Karlsruhe Reports in Informatics 2016,12
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Practical Detection of Entropy Loss in Pseudo-Random Number Generators

Extended Version

Felix Dörre, Vladimir Klebanov

2016

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Practical Detection of Entropy Loss in Pseudo-Random Number Generators

Extended Version

Felix Dörre
Karlsruhe Institute of Technology, Germany
felix.doerre@student.kit.edu

Vladimir Klebanov
Karlsruhe Institute of Technology, Germany
klebanov@kit.edu

ABSTRACT

Pseudo-random number generators (PRNGs) are a critical infrastructure for cryptography and security of many computer applications. At the same time, PRNGs are surprisingly difficult to design, implement, and debug. This paper presents the first static analysis technique specifically for quality assurance of cryptographic PRNG implementations.

The analysis targets a particular kind of implementation defect, the *entropy loss*. Entropy loss occurs when the entropy contained in the PRNG seed is not utilized to the full extent for generating the pseudo-random output stream. The Debian OpenSSL disaster, probably the most prominent PRNG-related security incident, was one but not the only manifestation of such a defect.

Together with the static analysis technique, we present its implementation, a tool named Entroposcope. The tool offers a high degree of automation and practicality. We have applied the tool to five real-world PRNGs of different designs and show that it effectively detects both known and previously unknown instances of entropy loss.

Keywords

Pseudo-Random Number Generator; PRNG; entropy loss; information flow; OpenSSL; static analysis; bounded model checking

1. INTRODUCTION

Motivation and goal.

Somewhat simplified, a pseudo-random number generator (PRNG) is a software module that is *seeded* with a small amount of externally-sourced entropy (read randomness)[1] and "stretches" it into a stream that is indistinguishable from random to a computationally-bounded adversary. The seed is typically obtained from outside the immediate system scope: A user-space PRNG, for instance, may query the OS, which derives it, among other things, from physical noise in the hardware. The PRNG then produces a stream of pseudo-random data by, in cycles, permuting its internal state and deriving a fixed-length output chunk from a part of it. The latter part of the PRNG code, with which we will be concerned in this paper, is deterministic and contains both cryptographic and non-cryptographic parts.

On the implementation level, of all things, the non-cryptographic parts have shown a history of defects causing fatal security incidents. Many of these defects are *entropy losses*, where entropy supplied in the seed is overwritten with constant or predictable values, or otherwise remains unused during PRNG operation. This kind of defect makes it unnecessarily easy for an attacker to predict the PRNG output.

The probably most prominent PRNG security incident, the Debian OpenSSL disaster [28], was caused by such an entropy loss. While sufficient seed entropy was available (the code collecting entropy was working), only 15 bits of it (the ID of the current process) were used for generating output, resulting in merely $2^{15} = 32768$ distinct possible output streams (fixing endianness and native word size). As a consequence, for instance, only 2^{15} key pairs could be generated on any affected system, which allowed an attacker to easily brute-force the private key to any given public key generated on a vulnerable system.

For an attacker, a flawed PRNG is an attractive vector, as it constitutes a single point of failure for many services relying on cryptography. The Debian OpenSSL disaster effectively demonstrated this point by affecting the security of—among other things—DNS (BIND), Email (postfix, cyrus, uw-imapd), FTP, VPN (StrongSWAN, OpenVPN), SSH (OpenSSH clients and servers), Kerberos authentication, Tor, and WWW (Apache).[2] For almost two years, the general public lacked awareness that cryptographic measures securing these services on Debian systems were essentially turned off.

Attacks based on PRNG flaws are also particularly insidious, as they often require only passive access to the victim's communications. Breaking into the victim's system is typically not necessary. Such an attack leaves behind less evidence and is thus much more difficult to detect or, in aftermath, reconstruct. This point was illustrated by a series of unsolved bitcoin thefts probably going back to an entropy loss in the Android PRNG [23].

[1] Entropy is, strictly speaking, a measure of uncertainty, but, as customary, we overload the term to denote data with high entropy, i.e., data that is difficult to guess for an adversary.

[2] https://wiki.debian.org/SSLkeys

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978369

Despite the gravity of the situation, there are very few effective quality assurance techniques against implementation defects in cryptographic PRNGs. The first testing technique claiming potential to have prevented the Debian OpenSSL disaster was proposed only recently [27].

The overarching contribution of this work is the first static analysis technique for detecting entropy loss in PRNG implementations. An entropy loss is detected when the analysis finds two distinct seeds that produce the same output stream. The technique is implemented in a highly automatic analysis tool building on program verification technology and effectively applies to real-world implementations.

Contributions in detail.

The paper identifies *absence of entropy loss*, a particular important correctness property of PRNGs, and formulates it within the popular semantical framework of *information flow*. In contrast to functional specification, such a formulation is both succinct, easy to understand, and uniform across PRNGs. In contrast to the majority of established security research scenarios, PRNG correctness is concerned with maximizing and not minimizing information flow, rendering most of the existing security analysis tools inapplicable. The paper proposes a method to find deviations from flow maximality and thus instances of entropy loss.

A particular challenge in this context is the use of cryptographic primitives in PRNGs. Since our analysis is purely information-theoretic, we propose a way to deal with the issue by replacing such primitives with idealizations.

Implementing the method, the paper presents a tool for PRNG analysis, built on top of the CBMC bounded model checker for C and Java. Due to the carefully defined application scenario, the tool enables a fast feedback cycle, with counterexamples, i.e., potential witnesses of entropy loss, aiding the developer in understanding the problem.

We have applied the analyzer to five popular PRNG implementations, including the OpenSSL PRNG and Apple's version of Yarrow. We report our experiences in detail using the example of OpenSSL. One result is that the tool detected a previously undiscovered (if small) entropy loss in the OpenSSL PRNG. A larger previously undiscovered entropy loss was detected in the Libgcrypt PRNG.

The paper also includes a "mini-museum" of entropy loss with the goal to increase awareness of this kind of problem. This section presents and discusses several disparate instances of entropy loss, demonstrating the importance of the concept and the diversity of applications where the problem occurs.

2. ILLUSTRATION OF ENTROPY LOSS: ANDROID PRNG (2013)

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 Harmony PRNG was part of the Android platform up to and including Android 4.1. It was replaced when a problem with the PRNG became widely known [23], after a series of mysterious bitcoin thefts.

Bitcoin transactions are ECDSA signatures and include a nonce that is often generated by a PRNG. If the same nonce is used for two transactions signed by the same key, then anyone can reconstruct the private key of the victim from public information and divert their money without having to compromise their system. Entropy loss in the PRNG increases the probability of nonce reuse.

The entropy loss occurred in the main method of the Android PRNG, `engineNextBytes(byte[] bytes)`, which fills the caller-supplied array `bytes` with pseudo-random values. 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 seeding can happen either manually by calling `setSeed()` or automatically. In the latter case, the PRNG is seeded with 20 bytes of entropy requested from the OS kernel on first invocation of `engineNextBytes()`. This so-called *self-seeding* mode is typically considered preferable as less error-prone.

Figure 1(c) shows the essence of the PRNG's operation in this scenario. In cycle k, the 20 pseudo-random bytes are computed by combining the seed (words 0–4 in Figure 1(a)), the cycle counter k (as a 64-bit integer in words 5–6), and the output of cycle $k-1$ (resp. SHA-1 initialization vector in the initial cycle) with the SHA-1 compression function. The computation 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.

To compute the output, the seed and the cycle counter have to be suffixed by a standard-defined SHA-1 padding. 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. 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 1(b)). Twelve bytes of entropy are lost, and the actual amount of entropy in the PRNG amounts thus to 8 instead of 20 bytes.[3]

The goal of our analysis is to detect that two seeds differing in words 0–2 will produce the same output stream.

3. FOUNDATIONS

3.1 The PRNG Model

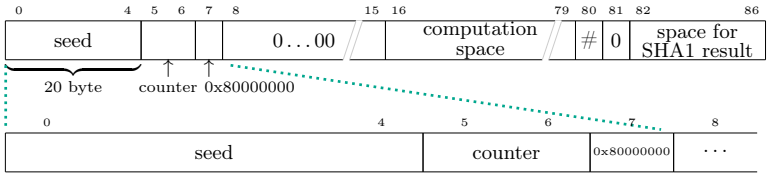
In this paper, we treat a PRNG as a function

$$g\colon \{0,1\}^m \to \{0,1\}^n \ ,$$

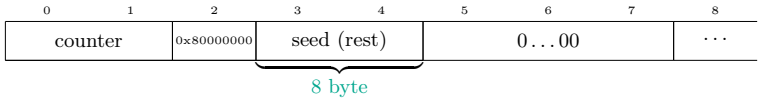
which translates a seed of m bits into a stream of n bits ($m, n > 0$). The fact that we only consider finite streams is inconsequential in the scope of this work. Since PRNG implementations typically use bytes as an atomic unit of information exchange, we establish the following convention. Whenever we use the parameters M and N, we are implying $m = 8M$ and $n = 8N$.

The above model of a PRNG as a function from seed to output goes hand in hand with the following assumptions that we make:

[3]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 loss, though these were much simpler technically [23].



(a) Intended operation



(b) Effect of the bug

$\texttt{out}_0 = \texttt{sha1(seed} \mid \texttt{0} \mid \texttt{sha1-iv)}$

$\texttt{out}_k = \texttt{sha1(seed} \mid \texttt{k} \mid \texttt{out}_{k-1}\texttt{)}$

(c) Equations describing the output of the Android PRNG in each cycle

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

1. The PRNG is seeded before it starts producing output.
2. The seed is chosen uniformly at random from the set $\{0,1\}^m$ for some $m > 0$.
3. An attacker neither knows the seed nor has control over its choice.
4. The PRNG is not re-seeded, i.e., the seed remains constant throughout the PRNG's lifetime.[4]
5. An attacker knows the source code of the PRNG but cannot inspect or corrupt its internal state.
6. We only consider sequential operation (i.e., no multithreading).

3.2 PRNG Security Concerns

Security of a PRNG means intuitively that a computationally-bounded attacker cannot predict its output with any practical probability. We will not state a complete formal model of PRNG security. Instead, we describe three major concerns that are its necessary prerequisites. In this paper, we focus on Concern 2, but to delineate the problem properly, we briefly discuss the others as well.

Concern 1. The entropy contained in the seed should be sufficiently large. While we leave open what exactly is considered sufficient, the issue can be further broken down as follows. First, the seed range 2^m (i.e., number of possible seeds) should be sufficiently large. For instance, the current time of the day in milliseconds provides only slightly more than $m = 26$ bits of entropy. Second, entropy is maximal for the uniform distribution. Skewed seed distributions will reduce entropy content. Third, in practice, entropy arguments are only sound relative to attacker knowledge. If one seeds the PRNG with current time but the attacker can roughly identify the moment when the seeding takes place, the effective unpredictability will be significantly below the theoretical 26 bits mentioned above.[5]

Since, hardware noise is typically an important source of PRNG seeds, ensuring sufficient seed entropy requires knowledge of said hardware (e.g., rotating hard drive vs. SSD) and its characteristics (e.g., seek time distribution). In the meantime, it has become widely known that proper seeding can be a substantial challenge in virtual machines and embedded devices [15].

Concern 2. Seed entropy should not be lost during PRNG operation. For an unbounded attacker, predicting the PRNG output of length $n \geqslant m$ should not be easier than predicting the m-bit seed itself.

Since the output is a deterministic function of the seed, it is clear that it is impossible to keep unpredictable both the output and the seed after observing the output. Barring permanent inflow of entropy, the only practical solution to this dilemma is to maximize the flow of information from the seed to the output and to prevent seed reconstruction by means of a cryptographic one-way function, shifting the adversary assumption from an unbounded to a computationally bounded one. This is indeed the way most cryptographic PRNGs operate.

Summarizing, absence of seed entropy loss (Concern 2), together with sufficient seed entropy (Concern 1), impedes an attacker at brute-forcing the seed resp. a part of the seed sufficient to predict the output.

Concern 3. It should be computationally infeasible to *analytically* invert g. A resource-bounded attacker should not be able to compute *seed* from an observed prefix of $g(seed)$.

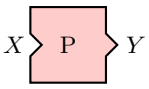
Examples of PRNGs susceptible to seed reconstruction are PRNGs that do not use cryptography, such as the linear congruential generators (LCG). A typical representative is the `drand48` generator in the C standard library.

A different example is the Dual_EC_DRBG, which utilizes a supposedly one-way elliptic curve primitive incorporating two parameters P and Q. The parameters were chosen by the NSA in an opaque manner, even though the (non-)invertibility of g and thus the security of the PRNG hinges critically on their choice. As [5] notes, "[. . .] it may be the case the adversary knows a d such that $dQ = P$. Then [. . .] a distinguisher could immediately recover the secret prestates from the output."

When considering more elaborate PRNG security models such as [7], further concerns can be identified. One can examine whether a PRNG can withstand an attacker that has some control over the choice of the seed or the capability

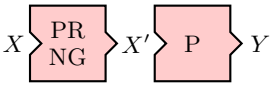
[4] Reseeding can be modeled by considering several PRNG functions. An example featuring the Yarrow PRNG is shown in Figure 4(b), Section 6.

[5] This circumstance was used in cracking the online poker PRNG in [2] or the hardware PRNG in encrypted hard drives [1]. In the latter case, the hardwired seed was close to the manufacturing time embossed on the drive.



An attacker tries to guess the secret input X after observing program output Y. The secret X typically reflects some information in the real world and has fixed entropy. Reduced information flow between X and Y makes X harder to guess, increasing security.

(a) Classical information flow analysis scenario



An attacker tries to guess X' after observing Y. The secret X' is generated by a PRNG, and its entropy depends on properties of the generator. Reduced information flow between the seed X and X' makes X' easier to guess, diminishing security.

(b) PRNG security scenario

Figure 2: Information flow scenarios illustrated

to temporarily inspect or corrupt the internal state of the PRNG. We consider such further concerns out of scope for this paper.

Of the three concerns elaborated above, the first concern cannot be solved by means of (mere) code analysis, as it involves the larger system into which the PRNG is embedded. The third concern has solution components in the form of widely-vetted one-way primitives, such as cryptographic hash functions of the SHA family. For the second concern, this paper makes a contribution in supplementing the usual manual code review with a practical technical solution.

3.3 Entropy and Information Flow

In general, entropy of a random variable X is a measure of an observer's uncertainty about its value. Information *flow* or *leakage* refers to the decrease of an (unbounded) attacker's uncertainty about a secret part of a program's initial state after observing a part of its final state. Information flow in a deterministic terminating program can be identified with the degree of injectivity of the input-output function induced by the program [18].

There is a vast body of work in information flow analysis, going back several decades. The entropy loss problems with PRNGs have entered public awareness in 2008 at the latest, but as far as we are aware the two have never been previously brought together. A part of the explanation is probably in the fact that information flow research concentrated on minimizing information flow for confidentiality, while maximizing information flow is needed for PRNG security (Figure 2).

Various entropy metrics have been defined, such as Shannon entropy, min-entropy, etc. In this paper we will be concentrating on min-entropy [25], which is a measure in bit of the probability to guess the value of X in one try. Similar arguments can be made for other metrics.

Definition 1 (Min-entropy). *The min-entropy of a random variable X*

$$H_\infty(X) := -\log \max_x \Pr[X = x] \ ,$$

where log *is a logarithm to the base* 2.

We choose min-entropy for its clear operational guarantees w.r.t. guessability. By construction, the probability of an adversary successfully guessing the value of X in one try is not larger than $2^{-H_\infty(X)}$.

Proposition 1. *If X follows a uniform distribution on a finite set of values, then*

$$H_\infty(X) = -\log \frac{1}{|\{x \in X\}|} = \log |\{x \in X\}| \ .$$

Concretely, if X follows a uniform distribution on $\{0,1\}^m$, then $H_\infty(X) = m$ bits.

In the PRNG setting, we are considering a pair of jointly distributed random variables X and Y, where $Y = g(X)$.

Proposition 2. *Let X and Y be random variables with $Y = g(X)$ and X following a uniform distribution.*

$$H_\infty(Y) = -\log \max_y \Pr[Y = y] = -\log \frac{\max_{y \in Y} |\{x \in X \mid g(x) = y\}|}{|\{x \in X\}|} = H_\infty(X) - \log \max_{y \in Y} |\{x \in X \mid g(x) = y\}| \ .$$

Corollary 3. *Under the assumptions of Proposition 2, $H_\infty(Y) \leqslant H_\infty(X)$ with $H_\infty(Y) = H_\infty(X)$ if and only if g is injective.*

In terms of information leakage, this result can be interpreted as $H_\infty(Y)$ being maximal iff the information that g leaks about X is maximal. Due to this constellation, many existing information flow analysis tools are not directly applicable to the problem, as they inherently provide upper bounds on leakage only. We, in contrast, need lower leakage bounds.

4. THE ANALYSIS METHOD

Per Corollary 3, assuring that no seed entropy is lost (i.e., the PRNG output is at least as unpredictable as the seed) is synonymous to establishing that the function g induced by a PRNG is injective. Thus, for a given PRNG implementation, the analysis computes the above-described function

$$g\colon \{0,1\}^m \to \{0,1\}^n$$

for given m and n ($m \leqslant n$) and checks whether the entropy preservation condition

$$g(seed_1) = g(seed_2) \to seed_1 = seed_2 \qquad (1)$$

holds. In other words, we check whether two distinct seeds produce distinct pseudo-random streams.

While the condition (1) is quite concise on this level of abstraction, checking it with existing program verification technology is not without challenges. Two major ones are: reasoning about cryptographic primitives and making the analysis *practical*.

4.1 Modeling Cryptographic Primitives Used in a PRNG

During its operation, a typical PRNG will invoke cryptographic primitives, such as, e.g., the SHA-1 hash function. The primitives are by design computationally hard to invert and are used, among other things, to prevent an attacker from calculating the seed from the observed PRNG output

(Concern 3). As a consequence, it is also computationally infeasible to reason about them with verification technology.[6]

For assuring absence of bugs in the implementation of the primitives, this infeasibility is not too problematic, as the primitives are standardized, with widely-available reference implementations and test suites. The non-cryptographic PRNG code, on the other hand, is not standardized, error-prone, and cannot be easily tested.

Our goal is thus to check absence of entropy loss in the latter, non-cryptographic parts, while *assuming* that cryptographic primitives do not contribute to it. We do not attempt this by considering only the code between the primitives, as reasoning about unstructured code is challenging and poorly supported in existing verification systems. Instead, we still consider a whole-program correctness property (i.e., entropy flow through the whole PRNG, from seed to output) but replace the cryptographic primitives with idealizations.

We assume that the definition of g contains several occurrences of a cryptographic function h:

$$h_i \colon \{0,1\}^{k_i} \to \{0,1\}^{l} \ .$$

Instead of the function g, we henceforth consider the function $\tilde{g}$ with the same signature, derived by replacing each h_i in g with a function $\tilde{h}_i$ of the same signature. Altogether, we consider two types of idealizations $\tilde{h}_i$.

The first type of idealization replaces h_i in g with a *projection*, i.e., a function of the form

$$\tilde{h}_i(x_1, \ldots, x_{j_i}, \ldots, x_{j_i+l-1}, \ldots, x_{k_i}) = (x_{j_i}, \ldots, x_{j_i+l-1})$$

for some user-specified $1 \leqslant j_i \leqslant k_i$. Since the output of h is l bits long, we expect the input to contain at least l bits of entropy. In practice, it suffices to assume that the l bits are supplied as one contiguous region $x_{j_i}, \ldots, x_{j_i+l-1}$; the other parts of the input can be discarded.

For each i, we let the user identify the entropic region start j_i by visually matching bit patterns occurring in the seed to the inputs of h_i. In general, there is only a small finite choice of possible values of j_i, and in practice, the choice is often even smaller, as most PRNGs supply a concatenation of a few longer bitvectors to h, each with a distinct purpose (see Section 6.4 for an example).

Now, while h and the above $\tilde{h}$ are both injections from $x_{j_i}, \ldots, x_{j_i+l-1}$, they are otherwise incomparable functions, and the properties of $\tilde{g}$ need thus not carry over to g (though very often they do). In other words, this type of idealization is unsound.

The second type of idealization replaces h_i in g with an uninterpreted function (i.e., a fresh function symbol) $\tilde{h}_i$, while adding the following injectivity axiom for $\tilde{h}_i$, based on a user-supplied j_i:

$$\begin{aligned} &\big(\tilde{h}_i(x_1, \ldots, x_{j_i}, \ldots, x_{j_i+l-1}, \ldots, x_{k_i}) = \\ &\ \tilde{h}_i(x'_1, \ldots, x'_{j_i}, \ldots, x'_{j_i+l-1}, \ldots, x'_{k_i})\big) \to \\ &\qquad \big(x_{j_i} = x'_{j_i} \wedge \ldots \wedge x_{j_i+l-1} = x'_{j_i+l-1}\big) \ . \end{aligned} \tag{2}$$

We assume that the cryptographic primitive h satisfies (2), which, e.g., for cryptographic hash functions, is a common information-theoretical approximation of the collision-resistance property. Since $\tilde{h}$ is otherwise uninterpreted, we are making strictly fewer assumptions about the nature of h in $\tilde{g}$ than in g. The properties of $\tilde{g}$ thus carry over to g, which makes this type of idealization sound.

[6] More precisely, it is not feasible to establish properties related to their injectivity (which we are interested in). In contrast, it is quite easy to prove, for instance, mere termination of the SHA-1 implementation.

Theorem 4 (Soundness). *Let $\tilde{g}$ be obtained from g by substituting each occurrence of h with an uninterpreted function $\tilde{h}$ satisfying (2). If $\tilde{g}$ is injective and h satisfies (2), then g is injective.*

Together with the soundness of the employed program verification technology, the theorem implies that the analysis is guaranteed to detect all instances of entropy loss in the scope given by m and n.

The reason for also having unsound idealization is that it produces injectivity counterexamples (pairs of program traces) that are easier to interpret for the user, as the output of each $\tilde{h}$ is just a copy of a part of its input. The user can thus easier track the flow of information in the PRNG by identifying occurrences of the same concrete bit pattern on the way from the seed to the output. This does not hold for the sound idealization. We typically perform the analysis with the unsound idealization first to find defects, and later automatically strengthen the idealization to the sound one to confirm their absence.

If no idealization can be synthesized to make (1) hold, then either the contiguous input region assumption is violated (we have not experienced this) or there is an entropy loss in the PRNG.

4.2 Scope and Limitations

While it is important to say what our analysis is designed to do, it is just as important to say what it is not designed to do.

The analysis does not consider entropy collection. The analysis will detect entropy loss in the non-cryptographic portions of any entropy extraction code, but otherwise makes no claims about its function. We are assuming that the provided seed contains maximal entropy. We are not discussing the size of the seed necessary for a desired level of security.

The analysis never examines the implementation of the cryptographic primitives, and cannot ensure that they are implemented correctly. We assume, the risk in this area is sufficiently mitigated by using reference implementations and the appropriate test suites.

The analysis only detects loss of entropy in the first N bytes of output, given a seed of M bytes. This limitation is key to achieving a tractable, automatic analysis that is sound and reasonably complete in its scope. The parameters M and N can be trivially adjusted, but checking with higher bounds consumes more resources. Values like $M = N = 40$, corresponding to 2–4 PRNG cycles are easily manageable (details in Section 6).

The analysis is complete, except for the overapproximation of the cryptographic primitive with its injectivity specification (2). If a PRNG relies for its injectivity on properties of a primitive other than (2), the analysis will produce a false alarm. This is a constellation that we yet have to encounter in practice.

If an entropy loss is detected, its severity and impact has to be established by a security analyst.

1. Complete the analysis driver template, fix m and n.
2. If necessary, disable self-seeding.
3. Configure the source code for analysis (complete the project-specific part of the Makefile).
4. Provide idealized cryptographic implementation (choose j in each occurrence of (2)).
5. Run a sanity check, checking that CBMC can symbolically execute the code.
6. Run the entropy loss analysis.
7. If analysis returns success, perform vacuity testing: introduce a bug in the implementation and repeat.
8. If analysis returns a counterexample, run the counterexample visualizer.
9. If counterexample is spurious, refine the idealized cryptographic implementation and repeat the analysis.
10. If counterexample is genuine, evaluate the defect's severity and fix it as appropriate.

Figure 3: General analysis procedure

4.3 Checking Non-Invertibility (Concern 3)

Though this is not our main concern in this paper, we can use a simple variation of the method above to check analytical non-invertibility of g. To this end, we replace the top-level injectivity assertion (1) with one of non-injectivity: $g(seed_1) = g(seed_2)$, and the assumption of injectivity of cryptographic primitives (2) with a corresponding non-injectivity assumption. This approach implements a standard check that there is no information flow from the seed to the output, or the flow passes through a one-way primitive. The one-way property of the primitive remains to be established by cryptanalysis (see the discussion of Dual_EC_DRBG in Section 3.2).

5. Entroposcope: A TOOL FOR DETECTING ENTROPY LOSS

We have implemented our analysis in an automatic detection tool named Entroposcope. The tool consists of an off-the-shelf verification tool (the bounded model checker CBMC [20]), a Java program for generating the injectivity verification condition and interpreting a potential satisfying assignment, an analysis driver template, an idealized hash function template, and a Makefile template. The overall analysis procedure is outlined in Figure 3. The individual components are described in more detail in the following.

Relational reasoner.

Entroposcope relies on CBMC to generate for a given PRNG implementation a propositional formula in conjunctive normal form

$$\phi(seed, out, aux) \tag{3}$$

that encodes the function g induced by the implementation. The arguments of ϕ are vectors of propositional variables. By construction, the formula $\exists aux.\, \phi(seed, out, aux)$ is true iff the PRNG produces the output encoded by out from the input encoded by $seed$. The auxiliary variables aux are used to represent intermediate states and for the Tseitin encoding[7].

```
RAND_add(in, M, M);
RAND_bytes(out, N);
```

(a) For the OpenSSL PRNG

```
PrngRef ref;
prngInitialize(&ref);

prngInput(ref, (BYTE*) in, M,
          SYSTEM_SOURCE, M * 8);
prngForceReseed(ref,0);
prngOutput(ref, (BYTE*) out, N);

prngInput(ref, (BYTE*) in2, M,
          SYSTEM_SOURCE, M * 8);
prngForceReseed(ref,0);
prngOutput(ref, (BYTE*) out2, N);
```

(b) For the Yarrow PRNG (incl. reseeding)

Figure 4: Analysis drivers (excerpts)

Entroposcope generates from the formula (3) produced by CBMC a formula of the form

$$\psi \wedge \phi(seed, out, aux) \wedge \phi(seed', out', aux') \wedge seed \neq seed' \wedge out = out' \quad . \tag{4}$$

This formula (4) is essentially a negation of the correctness condition (1). Its satisfiability coincides with a loss of injectivity of g and thus entropy in the PRNG. Satisfiability of (4) is checked with an off-the-shelf SAT solver (Minisat). The subformula ψ is a bit-level encoding of the idealization injectivity axiom (2).

In logic-based verification, it is often customary to encode a relational property such as (1) as a safety property via self-composition, a program transformation syntactically duplicating the program [4, 8]. Self-composition is technically challenging in presence of heap, pointer arithmetic, and complex global state. To sidestep this problem, Entroposcope operates on the logical formula level shown above rather than by transforming the program.

Analysis driver.

The starting point of the analysis is the analysis driver that defines the main function exercising the PRNG functionality. The driver consists of two parts: a generic and a PRNG-specific part. The PRNG-specific part is defined by the analyst in conformance with the PRNG API. It is supposed to seed the PRNG with an M-byte seed and generate an N-byte output. Figure 4 shows the PRNG-specific parts of the drivers for the OpenSSL and the Yarrow PRNG.

The generic part of the driver is the same for all PRNGs (not shown). It contains the scaffolding establishing the naming convention for the seed and output buffers (needed for generating (4)), as well as the code for counterexample visualization.

[7] A well-known method for encoding an arbitrary circuit as a formula in conjunctive normal form (CNF) required by a SAT solver.

SAT-based bounded model checker.

To generate the formula (3), ENTROPOSCOPE uses the SAT-based model checker CBMC [20]. CBMC is a very mature and popular bit-precise verification tool supporting almost all of ANSI C, including pointer constructs and dynamic memory allocation. Since recently, CBMC also supports programs written in Java.

CBMC executes the program symbolically starting from the given function (in our case, the main function of the analysis driver). During this process, called functions are inlined and loops are unwound to the user-specified depth. CBMC warns the user if the unwinding depth is insufficient to cover all of the program behaviors (this is known as *unwinding assertion* checking). The unwound program is transformed into the static-single-assignment (SSA) form. In this form, statements can be interpreted as equations over bitvectors. The equations are combined and reduced to a formula of propositional logic in a process resembling synthesis of arithmetic circuits. The formula is flattened into conjunctive normal form (CNF) $\bigwedge_i \bigvee_j L_{i,j}$, where each literal $L_{i,j}$ is either a propositional variable or its negation. The formula is exported in standard DIMACS format. The metadata embedded in the formula allows us to identify the variables representing the seed and the output.

Verification condition generator.

The subformula $\phi(seed', out', aux')$ in (4) is obtained by syntactically duplicating $\phi(seed, out, aux)$ with fresh variables. The schematic equality of bitvectors in (4) is, in reality, encoded on the individual bit level; for the inequality, Tseitin encoding is used to obtain an equisatisfiable formula in conjunctive normal form.

Counterexample visualizer.

If the SAT solver finds a satisfying assignment for (4), ENTROPOSCOPE interprets it and outputs a pair of distinct seeds and a common PRNG output that form a suspected witness for entropy loss. The user needs to analyze the counterexample to understand if it is spurious (i.e., due to the insufficient choice of the cryptographic primitive idealization) or if it is a real problem in the PRNG. To aid the user, the analyzer also generates C preambles from the satisfying assignment and runs the PRNG once for each of the seeds. The inputs and the idealized output of each cryptographic primitive invocation together with any other debugging output that the user may add are then displayed in a side-by-side diff.

6. ANALYZING REAL-WORLD PRNGS: OPENSSL AND OTHERS

We applied ENTROPOSCOPE to a number of PRNGs, carrying out the analysis according to the procedure outlined in Figure 3. In cases where no defects were found, we injected and detected synthetic bugs.

ENTROPOSCOPE finds the **Android PRNG** bug described in Section 2 and verifies the effectiveness of the official patch.

We found no entropy loss in the **Yarrow PRNG** [16] in the version used in Apple's XNU kernel (part of iOS and OS X). One of the scenarios we analyzed included reseeding (Figure 4(b)): ENTROPOSCOPE shows that the old and the new seed fully influence the output before and after reseeding respectively. On the other hand, attempts to analyze Yarrow with $M = 40, N = 40$ (without reseeding) immediately fail due to the fact that Yarrow's generator state is only 20 bytes large (a restriction prominently declared in [16]).

With ENTROPOSCOPE, we found a previously undiscovered entropy loss in the **Libgcrypt PRNG**, which we briefly describe in Section 6.9.

In the following we report in detail the results of our analysis of the **OpenSSL PRNG**. We also analyzed the **PRNG in BoringSSL**, Google's drop-in replacement for OpenSSL, and found no defects in the latter.

6.1 Structure of the OpenSSL PRNG

The relevant functions of the OpenSSL PRNG API 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. They are shown as pseudocode in Figure 5. The code has been simplified for presentation, eliding irrelevant features such as provisions for multi-threading (postponed as future work), function pointer indirection, etc. The PRNG is also parametric in the choice of a cryptographic hash function; we present the default instantiation with SHA-1. Nonetheless, please note that, unless explicitly mentioned otherwise below, we are analyzing the actual unmodified source code of OpenSSL.

OpenSSL PRNG maintains two entropy pools: a 20-byte buffer named `md` and a 1023-byte circular buffer named `state`. Entropy added to the PRNG using `RAND_add` is split into 20-byte chunks and hashed chunkwise into `state`. The chunks are chained together using `local_md`, a thread-local copy of the `md` buffer. Output generation proceeds in chunks of 10 bytes. First, a 20-byte hash is generated from the next 10 bytes of `state` and two counters; `local_md` is used for chaining. Then, the hash is split, with 10 bytes forming output, while the other 10 bytes are XORed 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.

6.2 Entropy Sources and the Analysis Driver

The analysis driver is straightforward, containing a call to `RAND_add` followed by a call to `RAND_bytes`. The core part of the driver is shown in Figure 4(a). A minor complication arises due to the fact that the OpenSSL PRNG self-seeds, i.e., automatically incorporates entropy from several sources beyond what is supplied by the driver: (i) entropy collected by an OS-specific routine (`RAND_poll`) added on first call to `RAND_bytes`, (ii) PID, time of day, and hardware RNG entropy (if available) added at every call to `RAND_bytes`, and (iii) the content of the (potentially uninitialized) output buffer supplied to `RAND_bytes`.

We choose to ignore these auxiliary sources, as they offer little conceptual added value from the analysis perspective. Technically, we set corresponding compilation flags (`PURIFY`, `GETPID_IS_MEANINGLESS`) and supply empty implementations for `RAND_poll` and `time`. Alternatively, it is trivial to include the output of relevant functions and content of buffers as an extra part of the conceptual seed in an extended scenario.

6.3 Pool Stirring

After self-seeding, but before generating first output, the OpenSSL PRNG "stirs" `state`. During the stirring process, `RAND_add` is called with a constant 20-byte input (literally 20 dot characters) for a total of at least 1023 bytes. The

```
1  long md_count_0  =  0;
2  long md_count_1  =  0;
3  char state[1023] = {0};
4  char md[20]      = {0};
5  int state_index  =  0;
6  int stirred_pool =  0;
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 = sha1(local_md
13       ¦ state[state_index..state_index+20>]
14       ¦ buf[i..i+20<]
15       ¦ md_count_0 ¦ md_count_1);
16     md_count_1++;
17     state[state_index..state_index+20>] ^=
           local_md;
18     state_index = (state_index + 20) % 1023;
19   }
20   md ^= local_md;
21 }
22
23 void RAND_bytes(const void *buf, int num) {
24   if(!stirred_pool++) stir_pool();
25   char local_md[] = copyOf(md);
26   md_count_0++;
27   int i = 0;
28   while(num > 0) {
29     local_md = sha1(local_md
30       ¦ md_count_0 ¦ md_count_1
31       ¦ state[state_index..state_index+10>] );
32     state[state_index..state_index+10>] ^=
           local_md[0..10];
33     buf[i..i+10<] = local_md[10..20];
34     state_index = (state_index + 10) % 1023;
35     num -= 10;
36     i += 10;
37   }
38   md = sha1(md_count_0 ¦ md_count_1 ¦ local_md ¦
         md);
39 }
```

Notation:

- We assume that every array has an associated implicit length.
- `buf[a..b]` denotes the sub-array of `buf` starting with `a` and ending with `b` (exclusive).
- Array actions such as assignments and XORs are to be understood component-wise. If the right-hand side of an assignment denotes more locations than the left-hand side (Line 33), the superflous locations are ignored.
- `buf[a..b>]` treats `buf` as cyclic and amounts to `buf[a..b]`, if `b<=len(buf)`, and `buf[a..len(buf)]`∪`buf[0..b-len(buf)]` otherwise.
- `buf[a..b<]` ignores accesses past the right array bound, i.e., it is equivalent to `buf[a..min(b,len(buf))]`.
- `sha1(a ¦ b ¦ c)` denotes the hash of the concatenation of `a`, `b` and `c`.

Figure 5: Simplified pseudocode of the OpenSSL PRNG

purpose is to better distribute the entropy throughout the pool via the chunk chaining mechanism. At the same time, stirring, obviously, does not increase the entropy content of the pool.

We abstract from pool stirring in the analysis scenario. It is a small and relatively simple piece of code: a single loop repeatedly calling `RAND_add`. At the same time, stirring imposes a prohibitively high computational tax on the analysis. Instead, we assume that we are already sufficiently exercising the `RAND_add` functionality through the analysis driver. Technically, we disable stirring by instructing the model checker to ignore the appropriate loop.

6.4 Idealized Cryptographic Functionality

Abstract description.

OpenSSL PRNG uses three static occurrences of a cryptographic primitive (SHA-1 hash per default). Each occurrence can be invoked several times depending on the parameters M, N. As described in Section 4.1, we replace the primitives by idealizations.

The first occurrence transfers the seed entropy into the pool and is invoked in `RAND_add` as:

```
h1(local_md ¦ state[state_index..state_index+20>]
   ¦ buf[i..i+20<] ¦ md_count_0 ¦ md_count_1)
```

We idealize it as an injection from `buf[i..i+20<]` (third argument), based both on its name and it carrying the seed material, as shown by the counterexample visualizer.

The second occurrence is for output generation and invoked in `RAND_bytes`:

```
h2(local_md ¦ md_count_0 ¦ md_count_1
   ¦ state[state_index..state_index+10>])
```

We discuss this occurrence in detail in Section 6.5.

The third occurrence updates the global state upon completion of `RAND_bytes`:

```
h3(md_count_0 ¦ md_count_1 ¦ local_md ¦ md)
```

We idealize it as an injection from `local_md`, though it is only relevant, if one calls `RAND_bytes` more than once.

As the example shows, there is only a very limited choice of argument for the injection, and roughly half of them—the counters—can be eliminated outright. For each invocation of a cryptographic primitive, Entroposcope's counterexample visualizer shows the exact callsite and the arguments supplied to the primitive, its output, which argument potentially contains or is influenced by the seed material, and whether each invocation and the overall PRNG are currently injective. This feedback effectively assists the user in synthesizing the appropriate idealized functionality, i.e., choosing the right j in (2).

Implementation.

On the implementation level, the hash function is implemented by a stateful object with a stream-based interface. The hash object is initialized by calling `MD_Init`, data to be hashed is supplied via `MD_Update`, and finally, the hash value is obtained by calling `MD_Final`. Passing a concatenation of several buffers as an argument to the hash function, which we use in the pseudocode, is implemented by repeated calls to `MD_Update`. We instrument each call with a counter that makes it possible to distinguish the three logical occurrences of the primitive shown above even though they use the same interface.

6.5 OpenSSL PRNG Loses Entropy in Output by Design

Attempts to synthesize an idealization for the hash function invocation `h2` shown in Section 6.4 show a peculiar problem. Abstracting away from details, the entropy flow from the 20 byte of `state` to the 20 byte of PRNG's output in `RAND_bytes` can be represented as follows (see Figure 5 for notation):

```
o1  = h2(... ¦ state[ 0..10]);
o2  = h2(... ¦ state[10..20]);
out = o1[10..20] ¦ o2[10..20];
```

Note that the entropy in `state[0..10]` and `state[10..20]` is "spread" over `o1` and `o2` respectively, yet `o1[0..10]` and `o2[0..10]` are discarded. Thus, some of the entropy contained in `state` will not make it to the output in this computation.

The fact that this particular construction is prominent and pervasive throughout the PRNG suggests that it is not an accident but a design decision by OpenSSL. While we do not agree with it, we choose to mask the issue and concentrate on uncovering other problems.

We achieve the masking by defining `h2` as injective from its last argument, `state[state_index..state_index+10>]`, to the last 10 byte of its output (as opposed to an injection from a 20-byte argument to the full 20-byte output as demanded by (2)). It is clear that the actual SHA-1 implementation does not have this property, but the unsound idealization makes the code fragment above injective from `state[0..20]` to `out`, as the discarded parts of `o1` and `o2` no longer contain entropy. This deviation from (2) silences the alarm.

6.6 Entroposcope Detects the Debian Incident

The Debian incident occurred when the Debian OpenSSL maintainer, as part of a campaign against memory-related errors, patched OpenSSL to be memory-safe. As mentioned above, OpenSSL uses uninitialized memory as an additional source of entropy for its PRNG, but the patch not only eliminated that but unintentionally disabled all PRNG seeding. Or rather almost all, as the current PID was still used as a source of entropy in `RAND_bytes` to prevent stream duplication upon forking (this part is elided from Figure 5).

The incident corresponds to the Debian maintainer having deleted Line 14 from the source code in Figure 5. This way, the entropy-containing `buf[i..i+20<]` is no longer read from in the rest of the PRNG. Defining an idealized hash function that ensures injectivity of the PRNG (like the first one in Section 6.4) becomes impossible. For any given idealization, Entroposcope produces a counterexample to injectivity (any two seeds will produce the same output after invoking SHA-1 in `RAND_add`), which immediately leads to the detection of the problem. The counterexample visualizer helps pinpoint the exact place where the problem occurs.

Once the idealized functionality is developed, Entroposcope can also be included as a completely automatic check into the continuous integration process in order to prevent introduction of bugs during maintenance.

6.7 Entroposcope Detects A Previously Undiscovered Anomaly in the OpenSSL PRNG

When we applied Entroposcope to the OpenSSL PRNG, the tool immediately detected a bug in a piece of code that manages entropy in the large pool. The defect is close in spirit to the one in the Android PRNG. Even though it turned out not exploitable in the larger context, the incident demonstrates that Entroposcope is effective at doing what it is designed to.

The problem occurs in the code implementing the circular buffer `state` that is the large entropy pool. The following code is encountered towards the end of `RAND_bytes`:

```
for (i=0; i<MD_DIGEST_LENGTH/2; i++) {
  state[st_idx++]^=local_md[i];
  if (st_idx >= st_num) st_idx=0;
  if (i < j) *(buf++)=local_md[i+MD_DIGEST_LENGTH/2];
}
```

In the pseudocode implementation of Figure 5, this code corresponds to Lines 32–33. After the analysis driver has seeded the PRNG with M bytes, we reach this code for the first time with `st_idx` = `st_num` = M. The former variable is the current index into `state`, while the latter is the number of bytes in the pool that are filled with seed data.

The code above is itself part of an outer loop. It is easy to see that the sequence of values of `st_idx` at inner loop entry is $M, 9, 19, 29, \ldots$ The correct sequence, in contrast, would have been $0, 10, 20, \ldots$ or at least $M, 10, 20, \ldots$ (a special case in the code reading from the pool treats `st_num` as zero). The significance of these values is that they are used to decompose the pool into chunks supplying entropy in each cycle of output generation. The code above forces chunks of the form `state[0..10]`, `state[9..19]`, `state[19..29]`, ... While each chunk is 10 bytes long, the first two chunks overlap by one byte. Accordingly, the last byte of entropy in the pool (byte number $M - 1$) is never used for output generation.

The code above thus suffers from a small but real entropy loss, though this defect is masked during normal operation. The bug can only manifest itself when the entropy pool is not full (`st_num` < 1023). In practice, though, the pool is full, alone by virtue of self-seeding and pool stirring. Under these circumstances, another instance of wrapping code comes into effect, correctly wrapping the index at the right bound of the `state` buffer.

The bug can be fixed—which Entroposcope verifies—by prepending the loop shown above with

```
if (st_idx >= st_num) st_idx=0;
```

From a larger perspective, though, an overhaul of the convoluted pool index manipulation in the whole PRNG could be beneficial.

6.8 Statistics

Entroposcope completes a single analysis run of the OpenSSL PRNG for $M = N = 40$ in 33 seconds on a modest desktop computer (Intel Core i7 860 2.80GHz CPU). Of this time, translation of the PRNG behavior into a formula by CBMC took 7 seconds, generating the relational proof obligation 16 seconds, SAT solving 7 seconds, and generation of a counterexample from a satisfying assignment 3 seconds. The size of the formula supplied to the SAT solver was 191 megabytes in DIMACS format. OpenSSL is the largest and most complex PRNG we considered.

6.9 Entroposcope Detects a Previously Undiscovered Entropy Loss in Libgcrypt PRNG

For reasons of space, we can only superficially describe the problem here; details can be found in the extended version of the paper [11].

The Libgcrypt PRNG describes itself as modeled after the one in [13]. Its critical function `mix_pool` perturbs the state of the generator by repeatedly hashing overlapping regions of the entropy pool back into the pool. The Libgcrypt implementation deviates from [13] in several ways. In particular, the region of the pool supplied to the hash function in Libgcrypt is not contiguous but contains a 20-byte "hole". The hole has the consequence that mixing a full pool reduces its entropy by at least 20 bytes. Other deviations from [13] make the flow of entropy difficult to understand. The problem was fixed by the Libgcrypt developer after we reported it.

7. A MINI-MUSEUM OF ENTROPY LOSS

Entropy loss is not limited to (cryptographic) PRNGs. All entropy-processing applications are susceptible. To demonstrate the importance of the concept, we briefly discuss three instances taken from different application domains. While Entroposcope detects injectivity failures in all of these cases, we are not claiming that it is the best means to deal with the problem. In scenarios like the third one (Linux kernel), the routine use of the tool, on the other hand, is probably advisable.

7.1 Debit Card PINs

A requirement of the early eurocheque debit card system ("EC-Karte" in Germany) was that an ATM can check the card PIN offline. To this end, the PIN was derived by encrypting and transforming the public data stored on the card. As the source of entropy the banks chose four hex digits of the ciphertext. To obtain a decimal PIN, the hex digits $A - F$ were replaced by $0 - 5$ correspondingly. In the final step, any occurrence of an initial zero was replaced by one, as the banks deemed a PIN beginning with a zero confusing for the customer. This decimalization produced a skewed PIN distribution, with small digits more common than large digits. Amplified by other features of the cards, the entropy loss resulted in four-digit PINs that were easier to guess than a uniformly distributed three-digit PIN [21]. The problem was solved when offline PIN checking was retired and online checking became mandatory.

7.2 Online Poker Card Deck Shuffling

ASF Software Inc. was one of the first online poker software providers. Since cheating by the operator is a concern in online gambling, the company published its deck shuffling source code with the intent to increase client confidence in its software.

The analysis of the software reported in [2] has shown that it fails on all three security concerns. Concern 1 was violated due to seeding the PRNG with current time, which was not only limited in range to 32 bits (a shuffled deck of cards contains nearly 226 bits of entropy) but also roughly known to the attacker. The software furthermore also failed Concern 2. The shuffling process produced a skewed distribution of decks containing less entropy than a properly shuffled deck or the PRNG output used to control the shuffle.

7.3 Linux ASLR Vulnerability

Address space layout randomization (ASLR) is a popular OS mechanism to mitigate buffer overflow attacks. An entropy loss in the Linux kernel prior to version 3.19-rc3 reduces the effectiveness of the stack randomization by a factor of four.[8]

In the following code (assuming x86_64 architecture), the entropy source are the lowest 22 bits (`STACK_RND_MASK=0x7ff`) of the value obtained from `get_random_int()` (Line 7). In Line 8, the result is shifted left by `PAGE_SHIFT` = 12 bits, causing an unsigned overflow of the 32-bit `random_variable`. $22 + 12 - 32 = 2$ bits of entropy inadvertently never reach the return value of the function. The solution is to declare `random_variable` as a 64-bit integer, as was intended.

```
1  static unsigned long randomize_stack_top(unsigned
       long stack_top)
2  {
3    unsigned int random_variable = 0;
4
5    if ((current->flags & PF_RANDOMIZE) &&
6      !(current->personality & ADDR_NO_RANDOMIZE)) {
7      random_variable = get_random_int() &
           STACK_RND_MASK;
8      random_variable <<= PAGE_SHIFT;
9    }
10 #ifdef CONFIG_STACK_GROWSUP
11   return PAGE_ALIGN(stack_top) + random_variable;
12 #else
13   return PAGE_ALIGN(stack_top) - random_variable;
14 #endif
15 }
```

Another entropy loss in the ASLR implementation on Linux has also been reported.[9]

8. RELATED WORK AND ALTERNATIVES

Previous own work.

This paper builds upon insights collected during the pilot study [10], where we proved correctness of the (fixed) Android PRNG in an interactive theorem prover. The current paper significantly extends and improves this work, both in the information-theoretical development and in practicality.

The fact that proof construction previously took hours and required substantial user interaction motivated us to redesign the approach, changing the underlying program verification architecture and the idealization of cryptographic primitives. The result is a vastly improved pragmatics, with drastically reduced need for user interaction, counterexamples to PRNG correctness, and turnaround times in the magnitude of seconds rather than hours.

Functional verification and testing.

Of course, it is possible to state and verify a functional specification of the PRNG implementation with existing verification technology. However, since PRNG output lacks intrinsic meaning, such a specification would have to closely mimic the implementation and thus be complex and tedious to write. It would be difficult to understand it and ascertain its adequacy; neither would it be possible to reuse it for another PRNG. The information flow specification (1), 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.

Functional testing is rare for similar reasons. The only test suite containing reference seeds and corresponding out-

[8] H. Marco, CVE-2015-1593, http://hmarco.org/bugs/linux-ASLR-integer-overflow.html

[9] http://hmarco.org/bugs/linux-ASLR-reducing-mmap-by-half.html

puts we are aware of is part of the NIST SP 800-90 standard for PRNGs implementing it. Note that such a suite essentially constitutes a *regression* test, i.e., it only assures that all implementations of the standard perform in the same way. Unit testing of PRNGs is often impeded by the lack of modularity in implementations.

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 SP800-22 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.

Given the single-stream nature of the mentioned tests, it is reasonably safe to expect that any modern cryptographic PRNG will pass them, even in presence of entropy loss. In [27], it was empirically demonstrated that the defective Debian OpenSSL PRNG passes the NIST SP800-22 test.

The first significant advance in quality assurance for cryptographic PRNGs has been made only recently in the form of LIL testing [27]. The test estimates the distance in a particular statistical metric between a set of bit sequences generated by running a PRNG and a similarly-sized set of truly random sequences. The authors show that the defective Debian OpenSSL PRNG is associated with a significantly larger distance to uniform randomness than the intact PRNG when considering 1000 sequences of 2GB each.

The LIL test is agnostic of the exact way the sequences are generated. The advice given by the authors is to generate them in a way reflecting the particular application of interest. The advantage of such a test (as of testing in general) is that it can be made to reflect the actual deployment scenario, including the larger system, up to the hardware layer. On the other hand, it requires insight into the exact planned PRNG usage pattern, as different patterns can produce significantly different results.

For example, a single-threaded Debian OpenSSL PRNG client fails the LIL test as carried out in [27]. The test was run in a simulated environment on a Windows system, where PIDs are recycled with a substantial probability. Since the PID is the only source of entropy in the broken Debian PRNG (in the single-threaded case), many generated sequences were identical. The same test on an actual Debian system would have (as far as we understand) succeeded, as Linux only recycles PIDs after they wrap around. Finally, testing cannot detect less severe (but still cryptography-affecting) instances of entropy loss where the probability of repeated streams is sufficiently low.

Information flow analyses.

Many information flow analyses have been developed—most of them with the use case of minimizing information flow. Maximizing information flow, in contrast, plays a much less prominent role. The first appearance we are aware of is as *required information release* in [6]. The concept has been revisited in [24] for the purpose of optimizing secure multi-party computation protocols. The latter work develops tooling based on symbolic execution and SMT solving and is the one closest in spirit to ours.

A research branch measuring the amount of information flowing in programs (rather than checking absence or maximality of flows) is Quantitative Information Flow analysis (QIF). Several methods and tools for QIF exist, including our own for C programs [19]. Yet, the complexity of PRNGs coupled with large desired flows puts this application beyond the reach of state-of-the-art QIF tools.

Some of the entropy loss instances described in this paper are "simple" enough to be detectable by some form of (to be developed) static or even dynamic dataflow analysis. This is the case where some part of the entropy-carrying state is never read (Sections 6.6, 6.7) or immediately overwritten with clearly irrelevant values (Section 2). Other instances (Section 6.9) feature complex flows that require, at the least, (unsound) cryptographic primitive idealization to be detectable without losing practical completeness. Furthermore, due to their lack of precision and impossibility to use sound idealizations, aforementioned analyses can only detect bugs. Entroposcope, on the other hand, can prove their absence within the analyzed scope. Nonetheless, dataflow analyses have the advantage of scalability and warrant further investigation.

Manual PRNG analysis.

A number of publications report results of manual analysis of individual cryptographic PRNGs or PRNG classes, among them [12–14, 17, 22, 23, 26] and [3, 7, 9]. 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.

Complementary research.

There exists a wide body of work on randomness in cryptography and security. Research subjects include secure theoretical PRNG constructions, cryptographic schemes that either do not require randomness or degrade gracefully in presence of bad randomness, empirical studies of randomness in practice through cryptographic artifacts collected from public sources, studies and methods of entropy collection in embedded devices or virtual machines. Since this research is orthogonal to the immediate goals or techniques of our work, we do not survey it here.

9. CONCLUSION

We have presented the first static analysis and one of the first technical quality assurance measures for cryptographic PRNG implementations. We implemented our analysis in a tool named Entroposcope and demonstrated its practicality by analyzing five real-world PRNGs implementing different designs, including one of the most complex in wide deployment today.

In the experiments, Entroposcope uncovered PRNG defects effectively and with reasonable effort. Five instances of entropy loss showcase the tool's capabilities, including previously undiscovered anomalies in OpenSSL and Libgcrypt. The detection is systematic (i.e., it is not based on examples, patterns, or heuristics) and sound. A negative analysis outcome guarantees absence of entropy loss in the analyzed scope.

The analysis we presented is intended to detect a large but specific class of implementation mistakes. It is thus not intended to replace expert review, nor, more generally, research into good PRNG designs and deployment practices. Yet, it is the first instance where the work of a PRNG security analyst in a particular area can be supported by an effective tool.

Interesting directions for future research include extensions to more powerful attacker models, scenarios with multithreading, and automatic counterexample-guided synthesis of cryptographic idealizations.

Acknowledgments

This work was in part supported by the German National Science Foundation (DFG) under the priority program 1496 "Reliably Secure Software Systems – RS3."

The authors would like to thank the anonymous CCS 2016 reviewers for the helpful suggestions for improving this paper and for pointing out [6] and [24]. Special thanks to the CBMC team and in particular Michael Tautschnig for the excellent tool and support.

10. REFERENCES

[1] G. Alendal, C. Kison, and modg. got HW crypto? On the (in)security of a self-encrypting drive series. Cryptology ePrint Archive, Report 2015/1002, 2015. https://eprint.iacr.org/2015/1002.

[2] B. Arkin, F. Hill, S. Marks, M. Schmid, T. J. Walls, and G. McGraw. How we learned to cheat at online poker: A study in software security. *Developer.com*. Originally published on 1999-09-28. Available at http://www.cigital.com/papers/download/developer_gambling.php.

[3] B. Barak and S. Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In *ACM CCS*, 2005.

[4] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *IEEE Computer Security Foundations Workshop*, 2004.

[5] D. R. L. Brown and K. Gjøsteen. A security analysis of the NIST SP 800-90 elliptic curve random number generator. In *CRYPTO*. Springer, 2007.

[6] S. Chong. Required information release. In *IEEE Computer Security Foundations Symposium (CSF)*, 2010.

[7] M. Cornejo and S. Ruhault. Characterization of real-life PRNGs under partial state corruption. In *ACM CCS*, 2014.

[8] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Security in Pervasive Computing*, 2005.

[9] 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 *ACM CCS*, 2013.

[10] F. Dörre and V. Klebanov. Pseudo-random number generator verification: A case study. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*. Springer, 2015.

[11] F. Dörre and V. Klebanov. Practical detection of entropy loss in pseudo-random number generators (extended version). Technical Report 2016-12, Karlsruhe Institute of Technology, 2016. http://dx.doi.org/10.5445/IR/1000058113.

[12] L. Dorrendorf, Z. Gutterman, and B. Pinkas. Cryptanalysis of the Windows random number generator. In *ACM CCS*, 2007.

[13] P. Gutmann. Software generation of practically strong random numbers. In *USENIX Security*, 1998.

[14] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the Linux random number generator. In *IEEE Security and Privacy*, 2006.

[15] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security*, 2012.

[16] J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator. In *Workshop on Selected Areas in Cryptography*, 1999.

[17] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudorandom number generators. In *Workshop on Fast Software Encryption (FSE)*, 1998.

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

[19] V. Klebanov, N. Manthey, and C. Muise. SAT-based analysis and quantification of information flow in programs. In *Quantitative Evaluation of Systems (QEST)*. Springer, 2013.

[20] D. Kroening and M. Tautschnig. CBMC – C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014.

[21] M. G. Kuhn. Probability theory for pickpockets—ec-PIN guessing. In *Workshop on Cryptography and Network Security*. DIMACS Research and Education Institute, 1997. Available at http://www.cl.cam.ac.uk/~mgk25/ec-pin-prob.pdf.

[22] P. Lacharme, A. Röck, V. Strubel, and M. Videau. The Linux pseudorandom number generator revisited. Cryptology ePrint Archive, Report 2012/251, 2012. http://eprint.iacr.org/2012/251.

[23] K. Michaelis, C. Meyer, and J. Schwenk. Randomly failed! The state of randomness in current Java implementations. In *Conference on Topics in Cryptology (CT-RSA)*. Springer, 2013.

[24] A. Rastogi, P. Mardziel, M. Hicks, and M. A. Hammer. Knowledge inference for optimizing secure multi-party computation. In *PLAS*. ACM, 2013.

[25] G. Smith. Quantifying information flow using min-entropy. In *Quantitative Evaluation of Systems (QEST)*. IEEE Computer Society, 2011.

[26] F. Strenzke. An analysis of OpenSSL's random number generator. In *EUROCRYPT*, 2016.

[27] Y. Wang and T. Nicol. On statistical distance based testing of pseudo random sequences and experiments with PHP and Debian OpenSSL. *Comput. Secur.*, 53(C):44–64, 2015.

[28] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *Conference on Internet Measurement (IMC)*. ACM, 2009.

APPENDIX

A. DETAILS OF THE PROBLEMS DISCOVERED IN THE LIBGCRYPT PRNG

In the following we describe a design flaw in the mixing function of the Libgcrypt PRNG (also used in GnuPG), which we discovered with the help of Entroposcope. Due to the flaw, mixing the full entropy pool reduces the stored entropy amount by at least 20 bytes. Furthermore, the flaw makes a part of the PRNG output completely predictable. This bug existed since at least 1998 in all GnuPG and Libgcrypt versions and is tracked as CVE-2016-6313. The flaw was fixed after we had reported our findings to the Libgcrypt author.

A.1 The Mixing Function

The mixing function in question is `mix_pool` in `random-csprng.c`. It is supposed to perturb the content of the entropy pool while maintaining its entropy content. The latter requirement (freedom from entropy loss) means that the mixing function ought to be injective, i.e., transform distinct pools into distinct pools. This was not the case.

It remains to note that the length L of the entropy pool is $30 \cdot 20 = 600$ bytes in the default configuration. A comment in the source states that the Libgcrypt PRNG is modeled after a proposal by Gutmann [13].

Original Proposal by Gutmann.

Figure 6 shows the original proposal for a mixing function presented in [13]. The mixing function operates in cycles. The top rectangle represents the entropy pool at the beginning of the (first) cycle, while the bottom rectangle shows the pool at the end of the cycle. A 20-byte hash of a sliding window first covering bytes [0,84) in the pool is computed and the result is used to overwrite the bytes [20,40). After that, the sliding window is shifted right by 20 bytes and the next cycle commences. If a part of the sliding window extends beyond the end of the pool, it is wrapped around. The function terminates when all bytes in the pool have been rewritten (i.e., after 30 cycles).

Implementation in Libgcrypt.

There are two relevant differences between the proposal in [13] and the Libgcrypt implementation. The more general difference is that the Libgcrypt sliding window has a "hole" (Figure 7b). The hash here is computed from the bytes $[0, 20) \cup [40, 84)$. The bytes [20,40), shown as hatched in the figure, are no longer part of the hash input. The more particular difference is that the first cycle (Figure 7a) deviates from the other cycles (Figure 7b). Here, the hash is computed from the bytes $[L - 20, L) \cup [0, 44)$.

Later on, we show that the hole in the sliding window decreases the security of the PRNG.

Properties of the Libgcrypt Mixing Function.

Proposition 5. *The pool is the only (effective) reservoir of entropy.*

We could identify only two potential threats to this claim in the code:

- There is a small auxiliary entropy buffer (physically trailing the pool) used in the hash calculation, but its

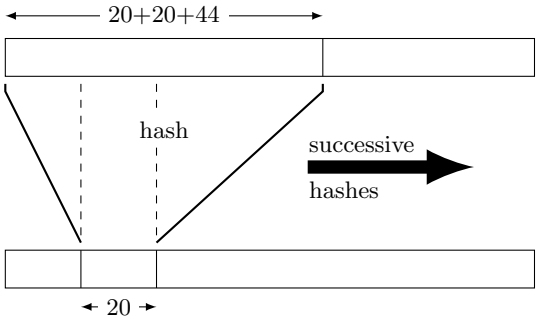
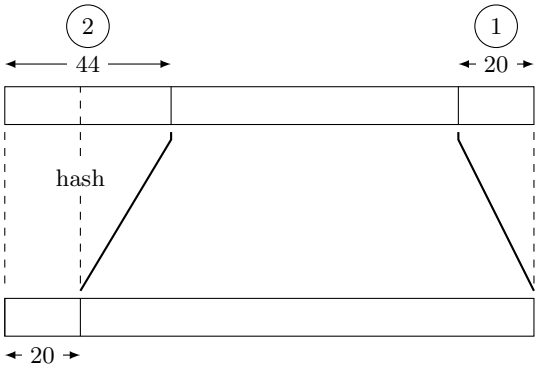
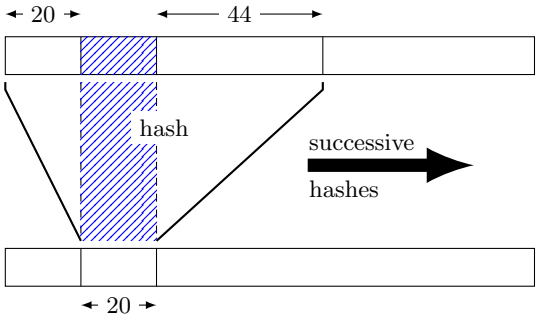


Figure 6: Mixing function proposed in [13]



(a) first cycle



(b) second and following cycles

Figure 7: Mixing function in Libgcrypt

content is completely overwritten by data from the pool at the beginning of each cycle.

- The hash context is reused between cycles and contains a 20-byte chaining buffer. Yet, we note that the buffer's content at the end of a cycle is identical with the output of the hash function. The hash function output from cycle i, in its turn, is stored in the pool and fed into the hash function in cycle $i+1$, after the sliding window shifts right. The chaining buffer, thus, injects no additional entropy into each hash operation beyond what is already contained in the pool.

COROLLARY 6. *Each cycle i, thus, induces a mathematical function f_i transforming a pool into another pool. The whole mixing function is a composition of such cycle functions:*

$$\mathit{mix\_pool} = f_{30} \circ \ldots \circ f_1 \ .$$

A.2 Entropy Loss

PROPOSITION 7. *Consider an entropy pool containing L bytes of data with an entropy of L bytes. After applying the Libgcrypt mixing function, the entropy content of the pool is at most $L - 20$ bytes.*

PROOF. It is clear that the mixing function can only be injective, if every cycle function f_i is injective.

Yet, for any $2 \leqslant i \leqslant 30$, the corresponding cycle function f_i as implemented in Libgcrypt (Figure 7b) is not injective. For example, any two pools differing (only) in the byte range [20,40) will produce the same pool after applying f_2. □

A.3 Output Predictability

In general, an entropy loss makes it easier for an attacker to brute-force the PRNG state. In this particular case, the flaw causing the entropy loss also caused a partial trivial output predictability.

PROPOSITION 8. *Whenever the Libgcrypt PRNG generates L bytes of output[10], the last 20 bytes are trivially predictable from the first $L - 20$ bytes.*

PROOF. The PRNG produces output by deriving a so-called key pool of length L from the main entropy pool (in a way irrelevant here), mixing it, and returning its content to the client. Due to the design of the mixing function, the bytes $[L-20, L)$ of the key pool (and thus of the output) are obtained by hashing the bytes $[L-40, L-20) \cup [0, 44)$. The hash context chaining buffer at this point coincides with the bytes $[L-40, L-20)$, as explained above. We note that all of the bytes in that range are contained in the first $L - 20$ bytes of the key pool and thus of the output. □

In other words, by taking the bytes $[L-40, L-20) \cup [0, 44)$ of the output, and hashing them with the hash context chaining buffer set to bytes $[L-40, L-20)$, an attacker can perfectly predict the bytes $[L-20, L)$ of the output. Simple proof-of-concept code confirms our finding.

[10] For simplicity, we assume that an L-multiple of bytes were generated previously.

A.4 Details of the Analysis Process

We began the analysis with preparing the analysis driver and distilling the relevant deterministic part of the PRNG. In the manner similar to the OpenSSL case (Section 6.2), we disabled self-seeding, fork detection and PID incorporation, provisions for memory scrubbing, etc. For tractability, we also reduced the pool size. The scenario $M = N = L = 120$ can be handled by the tool in under one minute.

For the idealization of the hash function, we needed to make a choice of the entropy source within the sliding window. We experimented with the first, second, and third 20-byte block of the 64-byte sliding window, but none of these choices allowed proving absence of entropy loss. At this point, we started to build a more detailed mental model of the PRNG operation, for which the description in [13] was helpful. Aided by the traces shown in the counterexample visualizer, we quickly confirmed our suspicion that we were facing a real flaw. A closer manual investigation of the flaw revealed that it was not only causing an entropy loss but partial trivial output predictability as well.

After fixing the PRNG by eliminating the hole in the sliding window, it was easy to check that choosing the second 20-byte block in the sliding window as the entropy source allows proving entropy preservation.

A.5 Conclusion and Aftermath

After finding the flaw, we reported it to Werner Koch, the author and maintainer of Libgcrypt. We would like to thank Werner for a productive discussion. Bugfix releases are available as of 2016-08-17[11]. Each mixing cycle now hashes a contiguous 64-byte region of the pool, maintaining injectivity of the mixing function.

Interestingly, the deviating design of the first cycle of the mixing function in the vulnerable PRNG makes the flaw undetectable with any dataflow analysis that does not replace the hash function by an idealization.

Please note that we make no claims about the effect of the flaw on the security of generated keys or other artifacts.

[11] https://lists.gnupg.org/pipermail/gnupg-announce/2016q3/000395.html