE INJECTIVE FROM in TO
                 out>
             ...

47           for(i = 0; i < OUTPUT_AMOUNT; i++){
48             output = out[i];
49             printf("%02x", out[i]);
50           }

             ...

60           assert(0);

             ...

64 }
```

Figure 1.5: The overall structure of the verification driver for C functions using CBMC.

1. **Sanity check for the other assertions** Remove the `assert(0)` in line 60 and verify that the CNF c_0 corresponding to this program is unsatisfiable. This proves that all other assertion throughout the code cannot fail.

2. **General sanity check for the tool chain** Add the `assert(0)` in line 60. CBMC will generate a CNF c of which the variable assignments correspond to execution paths leading to this assertion. Verify that this CNF c is satisfiable.

3. **Verification of injectivity** Now use this CNF c to construct another CNF c' to be shown as unsatisfiable as follows.

When transforming code to a CNF, CBMC creates new variables in the CNF for each program variable in each state in that it might change. During transformation CBMC generates a mapping from these program variables to the logical variables. This mapping is partly appended to the generated CNF in comments at the end of the output. The parameters generated CNF c can be split into three sets of variables.

1. in, all bits of the `in` array.

2. out, all bits of the `out` array.

3. x, all bits that correspond to intermediate values of other variables during program execution or to variables used for encoding other boolean expressions into CNF (e.g. generated by the Tseitin-transformation).

The program computes out from in, iff there is an x such that $c(in, out, x)$ is true.

Identifying input and output variables. CBMC does not output a mapping for array contents in the comments of the CNF. Therefore we cannot use the comments to identify the logical variables for each element of the array. To get those elements into the comments we use the two `for` loops in lines 29-31 and 47-50. Assigning each element of the array to the variable `input` makes CBMC create logical variables for each intermediate value of *input* and write their mapping down at the end of the CNF. Our tool uses these comments to find what variables of c are part of in and out.

Encoding injectivity as a SAT problem. Now injectivity of the C function is equivalent to the formula c' being unsatisfiable.

$$c'(in, in', out, out', x, x') = c(in, out, x) \wedge c(in', out', x') \wedge in \neq in' \wedge out = out'$$

We built a tool that transforms the original CNF c into a CNF c'' that is equisatisfiable to c'. The tool does:

1. outputs every clause of c twice, once with the original variables and once with variables from a fresh set of variables. This represents $c(in, out) \wedge c(in', out')$.

2. outputs clauses encoding $out = out'$. This can be expressed in CNF easily.

3. outputs clauses encoding $in \neq in'$. This is a more complex task because the resulting formula needs to be in CNF. Let $in_0, \ldots, in_n$ be the bits of in, $in'_0, \ldots, in'_n$ be the bits of in' and x_n fresh variables, then

$$\begin{aligned} in \neq in' \Leftrightarrow \bigvee_{i=0}^{n} in_i \not\leftrightarrow in'_i \Leftrightarrow \bigvee_{i=0}^{n} (in_i \wedge \overline{in'_i}) \vee (\overline{in_i} \wedge in'_i) \\ \Leftrightarrow^* \bigwedge_{i=0}^{n} \Big[(\overline{in_i} \vee in'_i \vee x_{2i}) \wedge (in_i \vee \overline{x_{2i}}) \wedge (\overline{in'_i} \vee \overline{x_{2i}}) \\ \wedge (\overline{in'_i} \vee in_i \vee x_{2i+1}) \wedge (in'_i \vee \overline{x_{2i+1}}) \wedge (\overline{in_i} \vee \overline{x_{2i+1}}) \Big] \\ \wedge (\bigvee_{i=0}^{n} x_i) \end{aligned}$$

   Where $\Leftrightarrow^*$ indicates equisatisfiability. This way we can express the inequality in a number of clauses linear in the length of in.

We can check satisfiability of this new CNF c'' with a SAT solver of our choice (e.g., minisat). This CNF must not be satisfiable. If a satisfying assignment for c'' is found, this constitutes a counter example containing the two inputs into the program that lead to the same output.

Running the tool and interpreting failed verification attempts. Our tool can be called in two modes. The first mode takes CBMC's output, the CNF c, in `example.cnf` and generates the combined CNF c'' in `example.cnf.composed`. To run our tool in this mode the following command needs to be executed:

```
cnfcomposer.sh example.cnf
```

This CNF can now be checked with minisat for satisfiability. If minisat finds an assignment it will write that to `example.cnf.composed.out`. The second mode of our tool interprets such an assignment of c'' and generates C-code that can be used to execute the counter example and debug it. The following call generates `ex_a.c` and `ex_b.c`. Both are the c files that contain code writing the counterexample into the `in` array from our verification driver:

```
cnfcomposer.sh example.cnf example.cnf.composed.out ex_a.c ex_b.c
```

1.7 Alternatives and Related Work

This section is based on [9].

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. 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 [15, 14].

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 [18], "modern" PRNGs have been studied in, e.g., [6, 8, 2]. 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.

2 Apache Harmony's `SHA1PRNG`

Apache Harmony is a reimplementation of the Java Standard Library that was created due to license issues. We started with the default secure PRNG from that library, `SHA1PRNG`, because it is the PRNG that was responsible for the Bitcoin incident mentioned above.

2.1 Structure of the PRNG

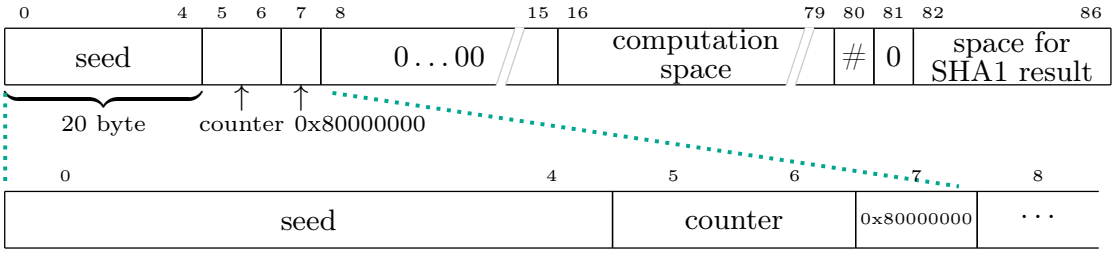
The PRNG consists of six methods in three classes, of which one class (`SHA1_Data`) only consists of constants. The other two classes are `SHA1PRNG_SecureRandomImpl` and `SHA1Impl`. `SHA1PRNG_SecureRandomImpl` is the main class, which contains methods to seed the PRNG (`void engineSetSeed(byte[] seed)`) and to generate new random bytes (`void engineNextBytes(byte[] bytes)`). Regarding our classification from Section 1.3.2 this PRNG would seed as SEEDING.4 and allow reseeding as ADD_ENTROPY.1 or ADD_ENTROPY.3, because the PRNG only partially resets its state before adding more seed material (i.e. the old seed data and some counters are kept). The PRNG holds most of the internal state in a big `int[]` called `seed`. Its structure with and without the flaw is shown in Figure 2.1, taken from [9].

Structure of the hash implementation. `SHA1Impl` is a partial implementation of the SHA1 hashing algorithm. Its interface consists of two methods:

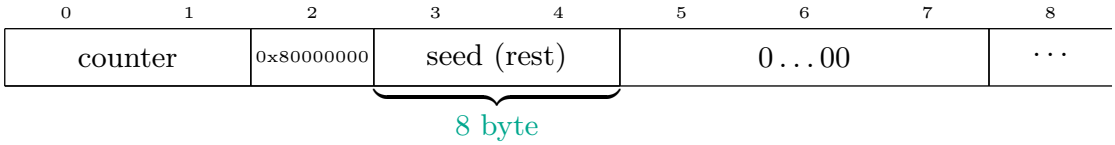
- `updateHash(int[] intArray, byte[] byteInput, int fromByte, int toByte)`
- `computeHash(int[] arrW)`

Both methods work on `int[]`s (`intArray` and `arrW`) that are structured like the state shown in Figure 2.1. In fact the complete state array is passed to this function in every call, when main the function `SHA1PRNG_SecureRandomImpl.engineNextBytes` uses `computeHash` to generate random bytes.

Necessary changes to the PRNG for verification. The PRNG operates in a slightly different mode, when seeded manually compared to self-seeding. But in that self-seeding code there was the error that caused the Bitcoin incident. The code from `engineNextBytes` that was wrong and caused the problem for the Bitcoin incident isn't even executed when seeding manually. In order to be able to prevent errors like the one that caused the Bitcoin incident and because self-seeding is the recommended mode of operation, we want to verify the code path used when the PRNG is self-seeding. In



(a) Intended operation



(b) Effect of the bug

Figure 2.1: Structure of the Android PRNG's main array (1 word = 1 `int` = 4 bytes) from [9]

order to do that, we have to slightly change the signature of `engineNextBytes`, because we need the seed data as part of the signature to use it in the information flow contract.

We focus on the case when the `engineNextBytes` function is called for the first time, does self-seeding and generates a first block of output. We want to specify information flow with the initial entropy used in self-seeding so we had to make that data an additional parameter to `engineNextBytes`, so that we can talk about it in its specification. The new method signature is then `engineNextBytes(byte[] bytes, byte[] extSource)`, where `bytes` is the array to place the random bytes into and `extSource` is the seeding entropy. When `engineNextBytes` is called as we specify (PRNG not initialized yet, byte arrays sane, not aliased, . . .), the method . . . :

1. . . . takes the `extSource` and copies it into `seed` (Figure 2.1) by calling `updateHash`. Internally `updateHash` packs every 4 bytes into one integer (as Java has 4-byte integers). and stores those 5 integers in the internal state array.

2. . . . calls `computeHash` on `seed`. The resulting hash is stored as 5 integers at index 82 in the same array (see Figure 2.1).

3. . . . unpacks the calculated 5 integers into 4 bytes each of the `nextBytes` array.

4. . . . copies those 20 bytes into the output array `bytes`.

Overall remarks on code quality. The code in the main methods is complex, dense and hard to follow. Although there are comments all over the code, using jittering

```
74      /*@ public normal_behavior
75          requires arrW.length==87;
76          ensures (\forall int i; 82 <= i & i <= 86; -2147483648 <= arrW[i] &
                arrW[i] <= 2147483647 );
77          assignable arrW[16..79],arrW[82..86];
78          accessible arrW[0..79],arrW[82..86];
79          determines \pre( (\seq_def int i;  0;  5; arrW[i])) \by
80                     \post((\seq_def int i; 82; 87; arrW[i]));
81      */
82      static /*@helper*/ void computeHash(int[] arrW) {
```

Figure 2.2: The specification of the hash function

terminology and sometimes being inconsistent with code, they pose more questions than they answer.

2.2 Information Flow Through the Hash Function

As mentioned above, the PRNG uses a version of the SHA1 hash function that has been implemented specifically for this PRNG. While those methods do calculations as described in the SHA1 standard they are not used to implement plain calculations of the SHA1 function. E.g. the length of the current data is not encoded in the padding and the function's state is not reset after each calculated hash. However the functions still pose much computational complexity.

Nevertheless want to specify the injectivity of this function. As its input and output are both located in the `seed` array (input at offsets $0-15$ and output at offsets $82-86$) we need to specify injectivity between those two intervals. However our input seed is always placed into bytes $0-4$ and the remainder of the input is only fixed data. The JML* specification that describes this information flow (i.e. claiming injectivity) and that states that the resulting integers are all in `int` range, is shown in Figure 2.2. It specifies:

1. No exceptions can be thrown: `normal_behaviour`
2. The input array needs to have a length of 87 (as the structure shown in Figure 2.1).
3. It guarantees the resulting integers in the hash part of the array are in `int` range.
4. The function only writes to the parts of the array called "computation space" and "SHA1 result".
5. The function is injective from the input data (interval `[0..4]`) to the "SHA1 result" (interval `[82..86]`)

The slightly modified version of `engineNextBytes` can be proven to fulfill this specification. Thereby we proof that this modified version hash has full information flow.

2.3 Proof Outline

2.3.1 Difficulties during the Proof

Byte packing and unpacking. Besides the overall size and complexity of the program under verification, `SHA1PRNG`, the packing and unpacking actions were the main problem. KeY cannot prove the injectivity through those actions as they involve bitwise or operations, which cannot be easily represented in standard mathematical algebra. Furthermore, in the original code those two actions are not in their own procedures, but directly embedded into the body of the `nextBytes` function, so we can not annotate those functions with their information flow contracts (as we did for the SHA1 implementation). We solved this problem by writing rules for KeY specifically designed to transform those formulas. The rules, together with example code, are shown in Figure 2.3 and Figure 2.4.

The "pack" rule (Figure 2.3) searches for the equivalence of two integers and replaces it with the byte equivalences while requiring in an auxiliary goal, that the bytes are in fact in byte range. The "unpack" rule (Figure 2.4) searches for the equivalence of the second byte of the int and replaces it with the int equality while requiring an in auxiliary goal that the other three bytes are equal as well.

Those rules have been proven with KeY's rule justification functionality. Because KeY still cannot prove the rules, the resulting proof obligation has been exported to the Z3 prover where the correctness of the rule has been proven.

Choosing integer semantics. We need to setup KeY to use its overflow-checking integer semantics. From this set of rules, we need to disable the rule that resolves casts to `byte`, because the only casts to byte in the given program are used for doing the integer packing. KeY's default rule for byte casts is to add a "modulo 256" operation to it. This is not beneficial for our proof as it only complicates the proof and brings Disabling this default rule allows and resolving the casts with the unpack rule make the rule and its manual application in the proof significantly easier.

2.3.2 User Interactions for the Proof with KeY

Proving the information flow contract for `engineNextBytes` results in a big formula in the schema from Figure 1.3. This formula contains two code fragments twice for representing the two executions of our program (in the schema called f). In order to keep track of what happens here we introduce notation. h^a and h^b denote heaps in the two different invocations. Let:

- h_0^a, h_0^b be the initial heaps at the beginning of `engineNextBytes`
- h_Before^a, h_Before^b the heaps before invocation of `computeHash`
- h_After^a, h_After^b the heaps after `computeHash`
- $h_\perp^a$,$h_\perp^b$ the heaps when `engineNextBytes` returns.

Rule:

```
\find(
  orJint(orJint(orJint(shiftleftJint(andJint(byte0b, 255), 24),
                       shiftleftJint(andJint(byte1b, 255), 16)),
                shiftleftJint(andJint(byte2b, 255), 8)),
       andJint(byte3b, 255))
  =
  orJint(orJint(orJint(shiftleftJint(andJint(byte0a, 255), 24),
                       shiftleftJint(andJint(byte1a, 255), 16)),
                shiftleftJint(andJint(byte2a, 255), 8)),
       andJint(byte3a, 255))

)\sameUpdateLevel
\replacewith(byte0b=byte0a & byte1b=byte1a
     & byte2b=byte2a & byte3b=byte3a);
\add(==>
  -128 <= byte0a & byte0a < 128 &
  -128 <= byte1a & byte1a < 128 &
  -128 <= byte2a & byte2a < 128 &
  -128 <= byte3a & byte3a < 128 &
  -128 <= byte0b & byte0b < 128 &
  -128 <= byte1b & byte1b < 128 &
  -128 <= byte2b & byte2b < 128 &
  -128 <= byte3b & byte3b < 128
)
```

Example:

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

Figure 2.3: The new KeY rule representing injectivity of packing 4 bytes into an int together with example code where the rule would be needed.

Rule:

```
\find(
  javaCastByte(unsignedshiftrightJint(intb, 8))
    = javaCastByte(unsignedshiftrightJint(inta, 8))
  )\sameUpdateLevel
  \replacewith(inta = intb);
\add( ==>
  inInt(inta)& inInt(intb)&
  javaCastByte(intb)
    = javaCastByte(inta) &
  javaCastByte(unsignedshiftrightJint(intb, 24))
    = javaCastByte(unsignedshiftrightJint(inta, 24)) &
  javaCastByte(unsignedshiftrightJint(intb, 16))
    = javaCastByte(unsignedshiftrightJint(inta, 16))
)
```

Example:

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

Figure 2.4: The new KeY rule representing injectivity of unpacking an int into 4 bytes together with example code where the rule would be needed.

Our proof follows this chain of implications:

$$bytes[*]@h^a_\perp = bytes[*]@h^b_\perp$$

$$\overset{(post)}{\rightarrow} seed[82..87]@h\_After^a = seed[82..87]@h\_After^b$$

$$\overset{(*)}{\rightarrow} seed[0..5]@h\_Before^a = seed[0..5]@h\_Before^b$$

$$\overset{(pre)}{\rightarrow} extSource[*]@h^a_0 = extSource[*]@h^b_0$$

Note that $=$ corresponds to component-wise equality of arrays. Implication $(*)$ is assumed as injectivity of the hash function. Implications (pre) and $(post)$[1] are to be shown during the proof.

To prove the big formula, we do the following:

1. Do symbolic execution for the code fragments and simplify the resulting formula. Using KeY's Strategy macro "Start auxiliary computation for self-composition proofs" we do the simplification steps only once and then apply the result on the two expressions in the main proof. This corresponds to simplification of the function f from Figure 1.3.

2. Use the information flow contract for `computeHash`. This results in 3 subgoals:

[1]The names correspond to names given by KeY to those two subproofs. They seem swapped because we are using the `determines` clause in opposite direction to the designers original intent.

a) The first one can be closed automatically. This goal represents proving the functional precondition of `computeHash` in both invocations (line 75 from Figure 2.2).

The remaining two goals represent full entropy flow up to and from the call to `computeHash`.

b) We show that given two outputs of `nextBytes` are equal ($bytes[*]@h^a_\perp = bytes[*]@h^b_\perp$), the "hash" part of the `seed` arrays after `computeHash` has to be equal ($seed[82..87]@h\_After^a = seed[82..87]@h\_After^b$). KeY calls this goal "pre" and it corresponds to the implication (pre) from above. This part represents full information flow from after `computeHash` to the function's end. To prove this claim several interactive steps are required.

   i. Instantiate the `\forall` in the term $bytes[*]@h^a_\perp = bytes[*]@h^b_\perp$ manually 20 times and remove unnecessary assumptions. This breaks the equality into 20 byte equalities.

   ii. Apply auto-mode to simplify the formula, so that the instantiated byte equivalences are simplified.

   iii. Apply "unpack"-rule on the 20 equivalences 5 times, which splits the proof 5 times.

   iv. The resulting 5 auxiliary goals can be closed automatically because the information that the other 3 byte pairs are equal is readily available in the sequent.

   v. Cut the proof into 5 sub cases that represent the 5 integer equalities of $seed[82..87]@h\_After^a = seed[82..87]@h\_After^b$.

   vi. Close those 5 subgoals using auto-mode.

c) Closing the second goal ("post") works symmetrically.

   i. Instantiate the `\forall` clause in $seed[0..5]@h\_Before^a = seed[0..5]@h\_Before^b$ 5 times and remove unnecessary assumptions.

   ii. Apply the auto-mode.

   iii. Apply the "pack" rule 5 times.

   iv. All 5 new auxiliary subgoals can be closed automatically.

   v. Do a case splitting into 20 subcases one for each of the 20 bytes of $extSource[*]@h^a_0 = extSource[*]@h^b_0$.

   vi. Close all those 20 subgoals using auto-mode.

Thereby all goals are closed, the proof is completed and the information flow property is proven for this PRNG.

3 BouncyCastle's `DigestRandomGenerator`

3.1 Structure of the PRNG

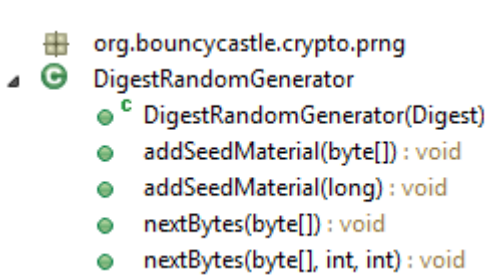
3.1.1 Class Structure

BouncyCastle's PRNG is contained in a single class. Its outline is shown in Figure 3.1a. Regarding our classification from Section 1.3.2 the PRNG seeds as SEEDING.3 and allows reseeding as ADD_ENTROPY.1. The PRNG is parametrized with a `Digest` object. This object wraps a hash function that is used inside the PRNG.

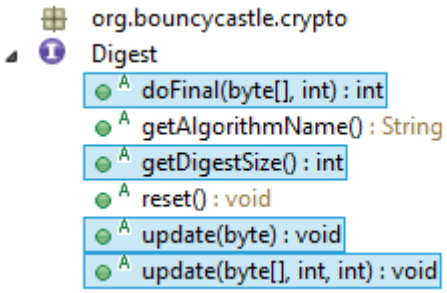
Before the PRNG can be used, it has to be seeded manually by calling `addSeedMaterial(byte[])` or `addSeedMaterial(long)`. The two methods are internally nearly identical. For the sake of simplicity we consider only the `byte[]` version.

In order to acquire output the user then has to call `nextBytes(byte[])` or `nextBytes(byte[], int, int)`. `nextBytes(byte[], int, int)` gets a `byte[]` where the random data should go, together with an offset and a length parameter, specifying the interval that should be filled. `nextBytes(byte[])` is a simple default parameter wrapper that assumes the offset to be 0 and the length to be the array length indicating that the whole array should be filled. We specify and prove information flow for `nextBytes(byte[], int, int)`, because the other one can be inlined easily and then the specification of the complex method can be used. When we verify our main method, which uses `nextBytes(byte[])`, KeY inlines this method and uses the (proven) information flow contract for `nextBytes(byte[], int, int)`.

In our main method (Figure 3.2), we used this calling convention for operating the PRNG. We verify information flow through this function in order to verify this PRNG.



(a) Outline of `DigestRandomGenerator`



(b) Outline of `Digest`

Figure 3.1: Outlines of the core interfaces of the BouncyCastle PRNG

```
20          public static void test(byte[] seed, byte[] output, Digest d) {
21                  DigestRandomGenerator drng = new DigestRandomGenerator(d);
22                  drng.addSeedMaterial(seed);
23                  drng.nextBytes(output);
24          }
```

Figure 3.2: The main function for the Bouncy Castle PRNG

```
35          public static void testHash(byte[] seed, byte[] output, Digest d) {
36                  d.update(seed, 0, seed.length);
37                  d.doFinal(output, 0);
38          }
```

Figure 3.3: Example for the usage of the `Digest` interface

3.1.2 Thread Safety

For all PRNGs we consider only single-threaded invocations, because multi-threaded invocations require specific tooling, are complex and often (as in this PRNG) not necessary to look at: All public methods of the `DigestRandomGenerator` class are wrapped in **`synchronized`** blocks on **`this`**. Also the **`this`** reference is not leaked in the constructor. This pattern ensures that method (and constructor) calls to the PRNG are atomic. Because of that, we can look at every parallel usage as a sequence of sequential method calls.

We can thus assume that the PRNG is used by a single thread. Because KeY is specialized on sequential programs, it currently has no rule for **`synchronized`** blocks (except for empty ones). We remove those blocks and replace them with their plain contents.

3.1.3 Structure of the Hash Function

The hash function that the PRNG uses is wrapped inside the `Digest` interface. An outline of this interface is shown in Figure 3.1b. The methods that are actually used by `DigestRandomGenerator` are highlighted.

This interface is stateful and streaming-based. This means the hash function has an internal state that is updated by `update` (again with offest, length version and a default parameter like `nextBytes` above) and then finally condensed into a hash by `doFinal`. Calculating the hash resets the state of the hash function (as if one would be calling `reset`), so that it can be used again cleanly. Normal use of the hash function would look like the method in Figure 3.3.

3.2 Information Flow Through the Hash Function

Overview As for the other PRNGs we want to assume that the hash function used in the PRNG is injective. This can be expressed by the means of an information flow

```
26          /*@
27           requires seed != output;
28           requires seed.length == 20;
29           requires output.length == 20;
30           requires d.digestState != null;
31           requires d.digestState != seed;
32           requires d.digestState != output;
33           determines \pre((\seq_def int i; 0; seed.length; seed[i])) \by
                 \post((\seq_def int i; 0; output.length; output[i]));
34           */
35          public static void testHash(byte[] seed, byte[] output, Digest d) {
```

Figure 3.4: Specification for the hash function example code (Figure 3.3)

```
7           /*@ ghost instance final byte[] digestState = new byte[5];*/
```

Figure 3.5: Ghost field for modeling the internal state of the hash function

specification like `determines x by hash(x)`. For our test function `testHash` this can be expressed with the specification in Figure 3.4. The specification additionally contains non-aliasing properties. Also we limit ourselves to 20-byte arrays, because 20 bytes is the size of a SHA1 hash and using a small and constant length keeps the proof manageable. Therefore also the `getDigestSize` method of the Digest interface is specified to always return 20. With KeY the specification of the hash function example code (Figure 3.3) can be proven to be correct using the specification of the `Digest` interface that we explain in the following.

Specification of the hash function interface. Since the hash function interface is stateful, this internal state has to be expressed in the specification. We use a ghost field `digestState` to model this internal state. A ghost field is a field that exists only for verification and specification purposes. The declaration of this field in order to specify the behavior of the `Digest` interface can be seen in Figure 3.5.

Now we specify the information flow behavior of the `update` and `doFinal` functions in Figure 3.6. For `update` the specification states that all information from the input parameter `in` (in the given range) and all information from the previous `digestState` flows into the new `digestState`. For `doFinal` the specification states that all information from `digestState` flows into the `out` parameter (at the given offset `outOff`). Note that there is no statement about the new contents of `digestState`. This is in consent with `doFinal` automatically resetting the hash function and not preserving information from the seed or the previous state.

Both specifications require "assignable" clauses. With `assignable digestState[*]` in the specification we assure that no other global state apart from the contents of `digestState` can be changed by the `update` function. Likewise the two assignable clauses for `doFinal` assert that only the content of `out` and `digestState` can be changed by that method.

```
58          /*@
59           normal_behavior
60           determines \pre((\seq_def int i; inOff; inOff+len; in[i])),
                 \pre((\seq_def int i; 0; digestState.length; digestState[i]))
                 \by \post((\seq_def int i; 0; digestState.length;
                 digestState[i]));
61           assignable digestState[*];
62           */
63          public/*@helper*/void update(byte[] in, int inOff, int len);

74          /*@
75           normal_behavior
76           requires outOff + getDigestSize() == out.length;
77           determines \pre((\seq_def int i; 0; digestState.length;
                 digestState[i])) \by \post((\seq_def int i; outOff; outOff +
                 getDigestSize(); out[i]));
78           assignable out[*];
79           assignable digestState[*];
80           */
81          public/*@helper*/int doFinal(byte[] out, int outOff);
```

Figure 3.6: Specification of `update` and `doFinal` methods from `Digest.java`

```
24          //@invariant seed != digest.digestState;
25          //@invariant digest.digestState != state;
26          //@invariant digest.digestState != null;
```

Figure 3.7: The class invariants for `Digest` that we moved to `DigestRandomGenerator`

All methods of the hash function interface require a `@helper` annotation. This annotation prevents KeY from assuming class invariants (the `Digest` class has non-empty default invariants) as precondition and requiring them to be verified as postconditions. We need this annotation due to a limitation that prevents KeY from resolving the invariants of a class different from the one that currently being proved. The necessary invariants have been moved to the `DigestRandomGenerator` class to work around this limitation. Specifically it is the general non-null invariant that JML introduces by default and non-aliasing invariants with the other arrays in `DigestRandomGenerator` (required to be sure that all those arrays are actually different arrays and not the same array object referenced by different fields).

Problems with the injectivity of hash functions due to domain sizes. Also note that the information for the `update` method technically cannot be true, because a function mapping a larger, finite input space to a smaller, finite output space cannot be injective. This is a manifestation of the problem with assuming that a hash function is injective that was mentioned in Chapter 1. We need to assure that the proof system doesn't use this contradiction by manual proof inspection.

```
30          /*@
31            normal_behavior
32            ensures seedCounter == 1;
33            ensures stateCounter == 1;
34            ensures \fresh(seed);
35            ensures \fresh(state);
36            ensures this.digest == digest;
37            assignable this.*;
```

Figure 3.8: The specification of the constructor of `DigestRandomGenerator`

```
11          /*@
12           requires seed != output;
13           requires seed.length == 20;
14           requires output.length == 20;
15           requires d.digestState != null;
16           requires d.digestState != seed;
17           requires d.digestState != output;
18           determines \pre((\seq_def int i; 0; seed.length; seed[i])) \by
                 \post((\seq_def int i; 0; output.length; output[i]));
19           */
```

Figure 3.9: Main proof goal for the BouncyCastle PRNG from `Launcher.java`

3.3 Specification for the Constructor of `DigestRandomGenerator`

As one already could see in Figure 3.2, we call the constructor of `DigestRandomGenerator` in our main method. We do that instead of specifying an initial clean state as precondition for our main method, because the constructor establishes such a clean state. The specification of the constructor is shown in Figure 3.8. It ensures the initial values for the counters (required to decide whether certain conditionals are taken), fresh state arrays (for proving the non-aliasing properties and invariants) and that the parameter `Digest` object is installed into its designated field.

3.4 Information Flow Target

The main proof goal is the information flow through the `test` method (see Figure 3.2) that calls the PRNG form an external class. The method receives the seed, an array to write the output to, and a digest object. It then creates a new PRNG object, seeds it with the given seed and let's the PRNG generate random output. The required specification for this method is straight forward. All information from `seed` needs to flow into `output`. This information flow contract is shown in the specification of this method (see Figure 3.9). The only remaining parts of that specifications are nonNull checks, non-aliasing checks and checking that the arrays have the length of exactly one digest. The digest length has been fixed to 20 bytes in order to get the proof manageable.

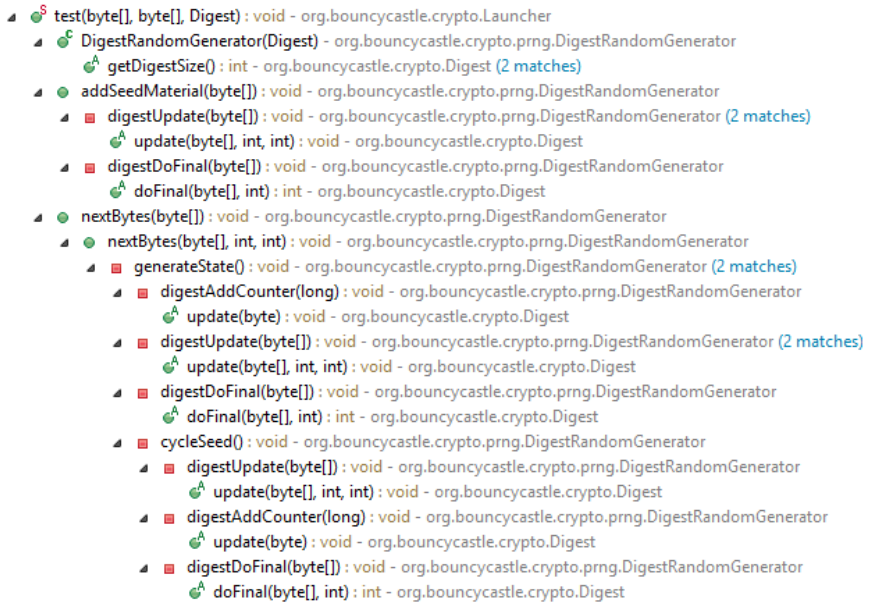


Figure 3.10: The call hierarchy of the BouncyCastle PRNG.

3.5 Proof Outline

The proof has been carried out using KeY, by proving all the contracts and subcontracts individually. KeY has been modified to not apply information flow specific rules (the ones that use the `RELATED_BY`-markers) automatically, because it often applied them in cases where they didn't help the proof. An overview of all proved sub parts is shown in Figure 3.10.

Integer arithmetics and loop treatment. The only method involving bit arithmetics is `digestAddCounter(long)`. Thus we use "javaSemantics" for this method and "arithmeticSemanticsCheckingOF" for all other methods. Because all loops have finite and small borders we can switch loop treatment to "Expand".

Proving the contracts. Proving works similar for most methods. Generally one can start by using the "Start auxiliary computation for self-composition proofs" macro as we did for the Apache SHA1PRNG. Then one can apply KeY's automated strategy, finish the auxiliary calculation and use the new rule from the auxiliary computation in the main proof. Now KeY simplifies that new formula so that all `RELATED_BY` predicates are on top level. This is necessary in order to apply the `RELATED_BY` rules, because they search for the markers only on top level. These rules are now applied (manually) to a pair of markers from the two different invocations that are their syntactic counterpart. From there on, KeY manages to close all proofs on its own.

4 OpenSSL's ssleay_rand

4.1 Structure of the PRNG

Like BouncyCastle's `DigestRandomGenerator` OpenSSL's PRNG is also independent from its hash function. The API is similar. The relevant functions are `rand_add(const void *buf, int num, double add_entropy)` to add entropy and `rand_bytes(unsigned char *buf, int num)` to generate output. The methods are shown as simplified pseudocode in Figure 4.1.

When entropy is added using `rand_add`, it is split into 20-bytes blocks and hashed blockwise into a static array of 1043 bytes, that is called `state`. Each addition of a block changes at most 20 bytes of `state`. The data is also hashed into the 20-byte array `md`. To generate output we select 10 bytes of `state` and hash those together with two counters into `md`. Then we split the resulting hash and use 10 bytes as output and XOR the other 10 bytes back into `state`. This process is repeated until enough output bytes are generated. If the requested amount of output bytes is not a multiple of 10, superfluous bytes are discarded.

Regarding our classification from Section 1.3.2 the PRNG seeds as SEEDING.5 and allows reseeding as ADD_ENTROPY.1.

Structure of the hash implementation. The hash function interface visible to the PRNG are three methods: `MD_Init`, `MD_Update` and `MD_Final`. They operate similarly to the hash function in BouncyCastle, but in this case `MD_Final` doesn't reset the hash function, but leaves it in an undefined state. `MD_Init` has to be called on the hash function object to reuse it for another hash calculation.

4.2 Information Flow Through the Hash Function

In order to assume full information flow through the hash function, we had to replace it with a dummy variant, that is aware of the manner the PRNG uses it. To touch the surrounding code as little as possible, we created a new OpenSSL hash function and told the PRNG to use that hash function.

In order to implement our own hash function, we need to provide `init`, `update` and `final`. Those methods then get wrapped into their `MD_` variants that are used by the PRNG.

The hash function we used is shown in Figure 4.2. It uses XOR to fold longer inputs into its hash. Additionally it keeps two counters:

```
1  long md_count_0;
2  long md_count_1;
3  char state[1043];
4  char md[20];
5  int state_index;
6  int initialized;
7
8  void rand_add(const void *buf, int num) {
9    char local_md[] = copyOf(md);
10
11   for(int i = 0; i < num; i += 20) {
12     local_md = hash(local_md
13          | state[state_index...(state_index+20) % 1043]
14          | buf[i..min(i+20,num-i)] | md_count_0 | md_count_1);
15     md_count_1++;
16     state[state_index...(state_index+20) % 1043] ^= local_md;
17   }
18   md ^= local_md;
19 }
20
21 void rand_bytes(const void *buf, int num) {
22   if(!initialized) stirr();
23   char local_md[] = copyOf(md);
24   int i = 0;
25   while(num > 0) {
26     local_md = hash(local_md
27          | md_count_0 | md_count_1
28          | state[state_index...(state_index+10) % 1043] );
29     state[state_index...(state_index+10) % 1043] ^= local_md[0..10];
30     state_index = (state_index + 10) % 1043;
31     buf[i..i+min(num,10)] = local_md[10..10+min(num,20)];
32     num -= 10;
33     i += 10;
34   }
35   md = hash(md_count_0 | md_count_1 | local_md | md);
36 }
```

Notation

- `buf[a..b]` denotes the sub-array of `buf` starting with `a` and ending with `b` (exclusive). If `b<a` the expression denotes the sequence starting with `b` over the sequence's end up to the element before `a`. With that syntax, `buf` is treated as a cyclic array.
- `hash(a|b|c)` denotes the hash of the concatenation of `a`, `b` and `c`.
- Array action like assignments and XORs are to be understood component-wise.

Figure 4.1: Simplified pseudocode of the OpenSSL PRNG

```
135 #define SHA_DIGEST_LENGTH 20
136 #define SHA_CBLOCK 20
137
138 // Counts the invocations of this hash functions
139 static int hash_function_invocation_count;
140
141 static int init(EVP_MD_CTX *ctx)
142         {
143           hash_function_invocation_count++;

147         memset(ctx->md_data,'\0', SHA_DIGEST_LENGTH + 1);
148         return 0;
149         }

151 static int update(EVP_MD_CTX *ctx,const void *data,size_t count)
152         {
153           char pos = (((char*)ctx->md_data)[SHA_DIGEST_LENGTH])++;
154         size_t i;

162         for (i = 0; i<count; i++)
163                 {
164                   ((char*)ctx->md_data)[(i + pos) % SHA_DIGEST_LENGTH] ^=
                          ((char*)data)[i];
165                 }

175         return 0;
176         }
177
178 static int final_fake(EVP_MD_CTX *ctx,unsigned char *md)
179         {
180         ((char*)ctx->md_data)[SHA_DIGEST_LENGTH] = 0;
181         size_t i;
182         int off = 1;
183         if(hash_function_invocation_count == 3){

187           off = 10;
188         }
189         for (i = 0; i<SHA_DIGEST_LENGTH; i++)
190                 {
191                 md[(i + off) % SHA_DIGEST_LENGTH] = ((char*)ctx->md_data)[i];
192                 ((char*)ctx->md_data)[i] = 0;
193                 }

202         return 0;
203         }
```

Figure 4.2: A stripped version of the dummy hash function we used for OpenSSL.

```
 9 extern int initialized;

...

42          RAND_add(in, INPUT_AMOUNT, 1);
43          initialized = 1;
44          RAND_bytes(out, OUTPUT_AMOUNT);
```

Figure 4.3: The calls to the OpenSSL PRNG from its verification driver

1. `pos` identifies different calls to `update`. We use it to XOR different data into different positions (line 164) to prevent canceling out of same data if it is fed into `update` multiple times.

2. `hash_function_invocation_count` identifies different calculations. We need this to make a slightly different output in the 3rd calculation (Figure 4.2 line 183). This is required because OpenSSL outputs two "half-hashes" (see Figure 4.1 line 31) and we need to present the second half as the second hash.

4.3 Verification Driver

We want to verify information flow through the first PRNG cycle, that is a call to `rand_add` and a call to `rand_bytes`. But the PRNG seeds itself (see item ADD_ENTROPY.1). This additional entropy that is not known to our prover needed to be removed. We can ignore the code path for self-seeding, because it consists only of gathering OS-specific entropy and adding it to the PRNG state with the `rand_add` function that we also use to add our entropy manually. We do this by suggesting the PRNG that it has already been initialized. The PRNG stores this information in a `static` variable. This prevents us from accessing it. We removed the static declaration so we could declare an `extern` variable that represents the same memory. We did this in line 9 of our verification driver shown in Figure 4.3. The shown code is then used inside the verification driver for proving injectivity shown in Figure 1.5. Additionally we had to make sure that all code is compiled (and verified) using the `PURIFY` preprocessor macro. This removes uninitialized buffers as entropy sources of the OpenSSL PRNG.

4.4 Proof Outline

4.4.1 Problems with the Original Source Code

The next step is to make CBMC translate this program into the CNF. While doing so, we ran into two problems.

Helping Constant Propagation. OpenSSL uses `structs` of function pointers when accessing e.g. the hash function. This indirection should be constant for all our use cases.

```
264 #ifdef __CPROVER__
265                   __CPROVER_assert(m.digest==EVP_sha1_fake(),"xx");
266                   m.digest=EVP_sha1_fake();
267 #endif
```

Figure 4.4: An example for helping constant propagation in `md_rand.c`.

We use code like show in Figure 4.4 to show CBMC what the value of a certain variable (in that case a `struct` of function pointers) will be. We prove all the assertions by proving the whole verification driver where the `assert(0)` at the end has been removed (check can be executed with the `sanity` target of our Makefile). After having proven the assertion we can safely add the assignment (the `EVP_sha1_fake()` just returns a constant value).

Preventing Pool Stirring. When used the first time, the OpenSSL PRNG "stirs" its state, so the random data gets "'evenly distributed' through 'state', our randomness pool" [20]. This process involves hashing data multiple times and XORing it all over the state. Those calculations make our proof far more difficult if not impossible. Probably we could incorporate this stirring in our dummy hash function and by that prove that also with stirring all of the entropy could theoretically come out of the PRNG again. We didn't do that, because it would increase the required complexity of the dummy hash function to an extent that would go far beyond the scope of this bachelor thesis. For the proof in this thesis, we disabled stirring of the pool as shown in line 422 of Figure 4.5.

4.4.2 Conducting the Proof

Doing the proof now works as outlined in Subsection 1.6.2. We verify that all assertions hold by removing the `assert(0)` from the verification driver and check that the resulting formula is unsatisfiable. For an additional sanity check, we then add the `assert(0)` again and check that now the formula is satisfiable. Finally we construct the self-composition of that formula with our tool and check that this formula is unsatisfiable.

4.4.3 Conclusion from the Proof

By completing the proof we would expect the output to contain 20 bytes of entropy. However, OpenSSL is not too gentle with the hashes of our dummy hash function. OpenSSLs way of generating output is similar to the example in Subsection 1.4.4, that is OpenSSL uses two partial hashes as output. As explained in the example, when seeded with 20 bytes of entropy the expected entropy in the first 20 bytes of the PRNG's output is in fact only 15 bytes, but the maximum entropy is 20 bytes. This issue and its impact need further investigation, which we leave for future work.

```
422                 while (n > 0 && 0)
423                         {
424 #if MD_DIGEST_LENGTH > 20
425 # error "Please␣adjust␣DUMMY_SEED."
426 #endif
427 #define DUMMY_SEED "...................." /* at least MD_DIGEST_LENGTH */
428                         /* Note that the seed does not matter, it's just that
429                          * ssleay_rand_add expects to have something to hash.
                                */
430                         ssleay_rand_add(DUMMY_SEED, MD_DIGEST_LENGTH, 0.0);
431                         n -= MD_DIGEST_LENGTH;
432                         }
```

Figure 4.5: The disabled pool stirring code from `md_rand.c`.

5 Conclusion

Formal verification techniques are suitable for verifying PRNGs. We verified 3 real-world PRNGs for their correctness regarding using all entropy that is supplied to them when seeding. In the cases of the two that contained real-world bugs, we would have discovered those bugs because our proofs would have failed if those bugs were in the source code. The experiment show that formal verification can detect such problems and reduces the chances of such bugs.

Verifying PRNGs is possible in a reasonable amount of time. For every PRNG the main effort went into understanding how the PRNG is designed to operate. Specifying the desired property was similar for all PRNGs and therefore an easy task. Other problems were caused by monolithic and dense code style. They could have been avoided, if the PRNGs source code would have been well-structured. A code maintainer of the original code could clean up and modularize the original code to facilitate such proofs. Verification would be an incentive to keep the code well-structured. A good example for this is the OpenSSL PRNG where especially factoring out "pool stirring" would be beneficial. BouncyCastle's `DigestRandomGenerator` was by far the best-structured PRNG of the ones analyzed. The remaining work, resolving, debugging and circumventing other verification failures, is often manageable within a few days. Overall we could probably verify a new PRNG, that is not too awfully structured, in 1-2 weeks.

Proofs can be integrated into existing regression detection. While the two proofs for Java programs require user-interaction, our proof for OpenSSL is fully automated and could be incorporated in the OpenSSL regression test suite and regularly re-checked, if there is still maximum entropy flow. Changing core parts of the PRNG could then require an adjustment to the dummy hash function but that encourages additional thinking on whether the code changes are justified.

Future work. More properties of the PRNG could be specified and verified, e.g: "not knowing all previous outputs and the seed, the n^{th} output block of the PRNG contains maximum entropy" or the property that "stirring the pool" tries to ensure could be specified and verified.

Bibliography

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, number 8471 in Lecture Notes in Computer Science, pages 1–17. Springer-Verlag, 2014.

[2] Boaz Barak and Shai 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] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[4] Bitcoin.org. Android security vulnerability. `https://bitcoin.org/en/alert/2013-08-11-android`, 2013.

[5] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In K. Jensen and A. Podelski, editors, *TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer Berlin Heidelberg, 2004.

[6] Mario Cornejo and Sylvain 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.

[7] Debian weak key vulnerability. CVE-2008-0166. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166`, 2008.

[8] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel 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.

[9] Felix Dörre and Vladimir Klebanov. Pseudo-random number generator verification: A case study. In Arie Gurfinkel and Sanjit A. Seshia, editors, *Proceedings, Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2015.

[10] John-Mark Gurney. URGENT: RNG broken for last 4 months. `https://lists.freebsd.org/pipermail/freebsd-current/2015-February/054580.html`, 2015.

[11] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[12] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.

[13] The KeY tool. Website at `www.key-project.org`.

[14] Vladimir Klebanov. Precise quantitative information flow analysis – a symbolic approach. *Theoretical Computer Science*, 538(0):124–139, 2014.

[15] Vladimir Klebanov, Norbert Manthey, and Christian 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.

[16] Daniel Kroening and Michael Tautschnig. CBMC – C bounded model checker. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer Berlin Heidelberg, 2014.

[17] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[18] Kai Michaelis, Christopher Meyer, and Jörg 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.

[19] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Des. Codes Cryptography*, 30(2):201–217, September 2003.

[20] Openssl source code. `https://github.com/openssl/openssl/blob/OpenSSL_1_0_2d/crypto/rand/md_rand.c#L422`, 2015.

[21] Christoph Scheben. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014.

[22] Christoph Scheben and Peter H. Schmitt. Efficient self-composition for weakest precondition calculi. In *Proceedings, Formal Methods (FM), 19th International Symposium*, pages 579–594. Springer, 2014.

[23] Kurt Stenzel. A formally verified calculus for full Java Card. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *Algebraic Methodology and Software Technology*, volume 3116 of *LNCS*, pages 491–505. Springer, 2004.